O'REILLY®

Software Engineering at Google

Lessons Learned from Programming Over Time

Curated by Titus Winters,
Tom Manshreck & Hyrum Wright

Software Engineering at Google

Lessons Learned from Programming Over Time

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Titus Winters, Tom Manshreck, and Hyrum Wright

Software Engineering at Google

by Titus Winters , Tom Manshreck , and Hyrum Wright

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

- Acquisitions Editor: Chris Guzikowski
- Development Editors: Alicia Young and Nicole Taché
- Production Editor: Christopher Faucher
- Copyeditor: Octal Publishing, LLC.
- Proofreader: Holly Baier Forsyth
- Indexer: Ellen Troutman-Zaig
- Interior Designer: David Futato
- Cover Designer: Karen Montgomery
- Illustrator: Rebecca Demarest
- February 2020: First Edition

Revision History for the Early Release

- 2019-12-11: First Release
- 2020-01-27: Second Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781492082798* for release details.

978-1-492-08272-9

[LSI]

Foreword

I have always been endlessly fascinated with the details of how Google does things. I have grilled my Googler friends for information about the way things really work inside of the company. How do they manage such a massive monolithic code repository without falling over? How do tens of thousands of engineers successfully collaborate on thousands of projects? How do they maintain the quality of their systems?

Working with former Googlers has only increased my curiosity. If you've ever worked with a former Google engineer (or "Xoogler" as they're sometimes called), you've no doubt heard the phrase "at Google we…" Coming out of Google into other companies seems to be a shocking experience, at least from the engineering side of things. As far as this outsider can tell, the systems and processes for writing code at Google must be among the best in the world, given both the scale of the company and how often people sing their praises.

In *Software Engineering at Google*, a set of Googlers (and some Xooglers) gives us a lengthy blueprint for many of the practices, tools, and even cultural elements that underlie software engineering at Google. It's easy to overfocus on the amazing tools that Google has built to support writing code, and this book provides a lot of details about those tools. But it also goes beyond simply describing the tooling to give us the philosophy and processes that the teams at Google follow. These can be adapted to fit a variety of circumstances, whether or not you have the scale and tooling. To my delight, there are several chapters that go deep on various aspects of automated testing, a topic that continues to meet with too much resistance in our industry.

The great thing about tech is that there is never only one way to do something. Instead, there are a series of tradeoffs we all must make depending on the circumstances of our team and situation. What can we cheaply take from open source? What can our team build? What makes sense to support for our scale? When I was grilling my Googler friends, I wanted to hear about the world at the extreme end of scale: resource rich, in both talent and money, with high demands on the software being built. This anecdotal information gave me ideas on some options that I might not otherwise have considered.

With this book, we have those options written down for everyone to read. Of course, Google is a unique company, and it would be foolish to assume that

the right way to run your software engineering organization is to precisely copy their formula. Applied practically, this book will give you ideas on how things could be done, and a lot of information that you can use to bolster your arguments for adopting best practices like testing, knowledge sharing, and building collaborative teams.

You may never need to build Google yourself, and you may not even want to reach for the same techniques they apply in your organization. But if you aren't familiar with the practices Google has developed, you're missing a perspective on software engineering that comes from tens of thousands of engineers working collaboratively on software over the course of more than two decades. That knowledge is far too valuable to ignore.

Camille Fournier

Preface

This book is titled "Software Engineering at Google." What precisely do we mean by software engineering? What distinguishes "software engineering" from "programming" or "computer science"? And why would Google have a unique perspective to add to the corpus of previous software engineering literature written over the past 50 years?

The terms "programming" and "software engineering" have been used interchangeably for quite some time in our industry, although each term has a different emphasis and different implications. University students tend to study computer science and get jobs writing code as "programmers."

"Software engineering," however, sounds more serious, as if it implies the application of some theoretical knowledge to build something real and precise. Mechanical engineers, civil engineers, aeronautical engineers, and those in other engineering disciplines all practice engineering. They all work in the real world and use the application of their theoretical knowledge to create something real. Software engineers also create "something real," though it is less tangible than the things other engineers create.

Unlike those more established engineering professions, current software engineering theory or practice is not nearly as rigorous. Aeronautical engineers must follow rigid guidelines and practices, because errors in their calculations can cause real damage; programming, on the whole, has traditionally not followed such rigorous practices. But, as software becomes more integrated into our lives, we must adopt and rely on more rigorous engineering methods. We hope this book helps others see a path toward more reliable software practices.

Programming Over Time

We propose that "software engineering" encompasses not just the act of writing code, but all of the tools and processes an organization uses to build and maintain that code over time. What practices can a software organization introduce that will best keep its code valuable over the long term? How can engineers make a codebase more sustainable and the software engineering discipline itself more rigorous? We don't have fundamental answers to these

questions, but we hope that Google's collective experiences over the past two decades illuminates possible paths toward finding those answers.

One key insight we share in this book is that software engineering can be thought of as "programming integrated over time." What practices can we introduce to our code *sustainable*—able to react to necessary change—over its life cycle, from conception to introduction to maintenance to deprecation?

The book emphasizes three fundamental principles that we feel software organizations should keep in mind when designing, architecting, and writing their code:

- *Time and Change*, or how code will need to adapt over the length of its life
- *Scale and Growth*, or how an organization will need to adapt as it evolves
- *Trade-Offs and Costs*, or how an organization makes decisions, based on the lessons of Time and Change and Scale and Growth

Throughout the chapters, we have tried to tie back to these themes and point out ways in which such principles affect engineering practices and allow them to be sustainable. (See Chapter 1 for a full discussion.)

Google's Perspective

Google has a unique perspective on the growth and evolution of a sustainable software ecosystem, stemming from our scale and longevity. We hope that the lessons we have learned will be useful as your organization evolves and embraces more sustainable practices.

We've divided the topics in this book into three main aspects of Google's software engineering landscape:

- Culture
- Processes
- Tools

Google's culture is unique, but the lessons we have learned in developing our engineering culture are widely applicable. Our chapters on Culture emphasize the collective nature of a software development enterprise, that the

development of software is a team effort, and that proper cultural principles are essential for an organization to grow and remain healthy.

The techniques outlined in our Processes chapters are familiar to most software engineers, but Google's large size and long-lived codebase provides a more complete stress test for developing best practices. Within those chapters, we have tried to emphasize what we have found to work over time and at scale as well as identify areas where we don't yet have satisfying answers.

Finally, our Tools chapters illustrate how we leverage our investments in tooling infrastructure to provide benefits to our codebase as it both grows and ages. In some cases, these tools are specific to Google, though we point out open source or third-party alternatives where applicable. We expect that these basic insights apply to most engineering organizations.

The culture, processes, and tools outlined in this book describe the lessons that a typical software engineer hopefully learns on the job. Google certainly doesn't have a monopoly on good advice, and our experiences presented here are not intended to dictate what your organization should do. This book is our perspective, but we hope you will find useful, either by adopting these lessons directly or by using them as a starting point when considering your own practices, specialized for your own problem domain.

Neither is this book intended to be a sermon. Google itself still imperfectly applies many of the concepts within these pages. The lessons that we have learned, we learned through our failures: we still make mistakes, implement imperfect solutions, and need to iterate toward improvement. Yet the sheer size of Google's engineering organization ensures that there is a diversity of solutions for every problem. We hope that this book contains the best of that group.

What This Book Isn't

This book is not meant to cover software design, a discipline that requires its own book (and for which much content already exists). Although there is some code in this book for illustrative purposes, the principles are language neutral, and there is little actual "programming" advice within these chapters. As a result, this text doesn't cover many important issues in software development: project management, API design, security hardening, internationalization, user interface frameworks, or other language-specific concerns. Their omission in this book does not imply their lack of

importance. Instead, we choose not to cover them here knowing that we could not provide the treatment they deserve. We have tried to make the discussions in this book more about engineering and less about programming.

Parting Remarks

This text has been a labor of love on behalf of all who have contributed, and we hope that you receive it as it is given: as a window into how a large software engineering organization builds its products. We also hope that it is one of many voices that helps move our industry to adopt more forward-thinking and sustainable practices. Most important, we further hope that you enjoy reading it, and can adopt some of its lessons to your own concerns.

Tom Manshreck

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

> Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`

> Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`

> Shows commands or other text that should be typed literally by the user.

`Constant width italic`

> Shows text that should be replaced with user-supplied values or by values determined by context.

TIP

This element signifies a tip or suggestion.

NOTE

This element signifies a general note.

WARNING

This element indicates a warning or caution.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at *https://github.com/oreillymedia/title_title*.

If you have a technical question or a problem using the code examples, please send email to *bookquestions@oreilly.com*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but generally do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Software Engineering at Google* by Titus Winters, Tom Manshreck, and Hyrum Wright (O'Reilly). Copyright 2020 Google, LLC., 978-1-492-08279-8."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

O'Reilly Online Learning

NOTE

For more than 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, conferences, and our online learning

platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, please visit *http://oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

- O'Reilly Media, Inc.
- 1005 Gravenstein Highway North
- Sebastopol, CA 95472
- 800-998-9938 (in the United States or Canada)
- 707-829-0515 (international or local)
- 707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at https://oreil.ly/software-engineering-at-google.

Email *bookquestions@oreilly.com* to comment or ask technical questions about this book.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

A book like this would not be possible without the work of countless others. All of the knowledge within this book has come to all of us through the experience of so many others at Google throughout our careers. We are the messengers; others came before us, at Google and elsewhere, and taught us

what we now present to you. We cannot list all of you here, but we do wish to acknowledge you.

We'd also like to thank Melody Meckfessel for supporting this project in its infancy as well as Daniel Jasper and Danny Berlin for supporting it through its completion.

This book would not have been possible without the massive collaborative effort of our curators, authors, and editors. Although the authors and editors are specifically acknowledged in each chapter or callout, we'd like to take time to recognize those who contributed to each chapter by providing thoughtful input, discussion, and review.

- **What Is Software Engineering?:** Sanjay Ghemawat, Andrew Hyatt
- **Working Well on Teams:** Sibley Bacon, Joshua Morton
- **Knowledge Sharing:** Dimitri Glazkov, Kyle Lemons, John Reese, David Symonds, Andrew Trenk, James Tucker, David Kohlbrenner, Rodrigo Damazio Bovendorp
- **Engineering for Equity:** Kamau Bobb, Bruce Lee
- **How to Lead a Team:** Jon Wiley, Laurent Le Brun
- **Leading at Scale:** Bryan O'Sullivan, Bharat Mediratta, Daniel Jasper, Shaindel Schwartz
- **Measuring Engineering Productivity:** Andrea Knight, Collin Green, Caitlin Sadowski, Max-Kanat Alexander, Yilei Yang
- **Style Guides and Rules:** Max Kanat-Alexander, Titus Winters, Matt Austern, James Dennett
- **Code Review:** Max Kanat-Alexander, Brian Ledger, Mark Barolak
- **Documentation:** Jonas Wagner, Smit Hinsu, Geoffrey Romer
- **Testing Overview:** Erik Kufler, Andrew Trenk, Dillon Bly, Joseph Graves, Neal Norwitz, Jay Corbett, Mark Striebeck, Brad Green, Miško Hevery, Antoine Picard, Sarah Storck
- **Unit Testing:** Andrew Trenk, Adam Bender, Dillon Bly, Joseph Graves, Titus Winters, Hyrum Wright, Augiue Fackler
- **Testing Doubles:** Joseph Graves, Gennadiy Civil
- **Larger Testing:** Adam Bender, Andrew Trenk, Erik Kuefler, Matthew Beaumont-Gay
- **Deprecation:** Greg Miller, Andy Shulman
- **Version Control and Branch Management:** Rachel Potvin, Victoria Clarke
- **Code Search:** Jenny Wang
- **Build Systems and Build Philosophy:** Hyrum Wright, Titus Winters, Adam Bender, Jeff Cox, Jacques Pienaar

- **Critique: Google's Code Review Tool:** Mikołaj Dądela, Hermann Loose, Eva May, Alice Kober-Sotzek, Edwin Kempin, Patrick Hiesel
- **Static Analysis:** Jeffrey van Gogh, Ciera Jaspan, Emma Söderberg, Edward Aftandilian, Collin Winter, Eric Haugh
- **Dependency Management:** Russ Cox, Nicholas Dunn
- **Large-Scale Changes:** Matthew Fowles Kulukundis, Adam Zarek
- **Continuous Integration:** Jeff Listfield, John Penix, Kaushik Sridharan
- **Continuous Delivery:** Dave Owens, Sheri Shipe, Bobbi Jones, Matt Duftler, Brian Szuter
- **Compute Services:** Tim Hockin, Collin Winter, Jarek Kuśmierek

Additionally, we'd like to thank Betsy Beyer for sharing her insight and experience in having published the original *Site Reliability Engineering* book, which made our experience much smoother. Christopher Guzikowski and Alicia Young at O'Reilly did an awesome job launching and guiding this project to publication.

The curators would also like to personally thank the following people:

Tom Manshreck: To my mom and dad for making me believe in myself—and working with me at the kitchen table to do my homework.

Titus Winters: To dad, for my path. To mom, for my voice. To Victoria, for my heart. To Raf, for having my back. Also, to Mr. Snyder, Ranwa, Z, Mike, Zach, Tom (and all the Paynes), mec, Toby, cgd, and Melody for lessons, mentorship, and trust.

Hyrum Wright: To mom and dad for their encouragement. To Brian and the denizens of Bakerland, for my first foray into software. To Dewayne, for continuing that journey. To Hannah, Jonathan, Charlotte, Spencer, and Ben for their love and interest. To Heather for being there through it all.

Part I. Thesis

Chapter 1. What Is Software Engineering?

Written by Titus Winters

Edited by Tom Manshreck

Nothing is built on stone; All is built on sand, but we must build as if the sand were stone.

Jorge Luis Borges

We see three critical differences between programming and software engineering: time, scale, and the trade-offs at play. On a software engineering project, engineers need to be more concerned with the passage of time and the eventual need for change. In a software engineering organization, we need to be more concerned about scale and efficiency, both for the software we produce as well as for the organization that is producing it. Finally, as software engineers, we are asked to make more complex decisions with higher-stakes outcomes, often based on imprecise estimates of time and growth.

Within Google we sometimes say, "Software engineering is programming integrated over time." Programming is certainly a significant part of software engineering: after all, programming is how you generate new software in the first place. If you accept this distinction, it also becomes clear that we might need to delineate between programming tasks (development) and software engineering tasks (development, modification, maintenance). The addition of time adds an important new dimension to programming. Cubes aren't squares, distance isn't velocity. Software engineering isn't programming.

One way to see the impact of time on a program is to think about the question, "What is the expected life span[1] of your code?" Reasonable answers to this question vary by roughly a factor of 100,000. It is just as reasonable to think of code that needs to last for a few minutes as it is to imagine code that will live for decades. Generally, code on the short end of that spectrum is unaffected by time. It is unlikely that you need to adapt to a new version of your underlying libraries, operating system (OS), hardware, or language version for a program whose utility spans only an hour. These short-lived systems are effectively "just" a programming problem, in the same way that a cube compressed far enough in one dimension is a square.

As we expand that time to allow for longer life spans, change becomes more important. Over a span of a decade or more, most program dependencies, whether implicit or explicit, will likely change. This recognition is at the root of our distinction between software engineering and programming.

This distinction is at the core of what we call *sustainability* for software. Your project is *sustainable* if, for the expected life span of your software, you are capable of reacting to whatever valuable change comes along, for either technical or business reasons. Importantly, we are looking only for capability—you might choose not to perform a given upgrade, either for lack of value or other priorities.[2] When you are fundamentally incapable of reacting to a change in underlying technology or product direction, you're placing a high-risk bet on the hope that such a change never becomes critical. For short-term projects, that might be a safe bet. Over multiple decades, it probably isn't.[3]

Another way to look at software engineering is to consider scale. How many people are involved? What part do they play in the development and maintenance over time? A programming task is often an act of individual creation, but a software engineering task is a team effort. An early attempt to define software engineering produced a good definition for this viewpoint: "The multiperson development of multiversion programs."[4] This suggests the difference between software engineering and programming is one of both time and people. Team collaboration presents new problems, but also provides more potential to produce valuable systems than any single programmer could.

Team organization, project composition, and the policies and practices of a software project all dominate this aspect of software engineering complexity. These problems are inherent to scale: as the organization grows and its projects expand, does it become more efficient at producing software? Does our development workflow become more efficient as we grow, or do our version control policies and testing strategies cost us proportionally more? Scale issues around communication and human scaling have been discussed since the early days of software engineering, going all the way back to the *Mythical Man Month*[5]. Such scale issues are often matters of policy, and are fundamental to the question of software sustainability: how much will it cost to do the things that we need to do repeatedly?

We can also say that software engineering is different from programming in terms of the complexity of decisions that need to be made and their stakes. In software engineering, we are regularly forced to evaluate the trade-offs

between several paths forward, sometimes with high stakes and often with imperfect value metrics. The job of a software engineer, or a software engineering leader, is to aim for *sustainability* and management of the scaling costs for the organization, the product, and the development workflow. With those inputs in mind, evaluate your trade-offs and make rational decisions. We might sometimes defer maintenance changes, or even embrace policies that don't scale well, with the knowledge that we'll need to revisit those decisions. Those choices should be explicit and clear about the deferred costs.

Rarely is there a one-size-fits-all solution in software engineering, and the same applies to this book. Given a factor of 100,000 for reasonable answers on "How long will this software live," a range of perhaps a factor of 10,000 for "How many engineers are in your organization," and who-knows-how-much for "How many compute resources are available for your project," Google's experience will probably not match yours. In this book, we aim to present what we've found that works for us in the construction and maintenance of software that we expect to last for decades, with tens of thousands of engineers, and world-spanning compute resources. Most of the practices that we find are necessary at that scale will also work well for smaller endeavors: consider this a report on one engineering ecosystem that we think could be good as you scale up. In a few places, super-large scale comes with its own costs, and we'd be happier to not be paying extra overhead. We call those out as a warning. Hopefully if your organization grows large enough to be worried about those costs you can find a better answer.

Before we get to specifics about teamwork, culture, policies, and tools, let's first elaborate on these primary themes of time, scale, and trade-offs.

Time and Change

When a novice is learning to program, the life span of the resulting code is usually measured in hours or days. Programming assignments and exercises tend to be write-once, with little to no refactoring and certainly no long-term maintenance. These programs are often not rebuilt or executed ever again after their initial production. This isn't surprising in a pedagogical setting. Perhaps in secondary or post-secondary education we may find a team project course or hands-on thesis. If so, such projects are likely the only time student code is likely to live longer than a month or so. Those developers might need to refactor some code, perhaps as a response to changing

requirements, but it is unlikely they are being asked to deal with broader changes to their environment.

We also find developers of short-lived code in common industry settings. Mobile apps often have a fairly short life span,[6] and for better or worse, full rewrites are relatively common. Engineers at an early-stage startup might rightly choose to focus on immediate goals over long-term investments: the company might not live long enough to reap the benefits of an infrastructure investment that pays off slowly. A serial startup developer could very reasonably have 10 years of development experience, and little or no experience maintaining any piece of software expected to exist for longer than a year or two.

On the other end of the spectrum, some successful projects have an effectively unbounded life span: we can't reasonably predict an endpoint for Google Search, the Linux kernel, or the Apache HTTP Server project. For most Google projects, we must assume that they will live indefinitely: we cannot predict when we won't need to upgrade our dependencies, language versions, and so on. As their lifetimes grow, these long-lived projects *eventually* have a different feel to them than programming assignments or startup development.

Consider Figure 1-1, which demonstrates two software projects on opposite ends of this "expected life span" spectrum. For a programmer working on a task with an expected life span of hours, what types of maintenance are reasonable to expect? That is, if a new version of your OS comes out while you're working on a Python script that will be executed one time, should you drop what you're doing and upgrade? Of course not: the upgrade is not critical. But on the opposite end of the spectrum, Google Search being stuck on a version of our OS from the 1990s would be a clear problem.

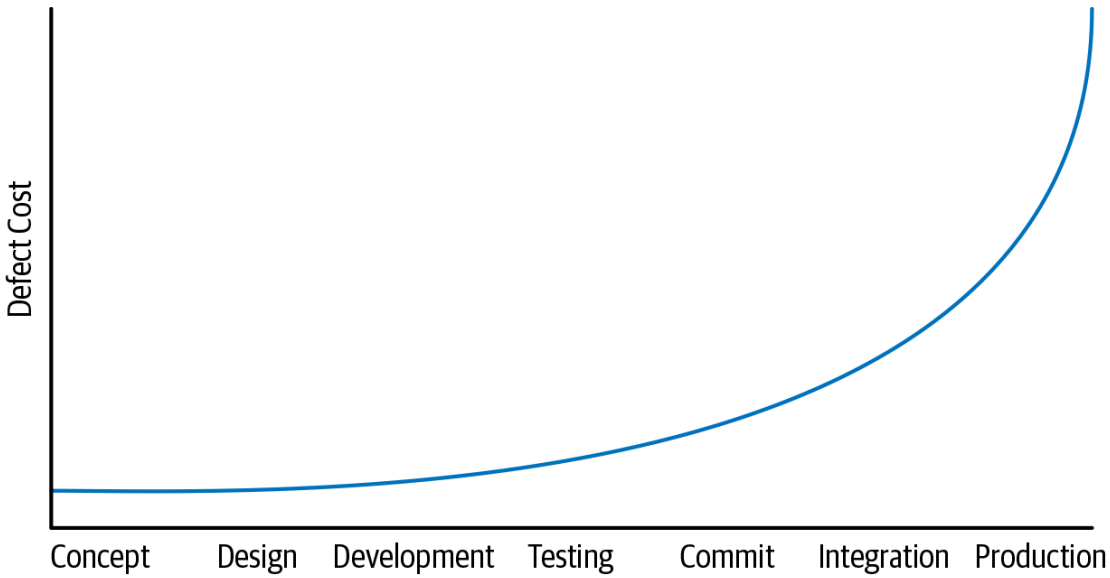

Figure 1-1. Life span and the importance of upgrades

These two points on the expected life span spectrum suggest that there's a transition somewhere. Somewhere along the line between a one-off program and a project that lasts for decades, a transition happens: a project must begin to react to changing externalities.[7] For any project that didn't plan for upgrades from the start, that transition is likely very painful for three reasons, each of which compounds the others:

- You're performing a task that hasn't yet been done for this project; more hidden assumptions have been baked-in.
- The engineers trying to do the upgrade are less likely to have experience in this sort of task.
- The size of the upgrade is often larger than usual, doing several years' worth of upgrades at once instead of a more incremental upgrade.

And thus after actually going through such an upgrade once (or giving up part way through), it's pretty reasonable to overestimate the cost of doing a subsequent upgrade and decide "Never again." Companies that come to this conclusion end up committing to just throwing things out and rewriting their code, or deciding to never upgrade again. Rather than take the natural approach by avoiding a painful task, sometimes the more responsible answer is to invest in making it less painful. It all depends on the cost of your upgrade, the value it provides, and the expected life span of the project in question.

Getting through not only that first big upgrade, but getting to the point at which you can reliably stay current going forward is the essence of long-term sustainability for your project. Sustainability requires planning and managing

the impact of required change. For many projects at Google, we believe we have achieved this sort of sustainability, largely through trial and error.

So, concretely, how does short-term programming differ from producing code with a much longer expected life span? Over time, we need to be much more aware of the difference between "happens to work" and "it is maintainable." There is no perfect solution for identifying these issues. That is unfortunate because keeping software maintainable for the long-term is a constant battle.

Hyrum's Law

If you are maintaining a project that is used by other engineers, the most important lesson about "it works" versus "it is maintainable" is what we've come to call *Hyrum's Law*:

With a sufficient number of users of an API, it does not matter what you promise in the contract: all observable behaviors of your system will be depended on by somebody.

In our experience, this axiom is a dominant factor in any discussion of changing software over time. It is conceptually akin to entropy: discussions of change and maintenance over time must be aware of Hyrum's Law[8] just as discussions of efficiency or thermodynamics must be mindful of entropy. Just because entropy never decreases doesn't mean we shouldn't try to be efficient. Just because Hyrum's Law will apply when maintaining software doesn't mean we can't plan for it or try to better understand it. We can mitigate it, but we know that it can never be eradicated.

Hyrum's Law represents the practical knowledge that—even with the best of intentions, the best engineers, and solid practices for code review—we cannot assume perfect adherence to published contracts or best practices. As an API owner, you will gain *some* flexibility and freedom by being clear about interface promises, but in practice the complexity and difficulty of a given change also depends on how useful a user finds some observable behavior of your API. If users cannot depend on such things, your API will be easy to change. Given enough time and enough users, even the most innocuous change *will* break something;[9] your analysis of the value of that change must incorporate the difficulty in investigating, identifying, and resolving those breakages.

Example: Hash Ordering

Consider the example of hash iteration ordering. If we insert five elements into a hash-based set, in what order do we get them out?

```
>>> for i in {"apple", "banana", "carrot", "durian",

              "eggplant"}: print(i)

...

durian

carrot

apple

eggplant

banana
```

Most programmers know that hash tables are non-obviously ordered. Few know the specifics of whether the particular hash table they are using is *intending* to provide that particular ordering forever. This might seem unremarkable, but over the past decade or two, the computing industry's experience using such types has evolved:

- *Hash flooding*[10] attacks provide an increased incentive for nondeterministic hash iteration.
- Potential efficiency gains from research into improved hash algorithms or hash containers require changes to hash iteration order.
- Per Hyrum's Law, programmers will write programs that depend on the order in which a hashtable is traversed, if they have the ability to do so.

As a result, if you ask any expert, "Can I assume a particular output sequence for my hash container?" that expert will presumably say, "No." By and large that is correct, but perhaps simplistic. A more nuanced answer is "If your code is short-lived, with no changes to your hardware, language runtime, or choice of data structure, such an assumption is fine. If you don't know how long your code will live, or you cannot promise that nothing you depend upon will ever change, such an assumption is incorrect." Moreover, even if your own implementation does not depend on hash container order, it might

be used by other code that implicitly creates such a dependency. For example, if your library serializes values into a Remote Procedure Call (RPC) response, the RPC caller might wind up depending on the order of those values.

This is a very basic example of the difference between "it works" and "it is correct." For a short-lived program, depending on the iteration order of your containers will not cause any technical problems. For a software engineering project, on the other hand, such reliance on a defined order is a risk—given enough time, something will make it valuable to change that iteration order. That value can manifest in a number of ways, be it efficiency, security, or merely future-proofing the data structure to allow for future changes. When that value becomes clear, you will need to weigh the trade-offs between that value and the pain of breaking your developers or customers.

Some languages specifically randomize hash ordering between library versions or even between execution of the same program, in an attempt to prevent dependencies. But even this still allows for some Hyrum's Law surprises: there is code that uses hash iteration ordering as an inefficient random number generator. Removing such randomness now would break those users. Just as entropy increases in every thermodynamic system, Hyrum's Law applies to every observable behavior.

Thinking over the differences between code written with a "works now" and a "works indefinitely" mentality, we can extract some clear relationships. Looking at code as an artifact with a (highly) variable lifetime requirement, we can begin to categorize programming styles: code that depends on brittle and unpublished features of its dependencies is likely to be described as "hacky" or "clever," whereas code that follows best practices and has planned for the future is more likely to be described as "clean" and "maintainable." Both have their purposes, but which one you select depends crucially on the expected life span of the code in question. We've taken to saying, "It's *programming* if "clever" is a compliment, but it's *software engineering* if "clever" is an accusation."

Why Not Just Aim for "Nothing Changes"?

Implicit in all of this discussion of time and the need to react to change is the assumption that change might be necessary. Is it?

As with effectively everything else in this book, it depends. We'll readily commit to "For most projects, over a long enough time period, everything underneath them might need to be changed." If you have a project written in

pure C with no external dependencies (or only external dependencies that promise great long-term stability, like POSIX), you might well be able to avoid any form of refactoring or difficult upgrade. C does a great job of providing stability—in many respects, that is its primary purpose.

Most projects have far more exposure to shifting underlying technology. Most programming languages and runtimes change much more than C does. Even libraries implemented in pure C might change to support new features, which can affect downstream users. Security problems are disclosed in all manner of technology, from processors to networking libraries to application code. *Every* piece of technology upon which your project depends has some (hopefully small) risk of containing critical bugs and security vulnerabilities that might come to light only after you've started relying on it. If you are incapable of deploying a patch for Heartbleed or mitigating speculative execution problems like Meltdown and Spectre because you've assumed (or promised) that nothing will ever change, that is a significant gamble.

Efficiency improvements further complicate the picture. We want to outfit our datacenters with cost-effective computing equipment, especially enhancing CPU efficiency. However, algorithms and data structures from early-day Google are simply less efficient on modern equipment: a linked-list or a binary search tree will still work fine, but the ever-widening gap between CPU cycles versus memory latency impacts what "efficient" code looks like. Over time, the value in upgrading to newer hardware can be diminished without accompanying design changes to the software. Backward compatibility ensures that older systems still function, but that is no guarantee that old optimizations are still helpful. Being unwilling or unable to take advantage of such opportunities risks incurring large costs. Efficiency concerns like this are particularly subtle: the original design might have been perfectly logical and following reasonable best practices. It's only after an evolution of backward-compatible changes that a new more-efficient option becomes important. No mistakes were made, but the passage of time still made change valuable.

Concerns like those just mentioned are why there are large risks for long-term projects that haven't invested in sustainability. We must be capable of responding to these sorts of issues and taking advantage of these opportunities, regardless of whether they directly affect us or manifest in only the transitive closure of technology we build upon. Change is not inherently good. We shouldn't change just for the sake of change. But we do need to be capable of change. If we allow for that eventual necessity, we should also consider whether to invest in making that capability cheap. As every system

administrator knows: it's one thing to know in theory that you can recover from tape, and another to know in practice exactly how to do it and how much it will cost when it becomes necessary. Practice and expertise are great drivers of efficiency and reliability.

Scale and Efficiency

As noted in the *Site Reliability Engineering* (SRE) book,[11] Google's production system as a whole is among the most complex machines created by humankind. The complexity involved in building such a machine and keeping it running smoothly has required countless hours of thought, discussion, and redesign from experts across our organization and around the globe. We have already written a book about the complexity of keeping that machine running at that scale.

Much of *this* book focuses on the complexity of scale of the organization that produces such a machine, and the processes that we use to keep that machine running over time. Consider again the concept of codebase sustainability: "Your organization's codebase is *sustainable* when you are *able* to change all of the things that you ought to change, safely, and can do so for the life of your codebase." Hidden in the discussion of capability is also one of costs: if changing something comes at inordinate cost, it will likely be deferred. If costs grow superlinearly over time, the operation clearly is not scalable.[12] Eventually, time will take hold and something unexpected will arise that you absolutely must change. When your project doubles in scope and you need to perform that task again, will it be twice as labor intensive? Will you even have the human resources required to address the issue next time?

Human costs are not the only finite resource that needs to scale. Just as software itself needs to scale well with traditional resources such as compute, memory, storage and bandwidth, the development of that software also needs to scale, both in terms of human time involvement and the compute resources that power your development workflow. If the compute cost for your test cluster grows superlinearly, consuming more compute resources per person each quarter, you're on an unsustainable path and need to make changes soon.

Finally, the most precious asset of a software organization—the codebase itself—also needs to scale. If your build system or version control system scales superlinearly over time, perhaps as a result of growth and increasing

changelog history, a point might come at which you simply cannot proceed. Many questions, such as "How long does it take to do a full build?", "How long does it take to pull a fresh copy of the repo?", or "How much will it cost to upgrade to a new language version?" aren't actively monitored and change at a slow pace. They can easily become like the metaphorical boiled frog; it is far too easy for problems to worsen slowly and never manifest as a singular moment of crisis. Only with an organization-wide awareness and commitment to scaling are you likely to keep on top of these issues.

Everything your organization relies upon to produce and maintain code should be scalable in terms of overall cost and resource consumption. In particular, everything your organization must do repeatedly should be scalable in terms of human effort. Many common policies don't seem to be scalable in this sense.

Policies that Don't Scale

With a little practice, it becomes easier to spot policies with bad scaling properties. Most commonly these can be identified by considering the work imposed on a single engineer and imagining the organization scaling up by 10 or 100 times. When we are 10 times larger, did we add 10 times more work with which our sample engineer needs to keep up? Does the amount of work our engineer must perform grow as a function of the size of the organization? Does the work scale up with the size of the codebase? If either of these are true, do we have any mechanisms in place to automate or optimize that work? If not, we have scaling problems.

Consider a traditional approach to deprecation. We discuss deprecation much more in Chapter 15, but the common approach to deprecation serves as a great example of scaling problems. A new Widget has been developed. The decision is made that everyone should use the new one and stop using the old one. To motivate this, project leads say, "We'll delete the old Widget on August 15th: make sure you've converted to the new Widget."

This type of approach might work in a small software setting, but quickly fails as both the depth and breadth of the dependency graph increases. Teams depend on an ever-increasing number of Widgets, and a single build break can affect a growing percentage of the company. Solving these problems in a scalable way means changing the way we do deprecation: instead of pushing migration work to customers, teams can internalize it themselves, with all the economies of scale that provides.

In 2012, we tried to put a stop to this with rules mitigating churn: infrastructure teams must do the work to move their internal users to new versions themselves or do the update in place, in backward-compatible fashion. This policy, which we've called the "Churn Rule," scales better: dependent projects are no longer spending progressively greater effort just to keep up. We've also learned that having a dedicated group of experts execute the change scales better than asking for more maintenance effort from every user: experts spend some time learning the whole problem in depth and then apply that expertise to every subproblem. Forcing users to respond to churn means that every affected team does a worse job ramping up, solves their immediate problem, and then throws away that now-useless knowledge. Expertise scales better.

The traditional use of development branches is another example of policy that has built-in scaling problems. An organization might identify that merging large features into trunk has destabilized the product and conclude, "We need tighter controls on when things merge. We should merge less frequently." This leads quickly to every team or every feature having separate dev branches. Whenever any branch is decided to be "complete," it is tested and merged into trunk, triggering some potentially expensive work for other engineers still working on their dev branch, in the form of resyncing and testing. Such branch management can be made to work for a small organization juggling 5 to 10 such branches. As the size of an organization (and the number of branches) increases, it quickly becomes apparent that we're paying an ever-increasing amount of overhead to do the same task. We'll need a different approach as we scale up, and we discuss that in Chapter 16.

Policies That Scale Well

What sorts of policies result in better costs as the organization grows? Or, better still, what sorts of policies can we put in place that provide superlinear value as the organization grows?

One of our favorite internal policies is a great enabler of infrastructure teams, protecting their ability to make infrastructure changes safely. "If a product experiences outages or other problems as a result of infrastructure changes, but the issue wasn't surfaced by tests in our Continuous Integration (CI) system, it is not the fault of the infrastructure change." More colloquially, this is phrased as "If you liked it, you should have put a CI test on it," which we call "The Beyoncé Rule.[13] From a scaling perspective, the Beyoncé Rule implies that complicated one-off bespoke tests that aren't triggered by our

common CI system do not count. Without this, an engineer on an infrastructure team could conceivably need to track down every team with any affected code and ask them how to run their tests. We could do that when there were a hundred engineers. We definitely cannot afford to do that anymore.

We've found that expertise and shared communication forums offer great value as an organization scales. As engineers discuss and answer questions in shared forums, knowledge tends to spread. New experts grow. If you have 100 engineers writing Java, a single friendly and helpful Java expert willing to answer questions will soon produce 100 engineers writing better Java code and several Java experts. Knowledge is viral, experts are carriers, and there's a lot to be said for the value of clearing away the common stumbling blocks for your engineers. We cover this in greater detail in Chapter 3.

Example: Compiler Upgrade

Consider the daunting task of upgrading your compiler. Theoretically, a compiler upgrade should be cheap given how much effort languages take to be backward compatible, but how cheap of an operation is it in practice? If you've never done such an upgrade before, how would you evaluate whether your codebase is compatible with that change?

In our experience, language and compiler upgrades are subtle and difficult tasks even when they are broadly expected to be backward compatible. A compiler upgrade will almost always result in minor changes to behavior: fixing miscompilations, tweaking optimizations, or potentially changing the results of anything that was previously undefined. How would you evaluate the correctness of your entire codebase against all of these potential outcomes?

The most storied compiler upgrade in Google's history took place all the way back in 2006. At that point, we had been operating for a few years and had several thousand engineers on staff. We hadn't updated compilers in about five years. Most of our engineers had no experience with a compiler change. Most of our code had been exposed to only a single compiler version. It was a difficult and painful task for a team of (mostly) volunteers, which eventually became a matter of finding shortcuts and simplifications in order to work around upstream compiler and language changes that we didn't know how to adopt.[14] In the end, the 2006 compiler upgrade was extremely painful. Many Hyrum's Law problems, big and small, had crept into the codebase and served to deepen our dependency on a particular compiler version. Breaking

those implicit dependencies was painful. The engineers in question were taking a risk: we didn't have the Beyoncé Rule yet, nor did we have a pervasive CI system, so it was difficult to know the impact of the change ahead of time or be sure they wouldn't be blamed for regressions.

This story isn't at all unusual. Engineers at many companies can tell a similar story about a painful upgrade. What is unusual is that we recognized after the fact that the task had been painful and began focusing on technology and organizational changes to overcome the scaling problems and turn scale to our advantage: automation (so that a single human can do more), consolidation/consistency (so that low-level changes have a limited problem scope), and expertise (so that a few humans can do more).

The more frequently you change your infrastructure, the easier it becomes to do so. We have found that most of the time, when code is updated as part of something like a compiler upgrade, it becomes less brittle and easier to upgrade in the future. In an ecosystem in which most code has gone through several upgrades, it stops depending on the nuances of the underlying implementation; instead, the actual abstraction guaranteed by the language or OS. Regardless of what exactly you are upgrading, expect the first upgrade for a codebase to be significantly more expensive than later upgrades, even controlling for other factors.

Through this and other experiences, we've discovered many factors that affect the flexibility of a codebase:

Expertise

> We know how to do this; for some languages we've now done hundreds of compiler upgrades, across many platforms.

Stability

> There is less change between releases because we adopt releases more regularly; for some languages, we're now deploying compiler upgrades every week or two.

Conformity

> There is less code that hasn't been through an upgrade already, again because we are upgrading regularly.

Familiarity

> Because we do this regularly enough, we can spot redundancies in the process of performing an upgrade and attempt to automate. This overlaps significantly with SRE views on toil.[15]

Policy

> We have processes and policies like the Beyoncé Rule. The net effect of these processes is that upgrades remain feasible because infrastructure teams do not need to worry about every unknown usage, only the ones that are visible in our CI systems.

The underlying lesson is not about the frequency or difficulty of compiler upgrades, but that as soon as we became aware that compiler upgrade tasks were necessary, we found ways to make sure to perform those tasks with a constant number of engineers, even as the codebase grew.[16] If we had instead decided that the task was too expensive and should be avoided in the future, we might still be using a decade-old compiler version. We would be paying perhaps 25% extra for computational resources as a result of missed optimization opportunities. Our central infrastructure could be vulnerable to significant security risks given that a 2006-era compiler is certainly not helping to mitigate speculative execution vulnerabilities. Stagnation is an option, but often not a wise one.

Shifting Left

One of the broad truths we've seen to be true is the idea that finding problems earlier in the developer workflow usually reduces costs. Consider a timeline of the developer workflow for a feature that progresses from left to right, starting from conception and design, progressing through implementation, review, testing, commit, canary, and eventual production deployment. Shifting problem detection to the "left," earlier on this timeline, makes it cheaper to fix than waiting longer, as shown in Figure 1-2.

This term seems to have originated from arguments that security mustn't be deferred until the end of the development process, with requisite calls to "shift left on security." The argument in this case is relatively simple: if a security problem is discovered only after your product has gone to production, you have a very expensive problem. If it were caught before deploying to production, it may still take a lot of work to identify and remedy the problem, but it's cheaper. If you can catch it before the original developer commits the flaw to version control, it's even cheaper: they already have an understanding of the feature; revising according to new security constraints is cheaper than committing and forcing someone else to triage it and fix it.

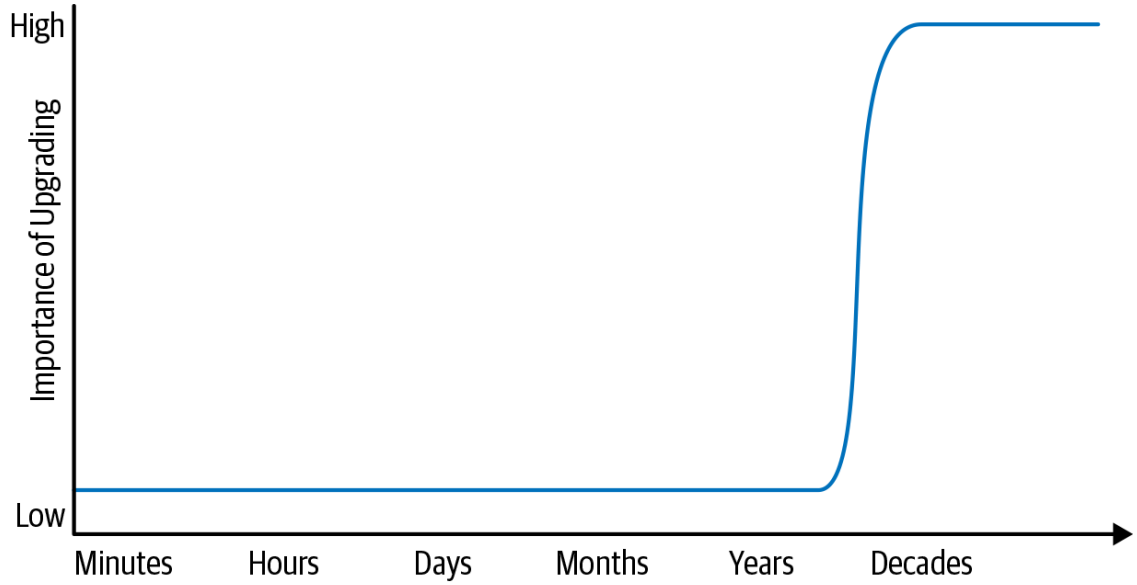

Figure 1-2. Timeline of the developer workflow

The same basic pattern emerges many times in this book. Bugs that are caught by static analysis and code review before they are committed are much cheaper than bugs that make it to production. Providing tools and practices that highlight quality, reliability, security early in the development process is a common goal for many of our infrastructure teams. No single process or tool needs to be perfect, we can assume a defense-in-depth approach, hopefully catching as many defects on the left side of the graph as possible.

Tradeoffs and Costs

If we understand how to program, understand the lifetime of the software we're maintaining, and understand how to maintain it as we scale up with more engineers producing and maintaining new features, all that is left is to make good decisions. This seems obvious: in software engineering, as in life, good choices lead to good outcomes. However, the ramifications of this observation are easily overlooked. Within Google, there is a strong distaste for "because I said so." It is important for there to be a decider for any topic, and clear escalation paths when decisions seem to be wrong, but the goal is consensus, not unanimity. It's fine and expected to see some instances of "I don't agree with your metrics/valuation, but I see how you can come to that conclusion." Inherent in all of this is the idea that there needs to be a reason for everything; "just because," "because I said so," or "because everyone else does it this way" are places where bad decisions lurk. Whenever it is efficient to do so, we should be able to explain our work when deciding between the general costs for two engineering options.

What do we mean by cost? We are not only talking about dollars here. "Cost" roughly translates to effort, and can involve any or all of these factors:

- Financial costs (e.g., money)
- Resource costs (e.g., CPU time)
- Personnel costs (e.g., engineering effort)
- Transaction costs (e.g., what does it cost to take action?)
- Opportunity costs (e.g., what does it cost to "not" take action?)
- Societal costs (e.g., what impact will this choice have on society at large?)

Historically, it's been particularly easy to ignore the question of societal costs. However, Google and other large tech companies can now credibly deploy products with billions of users. In many cases these products are a clear net benefit, but when we're operating at such a scale even small discrepancies in usability, accessibility, fairness, or potential for abuse are magnified, often to the detriment of groups that are already marginalized. Software pervades so many aspects of society and culture; thus, it is wise for us to be aware of both the good and the bad that we enable when making product and technical decisions. We discuss this much more in Chapter 4.

In addition to the aforementioned costs (or our estimate of them), there are biases: status quo bias, loss aversion, and others. When we evaluate cost, we need to keep all of the previously listed costs in mind: the health of an organization isn't just whether there is money in the bank, it's also whether its members are feeling valued and productive. In highly creative and lucrative fields like software engineering, financial cost is usually not the limiting factor: personnel cost usually is. Efficiency gains from keeping engineers happy, focused, and engaged can easily dominate other factors, simply because focus and productivity are so variable and a 10 to 20% difference is easy to imagine.

Example: Markers

In many organizations, whiteboard markers are treated as precious goods. They are tightly controlled and always in short supply. Invariably half of the markers at any given whiteboard are dry and unusable. How often have you been in a meeting that was disrupted by lack of a working marker? How often have you had your train of thought derailed by a marker running out? How often have all the markers just gone missing, presumably because some other

team ran out of markers and had to abscond with yours? All for a product that costs less than a dollar.

Google tends to have unlocked closets full of office supplies, including whiteboard markers, in most work areas. With a moment's notice it is easy to grab dozens of markers in a variety of colors. Somewhere along the line we made an explicit trade-off: it is far more important to optimize for obstacle-free brainstorming than to protect against someone wandering off with a bunch of markers.

We aim to have the same level of eyes-open and explicit weighing of the cost/benefit trade-offs involved for everything we do, from office supplies and employee perks through day-to-day experience for developers to how to provision and run global-scale services. We often say "Google is a data-driven culture." In fact that's a simplification: even when there isn't *data* there might still be *evidence*, *precedent*, and *argument*. Making good engineering decisions is all about weighing all of the available inputs and making informed decisions about the trade-offs. Sometimes, those decisions are based on instinct or accepted best practice, but only after we have exhausted approaches that try to measure or estimate the true underlying costs. Chapter 7

In the end, decisions in an engineering group should come down to very few things:

- We are doing this because we must (legal requirements, customer requirements)
- We are doing this because it is the best option (as determined by some appropriate decider) we can see at the time, based on current evidence

Decisions should not be "We are doing this because I said so."[17]

Inputs to Decision Making

When we are weighing data, we find two common scenarios:

- All of the quantities involved are measurable or can at least be estimated. This usually means that we're evaluating trade-offs between CPU and network, or dollars and RAM, or considering whether to spend two weeks of engineer-time in order to save *N* CPUs across our datacenters.

- Some of the quantities are subtle, or we don't know how to measure them. Sometimes this manifests in the form "We don't know how much engineer-time this will take." Sometimes it is even more nebulous: how do you measure the engineering cost of a poorly designed API? Or the societal impact of a product choice?

There is little reason to be deficient on the first type of decision. Any software engineering organization can and should track the current cost for compute resources, engineer-hours, and other quantities you interact with regularly. Even if you don't want to publicize to your organization the exact dollar amounts, you can still produce a conversion table: this many CPUs cost the same as this much RAM or this much network bandwidth.

With an agreed-upon conversion table in hand, every engineer can do their own analysis. "If I spend two weeks changing this linked-list into a higher-performance structure, I'm going to use five gibibytes more production RAM but save two thousand CPUs. Should I do it?" Not only does this question depend upon the relative cost of RAM and CPUs, but personnel costs (two weeks of support for a software engineer) and opportunity costs (what else could that engineer produce in two weeks).

For the second type of decision, there is no easy answer. We rely on experience, leadership, and precedent to negotiate these issues. We're investing in research to help us quantify the hard-to-quantify Chapter 7. However, the best broad suggestion that we have is to be aware that not everything is measurable or predictable, and to attempt to treat such decisions with the same priority and greater care. They are often just as important, but more difficult to manage.

Example: Distributed Builds

Consider your build. According to completely unscientific Twitter polling, something like 60 to 70% of developers build locally, even with today's large complicated builds. This leads directly to nonjokes like *https://xkcd.com/303/*—how much productive time in your organization is lost waiting for a build? Compare that to the cost to run something like `distcc` for a small group. Or, how much does it cost to run a small build farm for a large group? How many weeks/months does it take for those costs to be a net win?

Back in the mid 2000s, Google relied purely on a local build system: you checked out code and you compiled it locally. We had massive local machines in some cases (you could build Maps on your desktop!), but

compilation times became longer and longer as the codebase grew. Unsurprisingly, we incurred increasing overhead in personnel costs due to lost time, as well as increased resource costs for larger and more powerful local machines, and so on. These resource costs were particularly troublesome: of course we want people to have as fast a build as possible, but most of the time a high-performance desktop development machine will sit idle. This doesn't feel like the proper way to invest those resources.

Eventually, Google developed its own distributed build system. Development of this system incurred a cost, of course: it took engineers time to develop, it took more engineer time to change everyone's habits and workflow and learn the new system, and of course it cost additional computational resources. But the overall savings were clearly worth it: builds became faster, engineer time was recouped, and hardware investment could focus on managed shared infrastructure (in actuality, a subset of our production fleet) rather than ever-more-powerful desktop machines. Chapter 18 goes into more of the details on our approach to distributed builds and the relevant trade-offs.

So, we built a new system, deployed it to production, and sped up everyone's build. Is that the happy ending to the story? Not quite: providing a distributed build system made massive improvements to engineer productivity, but as time went on, the distributed builds themselves became bloated. What was constrained in the previous case by individual engineers (because they had a vested interest in keeping their local builds as fast as possible) was unconstrained within a distributed build system. Bloated or unnecessary dependencies in the build graph became all too common. When everyone directly felt the pain of a non-optimal build and was incentivized to be vigilant, incentives were better aligned. By removing those incentives and hiding bloated dependencies in a parallel distributed build, we created a situation in which consumption could run rampant, and almost nobody was incentivized to keep an eye on build bloat. This is reminiscent of Jevons Paradox: consumption of a resource may *increase* as a response to greater efficiency in its use.

Overall, the saved costs associated with adding a distributed build system far, far outweighed the negative costs associated with its construction and maintenance. But, as we saw with increased consumption, we did not foresee all of these costs. Having blazed ahead, we found ourselves in a situation in which we needed to reconceptualize the goals and constraints of the system and our usage, identify best practices (small dependencies, machine-management of dependencies), and fund the tooling and maintenance for the new ecosystem. Even a relatively simple trade-off of the form "We'll spend

$$$s for compute resources to recoup engineer time" had unforeseen downstream effects.

Example: Deciding Between Time and Scale

Much of the time, our major themes of time and scale overlap and work in conjunction. A policy like the Beyoncé Rule scales well, and helps us maintain things over time. A change to an OS interface might require many small refactorings to adapt to, but most of those changes will scale well because they are of a similar form: the OS change doesn't manifest differently for every caller and every project.

Occasionally time and scale come into conflict, and nowhere so clearly as in the basic question: should we add a dependency or fork/reimplement it to better suit our local needs?

This question can arise at many levels of the software stack because it is regularly the case that a bespoke solution customized for your narrow problem space may outperform the general utility solution that needs to handle all possibilities. By forking or reimplementing utility code and customizing it for your narrow domain, you can add new features with greater ease, or optimize with greater certainty, regardless of whether we are talking about a microservice, an in-memory cache, a compression routine, or anything else in our software ecosystem. Perhaps more important, the control you gain from such a fork isolates you from changes in your underlying dependencies: those changes aren't dictated by another team or third-party provider. You are in control of how and when to react to the passage of time and necessity to change.

On the other hand, if every developer forks everything used in their software project instead of reusing what exists, scalability suffers alongside sustainability. Reacting to a security issue in an underlying library is no longer a matter of updating a single dependency and its users: it is now a matter of identifying every vulnerable fork of that dependency and the users of those forks.

As with most software engineering decisions, there isn't a one-size-fits-all answer to this situation. If your project life span is short, forks are less risky. If the fork in question is provably limited in scope, that helps, as well—avoid forks for interfaces that could operate across time or project time boundaries (data structures, serialization formats, networking protocols). Consistency has great value, but generality comes with its own costs, and you can often win by doing your own thing—if you do it carefully.

Revisiting Decisions, Making Mistakes

One of the unsung benefits of committing to a data-driven culture is the combined ability and necessity of admitting to mistakes. A decision will be made at some point, based on the available data—hopefully based on good data and only a few assumptions, but implicitly based on currently available data. As new data comes in, contexts change, or assumptions are dispelled, it might become clear that a decision was in error or that it made sense at the time but no longer does. This is particularly critical for a long-lived organization: time doesn't only trigger changes in technical dependencies and software systems, but in data used to drive decisions.

We believe strongly in data informing decisions, but we recognize that the data will change over time, and new data may present itself. This means, inherently, that decisions will need to be revisited from time to time over the life span of the system in question. For long-lived projects, it's often critical to have the ability to change directions after an initial decision is made. And, importantly, it means that the deciders need to have the right to admit mistakes. Contrary to some people's instincts, leaders who admit mistakes are more respected, not less.

Be evidence driven, but also realize that things that can't be measured may still have value. If you're a leader, that's what you've been asked to do: exercise judgement, assert that things are important. We'll speak more on leadership in Chapters 5 and 6.

Software Engineering versus Programming

When presented with our distinction between software engineering and programming, you might ask whether there is an inherent value judgement in play. Is programming somehow worse than software engineering? Is a project that is expected to last a decade with a team of hundreds inherently more valuable than one that is useful for only a month built by two people?

Of course not. Our point is not that software engineering is superior, merely that these represent two different problem domains with distinct constraints, values, and best practices. Rather, the value in pointing out this difference comes from recognizing that some tools are great in one domain but not in the other. You probably don't need to rely on integration tests (see Chapter 14) and Continuous Deployment (CD) practices (see XREF(Continuous Deployment)) for a project that will last only a few days.

Similarly, all of our long-term concerns about semantic versioning (SemVer) and dependency management in software engineering projects (see Chapter 21) don't really apply for short-term programming projects: use whatever is available to solve the task at hand.

We believe it is important to differentiate between the related-but-distinct terms "programming" and "software engineering." Much of that difference stems from the management of code over time, the impact of time on scale, and decision making in the face of those ideas. Programming is the immediate act of producing code. Software engineering is the set of policies, practices, and tools that are necessary to make that code useful for as long as it needs to be used and allowing collaboration across a team.

Conclusion

This book discusses all of these topics: policies for an organization and for a single programmer, how to evaluate and refine your best practices, and the tools and technologies that go into maintainable software. Google has worked hard to have a sustainable codebase and culture. We don't necessarily think that our approach is the one true way to do things, but it does provide proof by example that it can be done. We hope it will provide a useful framework for thinking about the general problem: how do you maintain your code for as long as it needs to keep working?

TL;DRs

- "Software engineering" differs from "programming" in dimensionality: programming is about producing code. Software engineering extends that to include the maintenance of that code for the useful life span of that code.
- There is a factor of at least 100,000 times between the life spans of short-lived code and long-lived code. It is silly to assume that the same best practices apply universally on both ends of that spectrum.
- Software is *sustainable* when, for the expected life span of the code, we are capable of responding to changes in dependencies, technology, or product requirements. We may choose to not change things, but we need to be capable.
- Hyrum's Law: with a sufficient number of users of an API, it does not matter what you promise in the contract: all observable behaviors of your system will be depended on by somebody.

- Every task your organization has to do repeatedly should be scalable (linear or better) in terms of human input. Policies are a wonderful tool for making process scalable.
- Process inefficiencies and other software-development tasks tend to scale up slowly. Be careful about boiled-frog problems.
- Expertise pays off particularly well when combined with economies of scale.
- "Because I said so" is a terrible reason to do things.
- Being "data driven" is a good start, but, in reality, most decisions are based on a mix of data, assumption, precedent, and argument. It's best when objective data makes up the majority of those inputs, but it can rarely be *all* of them.
- Being "data driven" over time implies the need to change directions when the data changes (or when assumptions are dispelled). Mistakes or revised plans are inevitable.

1 We don't mean "execution lifetime" we mean "maintenance lifetime"—how long will the code continue to be built, executed, and maintained? How long will this software provide value?

2 This is perhaps a reasonable hand-wavy definition of technical debt: things that "should" be done, but aren't yet—the delta between our code and what we wish it was.

3 Also consider the issue of whether we know ahead of time that a project is going to be long lived.

4 There is some question as to the original attribution of this quote; consensus seems to be that it was originally phrased by Brian Randell or Margaret Hamilton, but it might have been wholly made up by Dave Parnas. The common citation for it is "Software Engineering Techniques: Report of a conference sponsored by the NATO Science Committee, Rome, Italy, 27–31 Oct. 1969, Brussels, Scientific Affairs Division, NATO."

5 Brooks, Frederick P., Jr., 1931-. *The Mythical Man-Month: Essays on Software Engineering*.

6 "Nothing is Certain Except Death, Taxes and a Short Mobile App Lifespan," Appcelerator.

7 Your own priorities and tastes will inform where exactly that transition happens. We've found that most projects seem to be willing to upgrade within five years. Somewhere between 5 and 10 years seems like a conservative estimate for this transition in general.

8 To his credit, Hyrum tried really hard to humbly call this "The Law of Implicit Dependencies," but "Hyrum's Law" is the shorthand that most people at Google have settled on.

9 "Workflow"

10 A type of Denial of Service (DoS) attack in which an untrusted user knows the structure of a hash table and the hash function and provides data in such a way as to degrade the algorithmic performance of operations on the table.

11 Site Reliability Engineering: How Google Runs Production Systems, O'Reilly Media, April 2016, Betsy Beyer, Chris Jones, Jennifer Petoff, Niall Richard Murphy

12 Whenever we use "scalable" in an informal context in this chapter, we mean "sublinear scaling wrt. human interactions."

13 This is a reference to the popular song *Single Ladies*, which includes the refrain "If you liked it then you shoulda put a ring on it."

14 Specifically: interfaces from the C++ standard library needed to be referred to in namespace std, and an optimization change for `std::string` turned out to be a significant pessimization for our usage, thus requiring some additional workarounds.

15 Ibid. Chapter 5: Eliminating Toil.

16 In our experience, an average software engineer (SWE) produces a pretty constant number of lines of code per unit time. For a fixed SWE population, a codebase grows linearly—proportional to the count of SWE-months over time. If your tasks require effort that scales with lines of code, that's concerning.

17 This is not to say that decisions need to be made unanimously, or even with broad consensus; in the end, someone must be the decider. This is primarily a statement of how the decision-making process should flow for whoever is actually responsible for the decision.

Part II. Culture

Chapter 2. How to Work Well on Teams

Written by Brian Fitzpatrick

Edited by Riona MacNamara

Because this chapter is about the cultural and social aspects of software engineering at Google, it makes sense to begin by focusing on the one variable over which you definitely have control: you.

People are inherently imperfect—we like to say that humans are mostly a collection of intermittent bugs. But before you can understand the bugs in your coworkers, you need to understand the bugs in yourself. We're going to ask you to think about your own reactions, behaviors, and attitudes—and in return, we hope you gain some real insight into how to become a more efficient and successful software engineer who spends less energy dealing with people-related problems and more time writing great code.

The critical idea in this chapter is that software development is a team endeavor. And to succeed on an engineering team—or in any other creative collaboration—you need to reorganize your behaviors around the core principles of humility, respect, and trust.

Before we get ahead of ourselves, let's begin by observing how software engineers tend to behave in general.

Help Me Hide My Code

For the past 20 years, my colleague Ben[1] and I have spoken at many programming conferences. In 2006, we launched Google's (now deprecated) open source Project Hosting service, and at first, we used to get lots of questions and requests about the product. But around mid-2008, we began to notice a trend in the sort of requests we were getting:

Can you please give Subversion on Google Code the ability to hide specific branches?

Can you make it possible to create open source projects that start out hidden to the world and then are revealed when they're ready?

Hi, I want to rewrite all my code from scratch, can you please wipe all the history?

Can you spot a common theme to these requests?

The answer is *insecurity*. People are afraid of others seeing and judging their work in progress. In one sense, insecurity is just a part of human nature—nobody likes to be criticized, especially for things that aren't finished. Recognizing this theme tipped us off to a more general trend within software development: insecurity is actually a symptom of a larger problem.

The Genius Myth

Many humans have the instinct to find and worship idols. For software engineers those might be Linus Torvalds, Guido Van Rossum, Bill Gates—all heroes who changed the world with heroic feats. Linus wrote Linux by himself, right?

Actually, what Linus did was write just the beginnings of a proof-of-concept Unix-like kernel and show it to an email list. That was no small accomplishment, and it was definitely an impressive achievement, but it was just the tip of the iceberg. Linux is hundreds of times bigger than that initial kernel and was developed by *thousands* of smart people. Linus's real achievement was to lead these people and coordinate their work; Linux is the shining result not of his original idea, but of the collective labor of the community. (And Unix itself was not entirely written by Ken Thompson and Dennis Ritchie, but by a group of smart people at Bell Labs.)

On that same note, did Guido Van Rossum personally write all of Python? Certainly, he wrote the first version. But hundreds of others were responsible for contributing to subsequent versions, including ideas, features, and bug fixes. Steve Jobs led an entire team that built the Macintosh, and although Bill Gates is known for writing a BASIC interpreter for early home computers, his bigger achievement was building a successful company around MS-DOS. Yet they all became leaders and symbols of the collective achievements of their communities. The Genius Myth is the tendency that we as humans need to ascribe the success of a team to a single person/leader.

And what about Michael Jordan?

It's the same story. We idolized him, but the fact is that he didn't win every basketball game by himself. His true genius was in the way he worked with

his team. The team's coach, Phil Jackson, was extremely clever, and his coaching techniques are legendary. He recognized that one player alone never wins a championship and so he assembled an entire "dream team" around MJ. This team was a well-oiled machine—at least as impressive as Michael himself.

So, why do we repeatedly idolize the individual in these stories? Why do people buy products endorsed by celebrities? Why do we want to buy Michelle Obama's dress or Michael Jordan's shoes?

Celebrity is a big part of it. Humans have a natural instinct to find leaders and role models, idolize them, and attempt to imitate them. We all need heroes for inspiration, and the programming world has its heroes, too. The phenomenon of "techie-celebrity" has almost spilled over into mythology. We all want to write something world-changing like Linux or design the next brilliant programming language.

Deep down many engineers secretly wish to be seen as geniuses. This fantasy goes something like this:

- You are struck by an awesome new concept.
- You vanish into your cave for weeks or months, slaving away at a perfect implementation of your idea.
- You then "unleash" your software on the world, shocking everyone with your genius.
- Your peers are astonished by your cleverness.
- People line up to use your software.
- Fame and fortune follow naturally.

But hold on: time for a reality check. You're probably not a genius.

No offense, of course—we're sure that you're a very intelligent person. But do you realize how rare actual geniuses really are? Sure, you write code, and that's a tricky skill. But even if you are a genius, it turns out that that's not enough. Geniuses still make mistakes, and having brilliant ideas and elite programming skills doesn't guarantee that your software will be a hit. Worse, you might find yourself solving only analytical problems and not *human* problems. Being a genius is most definitely not an excuse for being a jerk: anyone—genius or not—with poor social skills tends to be a poor teammate. The vast majority of the work at Google (and at most companies!) doesn't require genius-level intellect, but 100% of the work

requires a minimal level of social skills. What will make or break your career, especially at a company like Google, is how well you collaborate with others.

It turns out that this Genius Myth is just another manifestation of our insecurity. Many programmers are afraid to share work they've only just started because it means peers will see their mistakes and know the author of the code is not a genius. To quote a friend:

I know I get SERIOUSLY insecure about people looking before something is done. Like they are going to seriously judge me and think I'm an idiot.

This is an extremely common feeling among programmers, and the natural reaction is to hide in a cave, work, work, work, and then polish, polish, polish, sure that no one will see your goof-ups and that you'll still have a chance to unveil your masterpiece when you're done. Hide away until your code is perfect.

Another common motivation for hiding your work is the fear that another programmer might take your idea and run with it before you get around to working on it. By keeping it secret, you control the idea.

We know what you're probably thinking now: so what? Shouldn't people be allowed to work however they want?

Actually, no. In this case, we assert that you're doing it wrong, and it *is* a big deal. Here's why.

Hiding Considered Harmful

If you spend all of your time working alone, you're increasing the risk of unnecessary failure and cheating your potential for growth. Even though software development is deeply intellectual work that can require deep concentration and alone time, you must play that off against the value (and need!) for collaboration and review.

First of all, how do you even know whether you're on the right track?

Imagine you're a bicycle-design enthusiast, and one day you get a brilliant idea for a completely new way to design a gear shifter. You order parts and proceed to spend weeks holed up in your garage trying to build a prototype. When your neighbor—also a bike advocate—asks you what's up, you decide not to talk about it. You don't want anyone to know about your project until

it's absolutely perfect. Another few months go by and you're having trouble making your prototype work correctly. But because you're working in secrecy, it's impossible to solicit advice from your mechanically inclined friends.

Then, one day your neighbor pulls his bike out of his garage with a radical new gear-shifting mechanism. Turns out he's been building something very similar to your invention, but with the help of some friends down at the bike shop. At this point, you're exasperated. You show him your work. He points out that your design had some simple flaws—ones that might have been fixed in the first week if you had shown him.

There are a number of lessons to learn here.

Early Detection

If you keep your great idea hidden from the world and refuse to show anyone anything until the implementation is polished, you're taking a huge gamble. It's easy to make fundamental design mistakes early on. You risk reinventing wheels.[2] And you forfeit the benefits of collaboration, too: notice how much faster your neighbor moved by working with others? This is why people dip their toes in the water before jumping in the deep end: you need to make sure that you're working on the right thing, you're doing it correctly, and it hasn't been done before. The chances of an early misstep are high. The more feedback you solicit early on, the more you lower this risk.[3] Remember the tried-and-true mantra of "Fail early, fail fast, fail often."

Early sharing isn't just about preventing personal missteps and getting your ideas vetted. It's also important to strengthen what we call the bus factor of your project.

The Bus Factor

Bus factor (noun): the number of people that need to get hit by a bus before your project is completely doomed.

How dispersed is the knowledge and know-how in your project? If you're the only person who understands how the prototype code works, you might enjoy good job security—but if you get hit by a bus, the project is toast. If you're working with a colleague, however, you've doubled the bus factor. And if you have a small team designing and prototyping together, things are even better—the project won't be marooned when a team member disappears. Remember: team members might not literally be hit by buses, but other

unpredictable life events still happen. Someone might get married, move away, leave the company, or take leave to care for a sick relative. Ensuring that there is *at least* good documentation in addition to a primary and a secondary owner for each area of responsibility helps future-proof your project's success and increases your project's bus factor. Hopefully most engineers recognize that it is better to be one part of a successful project than the critical part of a failed project.

Beyond the bus factor, there's the issue of overall pace of progress. It's easy to forget that working alone is often a tough slog, much slower than people want to admit. How much do you learn when working alone? How fast do you move? Google and Stack Overflow are great sources of opinions and information, but they're no substitute for actual human experience. Working with other people directly increases the collective wisdom behind the effort. When you become stuck on something absurd, how much time do you waste pulling yourself out of the hole? Think about how different the experience would be if you had a couple of peers to look over your shoulder and tell you—instantly—how you goofed and how to get past the problem. This is exactly why teams sit together (or do pair programming) in software engineering companies. Programming is hard. Software engineering is even harder. You need that second pair of eyes.

Pace of Progress

Here's another analogy. Think about how you work with your compiler. When you sit down to write a large piece of software, do you spend days writing 10,000 lines of code, and then, after writing that final, perfect line, press the "compile" button for the very first time? Of course you don't. Can you imagine what sort of disaster would result? Programmers work best in tight feedback loops: Write a new function, compile. Add a test, compile. Refactor some code, compile. This way, we discover and fix typos and bugs as soon as possible after generating code. We want the compiler at our side for every little step; some environments can even compile our code as we type. This is how we keep code quality high and make sure our software is evolving correctly bit by bit. The current DevOps philosophy toward tech productivity is explicit about these sorts of goals: get feedback as early as possible, test as early as possible, think about security and production environments as early as possible. This is all bundled into the idea of "shifting left" in the developer workflow; the earlier we find a problem, the cheaper it is to fix it.

The same sort of rapid feedback loop is needed not just at the code level, but at the whole-project level, too. Ambitious projects evolve quickly and must adapt to changing environments as they go. Projects run into unpredictable design obstacles or political hazards, or we simply discover that things aren't working as planned. Requirements morph unexpectedly. How do you get that feedback loop so that you know the instant your plans or designs need to change? Answer: by working in a team. Most engineers know the quote, "Many eyes make all bugs shallow," but a better version might be, "Many eyes make sure your project stays relevant and on track." People working in caves awaken to discover that while their original vision might be complete, the world has changed and their project has become irrelevant.

Engineers and Offices: Controversial Opinions

Twenty-five years ago, conventional wisdom stated that for an engineer to be productive, they needed to have their own office with a door that closed. This was supposedly the only way they could have big uninterrupted slabs of time to deeply concentrate on writing reams of code.

I think that it's not only unnecessary for most engineers[4] to be in a private office, it's downright dangerous. Software today is written by teams, not individuals, and a high-bandwidth, readily available connection to the rest of your team is even more valuable than your internet connection. You can have all the uninterrupted time in the world, but if you're using it to work on the wrong thing, you're wasting your time.

Unfortunately, it seems that modern-day tech companies (including Google, in some cases) have swung the pendulum to the exact opposite extreme. Walk into their offices and you'll often find engineers clustered together in massive rooms—100 or more people together—with no walls whatsoever. This "open floor plan" is now a topic of huge debate and as a result, hostility toward open offices is on the rise. The tiniest conversation becomes public, and people end up not talking for risk of annoying dozens of neighbors. This is just as bad as private offices!

We think the middle ground is really the best solution. Group teams of four to eight people together in small rooms (or large offices), so as to make it easy (and non-embarrassing) for spontaneous conversation to happen.

Of course, in any situation, individual engineers still need a way to filter out noise and interruptions, which is why most teams I've seen have developed a way to communicate that they're currently busy and that you should limit interruptions. Some of us used to work on a team with a vocal interrupt

protocol: if you wanted to talk, you would say, “breakpoint Mary,” where Mary was the name of the person you wanted to talk to. If Mary was at a point where she could stop, she would swing her chair around and listen. If Mary was too busy, she’d just say “ack” and you’d go on with other things until she finished with her current head state.

Other teams have tokens or stuffed animals that team members put on their monitor to signify that they should be interrupted only in case of emergency. Still other teams give out noise-canceling headphones to engineers to make it easier to deal with background noise—in fact, in many companies the very act of wearing headphones is a common signal that means “don’t disturb me unless it’s really important.” Many engineers tend to go into headphones-only mode when coding, which may be useful for short spurts but, if used all the time, can be just as bad for collaboration as walling yourself off in an office.

Don’t misunderstand us—we still think engineers need uninterrupted time to focus on writing code, but we think they need a high-bandwidth, low-friction connection to their team just as much. If less-knowledgeable people on your team feel that there’s a barrier to asking you a question, it’s a problem: finding the right balance is an art.

In Short, Don’t Hide

So, what “hiding” boils down to is this: working alone is inherently riskier than working with others. Even though you might be afraid of someone stealing your idea or thinking you’re not intelligent, you should be much more concerned about wasting huge swaths of your time toiling away on the wrong thing.

Don’t become another statistic.

It’s All About the Team

So, let’s back up now and put all of these ideas together.

The point we’ve been hammering away at is that in the realm of programming, lone craftspeople are extremely rare—and even when they do exist, they don’t perform superhuman achievements in a vacuum; their world-changing accomplishment is almost always the result of a spark of inspiration followed by a heroic team effort.

A great team makes brilliant use of its superstars, but the whole is always greater than the sum of its parts. But creating a superstar team is fiendishly difficult.

Let's put this idea into simpler words: *Software engineering is a team endeavor*.

This concept directly contradicts the inner Genius Programmer fantasy so many of us hold, but it's not enough to be brilliant when you're alone in your hacker's lair. You're not going to change the world or delight millions of computer users by hiding and preparing your secret invention. You need to work with other people. Share your vision. Divide the labor. Learn from others. Create a brilliant team.

Consider this: how many pieces of widely used, successful software can you name that were truly written by a single person? (Some people might say "LaTeX," but it's hardly "widely used," unless you consider the number of people writing scientific papers to be a statistically significant portion of all computer users!)

High-functioning teams are gold and the true key to success. You should be aiming for this experience however you can.

The Three Pillars of Social Interaction

So, if teamwork is the best route to producing great software, how does one build (or find) a great team?

To reach collaborative nirvana, you first need to learn and embrace what I call the "three pillars" of social skills. These three principles aren't just about greasing the wheels of relationships; they're the foundation on which all healthy interaction and collaboration are based:

Pillar 1: Humility

> You are not the center of the universe (or your code!). You're neither omniscient nor infallible. You're open to self-improvement.

Pillar 2: Respect

> You genuinely care about others you work with. You treat them kindly and appreciate their abilities and accomplishments.

Pillar 3: Trust

> You believe others are competent and will do the right thing, and you're OK with letting them drive when appropriate.[5]

If you perform a root-cause analysis on almost any social conflict, you can ultimately trace it back to a lack of humility, respect, and/or trust. That might sound implausible at first, but give it a try. Think about some nasty or uncomfortable social situation currently in your life. At the basest level, is everyone being appropriately humble? Are people really respecting one another? Is there mutual trust?

Why Do These Pillars Matter?

When you began this chapter, you probably weren't planning to sign up for some sort of weekly support group. We empathize. Dealing with social problems can be difficult: people are messy, unpredictable, and often annoying to interface with. Rather than putting energy into analyzing social situations and making strategic moves, it's tempting to write off the whole effort. It's much easier to hang out with a predictable compiler, isn't it? Why bother with the social stuff at all?

Here's a quote from a famous lecture by Richard Hamming:[6]

By taking the trouble to tell jokes to the secretaries and being a little friendly, I got superb secretarial help. For instance, one time for some idiot reason all the reproducing services at Murray Hill were tied up. Don't ask me how, but they were. I wanted something done. My secretary called up somebody at Holmdel, hopped [into] the company car, made the hour-long trip down and got it reproduced, and then came back. It was a payoff for the times I had made an effort to cheer her up, tell her jokes and be friendly; it was that little extra work that later paid off for me. By realizing you have to use the system and studying how to get the system to do your work, you learn how to adapt the system to your desires.

The moral is this: do not underestimate the power of playing the social game. It's not about tricking or manipulating people; it's about creating relationships to get things done. Relationships always outlast projects. When you've got richer relationships with your coworkers, they'll be more willing to go the extra mile when you need them.

Humility, Respect, and Trust in Practice

All of this preaching about humility, respect, and trust sounds like a sermon. Let's come out of the clouds and think about how to apply these ideas in real-

life situations. We're going to examine a list of specific behaviors and examples that you can start with. Many of them might sound obvious at first, but after you begin thinking about them, you'll notice how often you (and your peers) are guilty of not following them—we've certainly noticed this about ourselves!

LOSE THE EGO

OK, this is sort of a simpler way of telling someone without enough humility to lose their 'tude. Nobody wants to work with someone who consistently behaves like they're the most important person in the room. Even if you know you're the wisest person in the discussion, don't wave it in people's faces. For example, do you always feel like you need to have the first or last word on every subject? Do you feel the need to comment on every detail in a proposal or discussion? Or do you know somebody who does these things?

Although it's important to be humble, that doesn't mean you need to be a doormat; there's nothing wrong with self-confidence. Just don't come off like a know-it-all. Even better, think about going for a "collective" ego, instead; rather than worrying about whether you're personally awesome, try to build a sense of team accomplishment and group pride. For example, the Apache Software Foundation has a long history of creating communities around software projects; these communities have incredibly strong identities and reject people who are more concerned with self-promotion.

Ego manifests itself in many ways, and a lot of the time it can get in the way of your productivity and slow you down. Here's another great story from Hamming's lecture that illustrates this point perfectly (emphasis ours):

John Tukey almost always dressed very casually. He would go into an important office and it would take a long time before the other fellow realized that this is a first-class man and he had better listen. For a long time, John has had to overcome this kind of hostility. It's wasted effort! I didn't say you should conform; I said, "The appearance of conforming gets you a long way." If you chose to assert your ego in any number of ways, "I am going to do it my way," you pay a small steady price throughout the whole of your professional career. And this, over a whole lifetime, adds up to an enormous amount of needless trouble. […] By realizing you have to use the system and studying how to get the system to do your work, you learn how to adapt the system to your desires. Or you can fight it steadily, as a small, undeclared war, for the whole of your life.

LEARN TO GIVE *AND* TAKE CRITICISM

A few years ago, Joe started a new job as a programmer. After his first week, he really began digging into the codebase. Because he cared about what was going on, he started gently questioning other teammates about their contributions. He sent simple code reviews by email, politely asking about design assumptions or pointing out places where logic could be improved. After a couple of weeks, he was summoned to his director's office. "What's the problem?" Joe asked. "Did I do something wrong?" The director looked concerned: "We've had a lot of complaints about your behavior, Joe. Apparently, you've been really harsh toward your teammates, criticizing them left and right. They're upset. You need to tone it down." Joe was utterly baffled. Surely, he thought, his code reviews should have been welcomed and appreciated by his peers. In this case, however, Joe should have been more sensitive to the team's widespread insecurity and should have used a subtler means to introduce code reviews into the culture—perhaps even something as simple as discussing the idea with the team in advance and asking team members to try it out for a few weeks.

In a professional software engineering environment, criticism is almost never personal—it's usually just part of the process of making a better project. The trick is to make sure you (and those around you) understand the difference between a constructive criticism of someone's creative output and a flat-out assault against someone's character. The latter is useless—it's petty and nearly impossible to act on. The former can (and should!) be helpful and give guidance on how to improve. And, most important, it's imbued with respect: the person giving the constructive criticism genuinely cares about the other person and wants them to improve themselves or their work. Learn to respect your peers and give constructive criticism politely. If you truly respect someone, you'll be motivated to choose tactful, helpful phrasing—a skill acquired with much practice. We cover this much more in Chapter 9.

On the other side of the conversation, you need to learn to accept criticism, as well. This means not just being humble about your skills, but trusting that the other person has your best interests (and those of your project!) at heart and doesn't actually think you're an idiot. Programming is a skill like anything else: it improves with practice. If a peer pointed out ways in which you could improve your juggling, would you take it as an attack on your character and value as a human being? We hope not. In the same way, your self-worth shouldn't be connected to the code you write—or any creative project you build. To repeat ourselves: *you are not your code*. Say that over and over.

You are not what you make. You need to not only believe it yourself, but get your coworkers to believe it, too.

For example, if you have an insecure collaborator, here's what not to say: "Man, you totally got the control flow wrong on that method there. You should be using the standard xyzzy code pattern like everyone else." This feedback is full of antipatterns: you're telling someone they're "wrong" (as if the world were black and white), demanding they change something, and accusing them of creating something that goes against what everyone else is doing (making them feel stupid). Your coworker will immediately be put on the offense, and their response is bound to be overly emotional.

A better way to say the same thing might be, "Hey, I'm confused by the control flow in this section here. I wonder if the xyzzy code pattern might make this clearer and easier to maintain?" Notice how you're using humility to make the question about you, not them. They're not wrong; you're just having trouble understanding the code. The suggestion is merely offered up as a way to clarify things for poor little you while possibly helping the project's long-term sustainability goals. You're also not demanding anything—you're giving your collaborator the ability to peacefully reject the suggestion. The discussion stays focused on the code itself, not on anyone's value or coding skills.

FAIL FAST AND ITERATE

There's a well-known urban legend in the business world about a manager who makes a mistake and loses an impressive $10 million. He dejectedly goes into the office the next day and starts packing up his desk, and when he gets the inevitable "the CEO wants to see you in his office" call, he trudges into the CEO's office and quietly slides a piece of paper across the desk.

"What's this?" asks the CEO.

"My resignation," says the executive. "I assume you called me in here to fire me."

"Fire you?" responds the CEO, incredulously. "Why would I fire you? I just spent $10 million training you!"[7]

It's an extreme story, to be sure, but the CEO in this story understands that firing the executive wouldn't undo the $10 million loss, and it would compound it by losing a valuable executive who he can be very sure won't make that kind of mistake again.

At Google, one of our favorite mottos is "Failure is an option." It's widely recognized that if you're not failing now and then, you're not being innovative enough or taking enough risks. Failure is viewed as a golden opportunity to learn and improve for the next go-around.[8] In fact, Thomas Edison is often quoted as saying, "If I find 10,000 ways something won't work, I haven't failed. I am not discouraged, because every wrong attempt discarded is another step forward."

Over in Google X—the division that works on "moonshots" like self-driving cars and internet access delivered by balloons—failure is deliberately built into its incentive system. People come up with outlandish ideas and coworkers are actively encouraged to shoot them down as fast as possible. Individuals are rewarded (and even compete) to see how many ideas they can disprove or invalidate in a fixed period of time. Only when a concept truly cannot be debunked at a whiteboard by all peers, does it proceed to early prototype.

Blameless Postmortem Culture

The key to learning from your mistakes is to document your failures by performing a root-cause analysis and writing up a "postmortem," as it's called at Google (and many other companies). Take extra care to make sure the postmortem document isn't just a useless list of apologies or excuses or finger-pointing—that's not its purpose. A proper postmortem should always contain an explanation of what was learned and what is going to change as a result of the learning experience. Then, make sure that the postmortem is readily accessible and that the team really follows through on the proposed changes. Properly documenting failures also makes it easier for other people (present and future) to know what happened and avoid repeating history. Don't erase your tracks—light them up like a runway for those who follow you!

A good postmortem should include the following:

- A brief summary of the event
- A timeline of the event, from discovery through investigation to resolution
- The primary cause of the event
- Impact and damage assessment
- A set of action items (with owners) to fix the problem immediately

- A set of action items to prevent the event from happening again
- Lessons learned

LEARN PATIENCE

Years ago, I was writing a tool to convert CVS repositories to Subversion (and later, Git). Due to the vagaries of CVS, I kept unearthing bizarre bugs. Because my longtime friend and coworker Karl knew CVS quite intimately, we decided we should work together to fix these bugs.

A problem arose when we began pair programming: I'm a bottom-up engineer who is content to dive into the muck and dig my way out by trying a lot of things quickly and skimming over the details. Karl, however, is a top-down engineer who wants to get the full lay of the land and dive into the implementation of almost every method on the call stack before proceeding to tackle the bug. This resulted in some epic interpersonal conflicts, disagreements, and the occasional heated argument. It got to the point at which the two of us simply couldn't pair-program together: it was too frustrating for us both.

That said, we had a longstanding history of trust and respect for each other. Combined with patience, this helped us work out a new method of collaborating. We would sit together at the computer, identify the bug, and then split up and attack the problem from two directions at once (top-down and bottom-up) before coming back together with our findings. Our patience and willingness to improvise new working styles not only saved the project, but also our friendship.

BE OPEN TO INFLUENCE

The more open you are to influence, the more you are able to influence; the more vulnerable you are, the stronger you appear. These statements sound like bizarre contradictions. But everyone can think of someone they've worked with who is just maddeningly stubborn; no matter how much people try to persuade them, they dig their heels in even more. What eventually happens to such team members? In our experience, people stop listening to their opinions or objections; instead, they end up "routing around" them like an obstacle everyone takes for granted. You certainly don't want to be that person, so keep this idea in your head: it's OK for someone else to change your mind. In the opening chapter of this book, we said that engineering is inherently about trade-offs. It's impossible for you to be right about everything all the time unless you have an unchanging environment and perfect knowledge, so of course you should change your mind when

presented with new evidence. Choose your battles carefully: to be heard properly, you first need to listen to others. It's better to do this listening *before* putting a stake in the ground or firmly announcing a decision—if you're constantly changing your mind, people will think you're wishy-washy.

The idea of vulnerability can seem strange, too. If someone admits ignorance of the topic at hand or the solution to a problem, what sort of credibility will she have in a group? Vulnerability is a show of weakness, and that destroys trust, right?

Not true. Admitting that you've made a mistake or you're simply out of your league can increase your status over the long run. In fact, the willingness to express vulnerability is an outward show of humility; it demonstrates accountability and the willingness to take responsibility; and it's a signal that you trust others' opinions. In return, people end up respecting your honesty and strength. Sometimes, the best thing you can do is just say, "I don't know."

Professional politicians, for example, are notorious for never admitting error or ignorance, even when it's patently obvious that they're wrong or unknowledgeable about a subject. This behavior exists primarily because politicians are constantly under attack by their opponents, and it's why most people don't believe a word that politicians say. When you're writing software, however, you don't need to be continually on the defensive—your teammates are collaborators, not competitors. You all have the same goal.

Being Googley

At Google, we have our own internal version of the principles of "humility, respect, and trust" when it comes to behavior and human interactions.

From the earliest days of our culture, we often referred to actions as being "Googley" or "not Googley." The word was never explicitly defined; rather, everyone just sort of took it to mean "don't be evil" or "do the right thing" or "be good to each other." Over time, people also started using the term "Googley" as an informal test for culture-fit whenever we would interview a candidate for an engineering job, or when writing internal performance reviews of one another. People would often express opinions about others using the term; for example, "the person coded well, but didn't seem to have a very Googley attitude."

Of course, we eventually realized that the term "Googley" was being overloaded with meaning; worse yet, it could become a source of unconscious bias in hiring or evaluations. If "Googley" means something different to every employee, we run the risk of the term starting to mean *"is just like me*." Obviously, that's not a good test for hiring—we don't want to hire people "just like me," but people from diverse set of backgrounds and with different opinions and experiences. An interviewer's personal desire to have a beer with a candidate (or coworker) should *never* be considered a valid signal about somebody else's performance or ability to thrive at Google.

Eventually Google fixed the problem by explicitly defining a rubric for what we mean by "Googleyness"—a set of attributes and behaviors that we look for, which represent strong leadership and exemplify "humility, respect, and trust":

Thrives in ambiguity

> Can deal with conflicting messages or directions, build consensus, and make progress against a problem, even when the environment is constantly shifting.

Values feedback

> Has humility to both receive and give feedback gracefully and understands how valuable feedback is for personal (and team) development.

Challenges status quo

> Is able to set ambitious goals and pursue them even when there might be resistance or inertia from others.

Puts the user first

> Has empathy and respect for users of Google's products and pursues actions that are in their best interests.

Cares about the team

> Has empathy and respect for coworkers and actively works to help them without being asked, improving team cohesion.

Does the right thing

> Has a strong sense of ethics about everything they do; willing to make difficult or inconvenient decisions to protect the integrity of the team and product.

Now that we have these best-practice behaviors better defined, we've begun to shy away from using the term "Googley." It's always better to be specific about expectations!

Conclusion

The foundation for almost any software endeavor, of almost any size, is a well-functioning team. Although the "genius myth" of the solo software developer still persists, the truth is that no one really goes it alone. For a software organization to stand the test of time, it must have a healthy culture, rooted in humility, trust and respect that revolves around the team, rather than the individual. As well, the creative nature of software development *requires* that people take risks and occasionally fail; for people to accept that failure, a healthy team environment must exist.

TL;DRs

- Be aware of the trade-offs of working in isolation.
- Acknowledge the amount of time that you and your team spend communicating and in interpersonal conflict. A small investment in understanding personalities and working styles of yourself and others can go a long way to improving productivity.
- If you want to work effectively with a team or a large organization, be aware of your preferred working style and that of others.

1 Ben Collins-Sussman, also an author within this book

2 Literally, if you are, in fact, a bike designer.

3 I should note that sometimes it's dangerous to get too much feedback too early in the process if you're still unsure of your general direction or goal.

4 I do, however, acknowledge that serious introverts likely need more peace, quiet, and alone time than most people and might benefit from a quieter environment if not their own office.

5 This is incredibly difficult if you've been burned in the past by delegating to incompetent people.

6 "You and Your Research"

7 You can find a dozen variants of this legend on the web, attributed to different famous managers.

8 By the same token, if you do the same thing over and over and keep failing, it’s not failure, it’s incompetence.

Chapter 3. Knowledge Sharing

Written by Nina Chen and Mark Barolak

Edited by Riona MacNamara

Your organization understands your problem domain better than some random person on the internet; your organization should be able to answer most of its own questions. To achieve that, you need both experts that know the answers to those questions *and* mechanisms to distribute their knowledge, which is what we'll explore in this chapter. These mechanisms range from the utterly simple (Ask questions; Write down what you know) to the much more structured, such as tutorials and classes. Most important, however, your organization needs a *culture of learning*, and that requires creating the psychological safety that permits people to admit to a lack of knowledge.

Challenges to Learning

Sharing expertise across an organization is not an easy task. Without a strong culture of learning, challenges can emerge. Google has experienced a number of these challenges, especially as the company has scaled:

Lack of psychological safety

> An environment in which people are afraid to take risks or make mistakes in front of others because they fear being punished for it. This often manifests as a culture of fear or a tendency to avoid transparency.

Information islands

> Knowledge fragmentation that occurs in different parts of an organization that don't communicate with one another or use shared resources. In such an environment, each group develops its own way of doing things.[1] This often leads to the following:
>
> - *Information fragmentation*: Each island has an incomplete picture of the bigger whole.
> - *Information duplication*: Each island has reinvented its own way to do something.
> - *Information skew*: each island has its own way of doing the same thing, and these might or might not conflict.

Single point of failure (SPOF)

A bottleneck that occurs when critical information is available from only a single person. This is related to *bus factor*, which we discuss in more detail in Chapter 2.

SPOFs can arise out of good intentions: it can be easy to fall into a habit of "Let me take care of that for you." But this approach optimizes for short-term efficiency ("It's faster for me to do it") at the cost of poor long-term scalability (the team never learns how to do whatever it is that needs to be done). This mindset also tends to lead to *all or nothing expertise*.

All or nothing expertise

A group of people that is split between people who know "everything" and novices, with little middle ground. This problem often reinforces itself if experts always do everything themselves and don't take the time to develop new experts through mentoring or documentation. In this scenario, knowledge and responsibilities continue to accumulate on those who already have expertise, and new team members or novices are left to fend for themselves and ramp up more slowly.

Parroting

Mimicry without understanding. This is typically characterized by mindlessly copying patterns or code without understanding their purpose, often under the assumption that said code is needed for unknown reasons.

Haunted graveyards

Places, often in code, that people avoid touching or changing because they are afraid that something might go wrong. Unlike the aforementioned *parroting*, haunted graveyards are characterized by people avoiding action because of fear and superstition.

In the rest of this chapter, we dive into strategies that Google's engineering organizations have found to be successful in addressing these challenges.

Philosophy

Software engineering can be defined as the multiperson development of multiversion programs.[2] People are at the core of software engineering: code is an important output but only a small part of building a product. Crucially,

code does not emerge spontaneously out of nothing, and neither does expertise. Every expert was once a novice: an organization's success depends on growing and investing in its people.

Personalized, one-to-one advice from an expert is always invaluable. Different team members have different areas of expertise, and so the best teammate to ask for any given question will vary. But if the expert goes on vacation or switches teams, the team can be left in the lurch. And although one person might be able to provide personalized help for one-to-many, this doesn't scale and is limited to small numbers of "many."

Documented knowledge, on the other hand, can better scale not just to the team but to the entire organization. Mechanisms such as a team wiki enable many authors to share their expertise with a larger group. But even though written documentation is more scalable than one-to-one conversations, that scalability comes with some trade-offs: it might be more generalized and less applicable to individual learners' situations, and it comes with the added maintenance cost required to keep information relevant and up to date over time.

Tribal knowledge exists in the gap between what individual team members know and what is documented. Human experts know these things that aren't written down. If we document that knowledge and maintain it, it is now available not only to somebody with direct one-to-one access to the expert today, but to anybody who can find and view the documentation.

So in a magical world in which everything is always perfectly and immediately documented, we wouldn't need to consult a person any more, right? Not quite. Written knowledge has scaling advantages, but so does targeted human help. A human expert can synthesize their expanse of knowledge. They can assess what information is applicable to the individual's use case, determine whether the documentation is still relevant, and know where to find it. Or, if they don't know where to find the answers, they might know who does.

Tribal and written knowledge complement each other. Even a perfectly expert team with perfect documentation needs to communicate with one another, coordinate with other teams, and adapt their strategies over time. No single knowledge-sharing approach is the correct solution for all types of learning, and the particulars of a good mix will likely vary based on your organization. Institutional knowledge evolves over time, and the knowledge-sharing methods that work best for your organization will likely change as it

grows. Train, focus on learning and growth, and build your own stable of experts: there is no such thing as too much engineering expertise.

Setting the Stage: Psychological Safety

Psychological safety is critical to promoting a learning environment.

To learn, you must first acknowledge that there are things you don't understand. We should welcome such honesty rather than punish it. (Google does pretty well, but sometimes engineers are reluctant to admit they don't understand something.)

An enormous part of learning is being able to try things and feeling safe to fail. In a healthy environment, people feel comfortable asking questions, being wrong, and learning new things. This is a baseline expectation for all Google teams; indeed, our research has shown that psychological safety is the most important part of an effective team.

Mentorship

At Google, we try to set the tone as soon as a "Noogler" (new Googler) engineer or internal transfer joins a new team. One important way of building psychological safety is to assign new team members a mentor—someone who is not their manager or tech lead—whose responsibilities explicitly include answering questions and helping the new team member ramp up. Having an officially assigned team mentor to ask for help makes it easier for the newcomer and means that they don't need to worry about taking up too much of their coworkers' time.

A mentor is a volunteer who has been at Google for more than a year and who is available to advise on anything from using Google infrastructure to navigating Google culture. Crucially, the mentor is there to be a safety net to talk to if the mentee doesn't know whom else to ask for advice. This mentor is *not* on the same team as the mentee, which can make the mentee feel more comfortable about asking for help in tricky situations.

Mentorship formalizes and facilitates learning, but learning itself is an ongoing process. There will always be opportunities for coworkers to learn from one another, whether it's a new employee joining the organization or an experienced engineer learning a new technology. With a healthy team, teammates will be open not just to answering but also to *asking* questions: showing that they don't know something and learning from one another.

Psychological Safety in Large Groups

Asking a nearby teammate for help is easier than approaching a large group of mostly strangers. But as we've seen, one-to-one solutions don't scale well. Group solutions are more scalable, but they are also scarier. It can be intimidating for novices to form a question and ask it of a large group, knowing that their question might be archived for many years. The need for psychological safety is amplified in large groups. Every member of the group has a role to play in creating and maintaining a safe environment that ensures that newcomers are confident asking questions and up-and-coming experts feel empowered to help those newcomers without the fear of having their answers attacked by established experts.

The most important way to achieve this safe and welcoming environment is for group interactions to be cooperative, not adversarial. Table 3-1 lists some examples of recommended patterns (and their corresponding antipatterns) of group interactions.

Recommended patterns (cooperative)	Antipatterns (adversarial)
Basic questions or mistakes are guided in the proper direction	Basic questions or mistakes are picked on, and the person asking the question is chastised
Explanations are given with the intent of helping the person asking the question learn	Explanations are given with the intent of showing off one's own knowledge
Responses are kind, patient, and helpful	Responses are condescending, snarky, and unconstructive
Interactions are shared discussions for finding solutions	Interactions are arguments with "winners" and "losers"

Table 3-1. Group interaction patterns

These antipatterns can emerge unintentionally: someone might be trying to be helpful but is accidentally condescending and unwelcoming. We find the Recurse Center's social rules to be helpful here:

No feigned surprise ("What?! I can't believe you don't know what the stack is!")

Feigned surprise is a barrier to psychological safety and makes members of the group afraid of admitting to a lack of knowledge.

No "well-actuallys"

Pedantic corrections that tend to be about grandstanding rather than precision.

No back-seat driving

Interrupting an existing discussion to offer opinions without committing to the conversation.

No subtle "-isms" ("It's so easy my grandmother could do it!")

Small expressions of bias (racism, ageism, homophobia) that can make individuals feel unwelcome, disrespected, or unsafe.

Growing Your Knowledge

Knowledge sharing starts with yourself. It is important to recognize that you always have something to learn. The following guidelines allow you to augment your own personal knowledge.

Ask Questions

If you take away only a single thing from this chapter, it is this: always be learning; always be asking questions.

We tell new Nooglers that ramping up can take around six months. This extended period is necessary to ramp up on Google's large, complex infrastructure, but it also reinforces the idea that learning is an ongoing, iterative process. One of the biggest mistakes that beginners make is not to ask for help when they're stuck. You might be tempted to struggle through it alone or feel fearful that your questions are "too simple." "I just need to try harder before I ask anyone for help," you think. Don't fall into this trap! Your coworkers are often the best source of information: leverage this valuable resource.

There is no magical day when you suddenly always know exactly what to do in every situation—there's always more to learn. Engineers who have been at Google for years still have areas in which they don't feel like they know what they are doing, and that's okay! Don't be afraid to say "I don't know what

that is; could you explain it?" Embrace not knowing things as an area of opportunity rather than one to fear.[3]

It doesn't matter whether you're new to a team or a senior leader: you should always be in an environment in which there's something to learn. If not, you stagnate (and should find a new environment).

It's especially critical for those in leadership roles to model this behavior: it's important not to mistakenly equate "seniority" with "knowing everything." In fact, the more you know, the more you know you don't know. Openly asking questions[4] or expressing gaps in knowledge reinforces that it's okay for others to do the same.

On the receiving end, patience and kindness when answering questions fosters an environment in which people feel safe looking for help. Making it easier to overcome the initial hesitation to ask a question sets the tone early: reach out to solicit questions, and make it easy for even "trivial" questions to get an answer. Although engineers *could* probably figure out tribal knowledge on their own, they're not here to work in a vacuum. Targeted help allows engineers to be productive faster, which in turn makes their entire team more productive.

Understand Context

Learning is not just about understanding new things; it also includes developing an understanding of the decisions behind the design and implementation of existing things. Suppose that your team inherits a legacy codebase for a critical piece of infrastructure that has existed for many years. The original authors are long gone, and the code is difficult to understand. It can be tempting to rewrite from scratch rather than spending time learning the existing code. But instead of thinking "I don't get it" and ending your thoughts there, dive deeper: what questions should you be asking?

Consider the principle of "Chesterson's fence": before removing or changing something, first understand why it's there.

In the matter of reforming things, as distinct from deforming them, there is one plain and simple principle; a principle which will probably be called a paradox. There exists in such a case a certain institution or law; let us say, for the sake of simplicity, a fence or gate erected across a road. The more modern type of reformer goes gaily up to it and says, "I don't see the use of this; let us clear it away." To which the more intelligent type of reformer will do well to answer: "If you don't see the use of it, I certainly won't let you clear it away. Go away

and think. Then, when you can come back and tell me that you do see the use of it, I may allow you to destroy it."[5]

This doesn't mean that code can't lack clarity or that existing design patterns can't be wrong, but engineers have a tendency to reach for "this is bad!" far more quickly than is often warranted, especially for unfamiliar code, languages, or paradigms. Google is not immune to this. Seek out and understand context, especially for decisions that seem unusual. After you've understood the context and purpose of the code, consider whether your change still makes sense. If it does, go ahead and make it; if it doesn't, document your reasoning for future readers.

Many Google style guides explicitly include context to help readers understand the rationale behind the style guidelines instead of just memorizing a list of arbitrary rules. More subtly, understanding the rationale behind a given guideline allows authors to make informed decisions about when the guideline shouldn't apply or whether the guideline needs updating. See Chapter 8.

Scaling Your Questions: Ask the Community

Getting one-to-one help is high bandwidth but necessarily limited in scale. And as a learner, it can be difficult to remember every detail. Do your future self a favor: when you learn something from a one-to-one discussion, *write it down*.

Chances are that future newcomers will have the same questions you had. Do them a favor, too, and *share what you write down*.

Although sharing the answers you receive can be useful, it's also beneficial to seek help not from individuals but from the greater community. In this section, we examine different forms of community-based learning. Each of these approaches—group chats, mailing lists, and question-and-answer systems—have different trade-offs and complement one another But each of them enables the knowledge seeker to get help from a broader community of peers and experts, but also ensures that answers are broadly available to current and future members of that community.

Group Chats

When you have a question, it can sometimes be difficult to get help from the appropriate person. Maybe you're not sure who knows the answer or the

person you want to ask is busy. In these situations, group chats are great, because you can ask your question to many people at once and have a quick back-and-forth conversation with whomever is available. As a bonus, other members of the group chat can learn from the question and answer, and many forms of group chat can be automatically archived and searched later.

Group chats tend to be devoted either to topics or to teams. Topic-driven group chats are typically open so that anyone can drop in to ask a question. They tend to attract experts and can grow quite large, so questions are usually answered quickly. Team-oriented chats, on the other hand, tend to be smaller and restrict membership. As a result, they might not have the same reach as a topic-driven chat, but their smaller size can feel safer to a newcomer.

Although group chats are great for quick questions, they don't provide much structure, which can make it difficult to extract meaningful information from a conversation in which you're not actively involved. As soon as you need to share information outside of the group, or make it available to refer back to later, you should write a document or email a mailing list.

Mailing Lists

Most topics at Google have a topic-users@ or topic-discuss@ Google Groups mailing list that anyone at the company can join or email. Asking a question on a public mailing list is a lot like asking a group chat: the question reaches a lot of people who could potentially answer it and anyone following the list can learn from the answer. Unlike group chats, though, public mailing lists are easy to share with a wider audience: they are packaged into searchable archives, and email threads provide more structure than group chats. At Google, mailing lists are also indexed and can be discovered by Moma, Google's intranet search engine.

When you find an answer to a question you asked on a mailing list, it can be tempting to get on with your work. Don't do it! You never know when someone will need the same information in the future,[6] so it's a best practice to post the answer back to the list.

Mailing lists are not without their trade-offs. They're well suited for complicated questions that require a lot of context, but they're clumsy for the quick back-and-forth exchanges at which group chats excel. A thread about a particular problem is generally most useful while it is active. Email archives are immutable, and it can be hard to determine whether an answer discovered in an old discussion thread is still relevant to a present-day situation. Additionally, the signal-to-noise ratio can be lower than other mediums like

formal documentation because the problem that someone is having with their specific workflow might not be applicable to you.

EMAIL AT GOOGLE

Google culture is infamously email-centric and email-heavy. Google engineers receive hundreds of emails (if not more) each day, with varying degrees of actionability. Nooglers can spend days just setting up email filters to deal with the volume of notifications coming from groups that they've been autosubscribed to; some people just give up and don't try to keep up with the flow. Some groups CC large mailing lists onto every discussion by default, without trying to target information to those who are likely to be specifically interested in it; as a result, the signal-to-noise ratio can be a real problem.

Google tends toward email-based workflows by default. This isn't necessarily because email is a better medium than other communications options—it often isn't—rather, it's because that's what our culture is accustomed to. Keep this in mind as your organization considers what forms of communication to encourage or invest in.

YAQS: Question and Answer Platform

YAQS ("Yet Another Question System") is a Google-internal version of a Stack Overflow–like website, making it easy for Googlers to link to existing or work-in-progress code as well as discuss confidential information.

Like Stack Overflow, YAQS shares many of the same advantages of mailing lists and adds refinements: answers marked as helpful are promoted in the user interface, and users can edit questions and answers so that they remain accurate and useful as code and facts change. As a result, some mailing lists have been superseded by YAQS, whereas others have evolved into more general discussion lists that are less focused on problem solving.

Scaling Your Knowledge: You Always Have Something to Teach

Teaching is not limited to experts, nor is expertise a binary state in which you are either a novice or an expert. Expertise is a multidimensional vector of what you know: everyone has varying levels of expertise across different areas. This is one of the reasons why diversity is critical to organizational success: different people bring different perspectives and expertise to the table (see Chapter 4). Google engineers teach others in a variety of ways, such as office hours, giving tech talks, teaching classes, writing documentation, and reviewing code.

Office Hours

Sometimes it's really important to have a human to talk to, and in those instances office hours can be a good solution. Office hours are a regularly scheduled (typically weekly) event during which one or more people make themselves available to answer questions about a particular topic. Office hours are almost never the first choice for knowledge sharing: if you have an urgent question, it can be painful to wait for the next session for an answer; and if you're hosting office hours, they take up time and need to be regularly promoted. That said, they do provide a way for people to talk to an expert in person. This is particularly useful if the problem is still ambiguous enough that the engineer doesn't yet know what questions to ask (such as when they're just starting to design a new service) or whether the problem is about something so specialized that there just isn't documentation on it.

Tech Talks and Classes

Google has a robust culture of both internal and external[7] tech talks and classes. Our engEDU (Engineering Education) team focuses on providing Computer Science education to many audiences, ranging from Google engineers to students around the world. At a more grassroots level, our g2g (Googler2Googler) program lets Googlers sign up to give or attend talks and classes from fellow Googlers.[8] The program is wildly successful, with thousands of participating Googlers teaching topics from the technical (e.g., "Understanding Vectorization in Modern CPUs") to the just-for-fun (e.g., "Beginner Swing Dance").

Tech talks typically consist of a speaker presenting directly to an audience. Classes, on the other hand, can have a lecture component but often center on in-class exercises and therefore require more active participation from attendees. As a result, instructor-led classes are typically more demanding and expensive to create and maintain than tech talks and are reserved for the most important or difficult topics. That said, after a class has been created, it can be scaled relatively easily because many instructors can teach a class from the same course materials. We've found that classes tend to work best when the following circumstances exist:

- The topic is complicated enough that it's a frequent source of misunderstanding. Classes take a lot of work to create, so they should be developed only when they're addressing specific needs.

- The topic is relatively stable. Updating class materials is a lot of work, so if the subject is rapidly evolving, other forms of sharing knowledge will have a better bang for the buck.
- The topic benefits from having teachers available to answer questions and provide personalized help. If students can easily learn without directed help, self-serve mediums like documentation or recordings are more efficient. A number of introductory classes at Google also have self-study versions.
- There is enough demand to offer the class regularly. Otherwise, potential learners will get the information they need in other ways rather than waiting for the class. At Google, this is particularly a problem for small, geographically remote offices.

Documentation

Documentation is written knowledge whose primary goal is to help its readers learn something. Not all written knowledge is necessarily documentation, although it can be useful as a paper trail. For example, it's possible to find an answer to a problem in a mailing list thread, but the primary goal of the original question on the thread was to seek answers, and only secondarily to document the discussion for others.

In this section, we focus on spotting opportunities for contributing to and creating formal documentation, from small things like fixing a typo to larger efforts such as documenting tribal knowledge.

NOTE

For a more comprehensive discussion of documentation, see Chapter 10.

UPDATING DOCUMENTATION

The first time you learn something is the best time to see ways that the existing documentation and training materials can be improved. By the time you've absorbed and understood a new process or system, you might have forgotten what was difficult or what simple steps were missing from the "Getting Started" documentation. At this stage, if you find a mistake or omission in the documentation, fix it! Leave the campground cleaner than you found it,[9] and try to update the documents yourself, even when that documentation is owned by a different part of the organization.

At Google, engineers feel empowered to update documentation regardless of who owns it—and we often do—even if the fix is as small as fixing a typo.

This level of community upkeep increased notably with the introduction of g3doc,[10] which made it much easier for Googlers to find a documentation owner to review their suggestion. It also leaves an auditable trail of change history no different than that for code.

CREATING DOCUMENTATION

As your proficiency grows, write your own documentation and update existing docs. For example, if you set up a new development flow, document the steps. You can then make it easier for others to follow in your path by pointing them at your document. Even better, make it easier for people to find the document themselves. Any sufficiently undiscoverable or unsearchable documentation might as well not exist. This is another area in which g3doc shines because the documentation is predictably located right next to the source code, as opposed to off in an (unfindable) document or webpage somewhere.

Finally, make sure there's a mechanism for feedback. If there's no easy and direct way for readers to indicate that documentation is outdated or inaccurate, they are likely not to bother telling anyone, and the next newcomer will come across the same problem. People are more willing to contribute changes if they feel that someone will actually notice and consider their suggestions. At Google, you can file a documentation bug directly from the document itself.

In addition, Googlers can easily leave comments on g3doc pages. Other Googlers can see and respond to these comments and, because leaving a comment automatically files a bug for the documentation owner, the reader doesn't need to figure out who to contact.

PROMOTING DOCUMENTATION

Traditionally, encouraging engineers to document their work can be difficult. Writing documentation takes time and effort that could be spent on coding, and the benefits that result from that work are not immediate and are mostly reaped by others. Asymmetrical trade-offs like these are good for the organization as a whole given that many people can benefit from the time investment of a few, but without good incentives, it can be challenging to encourage such behavior. We discuss some of these structural incentives in the section "Incentives and Recognition".

However, a document author can often directly benefit from writing documentation. Suppose that team members always ask you for help

debugging certain kinds of production failures. Documenting your procedures requires an upfront investment of time, but after that work is done, you can save time in the future by pointing team members to the documentation and providing hands-on help only when needed.

Writing documentation also helps your team and organization scale. First, the information in the documentation becomes canonicalized as a reference: team members can refer to the shared document and even update it themselves. Second, the canonicalization may spread outside the team. Perhaps some parts of the documentation are not unique to the team's configuration and become useful for other teams looking to resolve similar problems.

Code

At a meta level, code *is* knowledge, so the very act of writing code can be considered a form of knowledge transcription. Although knowledge sharing might not be a direct intent of production code, it is often an emergent side effect, which can be facilitated by code readability and clarity.

Code documentation is one way to share knowledge; clear documentation not only benefits consumers of the library, but also future maintainers. Similarly, implementation comments transmit knowledge across time: you're writing these comments expressly for the sake of future readers (including Future You!). In terms of trade-offs, code comments are subject to the same downsides as general documentation: they need to be actively maintained or they can quickly become out of date, as anyone who has ever read a comment that directly contradicts the code can attest.

Code reviews (see Chapter 9) are often a learning opportunity for both author and reviewer(s). For example, a reviewer's suggestion might introduce the author to a new testing pattern, or a reviewer might learn of a new library by seeing the author use it in their code. Google standardizes mentoring through code review with the *readability process*, as detailed in the case study at the end of this chapter.

Scaling Your Organization's Knowledge

Ensuring that expertise is appropriately shared across the organization becomes more difficult as the organization grows. Some things, like culture, are important at every stage of growth, whereas others, like establishing canonical sources of information, might be more beneficial for more mature organizations.

Cultivating a Knowledge-Sharing Culture

Organizational culture is the squishy human thing that many companies treat as an afterthought. But at Google, we believe that focusing on the culture and environment first,[11] results in better outcomes than focusing on only the output—such as the code—of that environment.

Making major organizational shifts is difficult, and countless books have been written on the topic. We don't pretend to have all the answers, but we can share specific steps Google has taken to create a culture that promotes learning.

See the book *Work Rules!*[12] for a more in-depth examination of Google's culture.

RESPECT

The bad behavior of just a few individuals can make an entire team or community unwelcoming.[13] In such an environment, novices learn to take their questions elsewhere, and potential new experts stop trying and don't have room to grow. In the worst cases, the group reduces to its most toxic members. It can be difficult to recover from this state.

Knowledge sharing can and should be done with kindness and respect. In tech, tolerance—or worse, reverence—of the "brilliant jerk" is both pervasive and harmful, but being an expert and being kind are not mutually exclusive. The Leadership section of Google's software engineering job ladder outlines this clearly:

Although a measure of technical leadership is expected at higher levels, not all leadership is directed at technical problems. Leaders improve the quality of the people around them, improve the team's psychological safety, create a culture of teamwork and collaboration, defuse tensions within the team, set an example of Google's culture and values, and make Google a more vibrant and exciting place to work. Jerks are not good leaders.

This expectation is modeled by senior leadership: Urs Hölzle (senior vice president of technical infrastructure) and Ben Treynor Sloss (vice president, founder of Google SRE) wrote a regularly cited internal document ("No Jerks") about why Googlers should care about respectful behavior at work and what to do about it.

INCENTIVES AND RECOGNITION

Good culture must be actively nurtured, and encouraging a culture of knowledge sharing requires a commitment to recognizing and rewarding it at a systemic level. It's a common mistake for organizations to pay lip service to a set of values while actively rewarding behavior that does not enforce those values. People react to incentives over platitudes, and so it's important to put your money where your mouth is by putting in place a system of compensation and awards.

Google uses a variety of recognition mechanisms, from company-wide standards such as performance review and promotion criteria to peer-to-peer awards between Googlers.

Our software engineering ladder, which we use to calibrate rewards like compensation and promotion across the company, encourages engineers to share knowledge by noting these expectations explicitly. At more senior levels, the ladder explicitly calls out the importance of wider influence, and this expectation increases as seniority increases. At the highest levels, examples of leadership include the following:

- Growing future leaders by serving as mentors to junior staff, helping them develop both technically and in their Google role
- Sustaining and developing the software community at Google via code and design reviews, engineering education and development, and expert guidance to others in the field

See Chapter 5 and Chapter 6 for more on leadership.

Job ladder expectations are a top-down way to direct a culture, but culture is also formed from the bottom up. At Google, the peer bonus program is one way we embrace the bottom-up culture. Peer bonuses are a monetary award and formal recognition that any Googler can bestow on any other Googler for above-and-beyond work.[14] For example, when Ravi sends a peer bonus to Julia for being a top contributor to a mailing list—regularly answering questions that benefit many readers—he is publicly recognizing her knowledge-sharing work and its impact beyond her team. Because peer bonuses are employee driven, not management driven, they can have an important and powerful grassroots effect.

Similar to peer bonuses are kudos: public acknowledgement of contributions (typically smaller in impact or effort than those meriting a peer bonus) that boost the visibility of peer-to-peer contributions.

When a Googler gives another Googler a peer bonus or kudos, they can choose to copy additional groups or individuals on the award email, boosting recognition of the peer's work. It's also common for the recipient's manager to forward the award email to the team to celebrate each other's achievements.

A system in which people can formally and easily recognize their peers is a powerful tool for encouraging peers to keep doing the awesome things they do. It's not the bonus that matters: it's the peer acknowledgement.

Establishing Canonical Sources of Information

Canonical sources of information are centralized, company-wide corpuses of information that provide a way to standardize and propagate expert knowledge. They work best for information that is relevant to all engineers within the organization, which is otherwise prone to information islands. For example, a guide to setting up a basic developer workflow should be made canonical, whereas a guide for running a local Frobber instance is more relevant just to the engineers working on Frobber.

Establishing canonical sources of information requires higher investment than maintaining more localized information such as team documentation, but it also has broader benefits. Providing centralized references for the entire organization makes broadly required information easier and more predictable to find, and counters problems with information fragmentation that can arise when multiple teams grappling with similar problems produce their own—often conflicting—guides.

Because canonical information is highly visible and intended to provide a shared understanding at the organizational level, it's important that the content is actively maintained and vetted by subject matter experts. The more complex a topic, the more critical it is that canonical content has explicit owners. Well-meaning readers might see that something is out of date but lack the expertise to make the significant structural changes needed to fix it, even if tooling makes it easy to suggest updates.

Creating and maintaining centralized, canonical sources of information is expensive and time-consuming, and not all content needs to be shared at an organizational level. When considering how much effort to invest in this resource, consider your audience. Who benefits from this information? You? Your team? Your product area? All engineers?

DEVELOPER GUIDES

Google has a broad and deep set of official guidance for engineers, including style guides, official software engineering best practices,[15] guides for code review[16] and testing,[17] and Tips of the Week (TotW).[18]

The corpus of information is so large that it's impractical to expect engineers to read it all end to end, much less be able to absorb so much information at once. Instead, a human expert already familiar with a guideline can send a link to a fellow engineer, who then can read the reference and learn more. The expert saves time by not needing to personally explain a company-wide practice, and the learner now knows that there is a canonical source of trustworthy information that they can access whenever necessary. Such a process scales knowledge because it enables human experts to recognize and solve a specific information need by leveraging common, scalable resources.

GO/ LINKS

go/ links (sometimes referred to as goto/ links) are Google's internal URL shortener.[19] Most Google-internal references have at least one internal go/ link. For example, "go/spanner" provides information about Spanner, "go/python" is Google's Python developer guide. The content can live in any repository (g3doc, Google Drive, Google Sites, etc.) but having a go/ link that points to it provides a predictable, memorable way to access it. This yields some nice benefits:

- go/ links are so short that it's easy to share them in conversation ("You should check out go/frobber!"). This is much easier than having to go find a link and then send a message to all interested parties. Having a low-friction way to share references makes it more likely that that knowledge will be shared in the first place.
- go/ links provide a permalink to the content, even if the underlying URL changes. When an owner moves content to a different repository (for example, moving content from a Google doc to g3doc), they can simply update the go/ link's target URL. The go/ link itself remains unchanged.

go/ links are so ingrained into Google culture that a virtuous cycle has emerged: a Googler looking for information about Frobber will likely first check go/frobber. If the go/ link doesn't point to the Frobber Developer Guide (as expected), the Googler will generally configure the link

themselves. As a result, Googlers can usually guess the correct go/ link on the first try.

CODELABS

Google codelabs are guided, hands-on tutorials that teach engineers new concepts or processes by combining explanations, working best-practice example code, and code exercises.[20] A canonical collection of codelabs for technologies broadly used across Google is available at go/codelab. These codelabs go through several rounds of formal review and testing before publication. Codelabs are an interesting halfway point between static documentation and instructor-led classes, and they share the best and worst features of each. Their hands-on nature makes them more engaging than traditional documentation, but engineers can still access them on demand and complete them on their own; but they are expensive to maintain and are not tailored to the learner's specific needs.

STATIC ANALYSIS

Static analysis tools are a powerful way to share best practices that can be checked programmatically. Every programming language has its own particular static analysis tools, but they have the same general purpose: to alert code authors and reviewers to ways in which code can be improved to follow style and best practices. Some tools go one step further and offer to automatically apply those improvements to the code.

NOTE

See Chapter 20 for details on static analysis tools and how they're used at Google.

Setting up static analysis tools requires an upfront investment, but as soon as they are in place, they scale efficiently. When a check for a best practice is added to a tool, every engineer using that tool becomes aware of that best practice. This also frees up engineers to teach other things: the time and effort that would have gone into manually teaching the (now automated) best practice can instead be used to teach something else. Static analysis tools augment engineers' knowledge. They enable an organization to apply more best practices and apply them more consistently than would otherwise be possible.

Staying in the Loop

Some information is critical to do one's job, such as knowing how to do a typical development workflow. Other information, such as updates on popular productivity tools, is less critical but still useful. For this type of knowledge, the formality of the information sharing medium depends on the importance of the information being delivered. For example, users expect official documentation to be kept up to date, but typically have no such expectation for newsletter content, which therefore requires less maintenance and upkeep from the owner.

NEWSLETTERS

Google has a number of company-wide newsletters that are sent to all engineers, including *EngNews* (engineering news), *Ownd* (Privacy/Security news), and *Google's Greatest Hits* (report of the most interesting outages of the quarter). These are a good way to communicate information that is of interest to engineers but isn't mission critical. For this type of update, we've found that newsletters get better engagement when they are sent less frequently and contain more useful, interesting content. Otherwise, newsletters can be perceived as spam.

Even though most Google newsletters are sent via email, some are more creative in their distribution. *Testing on the Toilet* (testing tips) and *Learning on the Loo* (productivity tips) are single-page newsletters posted inside toilet stalls. This unique delivery medium helps the *Testing on the Toilet* and *Learning on the Loo* stand out from other newsletters, and all issues are archived online.

NOTE

See Chapter 11 for a history of how *Testing on the Toilet* came to be.

COMMUNITIES

Googlers like to form cross-organizational communities around various topics to share knowledge. These open channels make it easier to learn from others outside your immediate circle and avoid information islands and duplication. Google Groups are especially popular: Google has thousands of internal groups with varying levels of formality. Some are dedicated to troubleshooting; others, like the Code Health group, are more for discussion and guidance. Internal Google+ is also popular among Googlers as a source

of informal information because people will post interesting technical breakdowns or details about projects they are working on.

Readability: Standardized Mentorship Through Code Review

At Google, "readability" refers to more than just code readability; it is a standardized, Google-wide mentorship process for disseminating programming language best practices. Readability covers a wide breadth of expertise, including but not limited to language idioms, code structure, API design, appropriate use of common libraries, documentation, and test coverage.

Readability started as a one-person effort. In Google's early days, Craig Silverstein (employee ID #3) would sit down in person with every new hire and do a line-by-line "readability review" of their first major code commit. It was a nitpicky review that covered everything from ways the code could be improved to whitespace conventions. This gave Google's codebase a uniform appearance but, more important, it taught best practices, highlighted what shared infrastructure was available, and showed new hires what it's like to write code at Google.

Inevitably, Google's hiring rate grew beyond what one person could keep up with. So many engineers found the process valuable that they volunteered their own time to scale the program. Today 25 to 30% of Google engineers are participating in the readability process at any given time, as either reviewers or code authors.

What Is the Readability Process?

Code review is mandatory at Google. Every changelist (CL)[21] requires *readability approval*, which indicates that someone who has *readability certification* for that language has approved the CL. Certified authors implicitly provide readability approval of their own CLs; otherwise, one or more qualified reviewers must explicitly give readability approval for the CL. This requirement was added after Google grew to a point where it was no longer possible to enforce that every engineer received code reviews that taught best practices to the desired rigor.

NOTE

See Chapter 9 for an overview of the Google code review process and what Approval means in this context.

Within Google, having readability certification is commonly referred to as "having readability" for a language. Engineers with readability have demonstrated that they consistently write clear, idiomatic, and maintainable code that exemplifies Google's best practices and coding style for a given language. They do this by submitting CLs through the readability process, during which a centralized group of *readability reviewers* review the CLs and give feedback on how much it demonstrates the various areas of mastery. As authors internalize the readability guidelines, they receive fewer and fewer comments on their CLs until they eventually graduate from the process and formally receive readability. Readability brings increased responsibility: engineers with readability are trusted to continue to apply their knowledge to their own code and to act as reviewers for other engineers' code.

Around one to two percent of Google engineers are readability reviewers. All reviewers are volunteers, and anyone with readability is welcome to self-nominate to become a readability reviewer. Readability reviewers are held to the highest standards. Because they are expected not just to have deep language expertise but also an aptitude for teaching through code review. They are expected to treat readability as first and foremost a mentoring and cooperative process, not a gatekeeping or adversarial one. Readability reviewers and CL authors alike are encouraged to have discussions during the review process. Reviewers provide relevant citations for their comments so that authors can learn about the rationales that went into the style guidelines ("Chesterson's fence"). If the rationale for any given guideline is unclear, authors should ask for clarification ("ask questions").

Readability is deliberately a human-driven process that aims to scale knowledge in a standardized yet personalized way. As a complementary blend of written and tribal knowledge, readability combines the advantages of written documentation, which can be accessed with citable references, with the advantages of expert human reviewers, who know which guidelines to cite. Canonical guidelines and language recommendations are comprehensively documented—which is good!—but the corpus of information is so large[22] that it can be overwhelming, especially to newcomers.

Why Have This Process?

Code is read far more than it is written, and this effect is magnified at Google's scale and in our (very large) monorepo.[23] Any engineer can look at and learn from the wealth of knowledge that is the code of other teams, and powerful tools like Kythe make it easy to find references throughout the entire codebase (see Chapter 17). An important feature of documented best practices (see Chapter 8) is that they provide consistent standards for all Google code to follow. Readability is both an enforcement and propagation mechanism for these standards.

One of the primary advantages of the readability program is that it exposes engineers to more than just their own team's tribal knowledge. To earn readability in a given language, engineers must send CLs through a centralized set of readability reviewers who review code across the entire company. Centralizing the process makes a significant trade-off: the program is limited to scaling linearly rather than sublinearly with organization growth, but it makes it easier to enforce consistency, avoid islands, and (often unintentional) drifting from established norms.

The value of codebase-wide consistency cannot be overstated: even with tens of thousands of engineers writing code over decades, it ensures that code in a given language will look similar across the corpus. This enables readers to focus on what the code does rather than being distracted by why it looks different than code that they're used to. Large-scale change authors (see Chapter 22) can more easily make changes across the entire monorepo, crossing the boundaries of thousands of teams. People can change teams and be confident the way that the new team uses a given language is not drastically different than their previous team.

These benefits come with some costs: readability is a heavyweight process compared to other mediums like documentation and classes because it is mandatory and enforced by Google tooling (see Chapter 19). These costs are nontrivial and include the following:

- Increased friction for teams that do not have any team members with readability, because they need to find reviewers from outside their team to give readability approval on CLs.
- Potential for additional rounds of code review for authors who need readability review.

- Scaling disadvantages of being a human-driven process. Limited to scaling linearly to organization growth because it depends on human reviewers doing specialized code reviews.

The question, then, is whether the benefits outweigh the costs. There's also the factor of time: the full effect of the benefits versus the costs are not on the same timescale. The program makes a deliberate trade-off of increased short-term code-review latency and upfront costs for the long-term payoffs of higher-quality code, repository-wide code consistency, and increased engineer expertise. The longer timescale of the benefits comes with the expectation that code is written with a potential lifetime of years, if not decades.[24]

As with most—or perhaps all—engineering processes, there's always room for improvement. Some of the costs can be mitigated with tooling. A number of readability comments address issues that could be detected statically and commented on automatically by static analysis tooling. As we continue to invest in static analysis, readability reviewers can increasingly focus on higher-order areas, like whether a particular block of code is understandable by outside readers who are not intimately familiar with the codebase instead of automatable detections like whether a line has trailing whitespace.

But aspirations aren't enough. Readability is a controversial program: some engineers complain that it's an unnecessary bureaucratic hurdle and a poor use of engineer time. Are readability's trade-offs worthwhile? For the answer, we turned to our trusty Engineering Productivity Research (EPR) team.

The EPR team performed in-depth studies of readability, including but not limited to whether people were hindered by the process, learned anything, or changed their behavior after graduating. These studies showed that readability has a net positive impact on engineering velocity. CLs by authors with readability take statistically significantly less time to review and submit than CLs by authors who do not have readability.[25] Self-reported engineer satisfaction with their code quality—lacking more objective measures for code quality—is higher among engineers who have readability versus those who do not. A significant majority of engineers who complete the program report satisfaction with the process and find it worthwhile. They report learning from reviewers and changing their own behavior to avoid readability issues when writing *and* reviewing code.

NOTE

For an in-depth look at this study and Google's internal engineering productivity research, see XREF(Measuring Success and Failure).

Google has a very strong culture of code review, and readability is a natural extension of that culture. Readability grew from the passion of a single engineer to a formal program of human experts mentoring all Google engineers. It evolved and changed with Google's growth, and it will continue to evolve as Google's needs change.

Conclusion

Knowledge is in some ways the most important (though intangible) capital of a software engineering organization, and sharing of that knowledge is crucial for making an organization resilient and redundant in the face of change. A culture that promotes open and honest knowledge sharing distributes that knowledge efficiently across the organization and allows that organization to scale over time. In most cases, investments into easier knowledge sharing reap manyfold dividends over the life of a company.

TL;DRs

- *Psychological safety* is the foundation for fostering a knowledge-sharing environment.
- Start small: *ask questions* and *write things down*.
- Make it easy for people to get the help they need from both *human experts* and *documented references*.
- At a systemic level, *encourage and reward* those who take time to teach and broaden their expertise beyond just themselves, their team, or their organization.
- There is no silver bullet: empowering a knowledge-sharing culture requires a *combination of multiple strategies*, and the exact mix that works best for your organization will likely change over time.

1 In other words, rather than developing a single global maximum, we have a bunch of local maxima. *https://en.wikipedia.org/wiki/Maxima_and_minima*

2 Parnas, David Lorge. 2011. *Software engineering: multi-person development of multi-version programs*. Heidelberg, Germany: Springer-Verlag Berlin.

3 Impostor syndrome is not uncommon among high achievers, and Googlers are no exception—in fact, a majority of this book's authors have impostor syndrome. We acknowledge that fear of failure can be difficult for those with impostor syndrome and can reinforce an inclination to avoid branching out.

4 See "How to ask good questions"

5 *https://en.wikipedia.org/wiki/Wikipedia:Chesterton%27s_fence*

6 *https://xkcd.com/979/*

7 *https://talksat.withgoogle.com/* and *https://www.youtube.com/GoogleTechTalks*, to name a few.

8 The g2g program is detailed in: Bock, Laszlo. 2015. *Work Rules!: Insights from Inside Google That Will Transform How You Live and Lead*. New York: Twelve Books. It includes descriptions of different aspects of the program as well as how to evaluate impact and recommendations for what to focus on when setting up similar programs.

9 Martin, Robert C. 2010. "The Boy Scout Rule," In Henney, Kevin. "*97 Things Every Programmer Should Know*" O'Reilly Media, Inc.

10 g3doc stands for "google3 documentation." google3 is the name of the current incarnation of Google's monolithic source repository.

11 As referenced in: Bock, Laszlo. 2015. *Work Rules!: Insights from Inside Google That Will Transform How You Live and Lead*. New York: Twelve Books, culture beats strategy every time [Mer11]

12 Ibid.

13 *https://hbr.org/ideacast/2013/01/the-high-cost-of-rudeness-at-w.html*

14 Peer bonuses include a cash award and a certificate as well as being a permanent part of a Googler's award record in an internal tool called gThanks.

15 Such as books about software engineering at Google.

16 See Chapter 9.

17 See Chapter 11.

18 Available for multiple languages. Externally available for C++ at *https://abseil.io/tips/*.

19 go/ links are unrelated to the Go language.

20 External codelabs are available at *https://codelabs.developers.google.com/*.

21 A *changelist* is a list of files that make up a change in a version control system. A changelist is synonymous with a *changeset*.

22 As of 2019, just the Google C++ style guide is 40 pages long. The secondary material making up the complete corpus of best practices is many times longer.

23 For why Google uses a monorepo, see *https://cacm.acm.org/magazines/2016/7/204032-why-google-stores-billions-of-lines-of-code-in-a-single-repository/fulltext*. Note also that not all of Google's code lives within the monorepo; readability as described here applies only to the monorepo because it is a notion of within-repository consistency.

24 For this reason, code that is known to have a short time span is exempt from readability requirements. Examples include the *experimental/* directory (explicitly designated for experimental code and cannot push to production) and the Area 120 program, a workshop for Google's experimental products).

25 This includes controlling for a variety of factors, including tenure at Google and the fact that CLs for authors who do not have readability typically need additional rounds of review compared to authors who already have readability.

Chapter 4. Engineering for Equity

Written by Demma Rodriguez

Edited by Riona MacNamara

In earlier chapters, we've explored the contrast between *programming* as the production of code that addresses the problem of the moment, and *software engineering* as the broader application of code, tools, policies and processes to a dynamic and ambiguous problem that can span decades or even lifetimes. In this chapter, we'll discuss the unique responsibilities of an engineer when designing products for a broad base of users. Further, we evaluate how an organization, by embracing diversity, can design systems that work for everyone, and avoid perpetuating harm against our users.

As new as the field of software engineering is, we're newer still at understanding the impact it has on underrepresented people and diverse societies. We did not write this chapter because we know all the answers. We do not. In fact, understanding how to engineer products that empower and respect all our users is still something Google is learning to do. We have had many public failures in protecting our most vulnerable users, and so we are writing this chapter because the path forward to more equitable products begins with evaluating our own failures and encouraging growth.

We are also writing this chapter because the increasing imbalance of power between those who make development decisions that impact the world and those who simply must accept and live with those decisions that sometimes disadvantages already marginalized communities globally. It is important to share and reflect on what we've learned so far with the next generation of software engineers. It is even more important that we help influence the next generation of engineers to be better than we are today.

Just picking up this book means that you likely aspire to be an exceptional engineer. You want to solve problems. You aspire to build products that drive positive outcomes for the broadest base of people, including people who are the most difficult to reach. To do this, you will need to consider how the tools you build will be leveraged to change the trajectory of humanity, hopefully for the better.

Bias Is the Default

When engineers do not focus on users of different nationalities, ethnicities, races, genders, ages, socioeconomic statuses, abilities, and belief systems, even the most talented staff will inadvertently fail their users. Such failures are often intentional; all people have certain biases, and social scientists have recognized over the past several decades that most people exhibit unconscious bias, enforcing and promulgating existing stereotypes. Unconscious bias is insidious and often more difficult to mitigate than intentional acts of exclusion. Even when we want to do the right thing, we might not recognize our own biases. By the same token, our organizations must also recognize that such bias exists and work to address it in its workforce, product development, and user outreach.

Because of bias, Google has at times failed to represent users equitably within their products, with launches over the past several years that did not focus enough on underrepresented groups. Many users attribute our lack of awareness in these cases to the fact that our engineering population is mostly male, mostly White or Asian, and certainly not representative of all the communities that use our products. The lack of representation of such users in our workforce[1] means that we often do not have the requisite diversity to understand how the use of our products can affect underrepresented or vulnerable users.

Google Misses the Mark on Racial Inclusion

In 2015, software engineer Jacky Alciné pointed out[2] that the image recognition algorithms in Google Photos were classifying his black friends as "gorillas." Google was slow to respond to these mistakes and incomplete in addressing them.

What caused such a monumental failure? Several things:

- Image recognition algorithms depend on being supplied a "proper" (often meaning "complete") dataset. The photo data fed into Google's image recognition algorithm was clearly incomplete. In short, the data did not represent the population.
- Google itself (and the tech industry in general) did not (and does not) have much black representation[3] and that affects decisions subjective in the design of such algorithms and the collection of such datasets. The unconscious bias of the organization itself likely led to a more representative product being left on the table.

- Google's target market for image recognition did not adequately include such underrepresented groups. Google's tests did not catch these mistakes; as a result, our users did, which both embarrassed Google and harmed our users.

As late as 2018, Google still had not adequately addressed the underlying problem.[4]

In this example, our product was inadequately designed and executed, failing to properly consider all racial groups, and as a result, failed our users and caused Google bad press. Other technology suffers from similar failures: autocomplete can return offensive or racist results. Google's Ad system could be manipulated to show racist or offensive ads. YouTube might not catch hate speech, though it is technically outlawed on that platform.

In all of these cases, the technology itself is not really to blame. Autocomplete, for example, was not designed to target users or to discriminate. But it was also not resilient enough in its design to exclude discriminatory language that is considered hate speech. As a result, the algorithm returned results that caused harm to our users. The harm to Google itself should also be obvious: reduced user trust and engagement with the company. For example, Black, Latinx, and Jewish applicants could lose faith in Google as a platform or even as an inclusive environment itself, therefore undermining Google's goal of improving representation in hiring.

How could this happen? After all, Google hires technologists with impeccable education and/or professional experience; exceptional programmers, who write the best code and test their work. "Build for everyone" is a Google brand statement, but the truth is that we still have a long way to go before we can claim that we do. One way to address these problems is to help the software engineering organization itself look like the populations for whom we build products.

Understanding the Need for Diversity

At Google, we believe that being an exceptional engineer requires that you also focus on bringing diverse perspectives into product design and implementation. It also means that Googlers responsible for hiring or interviewing other engineers must contribute to building a more representative workforce. For example, if you interview other engineers for positions at your company, it is important to learn how biased outcomes happen in hiring. There are significant prerequisites for understanding how to

anticipate harm and prevent it. To get to the point where we can build for everyone, we first must understand our representative populations. We need to encourage engineers to have a wider scope of educational training.

The first order of business is to disrupt the notion that as a person with a computer science degree and/or work experience, you have all the skills you need to become an exceptional engineer. A computer science degree is often a necessary foundation. However, the degree alone (even when coupled with work experience) will not make you an engineer. It is also important to disrupt the idea that only people with computer science degrees can design and build products. Today, most programmers do have a computer science degree; they are successful at building code, establishing theories of change, and methodologies for problem solving. *However, as the aforementioned examples demonstrate, this approach is insufficient for inclusive and equitable engineering.*

Engineers should begin by focusing all work within the framing of the complete ecosystem they seek to influence. At minimum, they need to understand the population demographics of their users. Engineers should focus on people who are different than themselves, especially people who might attempt to use our products to cause harm. The most difficult users to consider are those who are disenfranchised by the processes and the environment in which they access technology. To address this challenge, engineering teams need to be representative of their existing and future users. In the absence of diverse representation on engineering teams, individual engineers need to learn how to build for all users.

Building Multicultural Capacity

One mark of an exceptional engineer is the ability to understand how products can advantage and disadvantage different groups of human beings. Engineers are expected to have technical aptitude, but they should also have the *discernment* to know when to build something and when not to. Discernment includes building the capacity to identify and reject features or products that drive adverse outcomes. This is a lofty and difficult goal, because there is an enormous amount of individualism that goes into being a high-performing engineer. Yet to succeed, we must extend our focus beyond our own communities to the next billion users or to current users who might be disenfranchised or left behind by our products.

Over time, you might build tools that billions of people use daily—tools that influence how people think about the value of human lives, tools that monitor human activity, and tools that capture and persist sensitive data, such as images of their children and loved ones as well as other types of sensitive data. As an engineer, you might wield more power than you realize: the power to literally change society. It's critical that on your journey to becoming an exceptional engineer, you understand the innate responsibility needed to exercise power without causing harm. The first step is to recognize the default state of your bias caused by many societal and educational factors. After you recognize this, you'll be able to consider the often-forgotten use cases or users who can benefit or be harmed by the products you build.

The industry continues to move forward, building new use cases for artificial intelligence (AI) and machine learning at an ever-increasing speed. To stay competitive, we drive toward scale and efficacy in building a high-talent engineering and technology workforce. Yet we need to pause and consider the fact that today some people have the ability to design the future of technology and others do not. We need to understand whether the software systems we build will eliminate the potential for entire populations to experience shared prosperity and provide equal access to technology.

Historically, companies faced with a decision between completing a strategic objective that drives market dominance and revenue and one that potentially slows momentum toward that goal, have opted for speed and shareholder value. This tendency is exacerbated by the fact that many companies value individual performance and excellence, yet often fail to effectively drive accountability on product equity across all areas. Focusing on underrepresented users is a clear opportunity to promote equity. To continue to be competitive in the technology sector, we need to learn to engineer for global equity.

Today, we worry when companies design technology to scan, capture, and identify people walking down the street. We worry about privacy and how governments might use this information now and in the future. Yet most technologists do not have the requisite perspective of underrepresented groups to understand the impact of racial variance in facial recognition or to understand how applying AI can drive harmful and inaccurate results.

Currently, AI-driven facial-recognition software continues to disadvantage people of color or ethnic minorities. Our research is not comprehensive enough and does not include a wide enough range of different skin tones. We cannot expect the output to be valid if both the training data and those

creating the software represent only a small subsection of people. In those cases, we should be willing to delay development in favor of trying to get more complete and accurate data, and a more comprehensive and inclusive product.

Data science itself is challenging for humans to evaluate, however. Even when we do have representation, a training set can still be biased and produce invalid results. A study completed in 2016 found that more than 117 million American adults are in a law enforcement facial recognition database.[5] Due to the disproportionate policing of Black communities, and disparate outcomes in arrests, there could be racially biased error rates in utilizing such a database in facial recognition. Although the software is being developed and deployed at ever-increasing rates, the independent testing is not. To correct for this egregious misstep, we need to have the integrity to slow down and ensure that our inputs contain as little bias as possible. Google now offers statistical training within the context of AI, to help ensure that datasets are not intrinsically biased.

Therefore, shifting the focus of your industry experience to include more comprehensive, multicultural, race and gender studies education is not only *your* responsibility, but also the *responsibility of your employer*. Technology companies must ensure that their employees are continually receiving professional development, and that this development is comprehensive and multidisciplinary. The requirement is not that one individual take it upon themselves to learn about other cultures or other demographics alone. Change requires that each of us, individually or as leaders of teams, invest in continuous professional development that builds not just our software development and leadership skills, but also our capacity to understand the diverse experiences throughout humanity.

Making Diversity Actionable

Systemic equity and fairness are attainable, if we are willing to accept that we are all accountable for the systemic discrimination we see in the technology sector. We are accountable for the failures in the system. Deferring or abstracting away personal accountability is ineffective, and depending on your role, it could be irresponsible. It is also irresponsible to fully attribute dynamics at your specific company or within your team to the larger societal issues that contribute to inequity. A favorite line among diversity proponents and detractors alike goes something like this: “We are working hard to fix (insert systemic discrimination topic), but accountability is hard. How do we

combat (insert hundreds of years) of historical discrimination?" This line of inquiry is a detour to a more philosophical or academic conversation and away from focused efforts to improve work conditions or outcomes. Part of building multicultural capacity requires a more comprehensive understanding of how systems of inequality in society impact the workplace, especially in the technology sector.

If you are an engineering manager working on hiring more people from underrepresented groups, deferring to the historical impact of discrimination in the world is a useful academic exercise. However, it is critical to move beyond the academic conversation to a focus on quantifiable and actionable steps that you can take to drive equity and fairness. For example, as a hiring software engineer manager, you're accountable for ensuring that your candidate slates are balanced. Are there women or other underrepresented groups in the pool of candidates' reviews? After you hire someone, what opportunities for growth have you provided, and is the distribution of opportunities equitable? Every technology lead or software engineering manager has the means to augment equity on their teams. It is important that we acknowledge that although there are significant systemic challenges, we are all part of the system. It is our problem to fix.

Reject Singular Approaches

We cannot perpetuate solutions that present a single philosophy or methodology for fixing inequity in the technology sector. Our problems are complex and multifactorial. Therefore, we must disrupt singular approaches to advancing representation in the workplace, even if they are promoted by people we admire or who have institutional power.

One singular narrative held dear in the technology industry is that lack of representation in the workforce can be addressed solely by fixing the hiring pipelines. Yes, that is a fundamental step, but that is not the immediate issue we need to fix. We need to recognize systemic inequity in progression and retention while simultaneously focusing on more representative hiring and educational disparities across lines of race, gender, socioeconomic, and immigration status, for example.

In the technology industry, many people from underrepresented groups are daily passed over for opportunities and advancement. Attrition among Black+ Google employees outpaces attrition from all other groups[6] and confounds progress on representation goals. If we want to drive change and increase

representation, we need to evaluate whether we're creating an ecosystem in which all aspiring engineers and other technology professionals can thrive.

Fully understanding an entire problem space is critical to determining how to fix it. This holds true for everything from a critical data migration to the hiring of a representative workforce. For example, if you are an engineering manager who wants to hire more women, don't just focus on building a pipeline. Focus on other aspects of the hiring, retention, and progression ecosystem, and how inclusive it might or might not be to women. Consider whether your recruiters are demonstrating the ability to identify strong candidates who are women as well as men. If you manage a diverse engineering team, focus on psychological safety and invest in increasing multicultural capacity on the team so that new team members feel welcome.

A common methodology today is to build for the majority use-case first, leaving improvements and features that address edge-cases for later. But this approach is flawed; it gives users who are already advantaged in access to technology a head start, which increases inequity. Relegating the consideration of all user groups to the point when design has been nearly completed is to lower the bar of what it means to be an excellent engineer. Instead, by building in inclusive design from the start and raising development standards for development to make tools delightful and accessible for people who struggle to access technology, we enhance the experience for *all* users.

Designing for the user who is least like you is not just wise, it's a best practice. There are pragmatic and immediate next steps that all technologists, regardless of domain, should consider when developing products that avoid disadvantaging or underrepresenting users. It begins with more comprehensive user-experience research. This research should be done with user groups that are multilingual and multicultural, and that span multiple countries, socioeconomic class, abilities, and age ranges. Focus on the most difficult or least represented use case first.

Challenge Established Processes

Challenging yourself to build more equitable systems goes beyond designing more inclusive product specifications. Building equitable systems sometimes means challenging established processes that drive invalid results.

Consider a recent case evaluated for equity implications. At Google, several engineering teams worked to build a global hiring requisition system. The

system supports both external hiring and internal mobility. The engineers and product managers involved did a great job of listening to the requests of what they considered to be their core user group: recruiters. The recruiters were focused on minimizing wasted time for hiring managers and applicants, and presented the development team with use cases focused on scale and efficiency for those people. To drive efficiency, the recruiters asked the engineering team to include a feature that would highlight performance ratings—specifically lower ratings—to the hiring manager and recruiter as soon as an internal transfer expressed interest in a job.

On its face, expediting the evaluation process and helping jobseekers save time is a great goal. So where is the potential equity concern? The following equity questions were raised:

- Are developmental assessments a predictive measure of performance?
- Are the performance assessments being presented to prospective managers free of individual bias?
- Are performance assessments scores standardized across organizations?

If the answer to any of these questions is "No," presenting performance ratings could still drive inequitable, and therefore invalid, results.

When an exceptional engineer questioned whether past performance was in fact predictive of future performance, the reviewing team decided to conduct a thorough review. In the end, it was determined that candidates who had received a poor performance rating were likely to overcome the poor rating if they found a new team. In fact, they were just as likely to receive a satisfactory or exemplary performance rating as candidates who had never received a poor rating. In short, performance ratings are indicative only of how a person is performing in their given role *at the time they are being evaluated*. Ratings, although an important way to measure performance during a specific period, are not predictive of future performance, and should not be used to gauge readiness for a future role or qualify an internal candidate for a different team. (They can, however, be used to evaluate whether an employee is properly or improperly slotted on their current team; therefore, they can provide an opportunity to evaluate how to better support an internal candidate moving forward.)

This analysis definitely took up significant project time, but the positive trade-off was a more equitable internal mobility process.

Values versus Outcomes

Google has a strong track record of investing in hiring. As the previous example illustrates, we also continually evaluate our processes in order to improve equity and inclusion. More broadly, our core values are based on respect and an unwavering commitment to a diverse and inclusive workforce. Yet, year after year, we have also missed our mark on hiring a representative workforce that reflects our users around the globe. The struggle to improve our equitable outcomes persists despite the policies and programs in place to help support inclusion initiatives and promote excellence in hiring and progression. The failure point is not in the values, intentions, or investments of the company, but rather in the application of those policies at the *implementation* level.

Old habits are hard to break. The users you might be used to designing for today—the ones you are used to getting feedback from—might not be representative of all the users you need to reach. We see this play out frequently across all kinds of products, from wearables that do not work for women's bodies to video-conferencing software that does not work well for people with darker skin tones.

So, what's the way out?

1. **Take a hard look in the mirror.** At Google, we have the brand slogan, "Build For Everyone." How can we build for everyone, when we do not have a representative workforce or engagement model that centralizes community feedback first? We can't. The truth is that we have at times very publicly failed to protect our most vulnerable users from racist, antisemitic, and homophobic content.
2. **Don't build for everyone. Build *with* everyone.** We are not building for everyone yet. That work does not happen in a vacuum, and it certainly doesn't happen when the technology is still not representative of the population as a whole. That said, we can't pack up and go home. So how do we build for everyone? We build with our users. We need to engage our users across the spectrum of humanity and be intentional about putting the most vulnerable communities at the center of our design. They should not be an afterthought.
3. **Design for the user who will have the most difficulty using your product.** Building for those with additional challenges will make the product better for everyone. Another way of thinking about this is: don't trade equity for short-term velocity.

4. **Don't assume equity; measure equity throughout your systems.** Recognize that decision makers are also subject to bias and might be undereducated about the causes of inequity. You might not have the expertise to identify or measure the scope of an equity issue. Catering to a single user base might mean disenfranchising another; these trade-offs can be difficult to spot and impossible to reverse. Partner with individuals or teams that are subject matter experts in diversity, equity, and inclusion.
5. **Change is possible.** The problems we're facing with technology today, from surveillance to disinformation to online harassment, are genuinely overwhelming. We can't solve these with the failed approaches of the past or with just the skills we already have. We need to change.

Stay Curious, Push Forward

The path to equity is long and complex. However, we can and should transition from simply building tools and services, to growing our understanding of how the products we engineer impact humanity. Challenging our education, influencing our teams and managers, and doing more comprehensive user research are all ways to make progress. Although change is uncomfortable and the path to high performance can be painful, it is possible through collaboration and creativity.

Lastly, as future exceptional engineers, we should focus first on the users most impacted by bias and discrimination. Together, we can work to accelerate progress by focusing on Continuous Improvement and owning our failures. Becoming an engineer is an involved and continual process. The goal is to make changes that push humanity forward without further disenfranchising the disadvantaged. As future exceptional engineers, we have faith that we can prevent future failures in the system.

Conclusion

Developing software, and developing a software organization, is a team effort. As a software organization scales, it must respond and adequately design for its user base, which in the interconnected world of computing today involves everyone, locally and around the world. More effort must be made to make both the development teams that design software and the products that they produce reflect the values of such a diverse and encompassing set of users. And, if an engineering organization wants to

scale, it cannot ignore underrepresented groups; not only do such engineers from these groups augment the organization itself, they provide unique and necessary perspectives for the design and implementation of software that is truly useful to the world at large.

TL;DRs

- Bias is the default.
- Diversity is necessary to design properly for a comprehensive user base.
- Inclusivity is critical not just to improving the hiring pipeline for underrepresented groups, but to providing a truly supportive work environment for all people.
- Product velocity must be evaluated against providing a product that is truly useful to all users. It's better to slow down than to release a product that might cause harm to some users.

1 Google's 2019 Diversity Report *https://diversity.google/annual-report/*

2 2015, jackyalcine. "Google Photos, Y'all Fucked up. My Friend's Not a Gorilla. Pic.twitter.com/SMkMCsNVX4." *Twitter*, Twitter, 29 June 2015, twitter.com/jackyalcine/status/615329515909156865.

3 Many reports in 2018-2019 pointed to lack of diversity across tech. Some notables include the National Center for Women & Information Technology, *https://www.ncwit.org/resources/numbers* and Diversity in Tech *https://informationisbeautiful.net/visualizations/diversity-in-tech/*.

4 Wired, "When It Comes to Gorillas, Google Photos Remains Blind" *https://www.wired.com/story/when-it-comes-to-gorillas-google-photos-remains-blind/*

5 Gaines, Stephen, and Sara Williams. "The Perpetual Lineup: Unregulated Police Face Recognition in America." *Center on Privacy & Technology at Georgetown Law*, 18 Oct. 2016.

6 https://diversity.google/annual-report/#!#_this-years-data

Chapter 5. How to Lead a Team

Written by Brian Fitzpatrick

Edited by Riona MacNamara

We've covered a lot of ground so far on the culture and composition of teams writing software, and in this chapter, we'll take a look at the person ultimately responsible for making it all work.

No team can function well without a leader, especially at Google where engineering is almost exclusively a team endeavor. At Google, we recognize two different leadership roles. A "Manager" is a leader of people, whereas while a "Tech Lead" leads technology efforts. Although the responsibilities of these two roles are quite different, they require quite similar skills.

A boat without a captain is nothing more than a floating waiting room: unless someone grabs the rudder and starts the engine, it's just going to drift along aimlessly with the current. A piece of software is just like that boat: if no one pilots it, you're left with a group of engineers burning up valuable time, just sitting around waiting for something to happen (or worse, still writing code that you don't need). Although this chapter is about people management and technical leadership, it is still worth a read if you're an individual contributor because it will likely help you understand your own leaders a bit better.

Managers and Tech Leads (and Both)

Whereas every engineering team generally has a leader, they acquire those leaders in different ways. This is certainly true at Google; sometimes an experienced manager comes in to run a team, and sometimes an individual contributor is promoted into a leadership position (usually of a smaller team).

In nascent teams, both roles will sometimes be filled by the same person: a "Tech Lead Manager" (TLM). On larger teams, an experienced people manager will step in to take on the management role, whereas a senior engineer with extensive experience will step into the tech lead role. Even though manager and tech lead each play an important part in the growth and productivity of an engineering team, the skills required to succeed in each role are wildly different.

The Engineering Manager

Many companies bring in trained people managers who might know little to nothing about software engineering to run their engineering teams. Google decided early on, however, that its software engineering managers should have an engineering background. This meant hiring experienced managers who used to be software engineers, or training software engineers to be managers (more on this later).

At the highest level, an engineering manager is responsible for the performance, productivity, and happiness of every person on their team—including their tech lead—while still making sure that the needs of the business are met with the product for which they are responsible. Because the needs of the business and the needs of individual team members don't always align, this can often place a manager in a difficult position.

The Tech Lead

The tech lead (TL) of a team—who will often report to the manager of that team—is responsible for (surprise!) the technical aspects of the product, including technology decisions and choices, architecture, priorities, velocity, and general project management (although on larger teams they might have program managers helping out with this). The TL will usually work hand in hand with the engineering manager to ensure that the team is adequately staffed for their product and that engineers are set to work on tasks that best match their skillset and skill level. Most TLs are also individual contributors, which often forces them to choose between doing something quickly themselves or delegating it to a team member to do (sometimes) more slowly. The latter is most often the correct decision for the TL as they grow the size and capability of their team.

The Tech Lead Manager

On small and nascent teams, for which engineering managers need a strong technical skillset, the default is often to have a tech lead manager (TLM): a single person who can handle both the people and technical needs of their team. Sometimes, a TLM is a more senior person, but more often than not, the role is taken on by someone who was, until recently, an individual contributor.

At Google, it's customary for larger, well-established teams to have a pair of leaders, one TL and one engineering manager, working together as partners.

The theory is that it's really difficult to do both jobs at the same time (well) without completely burning out, so it's better to have two specialists crushing each role with dedicated focus.

The job of TLM is a tricky one, and often requires the TLM to learn how to balance individual work, delegation, and people management. As such, it usually requires a high degree of mentoring and assistance from more experienced TLMs (in fact, we recommend that in addition to taking a number of classes that Google offers on this subject, a newly minted TLM seek out a senior mentor who can advise them regularly as they grow into the role).

Case Study: Influencing Without Authority

It's generally accepted that you can get folks who report to you to do the work that you need done for your products, but it's different when you need to get people outside of your organization—or heck, even outside of your product area sometimes—to do something that you think needs to be done. This "influence without authority" is one of the most powerful leadership traits that you can develop.

For example, for years Jeff Dean, senior engineering fellow and possibly the most well-known Googler *inside* of Google, led only a fraction of Google's engineering team, but his influence on technical decisions and direction reaches to the ends of the entire engineering organization and beyond (thanks to his writing and speaking outside of the company).

Another example is a team that I started called The Data Liberation Front: with a team of less than a half-dozen engineers, we managed to get more than 50 Google products to export their data through a product that we launched called Google Takeout. At the time, there was no formal directive from the executive level at Google for all products to be a part of Takeout, so how did we get hundreds of engineers to contribute to this effort? By identifying a strategic need for the company, showing how it linked to the mission and existing priorities of the company, and working with a small group of engineers to develop a tool that allowed teams to quickly and easily integrate with Takeout.

Moving from an Individual Contributor Role to a Leadership Role

Whether or not they're officially appointed, someone needs to get into the driver's seat if your product is ever going to go anywhere, and if you're the motivated, impatient type, that person might be you. You might find yourself sucked into helping your team resolve conflicts, make decisions, and coordinate people. It happens all the time, and often by accident. Maybe you never intended to become a "leader," but somehow it happened anyway. Some people refer to this affliction as "manageritis."

Even if you've sworn to yourself that you'll never become a manager, at some point in your career you're likely to find yourself in a leadership position, especially if you've been successful in your role. The rest of this chapter is intended to help you understand what to do when this happens.

We're not here to attempt to convince you to become a manager, but rather to help show why the best leaders work to serve their team using the principles of humility, respect, and trust. Understanding the ins and outs of leadership is a vital skill for influencing the direction of your work. If you want to steer the boat for your project and not just go along for the ride, you need to know how to navigate or you'll run yourself (and your project) onto a sandbar.

The Only Thing to Fear Is…Well, Everything

Aside from the general sense of malaise that most people feel when they hear the word "manager," there are a number of reasons that most people don't want to become managers. The biggest reason you'll hear in the software development world is that you spend much less time writing code. This is true whether you become a TL or an engineering manager, and I'll talk more about this later in the chapter, but first, let's cover some more reasons why most of us avoid becoming managers.

If you've spent the majority of your career writing code, you typically end a day with something you can point to—whether it's code, a design document, or a pile of bugs you just closed—and say, "That's what I did today." But at the end of a busy day of "management" you'll usually find yourself thinking, "I didn't do a damned thing today." It's the equivalent of spending years counting the number of apples you picked each day, and changing to a job growing bananas, only to say to yourself at the end of each day, "I didn't pick any apples," happily ignoring the flourishing banana trees sitting next to you. Quantifying management work is more difficult than counting widgets you turned out, but just making it possible for your team to be happy and productive is a big measure of your job. Just don't fall into the trap of counting apples when you're growing bananas.[1]

Another big reason for not becoming a manager is often unspoken but rooted in the famous "Peter Principle," which states that "In a hierarchy every employee tends to rise to his level of incompetence." Google generally avoids this by requiring that a person perform the job *above* their current level for a period of time (i.e., to "exceeds expectations" at their current level) before being promoted to that level. Most people have had a manager who was incapable of doing their job or was just really bad at managing people,[2] and we know some people who have worked only for bad managers. If you've been exposed only to crappy managers for your entire career, why would you *ever* want to be a manager? Why would you want to be promoted to a role that you don't feel able to do?

There are, however, great reasons to consider becoming a TL or manager. First, it's a way to scale yourself. Even if you're great at writing code, there's still an upper limit to the amount of code you can write. Imagine how much code a team of great engineers could write under your leadership! Second, you might just be really good at it—many people who find themselves sucked into the leadership vacuum of a project discover that they're exceptionally skilled at providing the kind of guidance, help, and air cover a team or a company needs. Someone has to lead, so why not you?

Servant Leadership

There seems to be a sort of disease that strikes managers in which they forget about all the awful things their managers did to them and suddenly begin doing these same things to "manage" the people that report to them. The symptoms of this disease include, but are by no means limited to, micromanaging, ignoring low performers, and hiring pushovers. Without prompt treatment, this disease can kill an entire team. The best advice I received when I first became a manager at Google was from Steve Vinter, an engineering director at the time. He said, "Above all, resist the urge to manage." One of the greatest urges of the newly minted manager is to actively "manage" their employees because that's what a manager does, right? This typically has disastrous consequences.

The cure for the "management" disease is a liberal application of "servant leadership," which is a nice way of saying the most important thing you can do as a leader is to serve your team, much like a butler or majordomo tends to the health and well-being of a household. As a servant leader, you should strive to create an atmosphere of humility, respect, and trust. This might mean removing bureaucratic obstacles that a team member can't remove by themselves, helping a team achieve consensus, or even buying dinner for the

team when they're working late at the office. The servant leader fills in the cracks to smooth the way for their team and advises them when necessary, but still isn't afraid of getting their hands dirty. The only managing that a servant leader does is to manage both the technical and social health of the team; as tempting as it might be to focus on purely the technical health of the team, the social health of the team is just as important (but often infinitely more difficult to manage).

The Engineering Manager

So, what is actually expected of a manager at a modern software company? Before the computing age, "management" and "labor" might have taken on almost antagonistic roles, with the manager wielding all of the power and labor requiring collective action to achieve its own ends. But that isn't how modern software companies work.

Manager Is a Four-Letter Word

Before talking about the core responsibilities of an engineering manager at Google, let's review the history of managers. The present-day concept of the pointy-haired manager is partially a carryover, first from military hierarchy and later adopted by the Industrial Revolution—more than 100 years ago! Factories began popping up everywhere, and they required (usually unskilled) workers to keep the machines going. Consequently, these workers required supervisors to manage them, and because it was easy to replace these workers with other people who were desperate for a job, the managers had little motivation to treat their employees well or improve conditions for them. Whether humane or not, this method worked well for many years when the employees had nothing more to do than perform rote tasks.

Managers frequently treated employees in the same way that cart drivers would treat their mules: they motivated them by alternately leading them forward with a carrot, and, when that didn't work, whipping them with a stick. This carrot-and-stick method of management survived the transition from the factory[3] to the modern office, where the stereotype of the tough-as-nails manager-as-mule-driver flourished in the middle part of the twentieth century when employees would work at the same job for years and years.

This continues today in some industries—even in industries that require creative thinking and problem solving—despite numerous studies suggesting that the anachronistic carrot and stick is ineffective and harmful to the

productivity of creative people. Whereas the assembly-line worker of years past could be trained in days and replaced at will, software engineers working on large codebases can take months to get up to speed on a new team. Unlike the replaceable assembly-line worker, these people need nurturing, time, and space to think and create.

Today's Engineering Manager

Most people still use the title "manager" despite the fact that it's often an anachronism. The title itself often encourages new managers to *manage* their reports. Managers can wind up acting like parents,[4] and consequently employees react like children. To frame this in the context of humility, respect, and trust: if a manager makes it obvious that they trust their employee, the employee feels positive pressure to live up to that trust. It's that simple. A good manager forges the way for a team, looking out for their safety and well-being, all while making sure their needs are met. If there's one thing you remember from this chapter, make it this:

Traditional managers worry about how to get things done, whereas great managers worry about what things get done…(and trust their team to figure out how to do it).

A new engineer, Jerry, joined my team a few years ago. Jerry's last manager (at a different company) was adamant that he be at his desk from 9:00 to 5:00 every day, and assumed that if he wasn't there, he wasn't working enough (which is, of course, a ridiculous assumption). On his first day working with me, Jerry came to me at 4:40 p.m. and stammered out an apology that he had to leave 15 minutes early because he had an appointment that he had been unable to reschedule. I looked at him, smiled, and told him flat out, "Look, as long as you get your job done, I don't care what time you leave the office." Jerry stared blankly at me for a few seconds, nodded, and went on his way. I treated Jerry like an adult; he always got his work done, and I never had to worry about him being at his desk, because he didn't need a babysitter to get his work done. If your employees are so uninterested in their job that they actually need traditional-manager babysitting to be convinced to work, *that* is your real problem.

Failure Is an Option

Another way to catalyze your team is to make them feel safe and secure so that they can take greater risks by building psychological safety—meaning that your team members feel like they can be themselves without fear of negative repercussions from you or their team members. Risk is a fascinating

thing; most humans are terrible at evaluating risk, and most companies try to avoid risk at all costs. As a result of this, the usual *modus operandi* is to work conservatively and focus on smaller successes even when taking a bigger risk might mean exponentially greater success. A common saying at Google is that if you try to achieve an impossible goal, there's a good chance you'll fail, but if you fail trying to achieve the impossible, you'll most likely accomplish far more than you would have accomplished had you merely attempted something you knew you could complete. A good way to build a culture in which risk taking is accepted is to let your team know that it's OK to fail.

So, let's get that out of the way: it's OK to fail. In fact, we like to think of failure as a way of learning a lot really quickly (providing that you're not repeatedly failing at the same thing). In addition, it's important to see failure as an opportunity to learn and not to point fingers or assign blame. Failing fast is good, because there's not a lot at stake. Failing slowly can also teach a valuable lesson, but it is more painful because more is at risk and more can be lost (usually engineering time). Failing in a manner that affects your customers is probably the least desirable failure that we encounter, but it's also one in which we have the greatest amount of structure in place to learn from failures. As mentioned earlier, every time there is a major production failure at Google, we perform a postmortem. This procedure is a way to document the events that led to the actual failure and to develop a series of steps that will prevent it from happening in the future. This is neither an opportunity to point fingers, nor is it intended to introduce unnecessary bureaucratic checks; rather, the goal is to strongly focus on the core of the problem and fix it once and for all. It's very difficult, but quite effective (and cathartic).

Individual successes and failures are a bit different. It's one thing to laud individual successes, but looking to assign individual blame in the case of failure is a great way to divide a team and discourage risk taking across the board. It's alright to fail, but fail as a team and learn from your failures. If an individual succeeds, praise them in front of the team. If an individual fails, give constructive criticism in private.[5] Whatever the case, take advantage of the opportunity and apply a liberal helping of humility, respect, and trust to help your team to learn from its failures.

Antipatterns

Before we go over a litany of "design patterns" for successful TLs and engineering managers, we're going to review a collection of the patterns that you *don't* want to follow if you want to be a successful manager. We've observed these destructive patterns in a handful of bad managers that we've encountered in our careers, and in more than a few cases, ourselves.

Antipattern: Hire Pushovers

If you're a manager and you're feeling insecure in your role (for whatever reason), one way to make sure no one questions your authority or threatens your job is to hire people you can push around. You can achieve this by hiring people who aren't as smart or ambitious as you are, or just people who are more insecure than you. Even though this will cement your position as the team leader and decision maker, it will mean a lot more work for you. Your team won't be able to make a move without you leading them like dogs on a leash. If you build a team of pushovers, you probably can't take a vacation; the moment you leave the room, productivity comes to a screeching halt. But surely this is a small price to pay for feeling secure in your job, right?

Instead, you should strive to hire people who are smarter than you and can replace you. This can be difficult because these very same people will challenge you on a regular basis (in addition to letting you know when you make a mistake). These very same people will also consistently impress you and make great things happen. They'll be able to direct themselves to a much greater extent, and some will be eager to lead the team, as well. You shouldn't see this as an attempt to usurp your power; instead, look at it as an opportunity for you to lead an additional team, investigate new opportunities, or even take a vacation without worrying about checking in on the team every day to make sure it's getting its work done. It's also a great chance to learn and grow—it's a lot easier to expand your expertise when surrounded by people who are smarter than you.

Antipattern: Ignore Low Performers

Early in my career as a manager at Google, the time came for me to hand out bonus letters to my team, and I grinned as I told my manager, "I love being a manager!" Without missing a beat, my manager, a long-time industry veteran, replied, "Sometimes you get to be the tooth fairy, other times you have to be the dentist."

It's never any fun to pull teeth. We've seen team leaders do all the right things to build incredibly strong teams, only to have these teams fail to excel

(and eventually fall apart) because of just one or two low performers. We understand that the human aspect is the most challenging part of writing software, but the most difficult part of dealing with humans is handling someone who isn't meeting expectations. Sometimes, people miss expectations because they're not working long enough or hard enough, but the most difficult cases are when someone just isn't capable of doing their job no matter how long or hard they work.

Google's Site Reliability Engineering (SRE) team has a motto: "Hope is not a strategy." And nowhere is hope more overused as a strategy than in dealing with a low performer. Most team leaders grit their teeth, avert their eyes, and just *hope* that the low performer either magically improves or just goes away. Yet it is extremely rare that this person does either.

While the leader is hoping and the low performer isn't improving (or leaving), high performers on the team waste valuable time pulling the low performer along and team morale leaks away into the ether. You can be sure that the team knows the low performer is there even if you're ignoring them—in fact, the team is *acutely* aware of who the low performers are, because they have to carry them.

Ignoring low performers is not only a way to keep new high performers from joining your team, but it's also a way to encourage existing high performers to leave. You eventually wind up with an entire team of low performers because they're the only ones who can't leave of their own volition. Lastly, you aren't even doing the low performer any favors by keeping them on the team; often, someone who wouldn't do well on your team could actually have plenty of impact somewhere else.

The benefit of dealing with a low performer as quickly as possible is that you can put yourself in the position of helping them up or out. If you immediately deal with a low performer, you'll often find that they merely need some encouragement or direction to slip into a higher state of productivity. If you wait too long to deal with a low performer, their relationship with the team is going to be so sour and you're going to be so frustrated that you're not going to be able to help them.

How do you effectively coach a low performer? The best analogy is to imagine that you're helping a limping person learn to walk again, then jog, then run alongside the rest of the team. It almost always requires temporary micromanagement, but still a whole lot of humility, respect, and trust—particularly respect. Set up a specific time frame (say, two months), and some very specific goals you expect him to achieve in that period. Make the goals

small, incremental, and measurable so that there's an opportunity for lots of small successes. Meet with the team member every week to check on progress, and be sure you set really explicit expectations around each upcoming milestone so that it's easy to measure success or failure. If the low performer can't keep up, it will become quite obvious to both of you early in the process. At this point, the person will often acknowledge that things aren't going well and decide to quit; in other cases, determination will kick in and they'll "up their game" to meet expectations. Either way, by working directly with the low performer, you're catalyzing important and necessary changes.

Antipattern: Ignore Human Issues

A manager has two major areas of focus for their team: the social and the technical. It's rather common for managers to be stronger in the technical side at Google, and because most managers are promoted from a technical job (for which the primary goal of their job was to solve technical problems), they can tend to ignore human issues. It's tempting to focus all of your energy on the technical side of your team because, as an individual contributor, you spend the vast majority of your time solving technical problems. When you were a student, your classes were all about learning the technical ins and outs of your work. Now that you're a manager, however, you ignore the human element of your team at your own peril.

Let's begin with an example of a leader ignoring the human element in his team. Years ago, Jake had his first child. Jake and Katie had worked together for years, both remotely and in the same office, so in the weeks following the arrival of the new baby, Jake worked from home. This worked out great for the couple, and Katie was totally fine with it because she was already used to working remotely with Jake. They were their usual productive selves until their manager, Pablo (who worked in a different office), found out that Jake was working from home for most of the week. Pablo was upset that Jake wasn't going into the office to work with Katie, despite the fact that Jake was just as productive as always and that Katie was fine with the situation. Jake attempted to explain to Pablo that he was just as productive as he would be if he came into the office and that it was much easier on him and his wife for him to mostly work from home for a few weeks. Pablo's response: "Dude, people have kids all the time. You need to go into the office." Needless to say, Jake (normally a mild-mannered engineer) was enraged and lost a lot of respect for Pablo.

There are numerous ways in which Pablo could have handled this differently: he could have showed some understanding that Jake wanted to spend more time at home for his wife and, if his productivity and team weren't being affected, just let him continue to do so for a while. He could have negotiated that Jake go into the office for one or two days a week until things settled down. Regardless of the end result, a little bit of empathy would have gone a long way toward keeping Jake happy in this situation.

Antipattern: Be Everyone's Friend

The first foray that most people have into leadership of any sort is when they become the manager or TL of a team of which they were formerly members. Many leads don't want to lose the friendships they've cultivated with their teams, so they will sometimes work extra hard to maintain friendships with their team members after becoming a team lead. This can be a recipe for disaster and for a lot of broken friendships. Don't confuse friendship with leading with a soft touch: when you hold power over someone's career, they might feel pressure to artificially reciprocate gestures of friendship.

Remember that you can lead a team and build consensus without being a close friend of your team (or a monumental hard-ass). Likewise, you can be a tough leader without tossing your existing friendships to the wind. We've found that having lunch with your team can be an effective way to stay socially connected to them without making them uncomfortable—this gives you a chance to have informal conversations outside the normal work environment.

Sometimes, it can be tricky to move into a management role over someone who has been a good friend and a peer. If the friend who is being managed is not self-managing and is not a hard worker, it can be stressful for everyone. We recommend that you avoid getting into this situation whenever possible, but if you can't, pay extra attention to your relationship with those folks.

Antipattern: Compromise the Hiring Bar

Steve Jobs once said: "A people hire other A people; B people hire C people." It's incredibly easy to fall victim to this adage, and even more so when you're trying to hire quickly. A common approach I've seen outside of Google is that a team needs to hire five engineers, so it sifts through a pile of applications, interviews 40 or 50 people, and picks the best five candidates regardless of whether they meet the hiring bar.

This is one of the fastest ways to build a mediocre team.

The cost of finding the appropriate person—whether by paying recruiters, paying advertising, or pounding the pavement for references—pales in comparison to the cost of dealing with an employee who you never should have hired in the first place. This "cost" manifests itself in lost team productivity, team stress, time spent managing the employee up or out, and the paperwork and stress involved in firing the employee. That's assuming, of course, that you try to avoid the monumental cost of just leaving them on the team. If you're managing a team for which you don't have a say over hiring and you're unhappy with the hires being made for your team, you need to fight tooth and nail for higher-quality engineers. If you're still handed substandard engineers, maybe it's time to look for another job. Without the raw materials for a great team, you're doomed.

Antipattern: Treat Your Team Like Children

The best way to show your team that you don't trust it is to treat team members like kids—people tend to act the way you treat them, so if you treat them like children or prisoners, don't be surprised when that's how they behave. You can manifest this behavior by micromanaging them or simply by being disrespectful of their abilities and giving them no opportunity to be responsible for their work. If it's permanently necessary to micromanage people because you don't trust them, you have a hiring failure on your hands. Well, it's a failure unless your goal was to build a team that you can spend the rest of your life babysitting. If you hire people worthy of trust and show these people you trust them, they'll usually rise to the occasion (sticking with the basic premise, as we mentioned earlier, that you've hired good people).

The results of this level of trust go all the way to more mundane things like office and computer supplies. As another example, Google provides employees with cabinets stocked with various and sundry office supplies (e.g., pens, notebooks, and other "legacy" implements of creation) that are free to take as employees need them. The IT department runs numerous "Tech Stops" that provide self-service areas that are like a mini electronics store. These contain lots of computer accessories and doodads (power supplies, cables, mice, USB drives, etc.) that would be easy to just grab and walk off with *en masse*, but because Google employees are being entrusted to check these items out, they feel a responsibility to Do The Right Thing. Many people from typical corporations react in horror to hearing this, exclaiming that surely Google is hemorrhaging money due to people "stealing" these items. That's certainly possible, but what about the costs of having a workforce that behaves like children or that has to waste valuable

time formally requesting cheap office supplies? Surely that's more expensive than the price of a few pens and USB cables.

Positive Patterns

Now that we've covered antipatterns, let's turn to positive patterns for successful leadership and management that we've learned from our experiences at Google, from watching other successful leaders and, most of all, from our own leadership mentors. These patterns are not only those that we've had great success implementing, but the patterns that we've always respected the most in the leaders who we follow.

Lose the Ego

We talked about "losing the ego" a few chapters ago when we first examined humility, respect, and trust, but it's especially important when you're a team leader. This pattern is frequently misunderstood as encouraging people to be a doormat and let others walk all over them, but that's not the case at all. Of course, there's a fine line between being humble and letting others take advantage of you, but humility is not the same as lacking confidence. You can still have self-confidence and opinions without being an egomaniac. Big personal egos are difficult to handle on any team, especially in the team's leader. Instead, you should work to cultivate a strong collective team ego and identity.

Part of "losing the ego" is trust: you need to trust your team. That means respecting the abilities and prior accomplishments of the team members, even if they're new to your team.

If you're not micromanaging your team, you can be pretty certain the folks working in the trenches know the details of their work better than you do. This means that although you might be the one driving the team to consensus and helping to set the direction, the nuts and bolts of how to accomplish your goals are best decided by the people who are putting the product together. This gives them not only a greater sense of ownership, but also a greater sense of accountability and responsibility for the success (or failure) of their product. If you have a good team and you let it set the bar for the quality and rate of its work, it will accomplish more than by you standing over team members with a carrot and a stick.

Most people new to a leadership role feel an enormous responsibility to get everything right, to know everything, and to have all the answers. We can

assure you that you will not get everything right, nor will you have all the answers, and if you act like you do, you'll quickly lose the respect of your team. A lot of this comes down to having a basic sense of security in your role. Think back to when you were an individual contributor; you could smell insecurity a mile away. Try to appreciate inquiry: when someone questions a decision or statement you made, remember that this person is usually just trying to better understand you. If you encourage inquiry, you're much more likely to get the kind of constructive criticism that will make you a better leader of a better team. Finding people who will give you good constructive criticism is incredibly difficult, and it's even more difficult to get this kind of criticism from people who "work for you." Think about the big picture of what you're trying to accomplish as a team, and accept feedback and criticism openly; avoid the urge to be territorial.

The last part of losing the ego is a simple one, but many engineers would rather be boiled in oil than do it: apologize when you make a mistake. And we don't mean you should just sprinkle "I'm sorry" throughout your conversation like salt on popcorn—you need to sincerely mean it. You are absolutely going to make mistakes, and whether or not you admit it, your team is going to know you've made a mistake. Your team members will know regardless of whether they talk to you (and one thing is guaranteed: they *will* talk about it with one another). Apologizing doesn't cost money. People have enormous respect for leaders who apologize when they screw up, and contrary to popular belief, apologizing doesn't make you vulnerable. In fact, you'll usually gain respect from people when you apologize, because apologizing tells people that you are level headed, good at assessing situations, and—coming back to humility, respect, and trust—humble.

Be a Zen Master

As an engineer, you've likely developed an excellent sense of skepticism and cynicism, but this can be a liability when you're trying to lead a team. This is not to say that you should be naïvely optimistic at every turn, but you would do well to be less vocally skeptical while still letting your team know you're aware of the intricacies and obstacles involved in your work. Mediating your reactions and maintaining your calm is more important as you lead more people, because your team will (both unconsciously and consciously) look to you for clues on how to act and react to whatever is going on around you.

A simple way to visualize this effect is to see your company's organization chart as a chain of gears, with the individual contributor as a tiny gear with just a few teeth all the way at one end, and each successive manager above

them as another gear, ending with the CEO as the largest gear with many hundreds of teeth. This means that every time that individual's "manager gear" (with maybe a few dozen teeth) makes a single revolution, the "individual's gear" makes two or three revolutions. And the CEO can make a small movement and send the hapless employee, at the end of a chain of six or seven gears, spinning wildly! The farther you move up the chain, the faster you can set the gears below you spinning, whether or not you intend to.

Another way of thinking about this is the maxim that the leader is always on stage. This means that if you're in an overt leadership position, you are always being watched: not just when you run a meeting or give a talk, but even when you're just sitting at your desk answering emails. Your peers are watching you for subtle clues in your body language, your reactions to small talk, and your signals as you eat lunch. Do they read confidence or fear? As a leader, your job is to inspire, but inspiration is a 24/7 job. Your visible attitude about absolutely everything—no matter how trivial—is unconsciously noticed and spreads infectiously to your team.

One of the early managers at Google, Bill Coughran, a VP of engineering, had truly mastered the ability to maintain calm at all times. No matter what blew up, no matter what crazy thing happened, no matter how big the firestorm, Bill would never panic. Most of the time he'd place one arm across his chest, rest his chin in his hand, and ask questions about the problem, usually to a completely panicked engineer. This had the effect of calming them and helping them to focus on solving the problem instead of running around like a chicken with its head cut off. Some of us used to joke that if someone came in and told Bill that 19 of the company's offices had been attacked by space aliens, Bill's response would be, "Any idea why they didn't make it an even 20?"

This brings us to another Zen management trick: asking questions. When a team member asks you for advice, it's usually pretty exciting because you're finally getting the chance to fix something. That's exactly what you did for years before moving into a leadership position, so you usually go leaping into solution mode, but that is the last place you should be. The person asking for advice typically doesn't want *you* to solve their problem, but rather to help them solve it, and the easiest way to do this is to ask this person questions. This isn't to say that you should replace yourself with a Magic 8 Ball, which would be maddening and unhelpful. Instead, you can apply some humility, respect, and trust and try to help the person solve the problem on their own by trying to refine and explore the problem. This will usually lead the employee to the answer,[6] and it will be that person's answer, which leads

back to the ownership and responsibility we went over earlier in this chapter. Whether or not you have the answer, using this technique will almost always leave the employee with the impression that you did. Tricky, eh? Socrates would be proud of you.

Be a Catalyst

In chemistry, a catalyst is something that accelerates a chemical reaction, but which itself is not consumed in the reaction. One of the ways in which catalysts (e.g., enzymes) work is to bring reactants into close proximity: instead of bouncing around randomly in a solution, the reactants are much more likely to favorably interact with one another when the catalyst helps bring them together. This is a role you'll often need to play as a leader, and there are a number of ways you can go about it.

One of the most common things a team leader does is to build consensus. This might mean that you drive the process from start to finish, or you just give it a gentle push in the right direction to speed it up. Working to build team consensus is a leadership skill that is often used by unofficial leaders because it's one way you can lead without any actual authority. If you have the authority, you can direct and dictate direction, but that's less effective overall than building consensus.[7] If your team is looking to move quickly, sometimes it will voluntarily concede authority and direction to one or more team leads. Even though this might look like a dictatorship or oligarchy, when it's done voluntarily, it's a form of consensus.

Remove Roadblocks

Sometimes, your team already has consensus about what you need to do, but it hit a roadblock and became stuck. This could be a technical or organizational roadblock, but jumping in to help the team get moving again is a common leadership technique. There are some roadblocks that, although virtually impossible for your team members to get past, will be easy for you to handle, and helping your team to understand that you're glad (and able) to help out with these roadblocks is valuable.

One time, a team spent several weeks trying to work past an obstacle with Google's legal department. When the team finally reached its collective wits' end and went to its manager with the problem, the manager had it solved in less than two hours simply because he knew the right person to contact to discuss the matter. Another time, a team needed some server resources and just couldn't get them allocated. Fortunately, the team's manager was in

communication with other teams across the company and managed to get the team exactly what it needed that very afternoon. Yet another time, one of the engineers was having trouble with an arcane bit of Java code. Although the team's manager was not a Java expert, she was able to connect the engineer to another engineer who knew exactly what the problem was. You don't need to know all the answers to help remove roadblocks, but it usually helps to know the people who do. In many cases, knowing the right person is more valuable than knowing the right answer.

Be a Teacher and a Mentor

One of the most difficult things to do as a TL is to watch a more junior team member spend three hours working on something that you know you can knock out in 20 minutes. Teaching people and giving them a chance to learn on their own can be incredibly difficult at first, but it's a vital component of effective leadership. This is especially important for new hires who, in addition to learning your team's technology and codebase, are learning your team's culture and the appropriate level of responsibility to assume. A good mentor must balance the trade-offs of a mentee's time learning versus their time contributing to their product as part of an effective effort to scale the team as it grows.

Much like the role of manager, most people don't apply for the role of mentor—they usually become one when a leader is looking for someone to mentor a new team member. It doesn't take a lot of formal education or preparation to be a mentor. Primarily, you need three things: experience with your team's processes and systems, the ability to explain things to someone else, and the ability to gauge how much help your mentee needs. The last thing is probably the most important—giving your mentee enough information is what you're supposed to be doing, but if you overexplain things or ramble on endlessly, your mentee will probably tune you out rather than politely tell you they got it.

Set Clear Goals

This is one of those patterns that, as obvious as it sounds, is solidly ignored by an enormous number of leaders. If you're going to get your team moving rapidly in one direction, you need to make sure that every team member understands and agrees on what the direction is. Imagine your product is a big truck (and not a series of tubes). Each team member has in their hand a rope tied to the front of the truck, and as they work on the product, they'll pull the truck in their own direction. If your intention is to pull the truck (or

product) northbound as quickly as possible, you can't have team members pulling every which way—you want them all pulling the truck north. If you're going to have clear goals, you need to set clear priorities and help your team decide how it should make trade-offs when the time comes.

The easiest way to set a clear goal and get your team pulling the product in the same direction is to create a concise mission statement for the team. After you've helped the team define its direction and goals, you can step back and give it more autonomy, periodically checking in to make sure everyone is still on the right track. This not only frees up your time to handle other leadership tasks, it also drastically increases the efficiency of your team. Teams can (and do) succeed without clear goals, but they typically waste a great deal of energy as each team member pulls the product in a slightly different direction. This frustrates you, slows progress for the team, and forces you to use more and more of your own energy to correct the course.

Be Honest

This doesn't mean that we're assuming you are lying to your team, but it merits a mention because you'll inevitably find yourself in a position in which you can't tell your team something or, even worse, you need to tell everyone something they don't want to hear. One manager we know tells new team members, "I won't lie to you, but I will tell you when I can't tell you something or if I just don't know."

If a team member approaches you about something you can't share, it's OK to just tell them you know the answer but are not at liberty to say anything. Even more common is when a team member asks you something you don't know the answer to: you can tell that person you don't know. This is another one of those things that seems blindingly obvious when you read it, but many people in a manager role feel that if they don't know the answer to something, it proves that they're weak or out of touch. In reality, the only thing it proves is that they're human.

Giving hard feedback is…well, hard. The first time you need to tell one of your reports that they made a mistake or didn't do their job as well as expected can be incredibly stressful. Most management texts advise that you use the "compliment sandwich" to soften the blow when delivering hard feedback. A compliment sandwich looks something like this:

You're a solid member of the team and one of our smartest engineers. That being said, your code is convoluted and almost impossible for anyone else on

the team to understand. But you've got great potential and a wicked cool T-shirt collection.

Sure, this softens the blow, but with this sort of beating around the bush most people will walk out of this meeting thinking only, "Sweet! I've got cool T-shirts!" We *strongly* advise against using the compliment sandwich, not because we think you should be unnecessarily cruel or harsh, but because most people won't hear the critical message, which is that something needs to change. It's possible to employ respect here: be kind and empathetic when delivering constructive criticism without resorting to the compliment sandwich. In fact, kindness and empathy are critical if you want the recipient to hear the criticism and not immediately go on the defensive.

Years ago, a colleague picked up a team member, Tim, from another manager who insisted that Tim was impossible to work with. He said that Tim never responded to feedback or criticism and instead just kept doing the same things he'd been told he shouldn't do. Our colleague sat in on a few of the manager's meetings with Tim to watch the interaction between the manager and Tim, and noticed that the manager made extensive use of the compliment sandwich so as not to hurt Tim's feelings. When they brought Tim onto their team, they sat down with him and very clearly explained that Tim needed to make some changes to work more effectively with the team:

We're quite sure that you're not aware of this, but the way that you're interacting with the team is alienating and angering them, and if you want to be effective, you need to refine your communication skills and we're committed to helping you do that.

They didn't give Tim any compliments or candy-coat the issue, but just as important, they weren't mean—they just laid out the facts as they saw them based on Tim's performance with the previous team. Lo and behold, within a matter of weeks (and after a few more "refresher" meetings), Tim's performance improved dramatically. Tim just needed very clear feedback and direction.

When you're providing direct feedback or criticism, your delivery is key to making sure that your message is heard and not deflected. If you put the recipient on the defensive, they're not going to be thinking of how they can change, but rather how they can argue with you to show you that you're wrong. Our colleague Ben once managed an engineer who we'll call Dean. Dean had extremely strong opinions and would argue with the rest of the team about anything. It could be something as big as the team's mission or as small as the placement of a widget on a web page; Dean would argue with

the same conviction and vehemence either way, and he refused to let anything slide. After months of this behavior, Ben met with Dean to explain to him that he was being too combative. Now, if Ben had just said, "Dean, stop being such a jerk," you can be pretty sure Dean would have disregarded it entirely. Ben thought hard about how he could get Dean to understand how his actions were adversely affecting the team, and he came up with the following metaphor:

Every time a decision is made, it's like a train coming through town—when you jump in front of the train to stop it, you slow the train down and potentially annoy the engineer driving the train. A new train comes by every 15 minutes, and if you jump in front of every train, not only do you spend a lot of your time stopping trains, but eventually one of the engineers driving the train is going to get mad enough to run right over you. So, although it's OK to jump in front of some trains, pick and choose the ones you want to stop to make sure you're stopping only the trains that really matter.

This anecdote not only injected a bit of humor into the situation, but also made it easier for Ben and Dean to discuss the effect that Dean's "train stopping" was having on the team in addition to the energy Dean was spending on it.

Track Happiness

As a leader, one way you can make your team more productive (and less likely to leave) in the long term is to take some time to gauge their happiness. The best leaders we've worked with have all been amateur psychologists, looking in on their team members' welfare from time to time, making sure they get recognition for what they do, and trying to make certain they are happy with their work. One TLM we know makes a spreadsheet of all the grungy, thankless tasks that need to be done and makes certain these tasks are evenly spread across the team. Another TLM watches the hours his team is working and uses comp time and fun team outings to avoid burnout and exhaustion. Yet another starts one-on-one sessions with his team members by dealing with their technical issues as a way to break the ice, and then takes some time to make sure each engineer has everything they need to get their work done. After they've warmed up, he talks to the engineer for a bit about how they're enjoying the work and what they're looking forward to next.

A good simple way to track your team's happiness[8] is to ask the team member at the end of each one-on-one meeting, "What do you need?" This simple question is a great way to wrap up and make sure each team member has what they need to be productive and happy, although you might need to

carefully probe a bit to get details. If you ask this every time you have a one-on-one, you'll find that eventually your team will remember this and sometimes even come to you with a laundry list of things it needs to make everyone's job better.

The Unexpected Question

Shortly after I started at Google, I had my first meeting with then-CEO Eric Schmidt, and at the end Eric asked me, "Is there anything you need?" I had prepared a million defensive responses to difficult questions or challenges but was completely unprepared for this. So I sat there, dumbstruck and staring. You can be sure I had something ready the next time I was asked that question!

It can also be worthwhile as a leader to pay some attention to your team's happiness outside the office. Our colleague Mekka starts his one-on-ones by asking his reports to rate their happiness on a scale of 1 to 10, and oftentimes his reports will use this as a way to discuss happiness in *and* outside of the office. Be wary of assuming that people have no life outside of work—having unrealistic expectations about the amount of time people can put into their work will cause people to lose respect for you, or worse, to burn out. We're not advocating that you pry into your team members' personal lives, but being sensitive to personal situations that your team members are going through can give you a lot of insight as to why they might be more or less productive at any given time. Giving a little extra slack to a team member who is currently having a tough time at home can make them a lot more willing to put in longer hours when your team has a tight deadline to hit later.

A big part of tracking your team members' happiness is tracking their careers. If you ask a team member where they see their career in five years, most of the time you'll get a shrug and a blank look. When put on the spot, most people won't say much about this, but there are usually a few things that everyone would like to do in the next five years: be promoted, learn something new, launch something important, and work with smart people. Regardless of whether they verbalize this, most people are thinking about it. If you're going to be an effective leader, you should be thinking about how you can help make all those things happen and let your team know you're thinking about this. The most important part of this is to take these implicit goals and make them explicit so that when you're giving career advice you have a real set of metrics on which to evaluate situations and opportunities.

Tracking happiness comes down to not just monitoring careers, but also giving your team members opportunities to improve themselves, be recognized for the work they do, and have a little fun along the way.

Other Tips and Tricks

Following are other, miscellaneous tips and tricks that we at Google would recommend when you're in a leadership position:

Delegate, but get your hands dirty

When moving from an individual contributor role to a leadership role, achieving a balance is one of the most difficult things to do. Initially, you're inclined to do all of the work yourself, and after being in a leadership role for a long time, it's easy to get into the habit of doing none of the work yourself. If you're new to a leadership role, you probably need to work hard to delegate work to other engineers on your team, even if it will take them a lot longer than you to accomplish that work. Not only is this one way for you to maintain your sanity, but also it's how the rest of your team will learn. If you've been leading teams for a while or if you pick up a new team, one of the easiest ways to gain the team's respect and get up to speed on what they're doing is to get your hands dirty—usually by taking on a grungy task that no one else wants to do. You can have a résumé and a list of achievements a mile long, but nothing lets a team know how skillful and dedicated (and humble) you are like jumping in and actually doing some hard work.

Seek to replace yourself

Unless you want to keep doing the exact same job for the rest of your career, seek to replace yourself. This starts, as we mentioned earlier, with the hiring process: if you want a member of your team to replace you, you need to hire people capable of replacing you, which we usually sum up by saying you need to "hire people smarter than you." After you have team members capable of doing your job, you need to give them opportunities to take on more responsibilities or occasionally lead the team. If you do this, you'll quickly see who has the most aptitude to lead as well as who wants to lead the team. Remember that some people prefer to just be high-performing

individual contributors, and that's OK. We've always been amazed at companies that take their best engineers and—against their wishes—throw these engineers into management roles. This usually subtracts a great engineer from your team and adds a subpar manager.

Know when to make waves

You will (inevitably and frequently) have difficult situations crop up in which every cell in your body is screaming at you to do nothing about it. It might be the engineer on your team whose technical chops aren't up to par. It might be the person who jumps in front of every train. It might be the unmotivated employee who is working 30 hours a week. "Just wait a bit and it will get better," you'll tell yourself. "It will work itself out," you'll rationalize. Don't fall into this trap—these are the situations for which you need to make the biggest waves and you need to make them now. Rarely will these problems work themselves out, and the longer you wait to address them, the more they'll adversely affect the rest of the team and the more they'll keep you up at night thinking about them. By waiting, you're only delaying the inevitable and causing untold damage in the process. So act, and act quickly.

Shield your team from chaos

When you step into a leadership role, the first thing you'll usually discover is that outside your team is a world of chaos and uncertainty (or even insanity) that you never saw when you were an individual contributor. When I first became a manager back in the 1990s (before going back to being an individual contributor), I was taken aback by the sheer volume of uncertainty and organizational chaos that was happening in my company. I asked another manager what had caused this sudden rockiness in the otherwise calm company, and the other manager laughed hysterically at my naïveté: the chaos had always been present, but my previous manager had shielded me and the rest of my team from it.

Give your team air cover

Whereas it's important that you keep your team informed about what's going on "above" them in the company, it's just as important that you defend them from a lot of the uncertainty and frivolous demands that can be imposed upon you from outside your team.

Share as much information as you can with your team, but don't distract them with organizational craziness that is extremely unlikely to ever actually affect them.

Let your team know when they're doing well

Many new team leads can get so caught up in dealing with the shortcomings of their team members that they neglect to provide positive feedback often enough. Just as you let someone know when they screw up, be sure to let them know when they do well, and be sure to let them (and the rest of the team) know when they knock one out of the park.

Lastly, here's something the best leaders know and use often when they have adventurous team members who want to try new things:

it's easy to say "yes" to something that's easy to undo

If you have a team member who wants to take a day or two to try using a new tool or library[9] that could speed up your product (and you're not on a tight deadline), it's easy to say, "Sure, give it a shot." If, on the other hand, they want to do something like launch a product that you're going to have to support for the next 10 years, you'll likely want to give it a bit more thought. Really good leaders have a good sense for when something can be undone, but more things are undoable than you think (and this applies to both technical and nontechnical decisions).

People Are Like Plants

My wife is the youngest of six children, and her mother was faced with the difficult task of figuring out how to raise six very different children, each of whom needed different things. I asked my mother-in-law how she managed this (see what I did there?), and she responded that kids are like plants: some are like cacti and need little water but lots of sunshine, others are like African violets and need diffuse light and moist soil, and still others are like tomatoes and will truly excel if you give them a little fertilizer. If you have six kids and give each one the same amount of water, light, and fertilizer, they'll all get equal treatment, but the odds are good that *none* of them will get what they actually need.

And so your team members are also like plants: some need more light, and some need more water (and some need more…fertilizer). It's your job as their leader to determine who needs what and then give it to them—except instead of light, water, and fertilizer, your team needs varying amounts of motivation and direction.

To get all of your team members what they need, you need to motivate the ones who are in a rut, and provide stronger direction to those who are distracted or uncertain of what to do. Of course, there are those who are "adrift" and need both motivation and direction. So, with this combination of motivation and direction you can make your team happy and productive. And you don't want to give them too much of either—because if they don't need motivation or direction and you try giving it to them, you're just going to annoy them.

Giving direction is fairly straightforward—it requires a basic understanding of what needs to be done, some simple organizational skills, and enough coordination to break it down into manageable tasks. With these tools in hand you can provide sufficient guidance for an engineer in need of directional help. Motivation, however, is a bit more sophisticated and merits some explanation.

Intrinsic versus Extrinsic Motivation

There are two types of motivation: *extrinsic*, which originates from outside forces (such as monetary compensation), and *intrinsic*, which comes from within. In his book *Drive*,[10] Dan Pink explains that the way to make people the happiest and most productive isn't to motivate them extrinsically (e.g., throw piles of cash at them); rather, you need to work to increase their intrinsic motivation. Dan claims you can increase intrinsic motivation by giving people three things: autonomy, mastery, and purpose.[11]

A person has autonomy when they have the ability to act on their own without someone micromanaging them[12]. With autonomous employees (and Google strives to hire mostly autonomous engineers), you might give them the general direction in which they need to take the product but leave it up to them to decide how to get there. This helps with motivation not only because they have a closer relationship with the product (and likely know better than you how to build it), but also because it gives them a much greater sense of ownership of the product. The bigger their stake is in the success of the product, the greater their interest is in seeing it succeed.

Mastery in its basest form simply means that you need to give someone the opportunity to improve existing skills and learn new ones. Giving ample opportunities for mastery not only helps to motivate people, but also makes them better over time, which makes for stronger teams.[13] An employee's skills are like the blade of a knife: you can spend tens of thousands of dollars to find people with the sharpest skills for your team, but if you use that knife for years without sharpening it, you will wind up with a dull knife that is inefficient, and in some cases useless. Google gives ample opportunities for engineers to learn new things and master their craft so as to keep them sharp, efficient, and effective.

Of course, all the autonomy and mastery in the world isn't going to help motivate someone if they're doing work for no reason at all, which is why you need to give their work purpose. Many people work on products that have great significance, but they're kept at arm's length from the positive effects their products might have on their company, their customers, or even the world. Even for cases in which the product might have a much smaller impact, you can motivate your team by seeking the reason for their efforts and making this reason clear to them. If you can help them to see this purpose in their work, you'll see a tremendous increase in their motivation and productivity.[14] One manager we know keeps a close eye on the email feedback that Google gets for its product (one of the "smaller-impact" products), and whenever she sees a message from a customer talking about how the company's product has helped the customer personally or helped the customer's business, she immediately forwards it to the engineering team. This not only motivates the team, but also frequently inspires team members to think about ways in which they can make their product even better.

Conclusion

Leading a team is a different task than that of being a software engineer. As a result, good software engineers do not always make good managers, and that's OK—effective organizations allow productive career paths to both individual contributors and people managers. Though Google has found that software engineering experience itself is invaluable for managers, the most important skills an effective manager brings to the table are social ones. Good managers enable their engineering teams by helping them work well, keeping them focused on proper goals, and insulating them from problems outside the group, all the while following the three pillars of humility, trust, and respect.

TL;DRs

- Don't "manage" in the traditional sense; focus on leadership, influence, and serving your team.
- Delegate where possible, don't DIY (Do It Yourself).
- Pay particular attention to the focus, direction, and velocity of your team.

1 Another difference that takes getting used to is that the things we do as managers typically pay off over a longer timeline.

2 Yet another reason companies shouldn't force people into management as part of a career path: if an engineer is able to write reams of great code and has no desire at all to manage people or lead a team, by forcing them into a management or TL role you're losing a great engineer and gaining a crappy manager. This is not only a bad idea, but it's actively harmful.

3 For more fascinating information on optimizing the movements of factory workers, read up on Scientific Management or Taylorism, especially its effects on worker morale.

4 If you have kids, the odds are good that you can remember with startling clarity the first time you said something to your child that made you stop and exclaim (perhaps even aloud), "Holy crap, I've become my mother."

5 Public criticism of an individual is not only ineffective (it puts people on the defense), but rarely necessary, and most often is just mean or cruel. You can be sure the rest of the team already knows when an individual has failed, so there's no need to rub it in.

6 See also "Rubber duck debugging."

7 Attempting to achieve 100% consensus can also be harmful. You need to be able to decide to proceed even if not everyone is on the same page or there is still some uncertainty.

8 Google also runs an annual employee survey called "Googlegeist" that rates employee happiness across many dimensions. This provides good feedback but isn't what we would call "simple."

9 To gain a better understanding of just how "undoable" technical changes can be, see Chapter 22.

10 See Dan's fantastic TED talk on this subject.

11 This assumes that the people in question are being paid well enough that income is not a source of stress.

12 This assumes that you have people on your team who don't need micromanagement.

13 Of course, it also means they're more valuable and marketable employees, so it's easier for them to pick up and leave you if they're not enjoying their work. See the pattern in "Track Happiness."

14 *http://bit.ly/task_significance*

Chapter 6. Leading at Scale

Written by Ben Collins-Sussman

Edited by Riona MacNamara

In Chapter 5, we talked about what it means to go from being an "individual contributor" to being an explicit leader of a team. It's a natural progression to go from leading one team to leading a set of related teams, and this chapter talks about how to be effective as you continue along the path of engineering leadership.

As your role evolves, all the best practices still apply. You're still a "servant leader"; you're just serving a larger group. That said, the scope of problems you're solving becomes larger and more abstract. You're gradually forced to become "higher level." That is, you're less and less able to get into the technical or engineering details of things, and you're being pushed to go "broad" rather than "deep." At every step, this process is frustrating: you mourn the loss of these details, and you come to realize that your prior engineering expertise is becoming less and less relevant to your job. Instead, your effectiveness depends more than ever on your *general* technical intuition and ability to galvanize engineers to move in good directions.

The process is often demoralizing—until one day you notice that you're actually having much more impact as a leader than you ever had as an individual contributor. It's a satisfying but bittersweet realization.

So, assuming that we understand the basics of leadership, what it does it take to scale yourself into a *really good* leader? That's what we talk about here, using what we call "the three Always of leadership": Always Be Deciding, Always Be Leaving, Always Be Scaling.

Always Be Deciding

Managing a team of teams means making ever more decisions at ever higher levels. Your job becomes more about high-level strategy rather than how to solve any specific engineering task. At this level, most of the decisions you'll make are about finding the correct set of trade-offs.

The Parable of the Airplane

Lindsay Jones is a friend of ours who is a professional theatrical sound designer and composer. He spends his life flying around the United States, hopping from production to production, and he's full of crazy (and true) stories about air travel. Here's one of our favorite stories:

It's 6 a.m., we're all boarded on the plane and ready to go. The captain comes on the PA system and explains to us that, somehow, someone has overfilled the fuel tank by 10,000 gallons. Now, I've flown on planes for a long time, and I didn't know that such a thing was possible. I mean, if I overfill my car by a gallon, I'm gonna have gas all over my shoes, right?

Well, so anyway, the captain then says that we have two options: we can either wait for the truck to come suck the fuel back out of the plane, which is going to take over an hour, or twenty people have to get off the plane right now to even out the weight.

No one moves.

*Now, there's this guy across the aisle from me in first class, and he is absolutely livid. He reminds me of Frank Burns on M*A*S*H; he's just super indignant and sputtering everywhere, demanding to know who's responsible. It's an amazing showcase, it's like he's Margaret Dumont in the Marx Brothers movies.*

So, he grabs his wallet and pulls out this massive wad of cash! And he's like "I cannot be late for this meeting!! I will give $40 to any person who gets off this plane right now!"

Sure enough, people take him up on it. He gives out $40 to 20 people (which is $800 in cash, by the way!) and they all leave.

So, now we're all set and we head out to the runway, and the captain comes back on the PA again. The plane's computer has stopped working. No one knows why. Now we gotta get towed back to the gate.

Frank Burns is apoplectic. I mean, seriously, I thought he was gonna have a stroke. He's cursing and screaming. Everyone else is just looking at each other.

We get back to the gate and this guy is demanding another flight. They offer to book him on the 9:30, which is too late. He's like, "Isn't there another flight before 9:30?"

The gate agent is like, "Well, there was another flight at 8, but it's all full now. They're closing the doors now."

And he's like, "Full?! Whaddya mean it's full? There's not one open seat on that plane?!?!?!"

The gate agent is like "No sir, that plane was wide open until 20 passengers showed up out of nowhere and took all the seats. They were the happiest passengers I've ever seen, they were laughing all the way down the jet bridge."

It was a very quiet ride on the 9:30 flight.

This story is, of course, about trade-offs. Although most of this book focuses on various technical trade-offs in engineering systems, it turns out that trade-offs also apply to human behaviors. As a leader, you need to make decisions about what your teams should do each week. Sometimes the trade-offs are obvious ("if we work on this project, it delays that other one…"); sometimes the trade-offs have unforeseeable consequences that can come back to bite you, as in the preceding story.

At the highest level, your job as a leader—either of a single team or a larger organization—is to guide people toward solving difficult, ambiguous problems. By *ambiguous*, we mean that the problem has no obvious solution and might even be unsolvable. Either way, the problem needs to be explored, navigated, and (hopefully) wrestled into a state in which it's under control. If writing code is analogous to chopping down trees, your job as a leader is to "see the forest through the trees" and find a workable path through that forest, directing engineers toward the important trees. There are three main steps to this process. First, you need to identify the *blinders*; next, you need to identify the *trade-offs*; and then you need to *decide* and iterate on a solution.

Identify the Blinders

When you first approach a problem, you'll often discover that a group of people has already been wrestling with it for years. These folks have been steeped in the problem for so long that they're wearing "blinders"—that is, they're no longer able to see the forest. They make a bunch of assumptions about the problem (or solution) without realizing it. "This is how we've always done it," they'll say, having lost the ability to consider the status quo critically. Sometimes, you'll discover bizarre coping mechanisms or rationalizations that have evolved to justify the status quo. This is where

you—with fresh eyes—have a great advantage. You can see these blinders, ask questions, and then consider new strategies. (Of course, being unfamiliar with the problem isn't a requirement for good leadership, but it's often an advantage.)

Identify the Key Trade-Offs

By definition, important and ambiguous problems do *not* have magic "silver bullet" solutions. There's no answer that works forever in all situations. There is only the *best answer for the moment*, and it almost certainly involves making trade-offs in one direction or another. It's your job to call out the trade-offs, explain them to everyone, and then help decide how to balance them.

Decide, Then Iterate

After you understand the trade-offs and how they work, you're empowered. You can use this information to make the best decision for this particular month. Next month, you might need to reevaluate and rebalance the trade-offs again; it's an iterative process. This is what we mean when we say *Always Be Deciding*.

There's a risk here. If you don't frame your process as continuous rebalancing of trade-offs, your teams are likely to fall into the trap of searching for the perfect solution, which can then lead to what some call "analysis paralysis." You need to make your teams comfortable with iteration. One way of doing this is to lower the stakes and calm nerves by explaining: "We're going to try this decision and see how it goes. Next month, we can undo the change or make a different decision." This keeps folks flexible and in a state of learning from their choices.

Case Study: Addressing the "Latency" of Web Search

In managing a team of teams, there's a natural tendency to move away from a single product and to instead own a whole "class" of products, or perhaps a broader problem that crosses products. A good example of this at Google has to do with our oldest product, Web Search.

For years, thousands of Google engineers have worked on the general problem of making search results better—improving the "quality" of the results page. But it turns out that this quest for quality has a side effect: it gradually makes the product slower. Once upon a time, Google's search results were not much more than a page of 10 blue links, each representing a relevant website. Over the past decade, however, thousands of tiny changes to improve "quality" have resulted in ever-richer results: images, videos, boxes with Wikipedia facts, even interactive UI elements. This means the servers need to do much more work to generate information: more bytes are being sent over the wire; the client (usually a phone) is being asked to render ever-more-complex HTML and data. Even though the speed of networks and computers have markedly increased over a decade, the speed of the search page has become slower and slower: its *latency* has increased. This might not seem like a big deal, but the latency of a product has a direct effect (in aggregate) on users' engagement and how often they use it. Even increases in rendering time as small as 10 ms matter. Latency creeps up slowly. This is not the fault of a specific engineering team, but rather represents a long, collective poisoning of the commons. At some point, the overall latency of Web Search grows until its effect begins to cancel out the improvements in user engagement that came from the improvements to the "quality" of the results.

A number of leaders struggled with this issue over the years, but failed to address the problem systematically. The blinders everyone wore assumed that the only way to deal with latency was to declare a latency "code yellow"[1] every two or three years, during which everyone dropped everything to optimize code and speed up the product. Although this strategy would work temporarily, the latency would begin creeping up again just a month or two later, and soon return to its prior levels.

So what changed? At some point, we took a step back, identified the blinders, and did a full reevaluation of the trade-offs. It turns out that the pursuit of "quality" has not one, but *two* different costs. The first cost is to the user: more quality usually means more data being sent out, which means more latency. The second cost is to Google: more quality means doing more work to generate the data, which costs more CPU time in our servers—what we call "serving capacity." Although leadership had often trodden carefully around the trade-off between quality and capacity, it had never treated latency as a full citizen in the calculus. As the old joke goes, "Good, Fast, Cheap—pick two." A simple way to depict the trade-offs is to draw a triangle of tension between Good (Quality), Fast (Latency), and Cheap (Capacity), as illustrated in Figure 6-1.

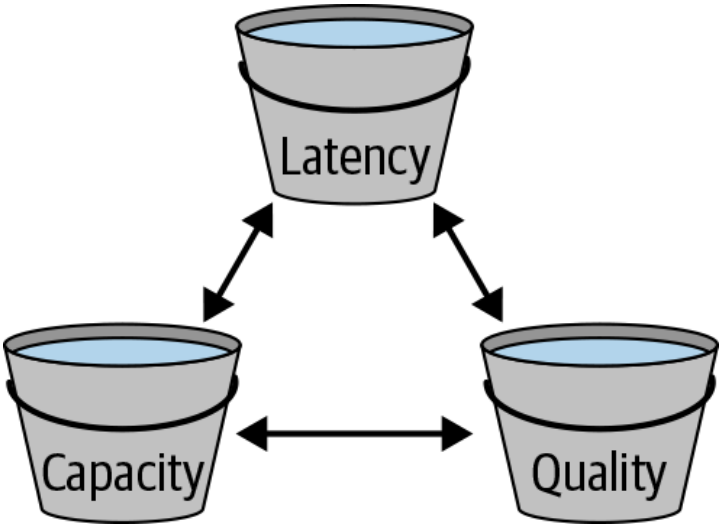

Figure 6-1. Trade-offs within Web Search; pick two!

That's exactly what was happening here. It's easy to improve any one of these traits by deliberately harming at least one of the other two. For example, you can improve quality by putting more data on the search results page—but it doing so will hurt capacity and latency. You can also do a direct trade-off between latency and capacity by changing the traffic load on your serving cluster. If you send more queries to the cluster, you get increased capacity in the sense that you get better utilization of the CPUs—more bang for your hardware buck. But higher load increases resource contention within a computer, making the average latency of a query worse. If you deliberately decrease a cluster's traffic (run it "cooler"), you have less serving capacity overall, but each query becomes faster.

The main point here is that this insight—a better understanding of *all* the trade-offs—allowed us to start experimenting with new ways of balancing. Instead of treating latency as an unavoidable and accidental side effect, we could now treat it as a first-class goal along with our other goals. This led to new strategies for us. For example, our data scientists were able to measure exactly how much latency hurt user engagement. This allowed them to construct a metric that pitted quality-driven improvements to short-term user engagement against latency-driven damage to long-term user engagement. This approach allows us to make more data-driven decisions about product changes. For example, if a small change improves quality, but also hurts latency, we can quantitatively decide whether the change is worth launching or not. We are *always deciding* whether our quality, latency, and capacity changes are in balance, and iterating on our decisions every month.

Always Be Leaving

At face value, *Always Be Leaving* sounds like terrible advice. Why would a good leader be trying to leave? In fact, this is a famous quote from Bharat Mediratta, a former Google engineering director. What he meant was that it's not just your job to solve an ambiguous problem, but to get your organization to solve it *by itself*, without you present. If you can do that, it frees you up to move to a new problem (or new organization), leaving a trail of self-sufficient success in your wake.

The antipattern here, of course, is a situation in which you've set yourself up to be a single point of failure (SPOF). As we noted earlier in this book, Googlers have a term for that, the bus factor: *the number of people that need to get hit by a bus before your project is completely doomed.*

Of course, the "bus" here is just a metaphor. People become sick; they switch teams or companies; they move away. As a litmus test, think about a difficult problem that your team is making good progress on. Now imagine that you, the leader, disappear. Does your team keep going? Does it continue to be successful? Here's an even simpler test: think about the last vacation you took that was at least a week long. Did you keep checking your work email? (Most leaders do.) Ask yourself *why*. Will things fall apart if you don't pay attention? If so, you have very likely made yourself an SPOF. You need to fix that.

Your Mission: Build a "Self-Driving" Team

Coming back to Bharat's quote: being a successful leader means building an organization that is able to solve the difficult problem by itself. That organization needs to have a strong set of leaders, healthy engineering processes, and a positive, self-perpetuating culture that persists over time. Yes, this is difficult; but it gets back to the fact that leading a team of teams is often more about organizing *people* rather than being a technical wizard. Again, there are three main parts to constructing this sort of self-sufficient group: dividing the problem space, delegating subproblems, and iterating as needed.

Dividing the problem space

Challenging problems are usually composed of difficult subproblems. If you're leading a team of teams, an obvious choice is to put a team in charge of each subproblem. The risk, however, is that the subproblems can change over time, and rigid team boundaries won't be able to notice or adapt to this

fact. If you're able, consider an organizational structure that is looser—one in which subteams can change size, individuals can migrate between subteams, and the problems assigned to subteams can morph over time. This involves walking a fine line between "too rigid" and "too vague." On the one hand, you want your subteams to have a clear sense of problem, purpose, and steady accomplishment; on the other hand, people need the freedom to change direction and try new things in response to a changing environment.

EXAMPLE: SUBDIVIDING THE "LATENCY PROBLEM" OF GOOGLE SEARCH

When approaching the problem of Search latency, we realized that the problem could, at a minimum, be subdivided into two general spaces: work that addressed the *symptoms* of latency, and different work that addressed the *causes* of latency. It was obvious that we needed to staff many projects to optimize our codebase for speed, but focusing *only* on speed wouldn't be enough. There were still thousands of engineers increasing the complexity and "quality" of search results, undoing the speed improvements as quickly as they landed, so we also needed people to focus on a parallel problem space of preventing latency in the first place. We discovered gaps in our metrics, in our latency analysis tools, and in our developer education and documentation. By assigning different teams to work on latency causes and symptoms at the same time, we were able to systematically control latency over the long term. (Also, notice how these teams owned the *problems*, not specific solutions!)

DELEGATING SUBPROBLEMS TO LEADERS

It's essentially a cliché for management books to talk about "delegation," but there's a reason for that: delegation is *really difficult* to learn. It goes against all our instincts for efficiency and achievement. That difficulty is the reason for the adage, "if you want something done right, do it yourself."

That said, if you agree that your mission is to build a self-driving organization, the main mechanism of teaching is through delegation. You must build a set of self-sufficient leaders, and delegation is absolutely the most effective way to train them. You give them an assignment, let them fail, and then try again and try again. Silicon Valley has well-known mantras about "failing fast and iterating." That philosophy doesn't just apply to engineering design, but to human learning, as well.

As a leader, your plate is constantly filling up with important tasks that need to be done. Most of these tasks are things that are fairly easy for you do. Suppose that that you're working diligently through your inbox, responding

to problems, and then you decide to put 20 minutes aside to fix a longstanding and nagging issue. But before you carry out the task, be mindful and stop yourself. Ask this critical question: *Am I really the only one who can do this work?*

Sure, it might be most *efficient* for you to do it, but then you're failing to train your leaders. You're not building a self-sufficient organization. Unless the task is truly time sensitive and on fire, bite the bullet and assign the work to someone else—presumably someone who you know can do it but will probably take much longer to finish. Coach them on the work if need be. You need to create opportunities for your leaders to grow; they need to learn to "level up" and do this work themselves so that you're no longer in the critical path.

The corollary here is that you need to be mindful of your own purpose as a leader of leaders. If you find yourself deep in the weeds, you're doing a disservice to your organization. When you get to work each day, ask yourself a different critical question: *What can I do that* nobody *else on my team can do?*

There are a number of good answers. For example, you can protect your teams from organizational politics; you can give them encouragement; you can make sure everyone is treating one another well, creating a culture of humility, trust, and respect. It's also important to "manage up," making sure your management chain understands what your group is doing, and staying connected to the company at large. But often the most common and important answer to this question is: "I can see the forest through the trees." In other words, you can *define a high-level strategy*. Your strategy needs to cover not just overall technical direction, but an organizational strategy, as well. You're building a blueprint for how the ambiguous problem is solved and how your organization can manage the problem over time. You're continuously mapping out the forest, and then assigning the tree-cutting to others.

ADJUSTING AND ITERATING

Let's assume that you've now reached the point at which you've built a self-sustaining machine. You're no longer an SPOF. Congratulations! What do you do now?

Before answering, note that you have actually liberated yourself—you now have the freedom to "Always Be Leaving." This could be the freedom to tackle a new, adjacent problem, or perhaps you could even move yourself to a

whole new department and problem space, making room for the careers of the leaders you've trained. This is a great way of avoiding personal burnout.

The simple answer to "what now?" is to *direct* this machine and keep it healthy. But unless there's a crisis, you should use a gentle touch. The book *Debugging Teams*[2] has a parable about making mindful adjustments:

There's a story about a Master of all things mechanical who had long since retired. His former company was having a problem that no one could fix, so they called in the Master to see if he could help find the problem. The Master examined the machine, listened to it, and eventually pulled out a worn piece of chalk and made a small X on the side of the machine. He informed the technician that there was a loose wire that needed repair at that very spot. The technician opened the machine and tightened the loose wire, thus fixing the problem. When the Master's invoice arrived for $10,000, the irate CEO wrote back demanding a breakdown for this ridiculously high charge for a simple chalk mark! The Master responded with another invoice, showing a $1 cost for the chalk to make the mark, and $9,999 for knowing where to put it.

To us, this is a story about wisdom: that a single, carefully considered adjustment can have gigantic effects. We use this technique when managing people. We imagine our team as flying around in a great blimp, headed slowly and surely in a certain direction. Instead of micromanaging and trying to make continuous course corrections, we spend most of the week carefully watching and listening. At the end of the week we make a small chalk mark in a precise location on the blimp, then give a small but critical "tap" to adjust the course.

This is what good management is about: 95% observation and listening, and 5% making critical adjustments in just the right place. Listen to your leaders and skip-reports. Talk to your customers, and remember that often (especially if your team builds engineering infrastructure) your "customers" are not end users out in the world, but your coworkers. Customers' happiness requires just as much intense listening as your reports' happiness. What's working and what isn't? Is this self-driving blimp headed in the proper direction? Your direction should be iterative, but thoughtful and minimal, making the minimum adjustments necessary to correct course. If you regress into micromanagement, you risk becoming an SPOF again! "Always Be Leaving" is a call to *macro*management.

TAKE CARE IN ANCHORING A TEAM'S IDENTITY

A common mistake is to put a team in charge of a specific product rather than a general problem. A product is a *solution* to a problem. The life expectancy

of solutions can be short, and products can be replaced by better solutions. However, a *problem*—if chosen well—can be evergreen. Anchoring a team identity to a specific solution ("We are the team that manages the Git repositories") can lead to all sorts of angst over time. What if a large percentage of your engineers want to switch to a new version control system? The team is likely to "dig in," defend its solution, and resist change, even if this is not the best path for the organization. The team clings to its blinders, because the solution has become part of the team's identity and self-worth. If the team instead owns the *problem* (e.g., "We are the team that provides version control to the company"), it is freed up to experiment with different solutions over time.

Always Be Scaling

A lot of leadership books talk about "scaling" in the context of learning to "maximize your impact"—strategies to grow your team and influence. We're not going to discuss those things here beyond what we've already mentioned. It's probably obvious that building a self-driving organization with strong leaders is already a great recipe for growth and success.

Instead, we're going to discuss team scaling from a *defensive* and personal point of view rather than an offensive one. As a leader, *your most precious resource is your limited pool of time, attention, and energy.* If you aggressively build out your teams' responsibilities and power without learning to protect your personal sanity in the process, the scaling is doomed to fail. And so we're going to talk about how to effectively scale *yourself* through this process.

The Cycle of Success

When a team tackles a difficult problem, there's a standard pattern that emerges, a particular cycle. It looks like this:

Analysis

> First, you receive the problem and start to wrestle with it. You identify the blinders, find all the trade-offs, and build consensus about how to manage them.

Struggle

> You start moving on the work, whether or not your team thinks it's ready. You prepare for failures, retries, and iteration. At this point, your job is mostly about herding cats. Encourage your leaders and experts on the ground to form opinions and then listen carefully and devise an overall strategy, even if you have to "fake it" at first.[3]

Traction

> Eventually your team begins to figure things out. You're making smarter decisions, and real progress is made. Morale improves. You're iterating on trade-offs, and the organization is beginning to drive itself around the problem. Nice job!

Reward

> Something unexpected happens. Your manager takes you aside and congratulates you on your success. You discover your reward isn't just a pat on the back, but a *whole new problem* to tackle. That's right: the reward for success is more work…and more responsibility! Often, it's a problem that is similar or adjacent to the first one, but equally difficult.

So now you're in a pickle. You've been given a new problem, but (usually) not more people. Somehow you need to solve *both* problems now, which likely means that the original problem still needs to be managed with *half* as many people in *half* the time. You need the other half of your people to tackle the new work! We refer to this final step as the compression *stage*: you're taking everything you've been doing and compressing it down to half the size.

So really, the "cycle" of success is more of a "spiral" (see Figure 6-2). Over months and years, your organization is scaling by tackling new problems and then figuring out how to compress them so that it can take on new, parallel struggles. If you're lucky, you're allowed to hire more people as you go. More often than not, though, your hiring doesn't keep pace with the scaling. Larry Page, one of Google's founders, would probably refer to this spiral as "uncomfortably exciting."

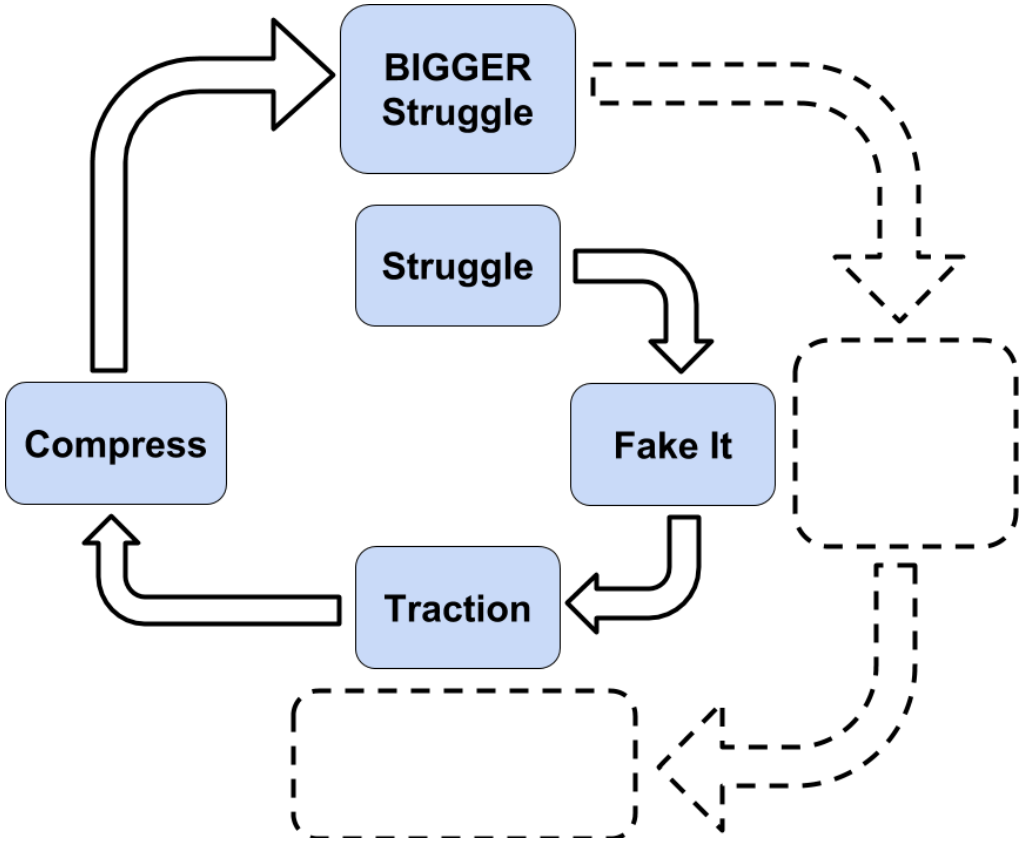

Figure 6-2. The spiral of success

The spiral of success is a conundrum—it's something that's difficult to manage, and yet it's the main paradigm for scaling a team of teams. The act of compressing a problem isn't just about figuring out how to maximize your team's efficiency, but also about learning to scale your *own* time and attention to match the new breadth of responsibility.

Important versus Urgent

Think back to a time when you weren't yet a leader, but still a carefree individual contributor. If you used to be a programmer, your life was likely calmer and more panic-free. You had a list of work to do, and each day you'd methodically work down your list, writing code and debugging problems. Prioritizing, planning, and executing your work was straightforward.

As you moved into leadership, though, you might have noticed that your main mode of work became less predictable and more about firefighting. That is, your job became less *proactive* and more *reactive*. The higher up in leadership you go, the more escalations you receive. You are the "finally" clause in a long list of code blocks! All of your means of communication—email, chat rooms, meetings—begin to feel like a Denial-of-Service attack against your time and attention. In fact, if you're not mindful, you end up

spending 100% of your time in reactive mode. People are throwing balls at you, and you're frantically jumping from one ball to the next, trying not to let any of them hit the ground.

A lot of books have discussed this problem. The management author Stephen Covey is famous for talking about the idea of distinguishing between things that are *important* versus things that are *urgent*. In fact, it was US President Dwight D. Eisenhower who popularized this idea in a famous 1954 quote:

I have two kinds of problems, the urgent and the important. The urgent are not important, and the important are never urgent.

This tension is one of the biggest dangers to your effectiveness as a leader. If you let yourself slip into pure reactive mode (which happens almost automatically), you spend every moment of your life on *urgent* things, but almost none of those things are *important* in the big picture. Remember that your job as a leader is to do things that *only you can do*, like mapping a path through the forest. Building that meta-strategy is incredibly important, but almost never urgent. It's always easier to respond to that next urgent email.

So how can you force yourself to work mostly on important things, rather than urgent things? Here are a few key techniques:

Delegate

> Many of the urgent things you see can be delegated back to other leaders in your organization. You might feel guilty if it's a trivial task; or you might worry that handing off an issue is inefficient because it might take those other leaders longer to fix. But it's good training for them, and it frees up your time to work on important things that only you can do.

Schedule dedicated time

> Regularly block out two hours or more to sit quietly and work *only* on important-but-not-urgent things—things like team strategy, career paths for your leaders, or how you plan to collaborate with neighboring teams.

Find a tracking system that works

> There are dozens of systems for tracking and prioritizing work. Some are software based (e.g., specific "to-do" tools), some are pen-and-paper based (the "Bullet Journal" method), and some systems are

agnostic to implementation. In this last category, David Allen's book, *Getting Things Done*, is quite popular among engineering managers; it's an abstract algorithm for working through tasks and maintaining a prized "inbox zero." The point here is to *try* these different systems and determine what works for you. Some of them will click with you and some will not, but you definitely need to find something more effective than tiny Post-It notes decorating your computer screen.

Learn to Drop Balls

There's one more key technique for managing your time, and on the surface it sounds radical. For many, it contradicts years of engineering instinct. As an engineer, you pay attention to detail; you make lists, you check things off lists, you're precise, and you finish what you start. That's why it feels so good to close bugs in a bug tracker, or whittle your email down to "inbox zero." But as a leader of leaders, your time and attention are under constant attack. No matter how much you try to avoid it, you end up dropping balls on the floor—there are just too many of them being thrown at you. It's overwhelming, and you probably feel guilty about this all the time.

So, at this point, let's step back and take a frank look at the situation. If dropping some number of balls is inevitable, isn't it better to drop certain balls *deliberately* rather than *accidentally*? At least then you have some semblance of control.

Here's a great way to do that.

Marie Kondo is an organizational consultant and the author of the extremely popular book *The Life-Changing Magic of Tidying Up*. Her philosophy is about effectively decluttering all of the junk from your house, but it works for abstract clutter, as well.

Think of your physical possessions as living in three piles. About 20% of your things are just useless—things that you literally never touch anymore, and all very easy to throw away. About 60% of your things are somewhat interesting; they vary in importance to you, and you sometimes use them, sometimes not. And then about 20% of your possessions are exceedingly important: these are the things you use *all* the time, that have deep emotional meaning, or, in Ms. Kondo's words, spark deep "joy" just holding them. The

thesis of her book is that most people declutter their lives incorrectly: they spend time tossing the bottom 20% in the garbage, but the remaining 80% still feels too cluttered. She argues that the *true* work of decluttering is about identifying the top 20%, not the bottom 20%. If you can identify only the critical things, you should then toss out the other 80%. It sounds extreme, but it's quite effective. It is greatly freeing to declutter so radically.

It turns out that you can also apply this philosophy to your inbox or task list—the barrage of balls being thrown at you. Divide your pile of balls into three groups: the bottom 20% are probably neither urgent nor important and very easy to delete or ignore. There's a middle 60%, which might contain some bits of urgency or importance, but it's a mixed bag. At the top, there's 20% of things that are absolutely, critically important.

And so now, as you work through your tasks, do *not* try to tackle the top 80%—you'll still end up overwhelmed and mostly working on urgent-but-not-important tasks. Instead, mindfully identify the balls that strictly fall in the top 20%—critical things that *only you can do*—and focus strictly on them. Give yourself explicit permission to drop the other 80%.

It might feel terrible to do so at first, but as you deliberately drop so many balls, you'll discover two amazing things. First, even if you don't delegate that middle 60% of tasks, your subleaders often notice and pick them up automatically. Second, if something in that middle bucket is truly critical, it ends up coming back to you anyway, eventually migrating up into the top 20%. You simply need to *trust* that things below your top-20% threshold will either be taken care of or evolve appropriately. Meanwhile, because you're focusing only on the critically important things, you're able to scale your time and attention to cover your group's ever-growing responsibilities.

Protecting Your Energy

We've talked about protecting your time and attention—but your personal energy is the other piece of the equation. All of this scaling is simply exhausting. In an environment like this, how do you stay charged and optimistic?

Part of the answer is that over time, as you grow older, your overall stamina builds up. Early in your career, working eight hours a day in an office can feel like a shock; you come home tired and dazed. But just like training for a marathon, your brain and body build up larger reserves of stamina over time.

The other key part of the answer is that leaders gradually learn to *manage* their energy more intelligently. It's something they learn to pay constant attention to. Typically, this means being aware of how much energy you have at any given moment, and making deliberate choices to "recharge" yourself at specific moments, in specific ways. Here are some great examples of mindful energy management:

Take real vacations

A weekend is not a vacation. It takes at least three days to "forget" about your work; it takes at least a week to actually feel refreshed. But if you check your work email or chats, you *ruin* the recharge. A flood of worry comes back into your mind, and all of the benefit of psychological distancing dissipates. The vacation recharges only if you are truly disciplined about disconnecting.[4] And, of course, this is possible only if you've built a self-driving organization.

Make it trivial to disconnect

When you disconnect, leave your work laptop at the office. If you have work communications on your phone, remove them. For example, if your company uses G Suite (Gmail, Google Calendar, etc.), a great trick is to install these apps in a "work profile" on your phone. This causes a second set of work-badged apps to appear on your phone. For example, you'll now have two Gmail apps: one for personal email, one for work email. On an Android phone, you can then press a single button to disable the entire work profile at once. All the work apps gray out, as if they were uninstalled, and you can't "accidentally" check work messages until you reenable the work profile.

Take real weekends, too

A weekend isn't as effective as a vacation, but it still has some rejuvenating power. Again, this recharge works only if you disconnect from work communications. Try truly signing out on Friday night, spend the weekend doing things you love, and then sign in again on Monday morning, when you're back in the office.

Take breaks during the day

Your brain operates in natural 90-minute cycles.[5] Use the opportunity to get up and walk around the office, or spend 10 minutes walking outside. Tiny breaks like this are only tiny recharges, but they can

make a tremendous difference in your stress levels and how you feel over the next two hours of work.

Give yourself permission to take a mental health day

Sometimes, for no reason, you just have a bad day. You might have slept well, eaten well, exercised—and yet you are still in a terrible mood anyway. If you're a leader, this is an awful thing. Your bad mood sets the tone for everyone around you, and it can lead to terrible decisions (emails you shouldn't have sent, overly harsh judgements, etc.) If you find yourself in this situation, just turn around and go home, declaring a sick day. Better to get nothing done that day than to do active damage.

In the end, managing your energy is just as important as managing your time. If you learn to master these things, you'll be ready to tackle the broader cycle of scaling responsibility and building a self-sufficient team.

Conclusion

Successful leaders naturally take on more responsibility as they progress (and that's a good and natural thing). Unless they effectively come up with techniques to properly make decisions quickly, delegate when needed, and manage their increased responsibility, they might end up feeling overwhelmed. Being an effective leader doesn't mean that you need to make perfect decisions, do everything yourself, or work twice as hard. Instead, strive to always be deciding, always be leaving, and always be scaling.

TL;DRs

- Always Be Deciding: Ambiguous problems have no magic answer; they're all about finding the right *trade-offs* of the moment, and iterating.
- Always Be Leaving: Your job, as a leader, is to build an organization that automatically solves a class of ambiguous problems—over *time*—without you needing to be present.
- Always Be Scaling: Success generates more responsibility over time, and you must proactively manage the *scaling* of this work in order to protect your scarce resources of personal time, attention, and energy.

1 “Code Yellow” is Google’s term for “emergency hackathon to fix a critical problem.” Affected teams are expected to suspend all work and focus 100% attention on the problem until the state of emergency is declared over.

2 Fitzpatrick, Brian W. and Ben Collins-Sussman. 2016. *Debugging Teams: Better Productivity through Collaboration*. O’Reilly Media, Inc.

3 It’s easy for *imposter syndrome* to kick in at this point. One technique for fighting the feeling that you don’t know what you’re doing is to simply pretend that *some* expert out there knows exactly what to do, and that they’re simply on vacation and you’re temporarily subbing in for them. It’s a great way to remove the personal stakes and give yourself permission to fail and learn.

4 You need to plan ahead and build around the assumption that your work simply won’t get done during vacation. Working hard (or smart) just before and after your vacation mitigates this issue.

5 You can read more about BRAC at *https://en.wikipedia.org/wiki/Basic_rest-activity_cycle*

Chapter 7. Measuring Engineering Productivity

Written by Ciera Jaspen

Edited by Riona Macnamara

Google is a data-driven company. We back up most of our products and design decisions with hard data. The culture of data-driven decision making, using appropriate metrics, has some drawbacks, but overall, relying on data tends to make most decisions objective rather than subjective, which is often a good thing. Collecting and analyzing data on the human side of things, however, has its own challenges. Specifically, within software engineering, Google has found that having a team of specialists focus on engineering productivity itself to be very valuable and important as the company scales and can leverage insights from such a team.

Why Should We Measure Engineering Productivity?

Let's presume that you have a thriving business (i.e., you run an online search engine), and you want to increase your business's scope (enter into the enterprise application market, or the cloud market, or the mobile market). Presumably, to increase the scope of your business, you'll need to also increase the size of your engineering organization. However, as organizations grow in size linearly, communication costs grow quadratically.[1] Adding more people will be necessary to increase the scope of your business, but the communication overhead costs will not scale linearly as you add additional personnel. As a result, you won't be able to scale the scope of your business linearly to the size of your engineering organization.

There is another way to address our scaling problem, though: *we could make each individual more productive*. If we can increase the productivity of individual engineers in the organization, we can increase the scope of our business without the commensurate increase in communication overhead.

Google has had to grow quickly into new businesses, which has meant learning how to make our engineers more productive. To do this, we needed to understand what makes them productive, identify inefficiencies in our

engineering processes, and fix the identified problems. Then, we would repeat the cycle as needed in a continuous improvement loop. By doing this, we would be able to scale our engineering organization with the increased demand on it.

However, this improvement cycle *also* takes human resources. It would not be worthwhile to improve the productivity of your engineering organization by the equivalent of 10 engineers per year if it took 50 engineers per year to understand and fix productivity blockers. *Therefore, our goal is to not only improve software engineering productivity, but to do so efficiently.*

At Google, we addressed these trade-offs by creating a team of researchers dedicated to understanding engineering productivity. Our research team includes people from the software engineering research field and generalist software engineers, but we also include social scientists from a variety of fields, including cognitive psychology and behavioral economics. The addition of people from the social sciences allows us to not only study the software artifacts that engineers produce, but to also understand the human-side of software development, including personal motivations, incentive structures, and strategies for managing complex tasks. The goal of the team is to take a data-driven approach to measuring and improving engineering productivity.

In this chapter, we walk through how our research team achieves this goal. This begins with the triage process: there are many parts of software development that we *can* measure, but what *should* we measure? After a project is selected, we walk through how the research team identifies meaningful metrics that will identify the problematic parts of the process. Finally, we look at how Google uses these metrics to track improvements to productivity.

For this chapter, we follow one concrete example posed by the C++ and Java language teams at Google: readability. For most of Google's existence, these teams have managed the "readability process" at Google. (For more on readability, see Chapter 3) The readability process was put in place in the early days at Google, before automatic formatters Chapter 8 and linters that block submission were commonplace (Chapter 9). The process itself is expensive to run because it requires hundreds of engineers performing "readability reviews" for engineers in order to grant readability to them. Some engineers viewed it as an archaic hazing process that no longer held utility, and it was a favorite topic to argue about around the lunch table. The

concrete question from the language teams was this: is the time spent on the readability process worthwhile?

Triage: Is It Even Worth Measuring?

Before we decide how to measure the productivity of engineers, we need to know when a metric is even worth measuring. The measurement itself is expensive: it takes people to measure the process, analyze the results, and disseminate them to the rest of the company. Furthermore, the measurement process itself might be onerous and slow down the rest of the engineering organization. Even if it is not slow, tracking progress might change engineer's behavior, possibly in ways that mask the underlying issues. We need to measure and estimate smartly; although we don't want to guess, we shouldn't waste time and resources measuring unnecessarily.

At Google, we've come up with a series of questions to help teams determine whether it's even worth measuring productivity in the first place. We first ask people to describe what they want to measure in the form of a concrete question; we find that the more concrete people can make this question, the more likely they are to derive benefit from the process. When the readability team approached us, its question was simple: are the costs of an engineer going through the readability process worth the benefits they might be deriving for the company?

We then ask them to consider the following aspects of their question:

1. *What result are you expecting, and why?* Even though we might like to pretend that we are neutral investigators, we are not. We do have preconceived notions about what ought to happen. By acknowledging this at the outset, we can try to address these biases and prevent *post hoc* explanations of the results.

   When this question was posed to the readability team, it noted that it was not sure. People were certain the costs had been worth the benefits at one point in time, but with the advent of autoformatters and static analysis tools, no one was entirely certain. There was a growing belief that the process now served as a hazing ritual. Although it might still provide engineers with benefits (and they had survey data showing that people did claim these benefits), it was not clear whether it was worth the time commitment of the authors or the reviewers of the code.
1. *If the data supports your expected result, what action will be taken?* We ask this because if no action will be taken, there is no point

in measuring. Notice that an action might in fact be "maintain the status quo" if there is a planned change that will occur if we didn't have this result.

When asked about this, the answer from the readability team was straightforward: if the benefit was enough to justify the costs of the process, they would link to the research and the data on the FAQ about readability and advertise it to set expectations.

1. *If we get a negative result, will appropriate action be taken?* We ask this question because in many cases, we find that a negative result will not change a decision. There might be other inputs into a decision that would override any negative result. If that is the case, it might not be worth measuring in the first place. This is the question that stops most of the projects that our research team takes on; we learn that the decision makers were interested in knowing the results, but for other reasons, they will not choose to change course.

   In the case of readability, however, we had a strong statement of action from the team. It committed that if our analysis showed that the costs either outweighed the benefit or the benefits were negligible, the team would kill the process. As different programming languages have different levels of maturity in formatters and static analyses, this evaluation would happen on a per-language basis.

1. *Who is going to decide to take action on the result, and when would they do it?* We ask this to ensure that the person requesting the measurement is the one who is empowered to take action (or is doing so directly on their behalf). Ultimately, the goal of measuring our software process is to help people make business decisions. It's important to understand who that individual is, including what form of data convinces them. Although the best research includes a variety of approaches (everything from structured interviews to statistical analyses of logs), there might be limited time in which to provide decision makers with the data they need. In those cases, it might be best to cater to the decision maker. Do they tend to make decisions by empathizing through the stories that can be retrieved from interviews?[2] Do they trust survey results or logs data? Do they feel comfortable with complex statistical analyses? If the decider doesn't believe the form of the result in principle, there is again no point in measuring the process.
   In the case of readability, we had a clear decision maker for each programming language. Two language teams, Java and C++, actively reached out to us for assistance, and the others were waiting to see what happened with those languages first.[3] The decision makers trusted

engineers' self-reported experiences for understanding happiness and learning, but the decision makers wanted to see "hard numbers" based on logs data for velocity and code quality. This meant that we needed to include both qualitative and quantitative analysis for these metrics. There was not a hard deadline for this work, but there was an internal conference that would make for a useful time for an announcement if there was going to be a change. That deadline gave us several months in which to complete the work

By asking these questions, we find that in many cases, measurement is simply not worthwhile…and that's ok! There are many good reasons to not measure the impact of a tool or process on productivity. Here are some examples that we've seen:

You can't afford to change the process/tools right now

There might be time constraints or financial constraints that prevent this. For example, you might determine that if only you switched to a faster build tool, it would save hours of time every week. However, the switchover will mean pausing development while everyone converts over, and there's a major funding deadline approaching such that you cannot afford the interruption. Engineering trade-offs are not evaluated in a vacuum—in a case like this, it's important to realize that the broader context completely justifies delaying action on a result.

Any results will soon be invalidated by other factors

Examples here might include measuring the software process of an organization just before a planned reorganization. Or measuring the amount of technical debt for a deprecated system.

The decision maker has strong opinions, and you are unlikely to be able to provide a large enough body of evidence, of the right type, to change their beliefs

This comes down to knowing your audience. Even at Google, we sometimes find people who have unwavering beliefs on a topic due to their past experiences. We have found stakeholders who never trust survey data because they do not believe self-reports. We've also found stakeholders who are swayed best by a compelling narrative that was informed by a small number of interviews. And, of course, there are stakeholders who are swayed only by logs analysis. In all cases, we attempt to triangulate on the truth using mixed-methods, but if a stakeholder is limited to believing only in methods that are not appropriate to the problem, there is no point in doing the work.

The results will be used only as vanity metrics to support something you were going to do anyway

> This is perhaps the most common reason we tell people at Google not to measure a software process. Many times, people have planned a decision for multiple reasons, and improving the software development process is only one benefit of several. For example, the release tool team at Google once requested a measurement to a planned change to the release workflow system. Due to the nature of the change, it was obvious that the change would not be worse than the current state, but they didn't know if it was a minor improvement or a large one. We asked the team, if it turns out to only be a minor improvement, would you spend the resources to implement the feature anyway, even if it didn't look to be worth the investment? The answer was yes! The feature happened to improve productivity, but this was a side effect: it was also more performant and lowered the release tool team's maintenance burden.

The only metrics available are not precise enough to measure the problem and can be confounded by other factors

> In some cases, the metrics needed (see the upcoming section on how to identify metrics) are simply unavailable. In these cases, it can be tempting to measure using other metrics that are less precise (lines of code written, for example). However, any results from these metrics will be uninterpretable. If the metric confirms the stakeholders' preexisting beliefs, they might end up proceeding with their plan without consideration that the metric is not an accurate measure. If it does not confirm their beliefs, the imprecision of the metric itself provides an easy explanation, and the stakeholder might, again, proceed with their plan.

When you are successful at measuring your software process, you aren't setting out to prove a hypothesis correct or incorrect; *success means giving a stakeholder the data they need to make a decision*. If that stakeholder won't use the data, the project is always a failure. We should only measure a software process when a concrete decision will be made based on the outcome. For the readability team, there was a clear decision to be made. If the metrics showed the process to be beneficial, they would publicize the result. If not, the process would be abolished. Most important, the readability team had the authority to make this decision.

Selecting Meaningful Metrics with Goals and Signals

After we decide to measure a software process, we need to determine what metrics to use. Clearly, lines of code (LOC) won't do,[4] but how do we actually measure engineering productivity?

At Google, we use the Goals/Signals/Metrics (GSM) framework to guide metrics creation.

- A *goal* is a desired end result. It's phrased in terms of what you want to understand at a high level and should not contain references to specific ways to measure it.
- A signal is how you might know that you've achieved the end result. Signals are things we would *like* to measure, but they might not be measurable themselves.
- A *metric* is proxy for a signal. It is the thing we actually can measure. It might not be the ideal measurement, but it is something that we believe is close enough.

The GSM framework encourages several desirable properties when creating metrics. First, by creating goals first, then signals, and finally metrics, it prevents the *streetlight effect*. The term comes from the full phrase "looking for your keys under the streetlight"; if you look only where you can see, you might not be looking in the right place. With metrics, this occurs when we use the metrics that we have easily accessible and that are easy to measure, regardless of whether that metric suits our needs. Instead, GSM forces us to think about which metrics will actually help us achieve our goals, rather than simply what we have readily available.

Second, GSM helps prevent both metrics creep and metrics bias by encouraging us to come up with the appropriate set of metrics, using a principled approach, *in advance* of actually measuring the result. Consider the case in which we select metrics without a principled approach and then the results do not meet our stakeholders expectations. At that point, we run the risk that stakeholders will propose that we use different metrics that they believe will produce the desired result. And because we didn't select based on a principled approach at the start, there's no reason to say that they're wrong! Instead, GSM encourages us to select metrics based on their ability to measure the original goals. Stakeholders can easily see that these metrics map to their original goals and agree, in advance, that this is the best set of metrics for measuring the outcomes.

Finally, GSM can show us where we have measurement coverage and where we do not. When we run through the GSM process, we list all our goals and create signals for each one. As we will see in the examples, not all signals are going to be measurable—and that's ok! With GSM, at least we have identified what is not measurable. By identifying these missing metrics, we can assess whether it is worth creating new metrics or even worth measuring at all.

The important thing is to maintain *traceability*. For each metric, we should be able to trace back to the signal that it is meant to be a proxy for and to the goal it is trying to measure. This ensures that we know which metrics we are measuring and why we are measuring them.

Goals

A goal should be written in terms of a desired property, without reference to any metric. By themselves, these goals are not measurable, but a good set of goals is something that everyone can agree on before proceeding onto signals and then metrics.

To make this work, we need to have identified the correct set of goals to measure in the first place. This would seem straightforward: surely the team knows the goals of their work! However, our research team has found that in many cases, people forget to include all the possible *trade-offs within productivity*, which could lead to mismeasurement.

Taking the readability example, let's assume that the team was so focused on making the readability process fast and easy that it had forgotten the goal about code quality. The team set up tracking measurements for how long it takes to get through the review process and how happy engineers are with the process. One of our teammates proposes the following:

I can make your review velocity very fast: just remove code reviews entirely.

Although this is obviously an extreme example, teams forget core trade-offs all the time when measuring: they become so focused on improving velocity that they forget to measure quality (or vice versa). To combat this, our research team divides productivity into five core components. These five components are in trade-off with one another, and we encourage teams to consider goals in each of these components to ensure that they are not inadvertently improving one while driving others downward. To help people remember all five components, we use the mnemonic "QUANTS":

***Qu**ality of the code*

What is the quality of the code produced? Are the test cases good enough to prevent regressions? How good is an architecture at mitigating risk and changes?

***A**ttention from engineers*

How frequently do engineers reach a state of flow? How much are they distracted by notifications? Does a tool encourage engineers to context switch?

***In**tellectual complexity*

How much cognitive load is required to complete a task? What is the inherent complexity of the problem being solved? Do engineers need to deal with unnecessary complexity?

***T**empo and velocity*

How quickly can engineers accomplish their tasks? How fast can they push their releases out? How many tasks do they complete in a given timeframe?

***S**atisfaction*

How happy are engineers with their tools? How well does a tool meet engineers needs? How satisfied are they with their work and their end product? Are engineers feeling burned out?

Going back to the readability example, our research team worked with the readability team to identify several productivity goals of the readability process:

Quality

Engineers write higher quality code as a result of the readability process; they write more consistent code as a result of the readability process; and they contribute to a culture of code health as a result of the readability process.

Attention

We did not have any attention goal for readability. This is ok! Not all questions about engineering productivity involve tradeoffs in all 5 areas.

Intellectual complexity

Engineers learn about the Google codebase and best coding practices as a result of the readability process, and they receive mentoring during the readability process.

Tempo and velocity

Engineers complete work tasks faster and more efficiently as a result of the readability process.

Satisfaction

Engineers see the benefit of the readability process and have positive feelings about participating in it.

Signals

A signal is the way in which we will know we've achieved our goal. Not all signals are measurable, but that's acceptable at this stage. There is not a 1:1 relationship between signals and goals. Every goal should have at least one signal, but they might have more. Some goals might also share a signal. Table 7-1 shows some example signals for the goals of the readability process measurement.

Goals	Signals
Engineers write higher quality code as a result of the readability process.	Engineers who have been granted readability judge their code to be of higher quality than engineers who have not been granted readability. The readability process has a positive impact on code quality.
Engineers learn about the Google codebase and best coding practices as a result of the readability process.	Engineers report learning from the readability process.
Engineers receive mentoring during the readability process.	Engineers report positive interactions with experienced Google engineers who serve as reviewers during the readability process.
Engineers complete work tasks faster and more efficiently as a result of the readability process.	Engineers who have been granted readability judge themselves to be more productive than engineers who have not been granted readability. Changes written by engineers who have been granted readability are faster to review than changes written by engineers who have not been granted readability.

Goals	Signals
Engineers see the benefit of the readability process and have positive feelings about participating in it.	Engineers view the readability process as being worthwhile.

Table 7-1. Signals and Goals

Metrics

Metrics are where we finally determine how we will measure the signal. Metrics are not the signal themselves; they are the measurable proxy of the signal. Because they are a proxy, they might not be a perfect measurement. For this reason, some signals might have multiple metrics as we try to triangulate on the underlying signal.

For example, to measure whether engineer's code is reviewed faster after readability, we might use a combination of both survey data and logs data. Neither of these metrics really provide the underlying truth (human perceptions are fallible, and logs metrics might not be measuring the entire picture of the time an engineer spends reviewing a piece of code or can be confounded by factors unknown at the time, like the size or difficulty of a code change). However, if these metrics show different results, it signals that possibly one of them is incorrect and we need to explore further. If they are the same, we have more confidence that we have reached some kind of truth.

Additionally, some signals might not have any associated metric because the signal might simply be unmeasurable at this time. Consider, for example, measuring code quality. Although academic literature has proposed many proxies for code quality, none of them have truly captured it. For readability, we had a decision of either using a poor proxy and possibly making a decision based on it, or simply acknowledging that this is a point that cannot currently be measured. Ultimately, we decided not to capture this as a quantitative measure, though we did ask engineers to self-rate their code quality.

Following the GSM framework is a great way to clarify the goals for why you are measuring your software process and how it will actually be measured. However, it's still possible that the metrics selected are not telling

the complete story because they are not capturing the desired signal. At Google, we use qualitative data to validate our metrics and ensure that they are capturing the intended signal.

Using Data to Validate Metrics

As an example, we once created a metric for measuring each engineer's median build latency; the goal was to capture the "typical experience" of engineers' build latencies. We then ran an *experience sampling study*. In this style of study, engineers are interrupted in context of doing a task of interest to answer a few questions. After an engineer started a build, we automatically sent them a small survey about their experiences and expectations of build latency. However, in a few cases, the engineers responded that they had not started a build! It turned out that automated tools were starting up builds, but the engineers were not blocked on these results and so it didn't "count" toward their "typical experience." We then adjusted the metric to exclude such builds.[5]

Quantitative metrics are useful because they give you power and scale. You can measure the experience of engineers across the entire company, over a large period of time, and have confidence in the results. However, they don't provide any context or narrative. Quantitative metrics don't explain why an engineer chose to use an antiquated tool to accomplish their task, or why they took an unusual workflow, or why they circumvented a standard process. Only qualitative studies can provide this information, and only qualitative studies can then provide insight on the next steps to improve a process.

Consider now the signals presented in Table 7-2. What metrics might you create to measure each of those? Some of these signals might be measurable by analyzing tool and code logs. Others are measurable only by directly asking engineers. Still others might not be perfectly measurable—how do we truly measure code quality, for example?

Ultimately, when evaluating the impact of readability on productivity, we ended up with a combination of metrics from three sources. First, we had a survey that was specifically about the readability process. This survey was given to people after they completed the process; this allowed us to get their immediate feedback about the process. This hopefully avoids recall bias,[6] but it does introduce both recency bias[7] and sampling bias.[8] Second, we used a large-scale quarterly survey to track items that were not specifically about readability; instead, they were purely about metrics that we expected

readability should affect. Finally, we used fine-grained logs metrics from our developer tools to determine how much time the logs claimed it took engineers to complete specific tasks.[9] Table 7-2 presents the complete list of metrics, with their corresponding signals and goals.

QUANTS	Goal	Signal	Metric
Quality	Engineers write higher quality code as a result of the readability process.	Engineers who have been granted readability judge their code to be of higher quality than engineers who have not been granted readability.	Quarterly Survey: Proportion of engineers who report being satisfied with the quality of their own code
		The readability process has a positive impact on code quality.	Readability Survey: Proportion of engineers reporting that readability reviews have no impact or negative impact on code quality
			Readability Survey: Proportion of engineers reporting that participating in the readability process has improved code quality for their team
	Engineers write more consistent code as a result of the readability process.	Engineers are given consistent feedback and direction in code reviews by readability reviewers as a part of the readability process.	Readability Survey: Proportion of engineers reporting inconsistency in readability reviewers' comments and readability criteria.

QUANTS	Goal	Signal	Metric
	Engineers contribute to a culture of code health as a result of the readability process.	Engineers who have been granted readability regularly comment on style and/or readability issues in code reviews.	Readability Survey: Proportion of engineers reporting that they regularly comment on style and/or readability issues in code reviews
Attention	n/a	n/a	n/a
iNtellectual	Engineers learn about the Google codebase and best coding practices as a result of the readability process.	Engineers report learning from the readability process.	Readability Survey: Proportion of engineers reporting that they learned about four relevant topics
			Readability Survey: Proportion of engineers reporting that learning or gaining expertise was a strength of the readability process
	Engineers receive mentoring during the readability process.	Engineers report positive interactions with experienced Google engineers who serve as	Readability Survey: Proportion of engineers reporting that working with readability reviewers was a strength of the readability process

QUANTS	Goal	Signal	Metric
		reviewers during the readability process.	
Tempo/ velocity	Engineers are more productive as a result of the readability process.	Engineers who have been granted readability judge themselves to be more productive than engineers who have not been granted readability.	Quarterly Survey: Proportion of engineers reporting that they're highly productive
		Engineers report that completing the readability process positively affects their engineering velocity.	Readability Survey: Proportion of engineers reporting that *not* having readability reduces team engineering velocity
		Changelists (CLs) written by engineers who have been granted readability are faster to review than CLs written by engineers who have not been granted readability.	Logs data: Median review time for CLs from authors with readability and without readability
		CLs written by engineers who have been granted readability are easier to shepherd through code	Logs data: Median shepherding time for CLs from authors with readability and without readability

QUANTS	Goal	Signal	Metric
		review than CLs written by engineers who have not been granted readability.	
		CLs written by engineers who have been granted readability are faster to get through code review CLs written by engineers who have not been granted readability.	Logs data: Median time to submit for CLs from authors with readability and without readability
		The readability process does not have a negative impact on engineering velocity	Readability Survey: Proportion of engineers reporting that the readability process negatively impacts their velocity
			Readability Survey: Proportion of engineers reporting that readability reviewers responded promptly
			Readability Survey: Proportion of engineers reporting that timeliness of reviews was a strength of the readability process

QUANTS	Goal	Signal	Metric
Satisfaction	Engineers see the benefit of the readability process and have positive feelings about participating in it.	Engineers view the readability process as being an overall positive experience.	Readability Survey: Proportion of engineers reporting that their experience with the readability process was positive overall
		Engineers view the readability process as being worthwhile	Readability Survey: Proportion of engineers reporting that the readability process is worthwhile
			Readability Survey: Proportion of engineers reporting that the quality of readability reviews is a strength of the process
			Readability Survey: Proportion of engineers reporting that thoroughness is a strength of the process
		Engineers do not view the readability process as frustrating.	Readability Survey: Proportion of engineers reporting that the readability process is uncertain, unclear, slow, or frustrating
			Quarterly Survey: Proportion of engineers

QUANTS	Goal	Signal	Metric
			reporting that they're satisfied with their own engineering velocity

Table 7-2. Goals, Signals, and Metrics

Taking Action and Tracking Results

Recall our original goal in this chapter: we want to take action and improve productivity. After performing research on a topic, the team at Google always prepares a list of recommendations for how we can continue to improve. We might suggest new features to a tool, improving latency of a tool, improving documentation, removing obsolete processes, or even changing the incentive structures for the engineers. Ideally, these recommendations are "tool-driven": it does no good to tell engineers to change their process or way of thinking if the tools do not support them in doing so. We instead always assume that engineers will make the appropriate trade-offs if they have the proper data available and the suitable tools at their disposal.

For readability, our study showed that it was overall worthwhile: engineers who had achieved readability were satisfied with the process and felt they learned from it. Our logs showed that they also had their code reviewed faster and submitted it faster, even accounting for no longer needing as many reviewers. Our study also showed places for improvement with the process: engineers identified pain points that would have made the process faster or more pleasant. The language teams took these recommendations and improved the tooling and process to make it faster and to be more transparent so that engineers would have a more pleasant experience.

Conclusion

At Google, we've found that staffing a team of engineering productivity specialists has widespread benefits to software engineering; rather than relying on each team to chart its own course to increase productivity, a centralized team can focus on broad-based solutions to complex problems. Such "human-based" factors are notoriously difficult to measure and it is important for experts to understand the data being analyzed given that many of the trade-offs involved in changing engineering processes are difficult to

measure accurately and often have unintended consequences. Such a team must remain data driven and aim to eliminate subjective bias.

TL;DRs

- Before measuring productivity, ask whether the result is actionable, regardless if the result is positive or negative. If you can't do anything with the result, it is likely not worth measuring.
- Select meaningful metrics using the GSM framework. A good metric is a reasonable proxy to the signal you're trying to measure, and it is traceable back to your original goals.
- Select metrics that cover all parts of productivity (QUANTS). By doing this, you ensure that you aren't improving one aspect of productivity (like developer velocity) and the cost of another (like code quality).
- Qualitative metrics are metrics, too! Consider having a survey mechanism for tracking longitudinal metrics about engineers' beliefs. Qualitative metrics should also align with the quantitative metrics; if they do not, it is likely the quantitative metrics that are incorrect.
- Aim to create recommendations that are built into the developer workflow and incentive structures. Even though it is sometimes necessary to recommend additional training or documentation, change is more likely to occur if it is built into the developer's daily habits.

1 Brooks, Frederick P. 1995. *The Mythical Man Month*. Addison-Wesley Professional.

2 It's worth pointing out here that our industry currently disparages "anecdata" and everyone has a goal of being "data driven." Yet anecdotes continue to exist because they are powerful. An anecdote can provide context and narrative that raw numbers cannot; it can provide a deep explanation that resonates with others because it mirrors personal experience. Although our researchers do not make decisions on anecdotes, we do use and encourage techniques such as structured interviews and case studies to deeply understand phenomena and provide context to quantitative data.

3 Java and C++ have the greatest amount of tooling support. Both have mature formatters and static analysis tools that catch common mistakes. Both are also heavily funded internally. Even though other language teams, like Python, were interested in the results, clearly there was not going to be a

benefit for Python to remove readability if we couldn't even show the same benefit for Java or C++.

4 "From there it is only a small step to measuring 'programmer productivity' in terms of 'number of lines of code produced per month'. This is a very costly measuring unit because it encourages the writing of insipid code, but today I am less interested in how foolish a unit it is from even a pure business point of view. My point today is that, if we wish to count lines of code, we should not regard them as 'lines produced' but as 'lines spent': the current conventional wisdom is so foolish as to book that count on the wrong side of the ledger." Edsger Dijkstra, on the cruelty of really teaching computing science, EWD Manuscript 1036.

5 It has routinely been our experience at Google that when the quantitative and qualitative metrics disagree, it was because the quantitative metrics were not capturing the expected result.

6 Recall bias is the bias from memory. People are more likely to recall events that are particularly interesting or frustrating.

7 Recency bias is another form of bias from memory in which people are biased toward their most recent experience. In this case, as they just successfully completed the process, they might be feeling particularly good about it.

8 Because we asked only those people who completed the process, we aren't capturing the opinions of those who did not complete the process.

9 There is a temptation to use such metrics to evaluate individual engineers, or perhaps even to identify high and low performers. Doing so would be counterproductive, though. If productivity metrics are used for performance reviews, engineers will be quick to game the metrics, and they will no longer be useful for measuring and improving productivity across the organization. The only way to make these measurements work is to let go of the idea of measuring individuals, and embrace measuring the aggregate effect.

Part III. Processes

Chapter 8. Style Guides and Rules

Written by Shaindel Schwartz

Edited by Tom Manshreck

Most engineering organizations have rules governing their codebases—rules about where source files are stored, rules about the formatting of the code, rules about naming and patterns and exceptions and threads. Most software engineers are working within the bounds of a set of policies that control how they operate. At Google, to manage our codebase, we maintain a set of style guides that define our rules.

Rules are laws. They are not just suggestions or recommendations, but strict, mandatory laws. As such, they are universally enforceable—rules may not be disregarded except as approved on a need-to-use basis. In contrast to rules, guidance provides recommendations and best practices. These bits are good to follow, even highly advisable to follow, but unlike rules, they usually have some room for variance.

We collect the rules that we define, the do's and don'ts of writing code that must be followed, in our programming style guides, which are treated as canon. "Style" might be a bit of a misnomer here, implying a collection limited to formatting practices. Our style guides are more than that; they are the full set of conventions that govern our code. That's not to say that our style guides are strictly prescriptive; style guide rules may call for judgement, such as the rule to use names that are "as descriptive as possible, within reason." Rather, our style guides serve as the definitive source for the rules to which our engineers are held accountable.

We maintain separate style guides for each of the programming languages used at Google.[1] At a high level, all of the guides have similar goals, aiming to steer code development with an eye to sustainability. At the same time, there is a lot of variation among them in scope, length, and content. Programming languages have different strengths, different features, different priorities, and different historical paths to adoption within Google's ever-evolving repositories of code. It is far more practical, therefore, to independently tailor each language's guidelines. Some of our style guides are concise, focusing on a few overarching principles like naming and formatting, as demonstrated in our Dart, R, and Shell guides. Other style guides include far more detail, delving into specific language features,

stretching into far lengthier documents, notably, our C++, Python, and Java guides. Some style guides put a premium on typical non-Google use of the language—our Go style guide is very short, adding just a few rules to a summary directive to adhere to the practices outlined in the externally recognized conventions. Others include rules that fundamentally differ from external norms; our C++ rules disallow use of exceptions, a language feature widely used outside of Google code.

The wide variance among even our own style guides makes it difficult to pin down the precise description of what a style guide should cover. The decisions guiding the development of Google's style guides stem from the need to keep our codebase sustainable. Other organizations' codebases will inherently have different requirements for sustainability that necessitate a different set of tailored rules. This chapter discusses the principles and processes that steer the development of our rules and guidance, pulling examples primarily from Google's C++, Python, and Java style guides.

Why Have Rules?

So why do we have rules? The goal of having rules in place is to encourage "good" behavior and discourage "bad" behavior. The interpretation of "good" and "bad" varies by organization, depending on what the organization cares about. Such designations are not universal preferences; good versus bad is subjective, and tailored to needs. For some organizations, "good" might promote usage patterns that support a small memory footprint or prioritize potential runtime optimizations. In other organizations, "good" might promote choices that exercise new language features. Sometimes, an organization cares most deeply about consistency, so that anything inconsistent with existing patterns is "bad." We must first recognize what a given organization values; we use rules and guidance to encourage and discourage behavior accordingly.

As an organization grows, the established rules and guidelines shape the common vocabulary of coding. A common vocabulary allows engineers to concentrate on what their code needs to say rather than how they're saying it. By shaping this vocabulary, engineers will tend to do the "good" things by default, even subconsciously. Rules thus give us broad leverage to nudge common development patterns in desired directions.

Creating the Rules

When defining a set of rules, the key question is not, "What rules should we have?" The question to ask is, "What goal are we trying to advance?" When we focus on the goal that the rules will be serving, identifying which rules support this goal makes it easier to distill the set of useful rules. At Google, where the style guide serves as law for coding practices, we do not ask, "What goes into the style guide?" but rather, "Why does something go into the style guide?" What does our organization gain by having a set of rules to regulate writing code?

Guiding Principles

Let's put things in context: Google's engineering organization is composed of more than 30,000 engineers. That engineering population exhibits a wild variance in skill and background. About 60,000 submissions are made each day to a codebase of more than two billion lines of code that will likely exist for decades. We're optimizing for a different set of values than most other organizations need, but to some degree, these concerns are ubiquitous—we need to sustain an engineering environment that is resilient to both scale and time.

In this context, the goal of our rules is to manage the complexity of our development environment, keeping the codebase manageable while still allowing engineers to work productively. We are making a trade-off here: the large body of rules that helps us toward this goal does mean we are restricting choice. We lose some flexibility and we might even offend some people, but the gains of consistency and reduced conflict furnished by an authoritative standard win out.

Given this view, we recognize a number of overarching principles that guide the development of our rules, which must:

- Pull their weight
- Optimize for the reader
- Be consistent
- Avoid error-prone and surprising constructs
- Concede to practicalities when necessary

RULES MUST PULL THEIR WEIGHT

Not everything should go into a style guide. There is a nonzero cost in asking all of the engineers in an organization to learn and adapt to any new rule that is set. With too many rules,[2] not only will it become harder for engineers to

remember all relevant rules as they write their code, but it also becomes harder for new engineers to learn their way. More rules also make it more challenging and more expensive to maintain the rule set.

To this end, we deliberately chose not to include rules expected to be self-evident. Google's style guide is not intended to be interpreted in a lawyerly fashion; just because something isn't explicitly outlawed does not imply that it is legal. For example, the C++ style guide has no rule against the use of `goto`. C++ programmers already tend to avoid it, so including an explicit rule forbidding it would introduce unnecessary overhead. If just one or two engineers are getting something wrong, adding to everyone's mental load by creating new rules doesn't scale.

OPTIMIZE FOR THE READER

Another principle of our rules is to optimize for the reader of the code rather than the author. Given the passage of time, our code will be read far more frequently than it is written. We'd rather have the code tedious to type than difficult to read. In our Python style guide, when discussing conditional expressions, we recognize that they are shorter than `if` statements and therefore more convenient for code authors. However, because they tend to be more difficult for readers to understand than the more verbose `if` statements, we restrict their usage. We value "simple to read" over "simple to write." We're making a trade-off here: it can cost more upfront when engineers must repeatedly type potentially longer, descriptive names for variables and types. We choose to pay this cost for the readability it provides for all future readers.

As part of this prioritization, we also require that engineers leave explicit evidence of intended behavior in their code. We want readers to clearly understand what the code is doing as they read it. For example, our Java, JavaScript, and C++ style guides mandate use of the override annotation or keyword whenever a method overrides a superclass method. Without the explicit in-place evidence of design, readers can likely figure out this intent, though it would take a bit more digging on the part of each reader working through the code.

Evidence of intended behavior becomes even more important when it might be surprising. In C++, it is sometimes difficult to track the ownership of a pointer just by reading a snippet of code. If a pointer is passed to a function, without being familiar with the behavior of the function, we can't be sure what to expect. Does the caller still own the pointer? Did the function take ownership? Can I continue using the pointer after the function returns or

might it have been deleted? To avoid this problem, our C++ style guide prefers the use of std::unique_ptr when ownership transfer is intended. `unique_ptr` is a construct that manages pointer ownership, ensuring that only one copy of the pointer ever exists. When a function takes a `unique_ptr` as an argument and intends to take ownership of the pointer, callers must explicitly invoke move semantics:

```
// Function that takes a Foo* and may or may not assume
ownership of

// the passed pointer.

void TakeFoo(Foo* arg);



// Calls to the function don't tell the reader anything about
what to

// expect with regard to ownership after the function returns.

Foo* my_foo(NewFoo());

TakeFoo(my_foo);
```

Compare this to the following:

```
// Function that takes a std::unique_ptr<Foo>.

void TakeFoo(std::unique_ptr<Foo> arg);



// Any call to the function explicitly shows that ownership is

// yielded and the unique_ptr cannot be used after the
function

// returns.

std::unique_ptr<Foo> my_foo(FooFactory());

TakeFoo(std::move(my_foo));
```

Given the style guide rule, we guarantee that all call sites will include clear evidence of ownership transfer whenever it applies. With this signal in place, readers of the code don't need to understand the behavior of every function call. We provide enough information in the API to reason about its interactions. This clear documentation of behavior at the call sites ensures that code snippets remain readable and understandable. We aim for local reasoning, where the goal is clear understanding of what's happening at the call site without needing to find and reference other code, including the function's implementation.

Most style guide rules covering comments are also designed to support this goal of in-place evidence for readers. Documentation comments (the block comments prepended to a given file, class, or function) describe the design or intent of the code that follows. Implementation comments (the comments interspersed throughout the code itself) justify or highlight non-obvious choices, explain tricky bits, and underscore important parts of the code. We have style guide rules covering both types of comments, requiring engineers to provide the explanations another engineer might be looking for when reading through the code.

BE CONSISTENT

Our view on consistency within our codebase is similar to the philosophy we apply to our Google offices. With a large, distributed engineering population, teams are frequently split among offices, and Googlers often find themselves traveling to other sites. Although each office maintains its unique personality, embracing local flavor and style, for anything necessary to get work done, things are deliberately kept the same. A visiting Googler's badge will work with all local badge readers; any Google devices will always get WiFi; the video conferencing setup in any conference room will have the same interface. A Googler doesn't need to spend time learning how to get this all set up; they know that it will be the same no matter where they are. It's easy to move between offices and still get work done.

That's what we strive for with our source code. Consistency is what enables any engineer to jump into an unfamiliar part of the codebase and get to work fairly quickly. A local project can have its unique personality, but its tools are the same, its techniques are the same, its libraries are the same, and it all Just Works.

ADVANTAGES OF CONSISTENCY

Even though it might feel restrictive for an office to be disallowed from customizing a badge reader or video conferencing interface, the consistency benefits far outweigh the creative freedom we lose. It's the same with code: being consistent may feel constraining at times, but it means more engineers get more work done with less effort:[3]

- When a codebase is internally consistent in its style and norms, engineers writing code and others reading it can focus on what's getting done rather than how it is presented. To a large degree, this consistency allows for expert chunking.[4] When we solve our problems with the same interfaces and format the code in a consistent way, it's easier for experts to glance at some code, zero in on what's important, and understand what it's doing. It also makes it easier to modularize code and spot duplication. For these reasons, we focus a lot of attention on consistent naming conventions, consistent use of common patterns, and consistent formatting and structure. There are also many rules that put forth a decision on a seemingly small issue solely to guarantee that things are done in only one way. For example, take the choice of the number of spaces to use for indentation or the limit set on line length.[5] It's the consistency of having one answer rather than the answer itself that is the valuable part here.
- Consistency enables scaling. Tooling is key for an organization to scale and consistent code makes it easier to build tools that can understand, edit, and generate code. The full benefits of the tools that depend on uniformity can't be applied if everyone has little pockets of code that differ—if a tool can keep source files updated by adding missing imports or removing unused includes, if different projects are choosing different sorting strategies for their import lists, the tool might not be able to work everywhere. When everyone is using the same components and when everyone's code follows the same rules for structure and organization, we can invest in tooling that works everywhere, building in automation for many of our maintenance tasks. If each team needed to separately invest in a bespoke version of the same tool, tailored for their unique environment, we would lose that advantage.
- Consistency helps when scaling the human part of an organization, too. As an organization grows, the number of engineers working on the codebase increases. Keeping the code that everyone is working on as consistent as possible enables better mobility across projects,

minimizing the ramp-up time for an engineer switching teams and building in the ability for the organization to flex and adapt as headcount needs fluctuate. A growing organization also means that people in other roles interact with the code—SREs, library engineers, and code janitors, for example. At Google, these roles often span multiple projects, which means engineers unfamiliar with a given team's project might jump in to work on that project's code. A consistent experience across the codebase makes this efficient.

- Consistency also ensures resilience to time. As time passes, engineers leave projects, new people join, ownership shifts, and projects merge or split. Striving for a consistent codebase ensures that these transitions are low cost and allows us nearly unconstrained fluidity for both the code and the engineers working on it, simplifying the processes necessary for long-term maintenance.

AT SCALE

A few years ago, our C++ style guide promised to almost never change style guide rules that would make old code inconsistent: "In some cases, there might be good arguments for changing certain style rules, but we nonetheless keep things as they are in order to preserve consistency."

When the codebase was smaller and there were fewer old, dusty corners, that made sense.

When the codebase grew bigger and older, that stopped being a thing to prioritize. This was (for the arbiters behind our C++ style guide, at least) a conscious change: when striking this bit, we were explicitly stating that the C++ codebase would never again be completely consistent, nor were we even aiming for that.

It would simply be too much of a burden to not only update the rules to current best practices, but to also require that we apply those rules to everything that's ever been written. Our Large Scale Change tooling and processes allow us to update almost all of our code to follow nearly every new pattern or syntax so that most old code exhibits the most recent approved style (see Chapter 22). Such mechanisms aren't perfect, however; when the codebase gets as large as it is, we can't be sure every bit of old code can conform to the new best practices. Requiring perfect consistency has reached the point where there's too much cost for the value.

Setting the standard

When we advocate for consistency, we tend to focus on internal consistency. Sometimes, local conventions spring up before global ones are adopted, and it isn't reasonable to adjust everything to match. In that case, we advocate a hierarchy of consistency: "Be consistent" starts locally, where the norms

within a given file precede those of a given team, which precede those of the larger project, which precede those of the overall codebase. In fact, the style guides contain a number of rules that explicitly defer to local conventions,[6] valuing this local consistency over a scientific technical choice.

However, it is not always enough for an organization to create and stick to a set of internal conventions. Sometimes, the standards adopted by the external community should be taken into account.

COUNTING SPACES

The Python style guide at Google initially mandated two-space indents for all of our Python code. The standard Python style guide, used by the external Python community, uses four-space indents. Most of our early Python development was in direct support of our C++ projects, not for actual Python applications. We therefore chose to use two-space indentation to be consistent with our C++ code, which was already formatted in that manner. As time went by, we saw that this rationale didn't really hold up. Engineers who write Python code read and write other Python code much more often than they read and write C++ code. We were costing our engineers extra effort every time they needed to look something up or reference external code snippets. We were also going through a lot of pain each time we tried to export pieces of our code into open source, spending time reconciling the differences between our internal code and the external world we wanted to join.

When the time came for Starlark (a Python-based language designed at Google to serve as the build description language) to have its own style guide, we chose to change to using four-space indents to be consistent with the outside world.[7]

If conventions already exist, it is usually a good idea for an organization to be consistent with the outside world. For small, self-contained, and short-lived efforts, it likely won't make a difference; internal consistency matters more than anything happening outside the project's limited scope. Once the passage of time and potential scaling become factors, the likelihood of your code interacting with outside projects or even ending up in the outside world increase. Looking long-term, adhering to the widely accepted standard will likely pay off.

AVOID ERROR-PRONE AND SURPRISING CONSTRUCTS

Our style guides restrict the use of some of the more surprising, unusual, or tricky constructs in the languages that we use. Complex features often have subtle pitfalls not obvious at first glance. Using these features without thoroughly understanding their complexities makes it easy to misuse them and introduce bugs. Even if a construct is well understood by a project's

engineers, future project members and maintainers are not guaranteed to have the same understanding.

This reasoning is behind our Python style guide ruling to avoid using power features such as reflection. The reflective Python functions `hasattr()` and `getattr()` allow a user to access attributes of objects using strings:

```
if hasattr(my_object, 'foo'):

some_var = getattr(my_object, 'foo')
```

Now, with that example, everything might seem fine. But consider this:

some_file.py:

```
A_CONSTANT = [

'foo',

'bar',

'baz',

]
```

other_file.py:

```
values = []

for field in some_file.A_CONSTANT:

values.append(getattr(my_object, field))
```

When searching through code, how do you know that the fields `foo`, `bar`, and `baz` are being accessed here? There's no clear evidence left for the reader. You don't easily see and therefore can't easily validate which strings are used to access attributes of your object. What if, instead of reading those values from `A_CONSTANT`, we read them from a Remote Procedure Call (RPC) request message or from a data store? Such obfuscated code could cause a major security flaw, one that would be very difficult to notice, simply by validating the message incorrectly. It's also difficult to test and verify such code.

Python's dynamic nature allows such behavior, and in very limited circumstances, using `hasattr()` and `getattr()` is valid. In most cases, however, they just cause obfuscation and introduce bugs.

Although these advanced language features might perfectly solve a problem for an expert who knows how to leverage them, power features are often more difficult to understand and are not very widely used. We need all of our engineers able to operate in the codebase, not just the experts. It's not just support for the novice software engineer, but it's also a better environment for SREs—if an SRE is debugging a production outage, they will jump into any bit of suspect code, even code written in a language in which they are not fluent. We place higher value on simplified, straightforward code that is easier to understand and maintain.

CONCEDE TO PRACTICALITIES

In the words of Ralph Waldo Emerson: "A foolish consistency is the hobgoblin of little minds." In our quest for a consistent, simplified codebase, we do not want to blindly ignore all else. We know that some of the rules in our style guides will encounter cases that warrant exceptions, and that's OK. When necessary, we permit concessions to optimizations and practicalities that might otherwise conflict with our rules.

Performance matters. Sometimes, even if it means sacrificing consistency or readability, it just makes sense to accommodate performance optimizations. For example, although our C++ style guide prohibits use of exceptions, it includes a rule that allows the use of noexcept, an exception-related language specifier that can trigger compiler optimizations.

Interoperability also matters. Code that is designed to work with specific non-Google pieces might do better if tailored for its target. For example, our C++ style guide includes an exception to the general CamelCase naming guideline that permits use of the standard library's snake_case style for entities that mimic standard library features.[8] The C++ style guide also allows exemptions for Windows programming, where compatibility with platform features requires multiple inheritance, something explicitly forbidden for all other C++ code. Both our Java and JavaScript style guides explicitly state that generated code, which frequently interfaces with or depends on components outside of a project's ownership, is out of scope for the guide's rules.[9] Consistency is vital; adaptation is key.

The Style Guide

So, what does go into a language style guide? There are roughly three categories into which all style guide rules fall:

- Rules to avoid dangers
- Rules to enforce best practices
- Rules to ensure consistency

AVOIDING DANGER

First and foremost, our style guides include rules about language features that either must or must not be done for technical reasons. We have rules about how to use static members and variables; rules about using lambda expressions; rules about handling exceptions; rules about building for threading, access control, and class inheritance. We cover which language features to use and which constructs to avoid. We call out standard vocabulary types that may be used and for what purposes. We specifically include rulings on the hard-to-use and the hard-to-use-correctly—some language features have nuanced usage patterns that might not be intuitive or easy to apply properly, causing subtle bugs to creep in. For each ruling in the guide, we aim to include the pros and cons that were weighed with an explanation of the decision that was reached. Most of these decisions are based on the need for resilience to time, supporting and encouraging maintainable language usage.

ENFORCING BEST PRACTICES

Our style guides also include rules enforcing some best practices of writing source code. These rules help keep the codebase healthy and maintainable. For example, we specify where and how code authors must include comments.[10] Our rules for comments cover general conventions for commenting and extend to include specific cases that must include in-code documentation—cases in which intent is not always obvious, such as fall-through in switch statements, empty exception catch blocks, and template metaprogramming. We also have rules detailing the structuring of source files, outlining the organization of expected content. We have rules about naming: naming of packages, of classes, of functions, of variables. All of these rules are intended to guide engineers to practices that support healthier, more sustainable code.

Some of the best practices enforced by our style guides are designed to make source code more readable. Many formatting rules fall under this category.

Our style guides specify when and how to use vertical and horizontal whitespace in order to improve readability. They also cover line length limits and brace alignment. For some languages, we cover formatting requirements by deferring to autoformatting tools—gofmt for Go, dartfmt for Dart. Itemizing a detailed list of formatting requirements or naming a tool that must be applied, the goal is the same: we have a consistent set of formatting rules designed to improve readability that we apply to all of our code.

Our style guides also include limitations on new and not-yet-well-understood language features. The goal is to preemptively install safety fences around a feature's potential pitfalls while we all go through the learning process. At the same time, before everyone takes off running, limiting use gives us a chance to watch the usage patterns that develop and extract best practices from the examples we observe. For these new features, at the outset, we are sometimes not sure of the proper guidance to give. As adoption spreads, engineers wanting to use the new features in different ways discuss their examples with the style guide owners, asking for allowances to permit additional use cases beyond those covered by the initial restrictions. Watching the waiver requests that come in, we get a sense of how the feature is getting used and eventually collect enough examples to generalize good practice from bad. After we have that information, we can circle back to the restrictive ruling and amend it to allow wider use.

INTRODUCING STD::UNIQUE_PTR

When C++11 introduced `std::unique_ptr`, a smart pointer type that expresses exclusive ownership of a dynamically allocated object and deletes the object when the `unique_ptr` goes out of scope, our style guide initially disallowed usage. The behavior of the `unique_ptr` was unfamiliar to most engineers and the related move semantics that the language introduced were very new and, to most engineers, very confusing. Preventing the introduction of `std::unique_ptr` in the codebase seemed the safer choice. We updated our tooling to catch references to the disallowed type and kept our existing guidance recommending other types of existing smart pointers.

Time passed. Engineers had a chance to adjust to the implications of move semantics and we became increasingly convinced that using `std::unique_ptr` was directly in line with the goals of our style guidance. The information regarding object ownership that a `std::unique_ptr` facilitates at a function call site makes it far easier for a reader to understand that code. The added complexity of introducing this new type, and the novel move semantics that come with it, was still a strong concern, but the significant improvement in the long-term overall state of the codebase made the adoption of `std::unique_ptr` a worthwhile trade-off.

BUILDING IN CONSISTENCY

Our style guides also contain rules that cover a lot of the smaller stuff. For these rules, we make and document a decision primarily to make and document a decision. Many rules in this category don't have significant technical impact. Things like naming conventions, indentation spacing, import ordering: there is usually no clear, measurable, technical benefit for one form over another, which might be why the technical community tends to keep debating them.[11] By choosing one, we've dropped out of the endless debate cycle and can just move on. Our engineers no longer spend time discussing two spaces versus four. The important bit for this category of rules is not *what* we've chosen for a given rule so much as the fact that we *have* chosen.

AND FOR EVERYTHING ELSE...

With all that, there's a lot that's not in our style guides. We try to focus on the things that have the greatest impact on the health of our codebase. There are absolutely best practices left unspecified by these documents, including many fundamental pieces of good engineering advice: don't be clever, don't fork the codebase, don't reinvent the wheel, and so on. Documents like our style guides can't serve to take a complete novice all the way to a master-level understanding of software engineering—there are some things we assume, and this is intentional.

Changing the Rules

Our style guides aren't static. As with most things, given the passage of time, the landscape within which a style guide decision was made and the factors that guided a given ruling are likely to change. Sometimes, conditions change enough to warrant reevaluation. If a new language version is released, we might want to update our rules to allow or exclude new features and idioms. If a rule is causing engineers to invest effort to circumvent it, we might need to reexamine the benefits the rule was supposed to provide. If the tools that we use to enforce a rule become overly complex and burdensome to maintain, the rule itself might have decayed and need to be revisited. Noticing when a rule is ready for another look is an important part of the process that keeps our rule set relevant and up to date.

The decisions behind rules captured in our style guides are backed by evidence. When adding a rule, we spend time discussing and analyzing the

relevant pros and cons as well as the potential consequences, trying to verify that a given change is appropriate for the scale at which Google operates. Most entries in Google's style guides include these considerations, laying out the pros and cons that were weighed during the process and giving the reasoning for the final ruling. Ideally, we prioritize this detailed reasoning and include it with every rule.

Documenting the reasoning behind a given decision gives us the advantage of being able to recognize when things need to change. Given the passage of time and changing conditions, a good decision made previously might not be the best current one. With influencing factors clearly noted, we are able to identify when changes related to one or more of these factors warrant reevaluating the rule.

CAMELCASE NAMING

At Google, when we defined our initial style guidance for Python code, we chose to use CamelCase naming style instead of snake_case naming style for method names. Although the public Python style guide (PEP 8) and most of the Python community used snake_case naming, most of Google's Python usage at the time was for C++ developers using Python as a scripting layer on top of a C++ codebase. Many of the defined Python types were wrappers for corresponding C++ types, and because Google's C++ naming conventions follow CamelCase style, the cross-language consistency was seen as key.

Later, we reached a point at which we were building and supporting independent Python applications. The engineers most frequently using Python were Python engineers developing Python projects, not C++ engineers pulling together a quick script. We were causing a degree of awkwardness and readability problems for our Python engineers, requiring them to maintain one standard for our internal code but constantly adjust for another standard every time they referenced external code. We were also making it more difficult for new hires who came in with Python experience to adapt to our codebase norms.

As our Python projects grew, our code more frequently interacted with external Python projects. We were incorporating third-party Python libraries for some of our projects, leading to a mix within our codebase of our own CamelCase format with the externally-preferred snake_case style. As we started to open-source some of our Python projects, maintaining them in an external world where our conventions were nonconformist added both complexity on our part and wariness from a community that found our style surprising and somewhat weird.

Presented with these arguments, after discussing both the costs (losing consistency with other Google code, reeducation for Googlers used to our Python style) and benefits (gaining consistency with most other Python code, allowing what was already

leaking in with third-party libraries), the style arbiters for the Python style guide decided to change the rule. With the restriction that it be applied as a file-wide choice, an exemption for existing code, and the latitude for projects to decide what is best for them, the Google Python style guide was updated to permit snake_case naming.

The Process

Recognizing that things will need to change, given the long lifetime and ability to scale that we are aiming for, we created a process for updating our rules. The process for changing our style guide is solution based. Proposals for style guide updates are framed with this view, identifying an existing problem and presenting the proposed change as a way to fix it. "Problems," in this process, are not hypothetical examples of things that could go wrong; problems are proven with patterns found in existing Google code. Given a demonstrated problem, because we have the detailed reasoning behind the existing style guide decision, we can reevaluate, checking whether a different conclusion now makes more sense.

The community of engineers writing code governed by the style guide are often best-positioned to notice when a rule might need to be changed. Indeed, here at Google, most changes to our style guides begin with community discussion. Any engineer can ask questions or propose a change, usually by starting with the language-specific mailing lists dedicated to style guide discussions.

Proposals for style guide changes might come fully-formed, with specific, updated wording suggested, or might start as vague questions about the applicability of a given rule. Incoming ideas are discussed by the community, receiving feedback from other language-users. Some proposals are rejected by community consensus, gauged to be unnecessary, too ambiguous, or not beneficial. Others receive positive feedback, gauged to have merit either as-is or with some suggested refinement. These proposals, the ones that make it through community review, are subject to final decision-making approval.

The Style Arbiters

At Google, for each language's style guide, final decisions and approvals are made by the style guide's owners—our style arbiters. For each programming language, a group of long-time language experts are the owners of the style guide and the designated decision makers. The style arbiters for a given language are often senior members of the language's library team and other long-time Googlers with relevant language experience.

The actual decision-making for any style guide change is a discussion of the engineering trade-offs for the proposed modification. The arbiters make decisions within the context of the agreed-upon goals for which the style guide optimizes. Changes are not made according to personal preference; they're trade-off judgements. In fact: The C++ style arbiter group currently consists of four members. This might seem strange: having an odd number of committee members would prevent tied votes in case of a split decision. However, because of the nature of the decision-making approach, where nothing is "because I think it should be this way" and everything is an evaluation of trade-off, decisions are made by consensus rather than by voting. The four-member group is happily functional as-is.

Exceptions

Yes, our rules are law, but yes, some rules warrant exceptions. Our rules are typically designed for the greater, general case. Sometimes, specific situations would benefit from an exemption to a particular rule. When such a scenario arises, the style arbiters are consulted to determine whether there is a valid case for granting a waiver to a particular rule.

Waivers are not granted lightly. In C++ code, if a macro API is introduced, the style guide mandates that it be named using a project-specific prefix. Because of the way C++ handles macros, treating them as members of the global namespace, all macros that are exported from header files must have globally unique names to prevent collisions. The style guide rule regarding macro naming does allow for arbiter-granted exemptions for some utility macros that are genuinely global. However, when the reason behind a waiver request asking to exclude a project-specific prefix comes down to preferences due to macro name length or project consistency, the waiver is rejected. The integrity of the codebase outweighs the consistency of the project here.

Exceptions are allowed for cases in which it is gauged to be more beneficial to permit the rule-breaking than to avoid it. The C++ style guide disallows implicit type conversions, including single-argument constructors. However, for types that are designed to transparently wrap other types, where the underlying data is still accurately and precisely represented, it's perfectly reasonable to allow implicit conversion. In such cases, waivers to the no-implicit-conversion rule are granted. Having such a clear case for valid exemptions might indicate that the rule in question needs to be clarified or amended. However, for this specific rule, enough waiver requests are received that appear to fit the valid case for exemption but in fact do not, either because the specific type in question is not actually a transparent

wrapper type or because the type is a wrapper but is not actually needed, that keeping the rule in place as-is is still worthwhile.

Guidance

In addition to rules, we curate programming guidance in various forms, ranging from long, in-depth discussion of complex topics to short, pointed advice on best practices that we endorse.

Guidance represents the collected wisdom of our engineering experience, documenting the best practices that we've extracted from the lessons learned along the way. Guidance tends to focus on things that we've observed people frequently getting wrong or new things that are unfamiliar and therefore subject to confusion. If the rules are the "musts," our guidance is the "shoulds."

One example of a pool of guidance that we cultivate is a set of primers for some of the predominant languages that we use. While our style guides are prescriptive, ruling on which language features are allowed and which are disallowed, the primers are descriptive, explaining the features that the guides endorse. They are quite broad in their coverage, touching on nearly every topic that an engineer new to the language's use at Google would need to reference. They do not delve into every detail of a given topic, but they provide explanations and recommended use. When an engineer needs to figure out how to apply a feature that they want to use, the primers aim to serve as the go-to guiding reference.

A few years ago, we began publishing a series of C++ tips that offered a mix of general language advice and Google-specific tips. We cover hard things—object lifetime, copy and move semantics, argument-dependent lookup; new things—C++ 11 features as they were adopted in the codebase, preadopted C++17 types like `string_view`, `optional`, and `variant`; and things that needed a gentle nudge of correction—reminders not to use `using` directives, warnings to remember to look out for implicit `bool` conversions. The tips grow out of actual problems encountered, addressing real programming issues that are not covered by the style guides. Their advice, unlike the rules in the style guide, are not true canon; they are still in the category of advice rather than rule. However, given the way they grow from observed patterns rather than abstract ideals, their broad and direct applicability set them apart from most other advice as a sort of "canon of the common." Tips are narrowly focused and relatively short, each one no more than a few minutes' read. This "Tip of

the Week" series has been extremely successful internally, with frequent citations during code reviews and technical discussions.[12]

Software engineers come in to a new project or codebase with knowledge of the programming language they are going to be using, but lacking the knowledge of how the programming language is used within Google. To bridge this gap, we maintain a series of "<Language>@Google 101" courses for each of the primary programming languages in use. These full-day courses focus on what makes development with that language different in our codebase. They cover the most frequently used libraries and idioms, in-house preferences, and custom tool usage. For a C++ engineer who has just become a Google C++ engineer, the course fills in the missing pieces that make them not just a good engineer, but a good Google codebase engineer.

In addition to teaching courses that aim to get someone completely unfamiliar with our setup up and running quickly, we also cultivate ready references for engineers deep in the codebase to find the information that could help them on the go. These references vary in form and span the languages that we use. Some of the useful references that we maintain internally include the following:

- Language-specific advice for the areas that are generally more difficult to get correct (such as concurrency and hashing)
- Detailed breakdowns of new features that are introduced with a language update and advice on how to use them within the codebase
- Listings of key abstractions and data structures provided by our libraries. This keeps us from reinventing structures that already exist and provides a response to, "I need a thing, but I don't know what it's called in our libraries."

Applying the Rules

Rules, by their nature, lend greater value when they are enforceable. Rules can be enforced socially, through teaching and training, or technically, with tooling. We have various formal training courses at Google that cover many of the best practices that our rules require. We also invest resources in keeping our documentation up to date to ensure that reference material remains accurate and current. A key part of our overall training approach when it comes to awareness and understanding of our rules is the role that code reviews play. The readability process that we run here at Google—where engineers new to Google's development environment for a given

language are mentored through code reviews—is, to a great extent, about cultivating the habits and patterns required by our style guides (see details on the readability process in the Chapter 3). The process is an important piece of how we ensure that these practices are learned and applied across project boundaries.

Although some level of training is always necessary—engineers must, after all, learn the rules so that they can write code that follows them—when it comes to checking for compliance, rather than exclusively depending on engineer-based verification, we strongly prefer to automate enforcement with tooling.

Automated rule enforcement ensures that rules are not dropped or forgotten as time passes or as an organization scales up. New people join; they might not yet know all the rules. Rules change over time; even with good communication, not everyone will remember the current state of everything. Projects grow and add new features; rules that had previously not been relevant are suddenly applicable. An engineer checking for rule compliance depends on either memory or documentation, both of which can fail. As long as our tooling stays up to date, in sync with our rule changes, we know that our rules are being applied by all our engineers for all our projects.

Another advantage to automated enforcement is minimization of the variance in how a rule is interpreted and applied. When we write a script or use a tool to check for compliance, we validate all inputs against a single, unchanging definition of the rule. We aren't leaving interpretation up to each individual engineer. Human engineers view everything with a perspective colored by their biases. Unconscious or not, potentially subtle, and even possibly harmless, biases still change the way people view things. Leaving enforcement up to engineers is likely to see inconsistent interpretation of the rules and inconsistent application of the rules, potentially with inconsistent expectations of accountability. The more that we delegate to the tools, the fewer entry points we leave for human biases to enter.

Tooling also makes enforcement scalable. As an organization grows, a single team of experts can write tools that the rest of the company can use. If the company doubles in size, the effort to enforce all rules across the entire organization doesn't double, it costs about the same as it did before.

Even with the advantages we get by incorporating tooling, it might not be possible to automate enforcement for all rules. Some technical rules explicitly call for human judgement. In the C++ style guide, for example: "Avoid complicated template metaprogramming." "Use auto to avoid type

names that are noisy, obvious, or unimportant - cases where the type doesn't aid in clarity for the reader." "Composition is often more appropriate than inheritance." In the Java style guide: "There's no single correct recipe for how to [order the members and initializers of your class]; different classes may order their contents in different ways." "It is very rarely correct to do nothing in response to a caught exception." "It is extremely rare to override `Object.finalize`." For all of these rules, judgement is required and tooling can't (yet!) take that place.

Other rules are social rather than technical, and it is often unwise to solve social problems with a technical solution. For many of the rules that fall under this category, the details tend to be a bit less well defined and tooling would become complex and expensive. It's often better to leave enforcement of those rules to humans. For example, when it comes to the size of a given code change, the number of files affected and lines modified, we recommend that engineers favor smaller changes. Small changes are easier for engineers to review, so reviews tend to be faster and more thorough. They're also less likely to introduce bugs because it's easier to reason about the potential impact and effects of a smaller change. The definition of small, however, is somewhat nebulous. A change that propagates the identical one-line update across hundreds of files might actually be easy to review. By contrast, a smaller, twenty-line change might introduce complex logic with side effects that are difficult to evaluate. We recognize that there are many different measurements of size, some of which may be subjective—particularly when taking the complexity of a change into account. This is why we do not have any tooling to autoreject a proposed change that exceeds an arbitrary line limit. Reviewers can (and do) push back if they judge a change to be too large. For this and similar rules, enforcement is up to the discretion of the engineers authoring and reviewing the code. When it comes to technical rules, however, whenever it is feasible, we favor technical enforcement.

Error Checkers

Many rules covering language usage can be enforced with static analysis tools. In fact, an informal survey of the C++ style guide by some of our C++ librarians in mid-2018 estimated that roughly 90% of its rules could be automatically verified. Error-checking tools take a set of rules or patterns and verify that a given code sample fully complies. Automated verification removes the burden of remembering all applicable rules from the code author. If an engineer only needs to look for violation warnings—many of which come with suggested fixes—surfaced during code review by an analyzer that has been tightly integrated into the development workflow, we

minimize the effort that it takes to comply with the rules. When we began using tools to flag deprecated functions based on source tagging, surfacing both the warning and the suggested fix in-place, the problem of having new usages of deprecated APIs disappeared almost overnight. Keeping the cost of compliance down makes it more likely for engineers to happily follow through.

We use tools like Clang-Tidy (for C++) and Error Prone (for Java) to automate the process of enforcing rules. See Chapter 20 for an in-depth discussion of our approach.

The tools we use are designed and tailored to support the rules that we define. Most tools in support of rules are absolutes; everybody must comply with the rules, so everybody uses the tools that check them. Sometimes, when tools support best practices where there's a bit more flexibility in conforming to the conventions, there are opt-out mechanisms to allow projects to adjust for their needs.

Code Formatters

At Google, we generally use automated style checkers and formatters to enforce consistent formatting within our code. The question of line lengths has stopped being interesting.[13] Engineers just run the style checkers and keep moving forward. When formatting is done the same way every time, it becomes a non-issue during code review, eliminating the review cycles that are otherwise spent finding, flagging, and fixing minor style nits.

In managing the largest codebase ever, we've had the opportunity to observe the results of formatting done by humans versus formatting done by automated tooling. The robots are better on average than the humans by a significant amount. There are some places where domain expertise matters—formatting a matrix, for example, is something a human can usually do better than a general-purpose formatter. Failing that, formatting code with an automated style checker rarely goes wrong.

We enforce use of these formatters with presubmit checks: before code can be submitted, a service checks whether running the formatter on the code produces any diffs. If it does, the submit is rejected with instructions on how to run the formatter to fix the code. Most code at Google is subject to such a presubmit check. For our code, we use clang-format for C++; an in-house wrapper around yapf for Python; gofmt for Go; dartfmt for Dart; and buildifier for our `BUILD` files.

Case Study: gofmt

By Sameer Ajmani

Google released the Go programming language as open source on November 10, 2009. Since then, Go has grown as a language for developing services, tools, cloud infrastructure, and open source software.[14]

We knew that we needed a standard format for Go code from day one. We also knew that it would be nearly impossible to retrofit a standard format after the open source release. So the initial Go release included `gofmt`, the standard formatting tool for Go.

MOTIVATIONS

Code reviews are a software engineering best practice, yet too much time was spent in review arguing over formatting. Although a standard format wouldn't be everyone's favorite, it would be good enough to eliminate this wasted time.[15]

By standardizing the format, we laid the foundation for tools that could automatically update Go code without creating spurious diffs: machine-edited code would be indistinguishable from human-edited code.[16]

For example, in the months leading up to Go 1.0 in 2012, the Go team used a tool called `gofix` to automatically update pre-1.0 Go code to the stable version of the language and libraries. Thanks to `gofmt`, the diffs `gofix` produced included only the important bits: changes to uses of the language and APIs. This allowed programmers to more easily review the changes and learn from the changes the tool made.

IMPACT

Go programmers expect that *all* Go code is formatted with `gofmt`. `gofmt` has no configuration knobs, and its behavior rarely changes. All major editors and IDEs use `gofmt` or emulate its behavior, so nearly all Go code in existence is formatted identically. At first, Go users complained about the enforced standard; now, users often cite `gofmt` as one of the many reasons they like Go. Even when reading unfamiliar Go code, the format is familiar.

Thousands of open source packages read and write Go code.[17] Because all editors and IDEs agree on the Go format, Go tools are portable, easily

integrated into new developer environments and workflows via the command line.

RETROFITTING

In 2012, we decided to automatically format all `BUILD` files at Google using a new standard formatter: `buildifier`. `BUILD` files contain the rules for building Google's software with Blaze, Google's build system. A standard `BUILD` format would enable us to create tools that automatically edit `BUILD` files without disrupting their format, just as Go tools do with Go files.

It took six weeks for one engineer to get the reformatting of Google's 200,000 `BUILD` files accepted by the various code owners, during which more than a thousand new `BUILD` files were added each week.[18] Google's nascent infrastructure for making large-scale changes greatly accelerated this effort. (See Chapter 22.)

Conclusion

For any organization, but especially for an organization as large as Google's engineering force, rules help us to manage complexity and build a maintainable codebase. A shared set of rules frames the engineering processes so that they can scale up and keep growing, keeping both the codebase and the organization sustainable for the long term.

TL;DRs

- Rules and guidance should aim to support resilience to time and scaling.
- Know the data so that rules can be adjusted.
- Not everything should be a rule.
- Consistency is key.
- Automate enforcement when possible.

[1] Many of our style guides have external versions, which you can find at *https://google.github.io/styleguide/*. We cite numerous examples from these guides within this chapter.

2 Tooling matters here. The measure for "too many" is not the raw number of rules in play, but how many an engineer needs to remember. For example, in the bad-old-days pre-clang-format, we needed to remember a ton of formatting rules. Those rules haven't gone away, but with our current tooling, the cost of adherence has fallen dramatically. We've reached a point at which somebody could add an arbitrary number of formatting rules and nobody would care, because the tool just does it for you.

3 Credit to H. Wright for the real-world comparison, made at the point of having visited around 15 different Google offices.

4 Chunking is a cognitive process that groups pieces of information together into meaningful "chunks" rather than keeping note of them individually. Expert chess players, for example, think about configurations of pieces rather than the positions of the individuals.

5 *https://google.github.io/styleguide/javaguide.html#s4.2-block-indentation* and *https://google.github.io/styleguide/cppguide.html#Spaces_vs._Tabs*. And then *https://google.github.io/styleguide/javaguide.html#s4.4-column-limit* and *https://google.github.io/styleguide/cppguide.html#Line_Length*.

6 *https://google.github.io/styleguide/cppguide.html#Use_of_const*, for example.

7 Style formatting for BUILD files implemented with Starlark is applied by buildifier. See *https://github.com/bazelbuild/buildtools*.

8 *https://google.github.io/styleguide/cppguide.html#Exceptions_to_Naming_Rules*. As an example, our open-sourced Abseil libraries use snake_case naming for types intended to be replacements for standard types. See the types defined in *https://github.com/abseil/abseil-cpp/blob/master/absl/utility/utility.h*. These are C++11 implementation of C++14 standard types and therefore use the standard's favored snake_case style instead of Google's preferred CamelCase form.

9 *https://google.github.io/styleguide/jsguide.html#policies-generated-code-mostly-exempt*

10 *https://google.github.io/styleguide/cppguide.html#Comments*, *http://google.github.io/styleguide/pyguide#38-comments-and-docstrings*, *https://google.github.io/styleguide/javaguide.html#s7-javadoc*, where multiple languages define general comment rules.

11 Such discussions are really just bikeshedding.

12 *https://abseil.io/tips/* has a selection of some of our most popular tips.

13 When you consider that it takes at least two engineers to have the discussion and multiply that by the number of times this conversation is likely to happen within a collection of more than 30,000 engineers, it turns out that "how many characters" can become a very expensive question.

14 In December 2018, Go is the #4 language on GitHub as measured by pull requests.

15 Robert Griesemer's 2015 talk, The Cultural Evolution of `gofmt`, provides details on the motivation, design, and impact of `gofmt` on Go and other languages.

16 Russ Cox explained in 2009 that `gofmt` was about automated changes: "So we have all the hard parts of a program manipulation tool just sitting waiting to be used. Agreeing to accept "gofmt style" is the piece that makes it doable in a finite amount of code."

17 The Go AST and format packages each have thousands of importers.

18 *https://rsc.users.x20web.corp.google.com/www/buildifierstatus/index.html*

Chapter 9. Code Review

Written by Tom Manshreck and Caitlin Sadowski

Edited by Lisa Carey

Code review is a process in which code is reviewed by someone other than the author, often before the introduction of that code into a codebase. Although that is a simple definition, implementations of the process of code review vary widely throughout the software industry. Some organizations have a select group of "gatekeepers" across the codebase that review changes. Others delegate code review processes to smaller teams, allowing different teams to require different levels of code review. At Google, essentially every change is reviewed before being committed, and every engineer is responsible for initiating reviews and reviewing changes.

Code reviews generally require a combination of a process and a tool supporting that process. At Google, we use a custom code review tool, Critique, to support our process.[1] Critique is an important enough tool at Google to warrant its own chapter in this book. This chapter focuses on the process of code review as it is practiced at Google rather than the specific tool, both because these foundations are older than the tool and because most of these insights can be adapted to whatever tool you might use for code review.

NOTE

For more information on Critique, see Chapter 19.

Some of the benefits of code review, such as detecting bugs in code before they enter a codebase, are well established[2] and somewhat obvious (if imprecisely measured). Other benefits, however, are more subtle. Because the code review process at Google is so ubiquitous and extensive, we've noticed many of these more subtle effects, including psychological ones, which provide many benefits to an organization over time and scale.

Code Review Flow

Code reviews can happen at many stages of software development. At Google, code reviews take place before a change can be committed to the

codebase; this stage is also known as a *precommit review*. The primary end goal of a code review is to get another engineer to consent to the change, which we denote by tagging the change as "looks good to me." We use this "looks good to me" as a necessary permissions "bit" (combined with other bits noted below) to allow the change to be committed.

A typical code review at Google goes through the following steps:

1. A user writes a change to the codebase in their workspace. This *author* then creates a snapshot of the change: a patch and corresponding description that are uploaded to the code review tool. This change produces a *diff* against the codebase, which is used to evaluate what code has changed.
2. The author can use this initial patch to apply automated review comments or do self-review. When the author is satisfied with the diff of the change, they mail the change to one or more reviewers. This process notifies those reviewers asking them to view and comment on the snapshot.
3. *Reviewers* open the change in the code review tool and post comments on the diff. Some comments request explicit resolution. Some are merely informational.
4. The author modifies the change and uploads new snapshots based on the feedback and then replies back to the reviewers. Steps 3 and 4 may be repeated multiple times.
5. After the reviewers are happy with the latest state of the change, they agree to the change and accept it by marking it as "looks good to me." Only one "looks good to me" is required by default, although convention might request that all reviewers agree to the change.
6. After a change is marked "looks good to me," the author is allowed to commit the change to the codebase, provided they *resolve all comments* and that the change is *approved*. We'll cover approval in the next section.

We'll go over this process in more detail later in this chapter.

Code Is a Liability

It's important to remember (and accept) that code itself is a liability. It might be a necessary liability, but by itself, code is simply a maintenance task to someone somewhere down the line. Much like the fuel that an airplane

carries, it has weight, though it is, of course, necessary for that airplane to fly.[3]

New features are often necessary, of course, but care should be taken before developing code in the first place to ensure that any new feature is warranted. Duplicated code not only is a wasted effort, it can actually cost more in time than not having the code at all; changes that could be easily performed under one code pattern often require more effort when there is duplication in the codebase. Writing entirely new code is so frowned upon, that some of us have a saying "If you're writing it from scratch, you're doing it wrong!"

This is especially true of library or utility code. Chances are, if you are writing a utility, someone else somewhere in a codebase the size of Google's has probably done something similar. Tools such as Chapter 17 are therefore critical for both finding such utility code and preventing the introduction of duplicate code. Ideally, this research is done beforehand, and a design for anything new has been communicated to the proper groups before any new code is written.

Of course, new features are often necessary. New projects happen, new techniques are introduced, new components are needed, and so on. All that said, a code review is not an occasion to rehash or debate previous design decisions. Design decisions often take time, requiring the circulation of design proposals, debate on the design in API reviews or some similar meetings, and so forth. and perhaps the development of prototypes. As much as a code review of entirely new code should not come out of the blue, the code review process itself should also not be viewed as an opportunity to revisit previous decisions.

How Code Review Works at Google

We've pointed out roughly how the typical code review process works, but the devil is in the details. This section outlines in specific how code review works at Google and how these practices allow it to scale properly over time.

There are three aspects of review that require "approval" for any given change at Google:

- A correctness and comprehension check from another engineer that the code is appropriate and does what the author claims it does. This is often a team member, though it does not need to be. This is reflected in the "looks good to me" permissions "bit," which will be set after a peer reviewer agrees that the code "looks good" to them.

- Approval from one of the code owners that the code is appropriate for this particular part of the codebase (and can be checked into a particular directory). This approval might be implicit if the author is such an owner. Google's codebase is a tree structure with hierarchical owners of particular directories. (See XREF(Source Control)). Owners act as gatekeepers for their particular directories. A change might be proposed by any engineer and "looks good to me'ed" by any other engineer, but an owner of the directory in question must also *approve* this addition to their part of the codebase. Such an owner might be a tech lead or other engineer deemed expert in that particular area of the codebase. It's generally up to each team to decide how broadly or narrowly to assign ownership privileges.
- Approval from someone with language "readability"[4] that the code conforms to the language's style and best practices, checking whether the code is written in the manner we expect. This approval, again, might be implicit if the author has such readability. These engineers are pulled from a company-wide pool of engineers who have been granted readability in that programming language.

Although this level of control sounds onerous—and, admittedly, it sometimes is—most reviews have one person assuming all three roles, which speeds up the process quite a bit. Importantly, the author can also assume the latter two roles, needing only an "looks good to me" from another engineer to check code into their own codebase, provided they already have readability in that language (which owners often do).

NOTE

For more information on readability, see Chapter 3.

These requirements allow the code review process to be quite flexible. A tech lead who is an owner of a project and has that code's language readability can submit a code change with only a "looks good to me" from another engineer. An intern without such authority can submit the same change to the same codebase, provided they get approval from an owner with language readability. The three aforementioned permission "bits" can be combined in any combination. An author can even request more than one "looks good to me" from separate people by explicitly tagging the change as wanting a "looks good to me" from all reviewers.

In practice, most code reviews that require more than one approval usually go through a two-step process: gaining a "looks good to me" from a peer

engineer, and then seeking approval from appropriate code owner/readability reviewer(s). This allows the two roles to focus on different aspects of the code review and saves review time. The primary reviewer can focus on code correctness and the general validity of the code change; the code owner can focus on whether this change is appropriate for their part of the codebase without having to focus on the details of each line of code. An approver is often looking for something different than a peer reviewer, in other words. After all, someone is trying to check in code to their project/directory. They are more concerned with questions such as: "Will this code be easy or difficult to maintain?" "Does it add to my technical debt?" "Do we have the expertise to maintain it within our team?"

If all three of these types of reviews can be handled by one reviewer, why not just have those types of reviewers handle all code reviews? The short answer is scale. Separating the three roles adds flexibility to the code review process. If you are working with a peer on a new function within a utility library, you can get someone on your team to review the code for code correctness and comprehension. After several rounds (perhaps over several days), your code satisfies your peer reviewer and you get a "looks good to me." Now, you need only get an OWNER of the library (and owners often have appropriate readability) to approve the change.

Ownership

By Hyrum Wright

When working on a small team in a dedicated repository, it's common to grant the entire team access to everything in the repository. After all, you know the other engineers, the domain is narrow enough that each of you can be experts, and small numbers constrain the effect of potential errors.

As the team grows larger, this approach can fail to scale. The result is either a messy repository split or a different approach to recording who has what knowledge and responsibilities in different parts of the repository. At Google, we call this set of knowledge and responsibilities *ownership* and the people to exercise them *owners*. This concept is different than possession of a collection of source code, but rather implies a sense of stewardship to act in the company's best interest with a section of the codebase. (Indeed, "stewards" would almost certainly be a better term if we had it to do over again.)

Specially named OWNERS files list usernames of people who have ownership responsibilities for a directory and its children. These files may also contain references to other OWNERS files or external access control lists, but eventually they resolve to a list of individuals. Each subdirectory may also contain a separate OWNERS file, and the relationship is hierarchically additive: a given file is generally owned by the union of the members of all the OWNERS files above it in the directory tree. OWNERS files may have as many entries as teams like, but we encourage a relatively small and focused list to ensure responsibility is clear.

Ownership of Google's code conveys approval rights for code within one's purview, but these rights also come with a set of responsibilities, such as understanding the code that is owned or knowing how to find somebody who does. Different teams have different criteria for granting ownership to new members, but we generally encourage them not to use ownership as a rite of initiation, and encourage departing members to yield ownership as soon as is practical.

This distributed ownership structure enables many of the other practices we've outlined in this book. For example, the set of people in the root OWNERS file can act as global approvers for large-scale changes (See Chapter 22), without having to bother local teams. Likewise, OWNERS files act as a kind of documentation, making it easy for people and tools to find those responsible for a given piece of code just by walking up the directory tree. When new projects are created, there's no central authority that has to register new ownership privileges: a new OWNERS file is sufficient.

This ownership mechanism is simple, yet powerful and has scaled well over the past two decades. It is one of the ways that Google ensures that tens of thousands of engineers can operate efficiently on billions of lines of code in a single repository.

Code Review Benefits

Across the industry, code review itself is not controversial, though it is far from a universal practice. Many (maybe even most) other companies and open source projects have some form of code review, and most view the process as important as a sanity check on the introduction of new code into a codebase. Software engineers understand some of the more obvious benefits of code review, even if they might not personally think it applies in all cases.

But at Google, this process is generally more thorough and more wide-spread than at most other companies.

Google's culture, like that of a lot of software companies, is based on giving engineers wide latitude in how they do their jobs. There is a recognition that strict processes tend not to work well for a dynamic company needing to respond quickly to new technologies, and that bureaucratic rules tend not to work well with creative professionals. Code review, however, is a mandate, one of the few blanket processes in which all software engineers at Google must participate. Google requires code review for almost[5] every code change to the codebase, no matter how small. This mandate does have a cost and effect on engineering velocity given that it does slow down the introduction of new code into a codebase and can impact time-to-production for any given code change. (Both of these are common complaints by software engineers of strict code review processes.) Why, then, do we require this process? Why do we believe that this is a long-term benefit?

A well-designed code review process and a culture of taking code review seriously provides the following benefits:

- Checks code correctness
- Ensures the code change is comprehensible to other engineers
- Enforces consistency across the codebase
- Psychologically promotes team ownership
- Enables knowledge sharing
- Provides a historical record of the code review itself

Many of these benefits are critical to a software organization over time, and many of them are beneficial to not only the author but also the reviewers. The following sections go into more specifics for each of these items.

Code Correctness

An obvious benefit of code review is that it allows a reviewer to check the "correctness" of the code change. Having another set of eyes look over a change helps ensure that the change does what was intended. Reviewers typically look for whether a change has proper testing, is properly designed and functions correctly and efficiently. In many cases, checking code correctness is checking whether the particular change can introduce bugs into the codebase.

Many reports point to the efficacy of code review in the prevention of future bugs in software. A study at IBM found that discovering defects earlier in a process, unsurprisingly, led to less time required to fix them later on.[6] The investment in the time for code review saved time otherwise spent in testing, debugging, and performing regressions, provided that the code review process itself was streamlined to keep it lightweight. This latter point is important; code review processes that are heavyweight, or don't scale properly, become unsustainable.[7] We will get into some best practices for keeping the process lightweight later in this chapter.

To prevent the evaluation of correctness from becoming more subjective than objective, authors are generally given deference to their particular approach, whether it be in the design or the function of the introduced change. A reviewer shouldn't propose alternatives because of personal opinion. Reviewers can propose alternatives, but only if they improve comprehension (by being less complex, for example) or functionality (by being more efficient, for example). In general, engineers are encouraged to approve changes that improve the codebase rather than wait for consensus on a more "perfect" solution. This focus tends to speed up code reviews.

As tooling becomes stronger, many correctness checks are performed automatically through techniques such as static analysis and automated testing (though tooling might never completely obviate the value for human-based inspection of code). (See Chapter 20 for more information). Though this tooling has its limits, it has definitely lessoned the need to rely on human-based code reviews for checking code correctness.

That said, checking for defects during the initial code review process is still an integral part of a general "shift left" strategy, aiming to discover and resolve issues at the earliest possible time, so that they don't require escalated costs and resources further down in the development cycle. A code review is not a panacea, nor the only check for such correctness, but is an element of a defense-in-depth against such problems in software. As a result, code review does not need to be "perfect" to achieve results.

Surprisingly enough, checking for code correctness is not the primary benefit Google accrues from the process of code review. Checking for code correctness generally ensures that a change works, but more importance is attached to ensuring that a code change is understandable and makes sense over time and as the codebase itself scales. To evaluate those aspects, we need to look at other factors other than whether the code is simply logically "correct" or understood.

Comprehension of Code

A code review typically is the first opportunity for someone other than the author to inspect a change. This perspective allows a reviewer to do something that even the best engineer cannot do: provide feedback unbiased by an author's perspective. *A code review is often the first test of whether a given change is understandable to a broader audience*. This perspective is vitally important because code will be read many more times than it is written, and understanding and comprehension are critically important.

It is often useful to find a reviewer who has a different perspective from the author, especially a reviewer who might need, as part of their job, to maintain or use the code being proposed within your change. Unlike the deference reviewers should give authors regarding design decisions, it's often useful to treat questions on code comprehension using the maxim "the customer is always right." In some respect, any questions you get now will be multiplied many-fold over time, so view each question on code comprehension as valid. This doesn't mean that you need to change your approach or your logic in response to the criticism, but it does mean that you might need to explain it more clearly.

Together, the code correctness and code comprehension checks are the main criteria for an "looks good to me" from another engineer, which is one of the approval bits needed for an approved code review. When an engineer marks a code review as "looks good to me", they are saying that the code does what it says, and that it is understandable. Google, however, also requires that the code be sustainably maintained, so we have additional approvals needed for code in certain cases.

Code Consistency

At scale, code that you write will be depended on, and eventually maintained, by someone else. Many others will need to read your code and understand what you did. Others (including automated tools) might need to refactor your code long after you've moved to another project. Code therefore needs to conform to some standards of consistency so that it can be understood and maintained. Code should also avoid being overly complex; simpler code is easier for others to understand and maintain, as well. Reviewers can assess how well this code lives up the standards of the codebase itself during code review. A code review therefore should act to ensure *code health*.

It is for maintainability that the "looks good to me" state of a code review (indicating code correctness and comprehension) is separated from that of readability approval. Readability approvals can be granted only by individuals who have successfully gone through the process of code readability training in a particular programming language. For example, Java code requires approval from an engineer who has "Java readability."

A readability approver is tasked with reviewing code to ensure that it follows agreed-on best practices for that particular programming language, is consistent with the codebase for that language within Google's code repository, and avoids being overly complex. Code that is consistent and simple is easier to understand and easier for tools to update when it comes time for refactoring, making it more resilient. If a particular pattern is always done in one fashion in the codebase, it's easier to write a tool to refactor it.

Additionally, code might be written only once, but it will be read dozens, hundreds, or even thousands of times. Having code that is consistent across the codebase improves comprehension for all of engineering, and this consistency even affects the process of code review itself. Consistency sometimes clashes with functionality; a readability reviewer may prefer a less complex change that may not be functionally "better," but is easier to understand.

With a more consistent codebase, it is easier for engineers to step in and review code on someone else's projects. Engineers might occasionally need to look outside the team for help in a code review. Being able to reach out and ask experts to review the code, knowing they can expect the code itself to be consistent, allows those engineers to focus more properly on code correctness and comprehension.

Psychological and Cultural Benefits

Code review also has important cultural benefits: it reinforces to software engineers that code is not "theirs" but in fact part of a collective enterprise. Such psychological benefits can be subtle but are still important. Without code review, most engineers would naturally gravitate toward personal style and their own approach to software design. The code review process forces an author to not only let others have input, but to compromise for the sake of the greater good.

It is human nature to be proud of one's craft and to be reluctant to open up one's code to criticism by others. It is also natural to be somewhat reticent to welcome critical feedback about code that one writes. The code review

process provides a mechanism to mitigate what might otherwise be an emotionally charged interaction. Code review, when it works best, provides not only a challenge to an engineer's assumptions, but also does so in a prescribed, neutral manner, acting to temper any criticism which might otherwise be directed to the author if provided in an unsolicited manner. After all, the process *requires* critical review (we in fact call our code review tool "Critique"), so you can't fault a reviewer from doing their job and being critical. The code review process itself, therefore, can act as the "bad cop," whereas the reviewer can still be seen as the "good cop."

Of course, not all, or even most, engineers, need such psychological devices. But buffering such criticism through the process of code review often provides a much gentler introduction for most engineers to the expectations of the team. Many engineers joining Google, or a new team, are intimidated by code review. It is easy to think that any form of critical review reflects negatively on a person's job performance. But over time, almost all engineers come to expect to be challenged when sending a code review and come to value the advice and questions offered through this process (though, admittedly, this sometimes takes a while).

Another psychological benefit of code review is validation. Even the most capable engineers can suffer from imposter syndrome and be too self-critical. A process like code review acts as validation and recognition for one's work. Often, the process involves an exchange of ideas and knowledge sharing (covered in the next section), which benefits both the reviewer and the reviewee. As an engineer grows in their domain knowledge, it's sometimes difficult for them to get positive feedback on how they improve. The process of code review can provide that mechanism.

The process of initiating a code review also forces all authors to take a little extra care with their changes. Many software engineers are not perfectionists; most will admit that code that "gets the job done" is better than code that is perfect but that takes too long to develop. Without code review, it's natural that many of us would cut corners, even with the full intention of correcting such defects later. "Sure, I don't have all of the unit tests done, but I can do that later." A code review forces an engineer to resolve those issues before sending the change. Collecting the components of a change for code review psychologically forces an engineer to make sure that all of their ducks are in a row. The little moment of reflection that comes before sending off your change is the perfect time to read through your change and make sure you're not missing anything.

Knowledge Sharing

One of the most important, but underrated, benefits of code review is in knowledge sharing. Most authors pick reviewers who are experts, or at least knowledgeable, in the area under review. The review process allows reviewers to impart domain knowledge to the author, allowing the reviewer(s) to offer suggestions, new techniques, or advisory information to the author. (Reviewers can even mark some comments "FYI," requiring no action; they are simply added as an aid to the author.) Authors who become particularly proficient in an area of the codebase will often become owners, as well, who then in turn will be able to act as reviewers for other engineers.

Part of the code review process of feedback and confirmation involves asking questions on why the change is done in a particular way. This exchange of information facilitates knowledge sharing. In fact, many code reviews involve an exchange of information both ways: the authors as well as the reviewers can learn new techniques and patterns from code review. At Google, reviewers may even directly share suggested edits with an author within the code review tool itself.

An engineer might may not read every email sent to them, but they tend to respond to every code review sent. This knowledge sharing can occur across time zones and projects, as well, using Google's scale to disseminate information quickly to engineers in all corners of the codebase. Code review is a perfect time for knowledge transfer: it is timely and actionable. (Many engineers at Google "meet" other engineers first through their code reviews!)

Given the amount of time Google engineers spend in code review, the knowledge accrued is quite significant. A Google engineer's primary task is still programming, of course, but a large chunk of their time is still spent in code review. The code review process provides one of the primary ways that software engineers interact with one another and exchange information about coding techniques. Often, new patterns are advertised within the context of code review, sometimes through refactorings such as large-scale changes.

Moreover, because each change becomes part of the codebase, code review acts as a historical record. Any engineer can inspect the Google codebase and determine when some particular pattern was introduced and bring up the actual code review in question. Often, that archeology provides insights to many more engineers than the original author and reviewer(s).

Code Review Best Practices

Code review can, admittedly, introduce friction and delay to an organization. Most of these issues are not problems with code review, per se, but with their chosen implementation of code review. Keeping the code review process running smoothly at Google is no different, and it requires a number of best practices to ensure that code review is worth the effort put into the process. Most of those practices emphasize keeping the process nimble and quick so that code review can scale properly.

Be Polite and Professional

As pointed out in the Culture section of this book, Google heavily fosters a culture of trust and respect. This filters down into our perspective on code review. A software engineer needs a "looks good to me" from only one other engineer to satisfy our requirement on code comprehension, for example. Many engineers make comments and "looks good to me" a change, with the understanding that the change can be submitted after those changes are made, without any additional rounds of review. That said, code reviews can introduce anxiety and stress to even the most capable engineers. It is critically important to keep all feedback and criticism firmly in the professional realm.

In general, reviewers should defer to authors on particular approaches, and only point out alternatives if the author's approach is deficient. If an author can demonstrate that several approaches are equally valid, the reviewer should accept the preference of the author. Even in those cases, if defects are found in an approach, consider the review a learning opportunity (for both sides!). All comments should remain strictly professional. Reviewers should be careful about jumping to conclusions based on a code author's particular approach. It's better to ask questions on why something was done the way it was before assuming that approach is wrong.

Reviewers should be prompt with their feedback. At Google, we expect feedback from a code review within 24 (working) hours. If a reviewer is unable to complete a review in that time, it's good practice (and expected) to respond that they've at least seen the change and will get to the review as soon as possible. Reviewers should avoid responding to the code review in piecemeal fashion. Few things annoy an author more than getting feedback from a review, addressing it, and then continuing to get unrelated further feedback in the review process.

As much as we expect professionalism on the part of the reviewer, we expect professionalism on the part of the author, as well. Remember that you are not your code, and that this change you propose is not "yours" but the team's. After you check that piece of code into the codebase, it is no longer yours in any case. Be receptive to questions on your approach, and be prepared to explain why you did things in certain ways. Remember that part of the responsibility of an author is to make sure this code is understandable and maintainable for the future.

It's important to treat each reviewer comment within a code review as a TODO item; a particular comment might not need to be accepted without question, but it should at least be addressed. If you disagree with a reviewer's comment, let them know, and let them know why and don't mark a comment as resolved until each side has had a chance to offer alternatives. One common way to keep such debates civil if an author doesn't agree with a reviewer is to offer an alternative and ask the reviewer to PTAL (please take another look). Remember that code review is a learning opportunity, for both the reviewer and the author. That insight often helps to mitigate any chances for disagreement.

By the same token, if you are an owner of code and responding to a code review within your codebase, be amenable to changes from an outside author. As long as the change is an improvement to the codebase, you should still give deference to the author that the change indicates something that could and should be improved.

Write Small Changes

Probably the most important practice to keep the code review process nimble is to keep changes small. A code review should ideally be easy to digest and focus on a single issue, both for the reviewer and the author. Google's code review process discourages massive changes consisting of fully formed projects, and reviewers can rightfully reject such changes as being too large for a single review. Smaller changes also prevent engineers from wasting time waiting for reviews on larger changes, reducing downtime. These small changes have benefits further down in the software development process, as well. It is far easier to determine the source of a bug within a change if that particular change is small enough to narrow it down.

That said, it's important to acknowledge that a code review process that relies on small changes is sometimes difficult to reconcile with the introduction of major new features. A set of small, incremental code changes can be easier to

digest individually, but more difficult to comprehend within a larger scheme. Some engineers at Google admittedly are not fans of the preference given to small changes. Techniques exist for managing such code changes (development on integration branches, management of changes using a diff base different than HEAD), but those techniques inevitably involve more overhead. Consider the optimization for small changes just that: an optimization, and allow your process to accommodate the occasional larger change.

"Small" changes should generally be limited to about 200 lines of code. A small change should be easy on a reviewer and, almost as important, not be so cumbersome that additional changes are delayed waiting for an extensive review. Most changes at Google are expected to be reviewed within about a day.[8] (This doesn't necessarily mean that the review is over within a day, but that initial feedback is provided within a day.) About 35% of the changes at Google are to a single file.[9] Being easy on a reviewer allows for quicker changes to the codebase and benefits the author, as well. The author wants a quick review; waiting on an extensive review for a week or so would likely impact follow-on changes. A small initial review also can prevent much more expensive wasted effort on an incorrect approach further down the line.

Because code reviews are typically small, it's common for almost all code reviews at Google to be reviewed by one and only one person. Were that not the case—if a team were expected to weigh in on all changes to a common codebase—there is no way the process itself would scale. By keeping the code reviews small, we enable this optimization. It's not uncommon for multiple people to comment on any given change—most code reviews are sent to a team member, but also CC'd to appropriate teams—but the primary reviewer is still the one whose "looks good to me" is desired, and only one "looks good to me" is necessary for any given change. Any other comments, though important, are still optional.

Keeping changes small also allows the "approval" reviewers to more quickly approve any given changes. They can quickly inspect whether the primary code reviewer did due diligence and focus purely on whether this change augments the codebase while maintaining code health over time.

Write Good Change Descriptions

A change description should indicate its type of change on the first line, as a summary. The first line is prime real estate and is used to provide summaries within the code review tool itself, to act as the subject line in any associated

emails, and become the visible line Google engineers see in a history summary within CodeSearch, so that first line is important.

Although the first line should be a summary of the entire change, the description should still go into detail on what is being changed *and why*. A description of "Bug fix" is not helpful to a reviewer or a future code archeologist. If several related modifications were made in the change (while still keeping it on message and small) enumerate them within a list. The description is the historical record for this change and tools such as CodeSearch allow you to find who wrote what line in any particular change in the codebase. Drilling down into the original change is often useful when trying to fix a bug.

Descriptions aren't the only opportunity for adding documentation to a change. When writing a public API, you generally don't want to leak implementation details, but by all means do so within the actual implementation, where you should comment liberally. If a reviewer does not understand why you did something, even if it is correct, it is a good indicator that such code needs better structure or better comments (or both). If, during the code review process, a new decision is reached, update the change description, or add appropriate comments within the implementation. A code review is not just something that you do in the present time; it is something you do to record what you did for posterity.

Keep Reviewers to a Minimum

Most code reviews at Google are reviewed by precisely one reviewer.[10] Because the code review process allows the bits on code correctness, owner acceptance, and language readability to be handled by one individual, the code review process scales quite well across an organization the size of Google.

There is a tendency within the industry, and within individuals, to try to get additional input (and unanimous consent) from a cross-section of engineers. After all, each additional reviewer can add their own particular insight to the code review in question. But we've found that this leads to diminishing returns; the most important "looks good to me" is the first one, and subsequent ones don't add as much as you might think to the equation. The cost of additional reviewers quickly outweighs their value.

The code review process is optimized around the trust we place in our engineers to do the right thing. In certain cases, it can be useful to get a

particular change reviewed by multiple people, but even in those cases, those reviewers should focus on different aspects of the same change.

Automate Where Possible

Code review is a human process, and that human input is important, but if there are components of the code process that can be automated, try to do so. Opportunities to automate mechanical human tasks should be explored; investments in proper tooling reap dividends. At Google, our code review tooling allows authors to automatically submit and automatically sync changes to the source control system upon approval (usually used for fairly simple changes).

One of the most important technological improvements regarding automation over the past few years is automatic static analysis of a given code change. (See Chapter 20). Rather than require authors to run tests, linters, or formatters, the current Google code review tooling provides most of that utility automatically through what is known as *presubmits*. A presubmit process is run when a change is initially sent to a reviewer. Before that change is sent, the presubmit process can detect a variety of problems with the existing change, reject the current change (and prevent sending an awkward email to a reviewer), and ask the original author to fix the change first. Such automation not only helps out with the code review process itself; it allows the reviewers to focus on more important concerns than formatting.

Types of Code Reviews

All code reviews are not alike! Different types of code review require different levels of focus on the various aspects of the review process. Code changes at Google generally fall into one of the following buckets (though there is sometimes overlap):

- Greenfield reviews and new feature development
- Behavioral changes, improvements, and optimizations
- Bug fixes and rollbacks
- Refactorings and large-scale changes

Greenfield Code Reviews

The least common type of code review is that of entirely new code, a so-called "greenfield" review. A greenfield review is the most important time to

evaluate whether the code will stand the test of time: that it will be easier to maintain as time and scale change the underlying assumptions of the code. Of course, the introduction of entirely new code should not come as a surprise. As mentioned earlier in this chapter, code is a liability, so the introduction of entirely new code should generally solve a real problem rather than simply provide yet another alternative. At Google, we generally require new code and/or projects to undergo an extensive design review, apart from a code review. A code review is not the time to debate design decisions already made in the past (and by the same token, a code review is not the time to introduce the design of a proposed API).

To ensure that code is sustainable, a greenfield review should ensure that an API matches an agreed design (which may require reviewing a design document) and is tested *fully*, with all API endpoints having some form of unit test, and that those tests fail when the code's assumptions change. (See Chapter 11). The code should also have proper owners (one of the first reviews in a new project is often of a single OWNERS file for the new directory), be sufficiently commented, and provide supplemental documentation, if needed. A greenfield review might also necessitate the introduction of a project into the continuous integration system. (See Chapter 23).

Behavioral Changes, Improvements, and Optimizations

Most changes at Google generally fall into the broad category of modifications to existing code within the codebase. These additions may include modifications to API endpoints, improvements to existing implementations, or optimizations for other factors such as performance. Such changes are the bread and butter of most software engineers.

In each of these cases, the guidelines that apply to a greenfield review also apply: is this change necessary, and does this change improve the codebase? Some of the best modifications to a codebase are actually deletions! Getting rid of dead or obsolete code is one of the best ways to improve the overall code health of a codebase.

Any behavioral modifications should necessarily include revisions to appropriate tests for any new API behavior. Augmentations to the implementation should be tested in a Continuous Integration (CI) system to ensure that those modifications don't break any underlying assumptions of the existing tests. As well, optimizations should of course ensure that they don't affect those tests and might need to include performance benchmarks

for the reviewers to consult. Some optimizations might require benchmark tests, as well.

Bug Fixes and Rollbacks

Inevitably, you will need to submit a change for a bug fix to your codebase. *When doing so, avoid the temptation to address other issues*. Not only does this risk increasing the size of the code review, it also makes it more difficult to perform regression testing or for others to rollback your change. A bug fix should focus solely on fixing the indicated bug, and (usually) updating associated tests to catch the error that occurred in the first place.

Addressing the bug with a revised test is often necessary. The bug surfaced because existing tests were either inadequate, or the code had certain assumptions which were not met. As a reviewer of a bug fix, it is important to ask for updates to unit tests if applicable.

Sometimes, a code change in a codebase as large as Google's causes some dependency to fail that was either not detected properly by tests, or unearths an untested part of the codebase. In those cases, Google allows such changes to be "rolled back," usually by the affected downstream customers. A rollback consists of a change that essentially undoes the previous change. Such rollbacks can be created in seconds because they just revert the previous change to a known state, but still require a code review.

It also becomes critically important that any change that could cause a potential rollback (and that includes all changes!) be as small and atomic as possible, so that a rollback, if needed, does not cause further breakages on other dependencies that can be difficult to untangle. At Google, we've seen developers start to depend on new code very quickly after it is submitted, and rollbacks sometimes break these developers as a result. Small changes help to mitigate these concerns, both because of their atomicity, and because reviews of small changes tend to be done quickly.

Refactorings and Large-Scale Changes

Many changes at Google are automatically generated: the author of the change isn't a person, but a machine. We discuss more about the large-scale change (LSC) process in Chapter (Chapter 22), but even machine-generated changes require review. In cases where the change is considered low risk, they are reviewed by designated reviewers, who have approval privileges for our entire codebase. But for cases in which the change might be risky or

otherwise requires local domain expertise, individual engineers might be asked to review automatically generated changes as part of their normal workflow.

At first look, a review for an automatically generated change should be handled the same as any other code review: the reviewer should check for correctness and applicability of the change. However, we encourage reviewers to limit comments in the associated change and only flag concerns which are specific to their code, not the underlying tool or LSC generating the changes. While the specific change might be machine generated, the overall process generating these changes has already been reviewed, and individual teams cannot hold a veto over the process, or it would not be possible to scale such changes across the organization. If there is a concern about the underlying tool or process, reviewers can escalate out of band to an LSC oversight group for more information.

We also encourage reviewers of automatic changes to avoid expanding their scope. When reviewing a new feature or a change written by a teammate, it is often reasonable to ask the author to address related concerns within the same change, so long as the request still follows the earlier advice to keep the change small. This does not apply to automatically generated changes, because the human running the tool might have hundreds of changes in flight, and even a small percentage of changes with review comments or unrelated questions limits the scale at which the human can effectively operate the tool.

Conclusion

Code review is one of the most important and critical processes at Google. Code review acts as the glue connecting engineers with one another, and the code review process is the primary developer workflow upon which almost all other processes must hang, from testing to static analysis to CI. A code review process must scale appropriately, and for that reason best practices, including small changes and rapid feedback and iteration, are important to maintain developer satisfaction and appropriate production velocity.

TL;DRs

- Code review has many benefits, including ensuring code correctness, comprehension, and consistency across a codebase.

- Always check your assumptions through someone else: optimize for the reader.
- Provide the opportunity for critical feedback while remaining professional.
- Code review is important for knowledge sharing throughout an organization.
- Automation is critical for scaling the process.
- The code review itself provides a historical record.

1 We also use Gerrit to review Git code, primarily for our open source projects. However, Critique is the primary tool of a typical software engineer at Google.

2 McConnell, Steve. *Code Complete*. Microsoft Press, 2004.

3 *https://twitter.com/TitusWinters/status/1132640942878613515*

4 At Google, "readability" does not refer simply to comprehension, but to the set of styles and best practices that allow code to be maintainable to other engineers. See Chapter 3.

5 Some changes to documentation and configurations might not require a code review, but often it is still preferable to obtain such a review.

6 "Advances in Software Inspection," *IEEE Transactions on Software Engineering*, SE-12(7): 744–751, July 1986. Granted, this study took place before robust tooling and automated testing had become so important in the software development process, but the results still seem relevant in the modern software age.

7 Rigby, Peter C. and Christian Bird. 2013. Convergent software peer review practices. In FSE.

8 "Modern Code Review: A Case Study at Google" Caitlin Sadowski, Emma Söderberg, Luke Church, Michal Sipko, and Alberto Bacchelli.

9 Ibid.

10 "Modern Code Review: A Case Study at Google" Caitlin Sadowski, Emma Söderberg, Luke Church, Michal Sipko, and Alberto Bacchelli.

Chapter 10. Documentation

Written by Tom Manshreck

Edited by Riona MacNamara

Of the complaints most engineers have about writing, using, and maintaining code, a singular common frustration is the lack of quality documentation. "What are the side effects of this method?" "I got an error after step 3" "What does this acronym mean?" "Is this document up to date?" Every software engineer has voiced complaints about the quality, quantity, or sheer lack of documentation throughout their career, and the software engineers at Google are no different.

Technical writers and project managers may help, but software engineers will always need to write most documentation themselves. Engineers therefore need the proper tools and incentives to do so effectively. The key to making it easier for them to write quality documentation is to introduce processes and tools that scale with the organization and that tie into their existing workflow.

Overall, the state of engineering documentation in the late 2010s is similar to the state of software testing in the late 1980s. Everyone recognizes that more effort needs to be made to improve it, but there is not yet organizational recognition of its critical benefits. That is changing, if slowly. At Google, our most successful efforts have been when documentation is *treated like code* and incorporated into the traditional engineering workflow, making it easier for engineers to write and maintain simple documents.

What Qualifies as Documentation?

When we refer to "documentation," we're talking about every supplemental text that an engineer needs to write to do their job: not only standalone documents, but code comments, as well. (In fact, most of the documentation an engineer at Google writes comes in the form of code comments.) We'll discuss the various types of engineering document further in this chapter.

Why Is Documentation Needed?

Quality documentation has tremendous benefits for an engineering organization. Code and APIs become more comprehensible, reducing mistakes. Project teams are more focused when their design goals and team objectives are clearly stated. Manual processes are easier to follow when the steps are clearly outlined. Onboarding new members to a team or code base takes much less effort if the process is clearly documented.

But because documentation's benefits are all necessarily downstream, they generally don't reap immediate benefits to the author. Unlike testing, which (as we've seen) quickly provides benefits to a programmer, documentation generally requires more effort upfront and doesn't provide clear benefits to an author until later. But, like investments in testing, the investment made in documentation will pay for itself over time. After all, you might write a document only once,[1] but it will be read hundreds, perhaps thousands of times afterward; its initial cost is amortized across all the future readers. Not only does documentation scale over time, but it is critical for the rest of the organization to scale, as well. It helps answer questions like these:

- Why were these design decisions made?
- Why did we implement this code in this manner?
- Why did *I* implement this code in this manner, if you're looking at your own code two years later?

If documentation conveys all these benefits, why is it generally considered "poor" by engineers? One reason, as we've mentioned, is that the benefits aren't *immediate*, especially to the writer. But there are several other reasons:

- Engineers often view writing as a separate skill than that of programming. (We'll try to illustrate that this isn't quite the case, and even where it is, it isn't necessarily a separate skill from that of *software engineering*.)
- Some engineers don't feel like they are capable writers. But you don't need a robust command of English[2] to produce workable documentation. You just need to step outside yourself a bit and see things from the audience's perspective.
- Writing documentation is often more difficult because of limited tools support or integration into the developer workflow.
- Documentation is viewed as an extra burden—something else to maintain—rather than something that will make maintenance of their existing code easier.

Not every engineering team needs a technical writer (and even if that were the case, there aren't enough of them). This means that engineers will, by and large, write most of the documentation themselves. So, instead of forcing engineers to become technical writers, we should instead think about how to make writing documentation easier for engineers. Deciding how much effort to devote to documentation is a decision your organization will need to make at some point.

Documentation benefits several different groups. Even to the writer, documentation provides the following benefits:

- It helps formulate an API. Writing documentation is one of the surest ways to figure out if your API makes sense. Often, the writing of the documentation itself leads engineers to reevaluate design decisions that otherwise wouldn't be questioned. If you can't explain it and can't define it, you probably haven't designed it well enough.
- It provides a road map for maintenance and a historical record. Tricks in code should be avoided, in any case, but good comments helps out a great deal when you're staring at code you wrote two years ago trying to figure out what's wrong.
- It makes your code look more professional and drive traffic. Developers will naturally assume that a well-documented API is a better-designed API. That's not always the case, but they are often highly correlated. Although this benefit sounds cosmetic, it's not quite so: whether a product has good documentation is usually a pretty good indicator of how well a product will be maintained.
- It will prompt fewer questions from other users. This is probably the biggest benefit over time to someone writing the documentation. If you have to explain something to someone more than once, it usually makes sense to document that process.

As great as these benefits are to the writer of documentation, the lion's share of documentation's benefits will naturally accrue to the reader. Google's C++ Style Guide notes the maxim "optimize for the reader." This maxim applies not just to code, but to the comments around code, or the documentation set attached to an API. Much like testing, the effort you put into writing good documents will reap benefits many times over its lifetime. Documentation is critical over time, and reaps tremendous benefits for especially critical code as an organization scales.

Documentation Is Like Code

Software engineers who write in a single, primary programming language still often reach for different languages to solve specific problems. An engineer might write shell scripts or Python to run command-line tasks, or they might write most of their backend code in C++ but write some middleware code in Java, and so on. Each language is a tool in the toolbox.

Documentation should be no different: it's a tool, written in a different language (usually English) to accomplish a particular task. Writing documentation is not much different than writing code. Like a programming language, it has rules, a particular syntax, and style decisions, often to accomplish a similar purpose as that within code: enforce consistency, improve clarity, and avoid (comprehension) errors. Within technical documentation, grammar is important not because one needs rules, but to standardize the voice and avoid confusing or distracting the reader. Google requires a certain comment style for many of its languages for this reason.

Like code, documents should also have owners. Documents without owners become stale and difficult to maintain. Clear ownership also makes it easier to handle documentation through existing developer workflows: bug tracking systems, code review tooling, and so forth. Of course, documents with different owners can still conflict with one another. In those cases, it is important to designate *canonical* documentation: determine the primary source and consolidate other associated documents into that primary source (or deprecate the duplicates).

The prevalent usage of "go/ links" at Google (see Chapter 3) makes this process easier. Documents with straightforward go/ links often become the canonical source of truth. One other way to promote canonical documents is to associate them directly with the code they document by placing them directly under source control and alongside the source code itself.

Documentation is often so tightly coupled to code that it should, as much as possible, be treated *as code*. That is, your documentation should:

- Have internal policies or rules to be followed
- Be placed under source control
- Have clear ownership responsible for maintaining the docs
- Undergo reviews for changes (and change *with* the code it documents)
- Have issues tracked, as bugs are tracked in code

- Be periodically evaluated (tested, in some respect)
- If possible, be measured for aspects such as accuracy, freshness, etc. (tools have still not caught up here)

The more engineers treat documentation as "one of" the necessary tasks of software development, the less they will resent the upfront costs of writing, and the more they will reap the long-term benefits. In addition, making the task of documentation easier reduces those upfront costs.

Callout: The Google Wiki

When Google was much smaller and leaner, it had few technical writers. The easiest way to share information was through our own internal wiki (GooWiki). At first, this seemed like a reasonable approach; all engineers shared a single documentation set and could update it as needed.

But as Google scaled, problems with a wiki-style approach became apparent. Because there were no true owners for documents, many became obsolete.[3] Because no process was put in place for adding new documents, duplicate documents and document sets began appearing. GooWiki had a flat namespace, and people were not good at applying any hierarchy to the documentation sets. At one point, there were 7 to 10documents (depending on how you counted them) on setting up Borg, our production compute environment, only a few of which seemed to be maintained, and most that were specific to certain teams with certain permissions and assumptions.

Another problem with GooWiki became apparent over time: the people who could fix the documents were not the people who used them. New users discovering bad documents either couldn't confirm that the documents were wrong or didn't have an easy way to report errors. They knew something was wrong (because the document didn't work) but they couldn't "fix" it. Conversely, the people best able to fix the documents often didn't need to consult them after they were written. The documentation became so poor as Google grew that the quality of documentation became Google's number one developer complaint on our annual developer surveys.

The way to improve the situation was to move important documentation under the same sort of source control that was being used to track code changes. Documents began to have their own owners, canonical locations within the source tree, and processes for identifying bugs and fixing them; the documentation began to dramatically improve. Additionally, the way documentation was written and maintained began to look the same as how

code was written and maintained. Errors in the documents could be reported within our bug tracking software. Changes to the documents could be handled using the existing code review process. Eventually, engineers began to fix the documents themselves, or send changes to technical writers (who were often the owners).

Moving documentation to source control was initially met with a lot of controversy. Many engineers were convinced that doing away with the GooWiki, that bastion of freedom of information, would lead to poor quality because the bar for documentation (requiring a review, requiring owners for documents, etc.) would be higher. But that wasn't the case. The documents became better.

The introduction of Markdown as a common documentation formatting language also helped because it made it easier for engineers to understand how to edit documents without needing specialized expertise in HTML or CSS. Google eventually introduced its own framework for embedding documentation within code: g3doc. With that framework, documentation improved further, as documents existed side by side with the source code within the engineer's development environment. Now, engineers could update the code and its associated documentation in the same change (a practice for which we're still trying to improve adoption).

The key difference was that maintaining documentation became a similar experience to maintaining code: engineers filed bugs, made changes to documents in changelists, sent changes to reviews by experts, and so on. Leveraging of existing developer workflows rather than creating new ones was a key benefit.

Know Your Audience

One of the most important mistakes that engineers make when writing documentation is to write only for themselves. It's natural to do so, and writing for yourself is not without value: after all, you might need to look at this code in a few years and try to figure out what you once meant. You also might be of approximately the same skill set as someone reading your document. But if you write only for yourself, you are going to make certain assumptions, and given that your document might be read by a very wide audience (all of engineering, external developers), even a few lost readers is a large cost. As an organization grows, mistakes in documentation become more prominent, and your assumptions often do not apply.

Instead, before you begin writing, you should (formally or informally) identify the audience(s) your documents need to satisfy. A design document might need to persuade decision makers. A tutorial might need to provide very explicit instructions to someone utterly unfamiliar with your codebase. An API might need to provide complete and accurate reference information for any users of that API, be they experts or novices. Always try to identify a primary audience and write to that audience.

Good documentation need not be polished or "perfect." One mistake engineers make when writing documentation is assuming they need to be much better writers. By that measure, few software engineers would write. Think about writing like you do about testing or any other process you need to do as an engineer. Write to your audience, in the voice and style that they expect. If you can read, you can write. Remember that your audience is standing where you once stood, but *without your new domain knowledge*. So you don't need to be a great writer; you just need to get someone like you as familiar with the domain as you now are. (And as long as you get a stake in the ground, you can improve this document over time.)

Types of Audiences

We've pointed out that you should write at the skill level and domain knowledge appropriate for your audience. But who precisely is your audience? Chances are, you have multiple audiences based on one or more of the following criteria:

- Experience level (expert programmers, or junior engineers who might not even be familiar—gulp!—with the language.)
- Domain knowledge (team members, or other engineers in your organization who are familiar only with API endpoints).
- Purpose (end users who might need your API to do a specific task and need to find that information quickly, or software gurus who are responsible for the guts of a particularly hairy implementation that you hope no one else needs to maintain).

In some cases, different audiences require different writing styles, but in most cases, the trick is to write in a way that applies as broadly to your different audience groups as possible. Often, you will need to explain a complex topic to both an expert and a novice. Writing for the expert with domain knowledge may allow you to cut corners, but you'll confuse the novice; conversely, explaining everything in detail to the novice will doubtless annoy the expert.

Obviously, writing such documents is a balancing act and there's no silver bullet, but one thing we've found is that it helps to keep your documents *short*. Write descriptively enough to explain complex topics to people unfamiliar with the topic, but don't lose or annoy experts. Writing a short document often requires you to write a longer one (getting all the information down) and then doing an edit pass, removing duplicate information where you can. This might sound tedious, but keep in mind that this expense is spread across all the readers of the documentation. As Blaise Pascal once said, "If I had more time, I would have written you a shorter letter." By keeping a document short and clear, you will ensure that it will satisfy both an expert and a novice.

Another important audience distinction is based on how a user encounters a document:

- *Seekers* are engineers who *know what they want* and want to know if what they are looking at fits the bill. A key pedagogical device for this audience is *consistency*. If you are writing reference documentation for this group (within a code file, for example), you will want to have your comments follow a similar format, for example, so that readers can quickly scan a reference and see whether they find what they are looking for.
- *Stumblers* might not know exactly what they want. They might have only a vague idea of how to implement what they are working with. The key for this audience is *clarity*. Provide overviews or introductions (at the top of a file, for example) that explain the purpose of the code they are looking at. It's also useful to identify when a doc is *not* appropriate for an audience. A lot of documents at Google begin with a "TL;DR; statement" such as "tldr; if you are not interested in C++ compilers at Google, you can stop reading now."

Finally, one important audience distinction is between that of a customer (e.g., a user of an API) and that of a provider (e.g., a member of the project team). As much as possible, documents intended for one should be kept apart from documents intended for the other. Implementation details are important to a team member for maintenance purposes; end users should not need to read such information. Often, engineers denote design decisions within the reference API of a library they publish. Such reasonings belong more appropriate in specific documents (design documents) or, at best, within the implementation details of code hidden behind an interface.

Documentation Types

Engineers write various different types of documentation as part of their work: design documents, code comments, how-to documents, project pages, and more. These all count as "documentation." But it is important to know the different types, and to *not mix types*. A document should have, in general, a singular purpose, and stick to it. Just as an API should do one thing and do it well, avoid trying to do several things within one document. Instead, break out those pieces more logically.

There are several main types of documents that software engineers often need to write:

- Reference documentation, including code comments
- Design documents
- Tutorials
- Conceptual documentation
- Landing pages

It was common in the early days of Google for teams to have monolithic wiki pages with bunches of links (many broken or obsolete), some conceptual information about how the system worked, an API reference, and so on, all sprinkled together. Such documents fail because they don't serve a single purpose (and they also get so long that no one will read them; some notorious wiki pages scrolled through several dozens of screens). Instead, make sure your document has a singular purpose, and if adding something to that page doesn't make sense, you probably want to find, or even create, another document for that purpose.

Reference Documentation

Reference documentation is the most common type that engineers need to write; indeed, often they need to write some form of reference documents every day. By reference documentation, we mean anything that documents the usage of code within the codebase. Code comments are the most common form of reference documentation that an engineer must maintain. Such comments can be divided into two basic camps: API comments versus implementation comments. Remember the audience differences between these two: API comments don't need to discuss implementation details or design decisions and can't assume a user is as versed in the API as the author. Implementation comments, on the other hand, can assume a lot more domain

knowledge of the reader, though be careful in assuming too much: people leave projects and sometimes it's safer to be methodical about exactly why you wrote this code the way you did.

Most reference documentation, even when provided as separate documentation from the code, is generated from comments within the codebase itself. (As it should; reference documentation should be single-sourced as much as possible.) Some languages such as Java or Python have specific commenting frameworks (Javadoc, PyDoc, GoDoc) meant to make generation of this reference documentation easier. Other languages, such as C++, have no standard "reference documentation" implementation, but because C++ separates out its API surface (in header or *.h* files) from the implementation (*.cc* files), header files are often a natural place to document a C++ API.

Google takes this approach: a C++ API deserves to have its reference documentation live within the header file. Other reference documentation is embedded directly in the Java, Python, and Go source code, as well. Because Google's Chapter 17 browser is so robust, we've found little benefit to providing separate generated reference documentation. Users in CodeSearch not only search code easily, they can usually find the original definition of that code as the top result. Having the documentation alongside the code's definitions also makes the documentation easier to discover and maintain.

We all know that code comments are essential to a well-documented API. But what precisely is a "good" comment? Earlier in this chapter, we identified two major audiences for reference documentation: seekers and stumblers. Seekers know what they want; stumblers don't. The key win for seekers is a consistently commented codebase so that they can quickly scan an API and find what they are looking for. The key win for stumblers is clearly identifying the purpose of an API, often at the top of a file header. We'll walk through some code comments in the subsections that follow. The code commenting guidelines that follow apply to C++, but similar rules are in place at Google for other languages.

FILE COMMENTS

Almost all code files at Google must contain a file comment. (Some header files that contain only one utility function, etc., might deviate from this standard.) File comments should begin with a header of the following form:

```
// ------------------------------------------------------------
-----------------
```

```
// str_cat.h

// ----------------------------------------------------------------
-----------------

//

// This header file contains functions for efficiently
concatenating and appending

// strings: StrCat() and StrAppend(). Most of the work within
these routines is

// actually handled through use of a special AlphaNum type,
which was designed

// to be used as a parameter type that efficiently manages
conversion to

// strings and avoids copies in the above operations.

…
```

Generally, a file comment should begin with an outline of what's contained in the code you are reading. It should identify the code's main use cases, and intended audience (in the preceding case, developers who want to concatenate strings). Any API that cannot be succinctly described in the first paragraph or two is usually the sign of an API that is not well thought out. Consider breaking the API into separate components, in those cases.

CLASS COMMENTS

Most modern programming languages are object oriented. Class comments are therefore important for defining the API "objects" in use in a codebase. All public classes (and structs) at Google must contain a class comment describing the class/struct, important methods of that class, and the purpose of the class. Generally, class comments should be "nouned" with documentation emphasizing their object aspect. That is, say "The Foo class contains x, y, z, allows you to do Bar, and has the following Baz aspects" and so on.

Class comments should generally begin with a comment of the following form:

```
// ------------------------------------------------------------
-----------------

// AlphaNum

// ------------------------------------------------------------
-----------------

//

// The AlphaNum class acts as the main parameter type for
StrCat() and

// StrAppend(), providing efficient conversion of numeric,
boolean, and

// hexadecimal values (through the Hex type) into strings.
```

FUNCTION COMMENTS

All free functions, or public methods of a class, at Google must also contain a function comment describing what the function *does*. Function comments should stress the *active* nature of their use, beginning with an indicative verb describing what the function does, and what is returned.

Function comments should generally begin with a comment of the following form:

```
// StrCat()

//

// Merges the given strings or numbers, using no delimiter(s),
returning the merged result as a string.

…
```

Note that starting a function comment with a declarative verb introduces consistency across a header file. A seeker can quickly scan an API and read just the verb to get an idea of whether the function is appropriate: “Merges, Deletes, Creates,” and so on.

Some documentation styles (and some documentation generators) require various forms of boilerplate on function comments, like “Returns:”, “Throws:”, and so forth, but at Google we haven’t found them to be

necessary. It is often clearer to present such information in a single prose comment that's not broken up into artificial section boundaries:

```
// Creates a new record for a customer with the given name and
address,

// and returns the record ID, or throws `DuplicateEntryError`
if a

// record with that name already exists.

int AddCustomer(string name, string address);
```

Notice how the postcondition, parameters, return value, and exceptional cases are naturally documented together (in this case, in a single sentence), because they are not independent of one another. Adding explicit boilerplate sections would make the comment more verbose and repetitive, but no clearer (and arguably less clear).

Design Docs

Most teams at Google require an approved design document before starting work on any major project. A software engineer typically writes the proposed design document using a specific design doc template approved by the team. Such documents are designed to be collaborative, so they are often shared in Google Docs, which has good collaboration tools. Some teams require such design documents to be discussed and debated at specific team meetings, where the finer points of the design can be discussed or critiqued by experts. In some respects, these design discussions act as a form of code review before any code is written.

Because the development of a design document is one of the first processes an engineer undertakes before deploying a new system, it is also a convenient place to ensure that various concerns are covered. The canonical design document templates at Google require engineers to consider aspects of their design such as security implications, internationalization, storage requirements and privacy concerns, and so on. In most cases, such parts of those design documents are reviewed by experts in those domains.

A good design document should cover the goals of the design, its implementation strategy, and propose key design decisions with an emphasis on their individual trade-offs. The best design documents suggest design goals and cover alternative designs, denoting their strong and weak points.

A good design document, once approved, also acts not only as a historical record, but as a measure of whether the project successfully achieved its goals. Most teams archive their design documents in an appropriate location within their team documents so that they can review them at a later time. It's often useful to review a design document before a product is launched to ensure that the stated goals when the design document was written remain the stated goals at launch (and if they do not, either the document or the product can be adjusted accordingly).

Tutorials

Every software engineer, when they join a new team, will want to get up to speed as quickly as possible. Having a tutorial that walks someone through the setup of a new project is invaluable; "Hello World" has established itself is one of the best ways to ensure that all team members start off on the right foot. This goes for documents as well as code. Most projects deserve a "Hello World" document that assumes nothing and gets the engineer to make something "real" happen.

Often, the best time to write a tutorial, if one does not yet exist, is when you first join a team. (It's also the best time to find bugs in any existing tutorial you are following.) Get a notepad or other way to take notes, and write down everything you need to do along the way, assuming no domain knowledge or special setup constraints; after you're done, you'll likely know what mistakes you made during the process—and why—and can then edit down your steps to get a more streamlined tutorial. Importantly, write *everything* you need to do along the way; try not to assume any particular setup, permissions, or domain knowledge. If you do need to assume some other setup, state that clearly in the beginning of the tutorial as a set of prerequisites.

Most tutorials require you to perform a number of steps, in order. In those cases, number those steps explicitly. If the focus of the tutorial is on the *user* (say, for external developer documentation), then number each action that a user needs to undertake. Don't number actions that the system may take in response to such user actions. It is critical and important to number explicitly every step when doing this. Nothing is more annoying than an error on step 4 because you forget to tell someone to properly authorize their username, for example.

EXAMPLE: A BAD TUTORIAL

1. Download the package from our server at http://example.com
2. Copy the shell script to your home directory.

3. Execute the shell script.
4. The foobar system will communicate with the authentication system.
5. Once authenticated, foobar will bootstrap a new database named "baz"
6. Test "baz" by executing a SQL command on the command line.
7. Type: CREATE DATABASE my_foobar_db;

In the preceding procedure, steps 4 and 5 happen on the server end. It's unclear whether the user needs to do anything, but they don't, so those side effects can be mentioned as part of step 3. As well, it's unclear whether step 6 and step 7 are different. (They aren't.) Combine all atomic user operations into single steps, so that the user knows they need to do something at each step in the process. Also, if your tutorial has user-visible input or output, denote that on separate lines (often using the convention of a `monospaced bold` font.)

EXAMPLE: A BAD TUTORIAL MADE BETTER

1. Download the package from our server at http://example.com:
   ```
   $ curl -I http://example.com
   ```
2. Copy the shell script to your home directory:
   ```
   $ cp foobar.sh ~
   ```
3. Execute the shell script in your home directory:
   ```
   $ cd ~; foobar.sh
   ```
   The foobar system will first communicate with the authentication system. Once authenticated, foobar will bootstrap a new database named "baz" and open an input shell.
4. Test "baz" by executing a SQL command on the command line:
   ```
   baz:$ CREATE DATABASE my_foobar_db;
   ```
   Note how each step requires specific user intervention. If, instead, the tutorial had a focus on some other aspect (e.g., a document about the "life of a server"), number those steps from the perspective of that focus (what the server does).

Conceptual Documentation

Some code requires deeper explanations or insights than can be obtained simply by reading the reference documentation. In those cases, we need conceptual documentation to provide overviews of the APIs or systems. Some examples of conceptual documentation might be a library overview for a popular API, a document describing the life cycle of data within a server, and so on. In almost all cases, a conceptual document is meant to augment,

not replace, a reference documentation set. Often this leads to duplication of some information, but with a purpose: to promote clarity. In those cases, it is not necessary for a conceptual document to cover all edge cases (though a reference should cover those cases religiously). In this case, sacrificing some accuracy is acceptable for clarity. The main point of a conceptual document is to impart understanding.

"Concept" documents are the most difficult forms of documentation to write. As a result, they are often the most neglected type of document within a software engineer's toolbox. One problem engineers face when writing conceptual documentation is that it often cannot be embedded directly within the source code because there isn't a canonical location to place it. Some APIs have a relatively broad API surface area, in which case a file comment might be an appropriate place for a "conceptual" explanation of the API. But often, an API works in conjunction with other APIs and/or modules. The only logical place to document such complex behavior is through a separate conceptual document. If comments are the unit tests of documentation, conceptual documents are the integration tests.

Even when an API is appropriately scoped, it often makes sense to provide a separate conceptual document. For example, Abseil's `StrFormat` library covers a variety of concepts that accomplished users of the API should understand. In those cases, both internally and externally, we provide a format concepts document.

A concept document needs to be useful to a broad audience: both experts and novices alike. Moreover, it needs to emphasize *clarity*, so it often needs to sacrifice completeness (something best reserved for a reference) and (sometimes) strict accuracy. That's not to say a conceptual document should intentionally be inaccurate; it just means that it should focus on common usage and leave rare usages or side effects for reference documentation.

Landing Pages

Most engineers are members of a team, and most teams have a "team page" somewhere on their company's intranet. Often, these sites are a bit of a mess: a typical landing page might contain some interesting links, sometimes several documents titled "read this first!", and some information both for the team and for its customers. Such documents start out useful but rapidly turn into disasters; because they become so cumbersome to maintain, they will eventually get so obsolete that they will be fixed by only the brave or the desperate.

Luckily, such documents look intimidating, but are actually straightforward to fix; ensure that a landing page clearly identifies its purpose, and then include *only* links to other pages for more information. If something on a landing page is doing more than being a traffic cop, it is *not doing its job*. If you have a separate setup document, link to that from the landing page as a separate document. If you have too many links on the landing page (your page should not scroll multiple screens), consider breaking up the pages by taxonomy, under different sections.

Most poorly configured landing pages serve two different purposes: they are the "goto" page for someone who is a user of your product or API, or they are the home page for a team. Don't have the page serve both masters—it will become confusing. Create a separate "team page" as an internal page or the main landing page. What the team needs to know is often quite different than what a customer of your API needs to know.

Documentation Reviews

At Google, all code needs to be reviewed and our code review process is well understood and accepted. In general, documentation also needs review (though this is less universally accepted). If you want to "test" whether your documentation works, you should generally have someone else review it.

A technical document benefits from three different types of reviews, each emphasizing different aspects:

- A technical review, for accuracy. This review is usually done by a subject matter expert, often another member of your team. Often, this is part of a code review itself.
- An audience review, for clarity. This is usually someone unfamiliar with the domain. This might be someone new to your team or a customer of your API.
- A writing review, for consistency. This is often a technical writer or volunteer.

Of course, some of these lines are sometimes blurred, but if your document is high profile, or might end up being externally published, you probably want to ensure that it receives more types of reviews. (We've used a similar review process for this book.) Any document tends to benefit from the aforementioned reviews, even if some of those reviews are ad hoc. That said,

even getting one reviewer to review your text is preferable to having no one review it.

Importantly, if documentation is tied into the engineering workflow, it will often improve over time. Most documents at Google now implicitly go through an audience review because at some point their audience will be using them, and hopefully letting you know when they aren't working (via bugs or other forms of feedback).

Callout: The Developer Guide Library

As mentioned earlier, there were problems associated with having most (almost all) engineering documentation contained within a shared wiki: little ownership of important documentation, competing documentation, obsolete information, and difficulty in filing bugs or issues with documentation. But this problem was not seen in some documents: the Google C++ style guide was owned by a select group of senior engineers (style arbiters) who managed it. The document was kept in good shape because certain people cared about it. They implicitly owned that document. The document was also canonical: there was only one C++ style guide.

As previously mentioned, documentation that sits directly within source code is one way to promote the establishment of canonical documents; if the documentation sits alongside the source code, it should usually be the most applicable (hopefully). At Google, each API usually has a separate *g3doc* directory where such documents live (written as Markdown files and readable within our CodeSearch browser). Having the documentation exist alongside the source code not only establishes de facto ownership, it makes the documentation seem more wholly "part" of the code.

Some documentation sets, however, cannot exist very logically within source code. A "C++ developer guide" for Googlers, for example, has no obvious place to sit within the source code. There is no master "C++" directory where people will look for such information. In this case (and others that crossed API boundaries), it became useful to create standalone documentation sets in their own depot. Many of these culled together associated existing documents into a common set, with common navigation and look-and-feel. Such documents were noted as "Developer Guides" and, like the code in the codebase, were under source control in a specific documentation depot, with this depot organized by topic rather than API. Often, technical writers managed these developer guides, because they were better at explaining topics across API boundaries.

Over time, these developer guides became canonical. Users who wrote competing or supplementary documents became amenable to adding their documents to the canonical document set after it was established, and then deprecating their competing documents. Eventually, the C++ style guide became part of a larger "C++ Developer Guide." As the documentation set became more comprehensive and more authoritative, its quality also improved. Engineers began logging bugs because they knew someone was maintaining these documents. Because the documents were locked down under source control, with proper owners, engineers also began sending changelists directly to the technical writers.

The introduction of go/ links (see Chapter 2) allowed most documents to, de facto, more easily establish themselves as canonical on any given topic. Our C++ Developer Guide became established at "go/cpp," for example. With better internal search, go/ links, and the integration of multiple documents into a common documentation set, such canonical documentation sets became more authoritative and robust over time.

Documentation Philosophy

Caveat: the following section is more of a treatise on technical writing best practices (and personal opinion) than of "how Google does it." Consider it optional for software engineers to fully grasp, though understanding these concepts will likely allow you to more easily write technical information.

WHO, WHAT, WHEN, WHERE, and WHY

Most technical documentation answers a "HOW" question. How does this work? How do I program to this API? How do I set up this server? As a result, there's a tendency for software engineers to jump straight into the "HOW" on any given document and ignore the other questions associated with it: the WHO, WHAT, WHEN, WHERE, and WHY. It's true that none of those are generally as important as the HOW—A design document is an exception because an equivalent aspect is often the WHY—but without a proper framing of technical documentation, documents end up confusing. Try to address the other questions in the first two paragraphs of any document:

- WHO was discussed previously: that's the audience. But sometimes you also need to explicitly call out and address the audience in a document. Example: "This document is for new engineers on the Secret Wizard project."

- WHAT identifies the purpose of this document: "This document is a tutorial designed to start a Frobber server in a test environment." Sometimes, merely writing the WHAT helps you frame the document appropriately. If you start adding information that isn't applicable to the WHAT, you might want to move that information into a separate document.
- WHEN identifies when this document was created, reviewed, or updated. Documents in source code have this date noted implicitly, and some other publishing schemes automate this, as well. But, if not, make sure to note the date on which the document was written (or last revised) on the document itself.
- WHERE is often implicit, as well, but decide where the document should live. Usually, the preference should be under some sort of version control, ideally *with the source code it documents*. But other formats work for different purposes, as well. At Google, we often use Google Docs for easy collaboration, particularly on design issues. At some point, however, any shared document becomes less of a discussion and more of a stable historical record. At that point, move it to someplace more permanent, with clear ownership, version control, and responsibility.
- WHY sets up the purpose for the document. Summarize what you expect someone to take away from the document after reading it. A good rule of thumb is to establish the WHY in the introduction to a document. When you write the summary, verify whether you've met your original expectations (and revise accordingly).

The Beginning, Middle, and End

All documents—indeed, all parts of documents—have a beginning, middle, and end. Although it sounds amazingly silly, most documents should often have, at a minimum, those three sections. A document with only one section has only one thing to say, and very few documents have only one thing to say. Don't be afraid to add sections to your document; they break up the flow into logical pieces and provide readers with a roadmap of what the document covers.

Even the simplest document usually has more than one thing to say. Our popular "C++ Tips of the Week" have traditionally been very short, focusing on one small piece of advice. However, even here, having sections help. Traditionally, the first section denotes the problem, the middle section goes through the recommended solutions, and the conclusion summarizes the

takeaways. Had the document consisted of only one section, some readers would doubtless have difficulty teasing out the important points.

Most engineers loathe redundancy, and with good reason. But in documentation, redundancy is often useful. An important point buried within a wall of text can be difficult to remember or tease out. On the other hand, placing that point at a more prominent location early often loses context provided later on. Usually, the solution is to introduce and summarize the point within an introductory paragraph, and then use the rest of the section to make your case in a more detailed fashion. In this case, redundancy helps the reader understand the importance of what is being stated.

The Parameters of Good Documentation

There are usually three aspects of good documentation: completeness, accuracy, and clarity. You rarely get all three within the same document; as you try to make a document more "complete," for example, clarity can begin to suffer. If you try to document every possible use case of an API, you might end up with an incomprehensible mess. For programming languages, being completely accurate in all cases (and documenting all possible side effects) can also affect clarity. For other documents, trying to be clear about a complicated topic can subtly affect the accuracy of the document; you might decide to ignore some rare side effects in a conceptual document, for example, because the point of the document is to familiarize someone with the usage of an API, not provide a dogmatic overview of all intended behavior.

In each case, a "good document" is defined as the document that is *doing its intended job*. As a result, you rarely want a document doing more than one job. For each document (and for each document type), decide on its focus and adjust the writing appropriately. Writing a conceptual document? You probably don't need to cover every part of the API. Writing a reference? You probably want this complete but perhaps must sacrifice some clarity. Writing a landing page? Focus on organization and keep discussion to a minimum. All of this adds up to quality, which, admittedly, is stubbornly difficult to accurately measure.

How can you quickly improve the quality of a document? Focus on the needs of the audience. Often, less is more. For example, one mistake engineers often make is adding design decisions or implementation details to an API document. Much like you should ideally separate the interface from an implementation within a well-designed API, you should avoid discussing

design decisions in an API document. Users don't need to know this information. Instead, put those decisions in a specialized document for that purpose (usually a design doc).

Deprecating Documents

Just like old code can cause problems, so can old documents. Over time, documents become stale, obsolete, or (often) abandoned. Try as much as possible to avoid abandoned documents, but when a document no longer serves any purpose, either remove it or identify it as obsolete (and, if available, where to go for new information). Even for unowned documents, someone adding a note that "This no longer works!" is more helpful than saying nothing and leaving something that seems authoritative but no longer works.

At Google, we often attach "freshness dates" to documentation. Such documents note the last time a document was reviewed, and metadata in the documentation set will send email reminders when the document hasn't been touched in, for example, three months. Such freshness dates, such as the following example—and tracking your documents as bugs—can help make a documentation set easier to maintain over time, which is the main concern for a document:

```
<!--*

# Document freshness: For more information, see go/fresh-
source.
freshness: { owner: `username` reviewed: '2019-02-27' }
*-->
```

Users who own such a document have an incentive to keep that freshness date current (and if the document is under source control, that requires a code review). As a result, it's a low-cost means to ensure that a document is looked over from time to time. At Google, we found that including the owner of a document in this freshness date within the document itself with a byline of "Last reviewed by..." led to increased adoption, as well.

When Do You Need Technical Writers?

When Google was young and growing, there weren't enough technical writers in software engineering. (That's still the case.) Those projects deemed important tended to receive a technical writer, regardless of whether that

team really needed one. The idea was that the writer could relieve the team of some of the burden of writing and maintaining documents and (theoretically) allow the important project to achieve greater velocity. This turned out to be a bad assumption.

We learned that most engineering teams can write documentation for themselves (their team) perfectly fine; it's only when they are writing documents for another audience that they tend to need help because it's difficult to write to another audience. The feedback loop within your team regarding documents is more immediate, the domain knowledge and assumptions are clearer, and the perceived needs are more obvious. Of course, a technical writer can often do a better job with grammar and organization, but supporting a single team isn't the best use of a limited and specialized resource; it doesn't scale. It introduced a perverse incentive: become an important project and your software engineers won't need to write documents. Discouraging engineers from writing documents turns out to be the opposite of what you want to do.

Because they are a limited resource, technical writers should generally focus on tasks that software engineers *don't* need to do as part of their normal duties. Usually, this involves writing documents that cross API boundaries. Project Foo might clearly know what documentation project Foo needs, but it probably has a less clear idea what Project Bar needs. A technical writer is better able to stand in as a person unfamiliar with the domain. In fact, it's one of their critical roles: to challenge the assumptions your team makes about the utility of your project. It's one of the reasons why many, if not most, software engineering technical writers tend to focus on this specific type of API documentation.

Conclusion

Google has made good strides in addressing documentation quality over the past decade, but to be frank, documentation at Google is not yet a first-class citizen. For comparison, engineers have gradually accepted that testing is necessary for any code change, no matter how small. As well, testing tooling is robust and varied, and plugged into an engineering workflow at various points. Documentation is not ingrained at nearly the same level.

To be fair, there's not necessarily the same need to address documentation as with testing. Tests can be made atomic (unit tests) and can follow prescribed form and function. Documents, for the most part, cannot. Tests can be

automated, and schemes to automate documentation are often lacking. Documents are necessarily subjective; the quality of the document is measured not by the writer, but by the reader, and often quite asynchronously. That said, there is a recognition that documentation is important, and processes around document development are improving. In this author's opinion, the quality of documentation at Google is better than in most software engineering shops.

To change the quality of engineering documentation, engineers—and the entire engineering organization—need to accept that they are both the problem and the solution. Rather than throw up their hands at the state of documentation, they need to realize that producing quality documentation is part of their job and saves them time and effort in the long run. For any piece of code that you expect to live more than a few months, the extra cycles you put in documenting that code will not only help others; it will help you maintain that code, as well.

TL;DRs

- Documentation is hugely important over time and scale.
- Documentation changes should leverage the existing developer workflow.
- Keep documents focused on one purpose.
- Write for your audience, not yourself.

1 Ok, you will need to maintain it and revise it occasionally.

2 English is still the primary language for most programmers and most technical documentation for programmers relies on an understanding of English.

3 When we deprecated GooWiki, we found that around 90% of the documents had no views or updates in the previous few months.

Chapter 11. Testing Overview

Written by Adam Bender

Edited by Tom Manshreck

Testing has always been a part of programming. In fact, the first time you wrote a computer program you almost certainly threw some sample data at it to see whether it performed as you expected. For a long time, the state of the art in software testing resembled a very similar process, largely manual and error prone. However, since the early 2000s, the software industry's approach to testing has evolved dramatically to cope with the size and complexity of modern software systems. Central to that evolution has been the practice of developer-driven, automated testing.

Automated testing can prevent bugs from escaping into the wild and affecting your users. The later in the development cycle a bug is caught, the more expensive it is; exponentially so in many cases.[1] However, "catching bugs" is only part of the motivation. An equally important reason why you want to test your software is to support the ability to change. Whether you're adding new features, doing a refactoring focused on code health, or undertaking a larger redesign, automated testing can quickly catch mistakes and this makes it possible to change software with confidence.

Companies that can iterate faster can adapt more rapidly to changing technologies, market conditions, and customer tastes. If you have a robust testing practice, you needn't fear change—you can embrace it as an essential quality of developing software. The more and faster you want to change your systems, the more you need a fast way to test them.

The act of writing tests also improves the design of your systems. As the first clients of your code, a test can tell you much about your design choices. Is your system too tightly coupled to a database? Does the API support the required use cases? Does your system handle all of the edge cases? Writing automated tests forces you to confront these issues early on in the development cycle. Doing so generally leads to more modular software that enables greater flexibility later on.

Much ink has been spilled about the subject of testing software, and for good reason: for such an important practice, doing it well still seems to be a mysterious craft to many. At Google while we have come a long way, we still

face difficult problems getting our processes to scale reliably across the company. In this chapter, we'll share what we have learned to help further the conversation.

Why Do We Write Tests?

To better understand how to get the most out of testing let's start from the beginning. When we talk about automated testing, what are we really talking about?

The simplest test is defined by:

- A single behavior you are testing, usually a method or API that you are calling
- A specific input, some value that you pass to the API
- An observable output or behavior
- A controlled environment such as a single isolated process

When you execute a test like this, passing the input to the system and verifying the output, you will learn whether the system behaves as you expect. Taken in aggregate, hundreds or thousands of simple tests (usually called a *test suite*) can tell you how well your entire product conforms to its intended design and, more important, when it doesn't.

Creating and maintaining a healthy test suite takes real effort. As a codebase grows, so too will the test suite. It will begin to face challenges like instability and slowness. A failure to address these problems will cripple a test suite. Keep in mind that tests derive their value from the trust engineers place in them. If testing becomes a productivity sink, constantly inducing toil and uncertainty, engineers will lose trust and begin to find workarounds. A bad test suite can be worse than no test suite at all.

In addition to empowering companies to build great products quickly, testing is becoming critical to ensuring the safety of important products and services in our lives. Software is more involved in our lives than ever before and defects can cause more than a little annoyance: they can cost massive amounts of money, loss of property, or, worst of all, loss of life.[2]

At Google, we have determined that testing cannot be an afterthought. Focusing on quality and testing is part of how we do our jobs. We have learned, sometimes painfully, that failing to build quality into our products

and services inevitably leads to bad outcomes, as a result, we have built testing into the heart of our engineering culture.

The Story of Google Web Server

In Google's early days, engineer-driven testing was often assumed to be of little importance. Teams regularly relied on smart people to get the software right. A few systems ran large integration tests, but mostly it was the Wild West. One product in particular seemed to suffer the worst, it was called the Google Web Server, also known as "GWS."

GWS is the web server responsible for serving Google Search queries and is as important to Google Search as Air Traffic Control is to an airport. Back in 2005, as the project swelled in size and complexity, productivity had slowed dramatically. Releases were becoming buggier, and it was taking longer and longer to push them out. Team members had little confidence when making changes to the service, and often found out something was wrong only when features stopped working in production. (At one point more than 80% of production pushes contained user-affecting bugs that had to be rolled back).

To address these problems the Tech Lead (TL) of GWS decided to institute a policy of engineer-driven, automated testing. As part of this policy, all new code changes were required to include tests and those tests would be run continuously. Within a year of instituting this policy, the number of emergency pushes *dropped by half*. This drop occurred despite the fact that the project was seeing a record number of new changes every quarter. Even in the face of unprecedented growth and change, testing brought renewed productivity and confidence to one of the most critical projects at Google. Today GWS has tens of thousands of tests, and releases almost every day with relatively few customer-visible failures.

The changes in GWS marked a watershed for testing culture at Google as teams in other parts of the company saw the benefits of testing and moved to adopt similar tactics.

One of the key insights the GWS experience taught us was that you can't rely on programmer ability alone to avoid product defects. Even if each engineer writes only the occasional bug, after you have enough people working on the same project you will be swamped by the ever-growing list of defects. Imagine a hypothetical 100-person team whose engineers are so good that they each write only a single bug a month. Collectively this group of amazing engineers still produces five new bugs every workday. Worse yet, in a

complex system, fixing one bug can often cause another, as engineers adapt to known bugs and code around them.

The best teams find ways to turn the collective wisdom of its members into a benefit for the entire team. That is exactly what automated testing does. After an engineer on the team writes a test it is added to the pool of common resources available to others. Everyone else on the team can now run the test and will benefit when it detects an issue. Contrast this to an approach based on debugging, wherein each time a bug occurs, an engineer must pay the cost of digging into it with a debugger. The cost in engineering resources is night and day, and was the fundamental reason GWS was able to turn its fortunes around.

Testing at the Speed of Modern Development

Software systems are growing larger and evermore complex. A typical application or service at Google is made up of thousands or millions of lines of code. It uses hundreds of libraries or frameworks, and must be delivered via unreliable networks to an increasing number of platforms running with an uncountable number of configurations. To make matters worse, new versions are pushed to users frequently, sometimes multiple times each day. This is a far cry from the world of shrink-wrapped software that saw updates only once or twice a year.

The ability for humans to manually validate every behavior in a system has been unable to keep pace with the explosion of features and platforms in most software. Imagine what it would take to manually test all of the functionality of Google Search, like finding flights, movie times, relevant images, and of course web search results (see Figure 11-1). Even if you can determine how to solve that problem, you then need to multiply that workload by every language, country, and device Google Search must support, and don't forget to check for things like accessibility and security. Attempting to assess product quality by asking humans to manually interact with every feature just doesn't scale. When it comes to testing there is one clear answer: automation.

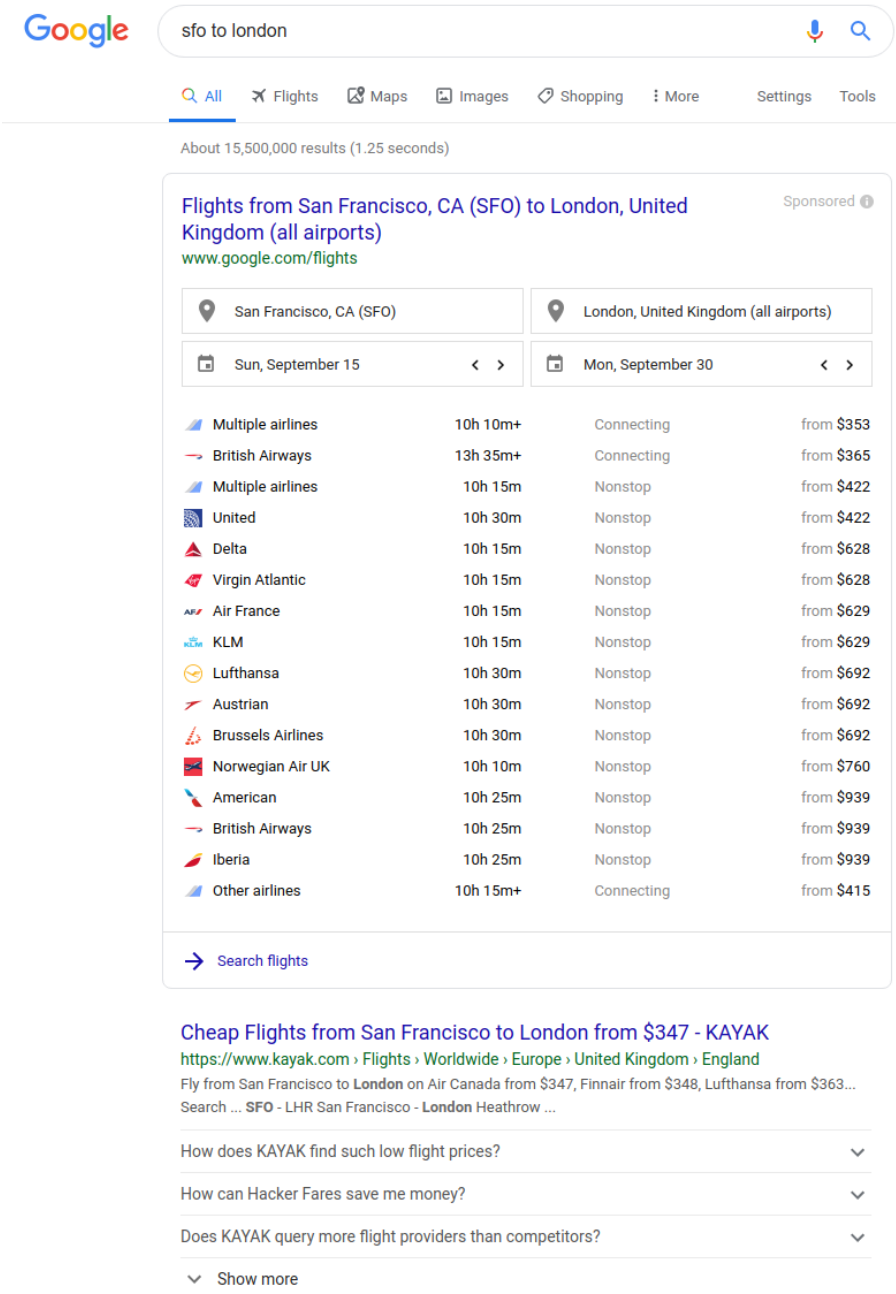

Figure 11-1. Screenshots of two complex Google search results

Write, Run, React

In its purest form, automating testing consists of three activities: writing tests, running tests, and reacting to test failures. An automated test is a small bit of code, usually a single function or method, that calls into an isolated part of a larger system that you want to test. The test code sets up an expected environment, calls into the system, usually with a known input, and verifies the result. Some of the tests are very small, exercising a single code path; others are much larger and can involve entire systems like a mobile operating system or web browser.

Figure 11-2 presents a deliberately simple test in Java using no frameworks or testing libraries. This is not how you would write an entire test suite, but at its core every automated test looks similar to this very simple example.

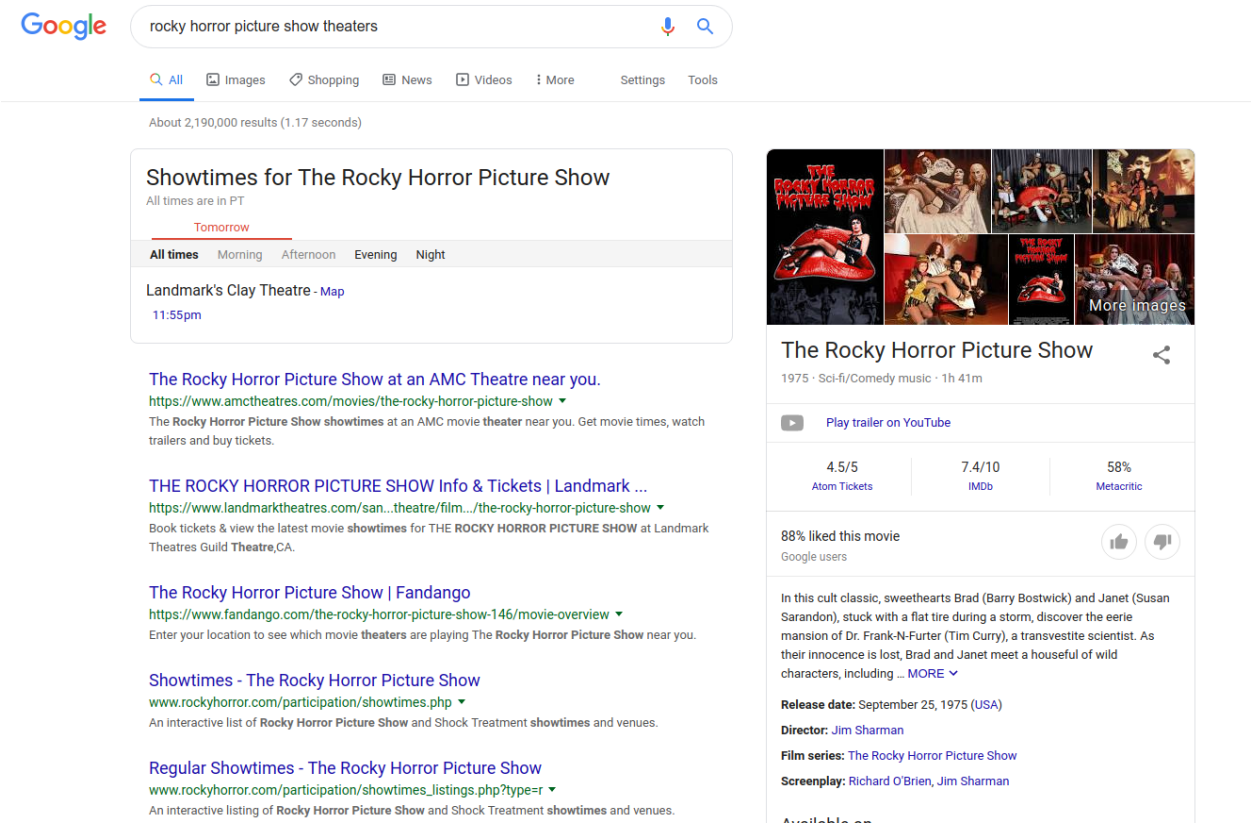

Figure 11-2. An example test

Unlike the QA processes of yore, in which rooms of dedicated software testers poured over new versions of a system exercising every possible behavior, the engineers who build systems today play an active and integral role in writing and running automated tests for their own code. Even in companies where QA is a prominent organization, developer-written tests are commonplace. At the speed and scale that today's systems are being developed, the only way to keep up is by sharing the development of tests around the entire engineering staff.

Of course, writing tests is different from writing *good tests*. It can be quite difficult to train tens of thousands of engineers to write good tests. We will discuss what we have learned about writing good tests in the chapters that follow.

Writing tests is only the first step in the process of automated testing. After you have written tests, you need to run them. Frequently. At its core, automated testing consists of repeating the same action over and over, only requiring human attention when something breaks. We will discuss this

Continuous Integration (CI) and testing in Chapter 23. By expressing tests as code instead of a manual series of steps, we can run them every time the code changes; easily thousands of times per day. Unlike human testers, machines never grow tired or bored.

Another benefit of having tests expressed as code is that it is easy to modularize them for execution in various environments. Testing the behavior of Gmail in Firefox requires no more effort than doing so in Chrome, provided you have configurations for both of these systems.[3] Running tests for a user interface (UI) in Japanese or German can be done using the same test code as for English.

Products and services under active development will inevitably experience test failures. What really makes a testing process effective is how it addresses test failures. Allowing failing tests to pile up quickly defeats any value they were providing so it is imperative not to let that happen. Teams that prioritize fixing a broken test within minutes of a failure are able to keep confidence high, failure isolation fast, and therefore derive more value out of their tests.

In summary, a healthy automated testing culture encourages everyone to share the work of writing tests. Such a culture also ensures that tests are run regularly. Last, and perhaps most important, it places an emphasis on fixing broken tests quickly so as to maintain high confidence in the process.

Benefits of Testing Code

To developers coming from organizations that don't have a strong testing culture, the idea of writing tests as a means of improving productivity and velocity might seem antithetical. After all, the act of writing tests can take just as long (if not longer!) than implementing a feature would take in the first place. On the contrary, at Google we've found that investing in software tests provides several key benefits to developer productivity:

Less debugging

> As you would expect, tested code has fewer defects when it is submitted. Critically, it also has fewer defects throughout its existence; most of them will be caught before the code is submitted. A piece of code at Google is expected to be modified dozens of times in its lifetime. It will be changed by other teams and even automated code maintenance systems. A test written once continues to pay dividends and prevent costly defects and annoying debugging sessions

through the lifetime of the project. Changes to a project, or the dependencies of a project, that break a test can be quickly detected by test infrastructure and rolled back before the problem is ever released to production.

Increased confidence in changes

All software changes. Teams with good tests can review and accept changes to their project with confidence because all important behaviors of their project are continuously verified. Such projects encourage refactoring. Changes that refactor code while preserving existing behavior should (ideally) require no changes to existing tests.

Improved documentation

Software documentation is notoriously unreliable. From outdated requirements, to missing edge cases, it is common for documentation to have a tenuous relationship to the code. Clear, focused tests that exercise one behavior at a time function as executable documentation. If you want to know what the code does in a particular case, look at the test for that case. Even better, when requirements change and new code breaks an existing test, we get a clear signal that the "documentation" is now out of date. Note that tests work best as documentation only if care is taken to keep them clear and concise.

Simpler reviews

All code at Google is reviewed by at least one other engineer before it can be submitted (see Chapter 9 for more details). A code reviewer spends less effort verifying code works as expected if the code review includes thorough tests that demonstrate code correctness, edge cases, and error conditions. Instead of the tedious effort needed to mentally walk each case through the code, the reviewer can verify that each case has a passing test.

Thoughtful design

Writing tests for new code is a practical means of exercising the API design of the code itself. If new code is difficult to test, it is often because the code being tested has too many responsibilities or difficult-to-manage dependencies. Well-designed code should be modular, avoiding tight coupling and focusing on specific

responsibilities. Fixing design issues early often means less rework later.

Fast, high-quality releases

With a healthy automated test suite, teams can release new versions of their application with confidence. Many projects at Google release a new version to production every day—even large projects with hundreds of engineers and thousands of code changes being submitted every day. This would not be possible without automated testing.

Designing a Test Suite

Today, Google operates at a massive scale, but we haven't always been so large and the foundations of our approach were laid long ago. Over the years, as our codebase has grown, we have learned a lot about how to approach the design and execution of a test suite, often by making mistakes and cleaning up afterward.

One of the lessons we learned fairly early on is that engineers favored writing larger, system-scale tests, but that these tests were slower, less reliable, and more difficult to debug than smaller tests. Engineers, fed up with debugging the system-scale tests, asked themselves, "Why can't we just test one server at a time?" or, "Why do we need to test a whole server at once? We could test smaller modules individually." Eventually, the desire to reduce pain led teams to develop smaller and smaller tests, which turned out to be faster, more stable, and generally less painful.

This led to a lot of discussion around the company about the exact meaning of "small." Does small mean unit test? What about integration tests, what size are those? We have come to the conclusion that there are two distinct dimensions for every test case: size and scope. Size refers to the resources that are required to run a test case: things like memory, processes, and time. Scope refers to the specific code paths we are verifying. Note that executing a line of code is different from verifying that it worked as expected. Size and scope are interrelated but distinct concepts.

Test Size

At Google, we classify every one of our tests into a size and encourage engineers to always write the smallest possible test for a given piece of

functionality. A test's size is determined not by its number of lines of code, but by how it runs, what it is allowed to do, and how many resources it consumes. In fact, in some cases our definitions of small, medium, and large are actually encoded as constraints the testing infrastructure can enforce on a test. We go into the details in a moment, but in brief, *small tests* run in a single process, *medium tests* run on a single machine, and *large tests* run wherever they want, as demonstrated in Figure 11-3.[4]

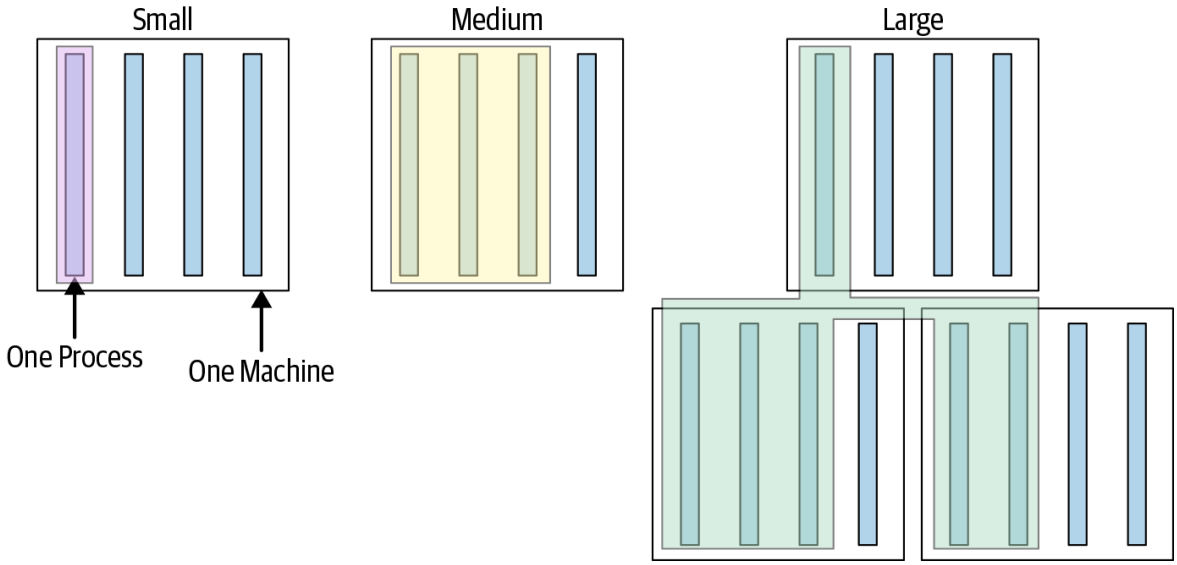

Figure 11-3. Test sizes

We make this distinction, as opposed to the more traditional "unit" or "integration," because the most important qualities we want from our test suite are speed and determinism, regardless of the scope of the test. Small tests, regardless of the scope, are almost always faster and more deterministic than tests that involve more infrastructure or consume more resources. Placing restrictions on small tests makes speed and determinism much easier to achieve. As test sizes grow, many of the restrictions are relaxed. Medium tests have more flexibility but also more risk of nondeterminism. Larger tests are saved for only the most complex and difficult testing scenarios. Let's take a closer look at the exact constraints imposed on each type of test.

SMALL TESTS

Small tests are the most constrained of the three test sizes. The primary constraint is that small tests must run in a single process. In many languages we restrict this even further to say that they must run on a single thread. This means that the code performing the test must run in the same process as the code being tested. You can't run a server and have a separate test process connect to it. It also means that you can't run a third-party program such as a database as part of your test.

The other important constraints on small tests are that they aren't allowed to sleep, perform I/O operations,[5] or make any other blocking calls. This means that small tests aren't allowed to access the network or disk. Testing code that relies on these sorts of operations requires the use of test doubles (see Chapter 13) to replace the heavyweight dependency with a lightweight, in-process dependency.

The purpose of these restrictions is to ensure that small tests don't have access to the main sources of test slowness or nondeterminism. A test that runs on a single process and never makes blocking calls can effectively run as fast as the CPU can handle. It's difficult (but certainly not impossible) to accidentally make such a test slow or nondeterministic. The constraints on small tests provide a sandbox that prevents engineers from shooting themselves in the foot.

These restrictions might seem excessive at first, but consider a modest suite of a couple hundred small test cases running throughout the day. If even a few of them fail nondeterministically (often called flaky tests) tracking down the cause becomes a serious drain on productivity. At Google's scale, such a problem could grind our testing infrastructure to a halt.

At Google, we encourage engineers to try to write small tests whenever possible, regardless of the scope of the test because it keeps the entire test suite running fast and reliably. For more discussion on small versus unit tests see Chapter 12.

MEDIUM TESTS

The constraints placed on small tests can be too restrictive for many interesting kinds of tests. The next rung up the ladder of test sizes is the medium test. Medium tests can span multiple processes, use threads, and can make blocking calls, including network calls to `localhost`. The only remaining restriction is that medium tests aren't allowed to make network calls to any system other than `localhost`. In other words, the test must be contained within a single machine.

The ability to run multiple processes opens up a lot of possibilities. For example, you could run a database instance to validate that the code you're testing integrates correctly in a more realistic setting. Or you could test a combination of web UI and server code. Tests of web applications often involve tools like WebDriver that start a real browser and control it remotely via the test process.

Unfortunately, with increased flexibility comes increased potential for tests to become slow and nondeterministic. Tests that span processes or are allowed to make blocking calls are dependent on the operating system and third-party processes to be fast and deterministic, which isn't something we can guarantee in general. Medium tests still provide a bit of protection by preventing access to remote machines via the network, which is far and away the biggest source of slowness and nondeterminism in most systems. Still, when writing medium tests, the "safety" is off, and engineers need to be much more careful.

LARGE TESTS

Finally, we have large tests. Large tests remove the `localhost` restriction imposed on medium tests, allowing the test and the system being tested to span across multiple machines. For example, the test might run against a system in a remote cluster.

As before, increased flexibility comes with increased risk. Having to deal with a system that spans multiple machines and the network connecting them increases the chance of slowness and nondeterminism significantly compared to running on a single machine. We mostly reserve large tests for full-system end-to-end tests that are more about validating configuration than pieces of code, and for tests of legacy components for which it is impossible to use test doubles. We'll talk more about use cases for large tests in Chapter 14. Teams at Google will frequently isolate their large tests from their small or medium tests, running them only during the build and release process so as not to impact developer workflow.

FLAKY TESTS ARE EXPENSIVE

If you have a few thousand tests, each with a very tiny bit of nondeterminism, running all day, occasionally one will probably fail (flake). As the number of tests grows, statistically so will the number of flakes. If each test has even a 0.1% of failing when it should not, and you run 10,000 tests per day, you will be investigating 10 flakes per day. Each investigation takes time away from something more productive that your team could be doing.

In some cases, you can limit the impact of flaky tests by automatically rerunning them when they fail. This is effectively trading CPU cycles for engineering time. At low levels of flakiness, this trade-off makes sense. Just keep in mind that rerunning a test is only delaying the need to address the root cause of flakiness.

If test flakiness continues to grow, you will experience something much worse than lost productivity: a loss of confidence in the tests. It doesn't take needing to

investigate many flakes before a team loses trust in the test suite. After that happens, engineers will stop reacting to test failures, eliminating any value the test suite provided. Our experience suggests that as you approach 1% flakiness the tests begin to lose value. At Google as our flaky rate hovers around 0.15%, which implies thousands of flakes every day. We fight hard to keep flakes in check including actively investing engineering hours to fix them.

In most cases, flakes appear because of nondeterministic behavior in the tests themselves. Software provides many sources of nondeterminism: clock time, thread scheduling, network latency, and more. Learning how to isolate and stabilize the effects of randomness is not easy. Sometimes, effects are tied to low-level concerns like hardware interrupts or browser rendering engines. A good automated test infrastructure should help engineers identify and mitigate any nondeterministic behavior.

PROPERTIES COMMON TO ALL TEST SIZES

All tests should strive to be hermetic: a test should contain all of the information necessary to set up, execute, and tear down its environment. Tests should assume as little as possible about the outside environment, such as the order in which the tests are run. For example, they should not rely on a shared database. This constraint becomes more challenging with larger tests, but effort should still be made to ensure isolation.

A test should contain *only* the information required to exercise the behavior in question. Keeping tests clear and simple aids reviewers in verifying that the code does what it says it does. Clear code also aids in diagnosing failure when they fail. We like to say that "a test should be obvious upon inspection." Because there are no tests for the tests themselves, they require manual review as an important check on correctness. As a corollary to this, we also strongly discourage the use of control flow statements like conditionals and loops in a test. More complex test flows risk containing bugs themselves, and make it more difficult to determine the cause of a test failure.

Remember that tests are often revisited only when something breaks. When you are called to fix a broken test that you have never seen before, you will be thankful someone took the time to make it easy to understand. Code is read far more than it is written, so make sure you write the test you'd like to read!

Test sizes in practice

Having precise definitions of test sizes has allowed us to create tools to enforce them. Enforcement enables us to scale our test suites and still make certain guarantees about speed, resource utilization, and stability. The extent to which these definitions are enforced at Google varies by language. For example, we run all Java tests using a custom security manager that will cause all tests tagged as small to fail if they attempt to do something prohibited, such as establish a network connection.

Test Scope

Though we at Google put a lot of emphasis on test size, another important property to consider is test scope. Test scope refers to how much code is being validated by a given test. Narrow-scoped tests (commonly called "unit tests") are designed to validate the logic in a small, focused part of the codebase, like an individual class or method. Medium-scoped tests (commonly called *integration tests*) are designed to verify interactions between a small number of components; for example, between a server and its database. Large-scoped tests (commonly referred to by names like *functional tests*, *end-to-end* tests, or *system tests*) are designed to validate the interaction of several distinct parts of the system, or emergent behaviors that aren't expressed in a single class or method.

It's important to note that when we talk about unit tests as being narrowly scoped, we're referring to the code that is being *validated*, not the code that is being *executed*. It's quite common for a class to have many dependencies or other classes it refers to, and these dependencies will naturally be invoked while testing the target class. Though some other testing strategies make heavy use of test doubles (fakes or mocks) to avoid executing code outside of the system under test, at Google we prefer to keep the real dependencies in place when it is feasible to do so. Chapter 13 discusses this issue in more detail.

Narrow-scoped tests tend to be small, and broad-scoped tests tend to be medium or large, but this isn't always the case. For example, it's possible to write a broad-scoped test of a server endpoint that covers all of its normal parsing, request validation, and business logic, which is nevertheless small because it uses doubles to stand in for all out-of-process dependencies like a database or filesystem. Similarly, it's possible to write a narrow-scoped test of a single method that must be medium sized. For example, modern web frameworks often bundle HTML and JavaScript together, and testing a UI

component like a Date Picker often requires running an entire browser, even to validate a single code path.

Just as we encourage tests of smaller size, at Google we also encourage engineers to write tests of narrower scope. As a very rough guideline, we tend to aim to have a mix of around 80% of our tests being narrow-scoped unit tests that validate the majority of our business logic; 15% medium-scoped integration tests that validate the interactions between two or more components; and 5% end-to-end tests that validate the entire system. Figure 11-3 depicts how we can visualize this as a pyramid.

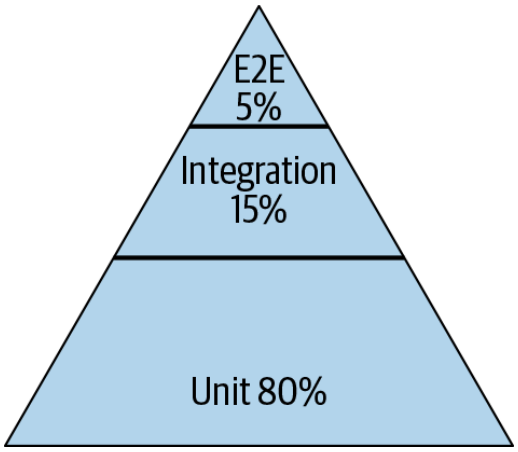

Figure 11-4. Google's version of Mike Cohn's test pyramid;[6] percentages are by test case count, and every team's mix will be a little different

Unit tests form an excellent base because they are fast, stable, and dramatically narrow the scope and reduce the cognitive load required to identify all the possible behaviors a class or function has. Additionally, they make failure diagnosis quick and painless. Two antipatterns to be aware of are the "ice cream cone" and the "hourglass," as illustrated in Figure 11-5.

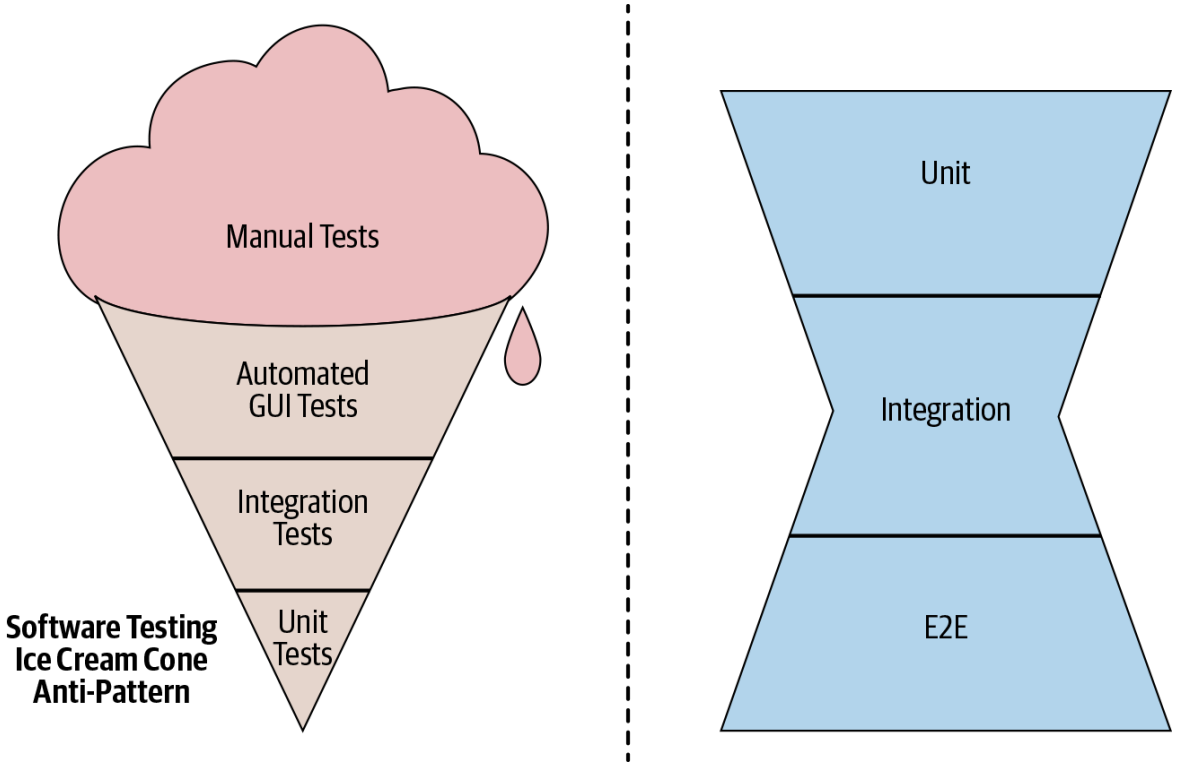

Figure 11-5. Test suite antipatterns

With the ice cream cone, engineers write many end-to-end tests but few integration or unit tests. Such suites tend to be slow, unreliable, and difficult to work with. This pattern often appears in projects that start as prototypes and are quickly rushed to production, never stopping to address testing debt.

The hourglass involves many end-to-end tests and many unit tests but few integration tests. It isn't quite as bad as the ice cream cone, but it still results in many end-to-end test failures that could have been caught quicker and more easily with a suite of medium-scope tests. The hourglass pattern occurs when tight coupling makes it difficult to instantiate individual dependencies in isolation.

Our recommended mix of tests is determined by our two primary goals: engineering productivity and product confidence. Favoring unit tests gives us high confidence quickly, and early in the development process. Larger tests act as sanity checks as the product develops; they should not be viewed as a primary method for catching bugs.

When considering your own mix, you might want a different balance. If you emphasize integration testing, you might discover that your test suites take longer to run but catch more issues between components. When you emphasize unit tests, your test suites can complete very quickly, and you will catch many common logic bugs. But, unit tests cannot verify the interactions between components, like a contract between two systems developed by

different teams. A good test suite contains a blend of different test sizes and scopes that are appropriate to the local architectural and organizational realities.

The Beyoncé Rule

We are often asked, when coaching new hires, which behaviors or properties actually need to be tested? The straightforward answer is: test everything that you don't want to break. In other words, if you want to be confident that a system exhibits a particular behavior, the only way to be sure it will is to write an automated test for it. This includes all of the usual suspects like testing performance, behavioral correctness, accessibility, and security. It also includes less obvious properties like testing how a system handles failure.

We have a name for this general philosophy: we call it the Beyoncé Rule. Succinctly, it can be stated as follows: "If you liked it, then you shoulda put a test on it". The Beyoncé Rule is often invoked by infrastructure teams that are responsible for making changes across the entire codebase. If unrelated infrastructure changes pass all of your tests but still break your team's product, you are on the hook for fixing it and adding the additional tests.

TESTING FOR FAILURE

One of the most important situations a system must account for is failure. Failure is inevitable, but waiting for an actual catastrophe to find out how well a system responds to a catastrophe is a recipe for pain. Instead of waiting for a failure, write automated tests that simulate common kinds of failures. This includes simulating exceptions or errors in unit tests and injecting Remote Procedure Call (RPC) errors or latency in integration and end-to-end tests. It can also include much larger disruptions that affect the real production network using techniques like Chaos Engineering. A predictable and controlled response to adverse conditions is a hallmark of a reliable system.

A Note on Code Coverage

Code coverage is a measure of which lines of feature code are exercised by which tests. If you have 100 lines of code and your tests execute 90 of them, you have 90% code coverage.[7] Code coverage is often held up as the gold standard metric for understanding test quality, and that is somewhat unfortunate. It is possible to exercise a lot of lines of code with a few tests, never checking that each line is doing anything useful. That's because code coverage only measures that a line was invoked, not what happened as a

result (we recommend only measuring coverage from small tests to avoid coverage inflation that occurs when executing larger tests).

An even more insidious problem with code coverage is that, like other metrics, it quickly becomes a goal unto itself. It is common for teams to establish a bar for expected code coverage; for instance, 80%. At first, that sounds eminently reasonable; surely you want to have at least that much coverage. In practice, what happens is that instead of treating 80% like a floor, engineers treat it like a ceiling. Soon, changes begin landing with no more than 80% coverage. After all, why do more work than the metric requires?

A better way to approach the quality of your test suite is to think about the behaviors that are tested. Do you have confidence that everything your customers expect to work, will work? Do you feel confident you can catch breaking changes in your dependencies? Are your tests stable and reliable? Questions like these are a more holistic way to think about a test suite. Every product and team is going to be different; some will have difficult-to-test interactions with hardware, some involve massive datasets. Trying to answer the question "do we have enough tests?" with a single number ignores a lot of context and is unlikely to be useful. Code coverage can provide some insight into untested code, but it is not a substitute for thinking critically about how well your system is tested.

Testing at Google Scale

Much of the guidance to this point can be applied to codebases of almost any size. However, we should spend some time on what we have learned testing at our very large scale. To understand how testing works at Google, you need an understanding of our development environment. The most important fact about which, is that most of Google's code is kept in a single monolithic repository (monorepo). Almost every line of code for every product and service we operate is all stored in one place. We have more than two billion lines of code in the repository today.

Google's codebase experiences close to 25 million lines of change every week. Roughly half of them are made by the tens of thousands of engineers working in our monorepo, and the other half by our automated systems, in the form of configuration updates, or large-scale changes (Chapter 22). Many of those changes are initiated from outside the immediate project. We don't place many limitations on the ability of engineers to reuse code.

The openness of our codebase encourages a level of co-ownership that lets everyone take responsibility for the codebase. One benefit of such openness is the ability to directly fix bugs in a product or service you use (subject to approval, of course) instead of complaining about it. This also implies that many people will make changes in a part of the codebase owned by someone else.

Another thing that makes Google a little different is that almost no teams use repository branching. All changes are committed to the repository head and are immediately visible for everyone to see. Furthermore, all software builds are performed using the last committed change that our testing infrastructure has validated. When a product or service is built, almost every dependency required to run it is also built from source, also from the head of the repository. Google manages testing at this scale by use of a CI system. One of the key components of our CI system is our Test Automated Platform (TAP).

NOTE

For more information on TAP and our CI philosophy, see Chapter 23.

Whether you are considering our size, our monorepo, or the number of products we offer, Google's engineering environment is complex. Every week it experiences millions of changing lines, billions of test cases being run, tens of thousands of binaries being built, and hundreds of products being updated—talk about complicated!

The Pitfalls of a Large Test Suite

As a codebase grows you will inevitably need to make changes to existing code. When poorly written, automated tests can make it more difficult to make those changes. Brittle tests—those which over-specify expected outcomes or rely on extensive and complicated boilerplate—can actually resist change. These poorly written tests can fail even when unrelated changes are made.

If you have ever made a five-line change to a feature only to find dozens of unrelated, broken tests, you have felt the friction of brittle tests. Over time, this friction can make a team reticent to perform necessary refactoring to keep a codebase healthy. The subsequent chapters will cover strategies that you can use to improve the robustness and quality of your tests.

Some of the worst offenders of brittle tests come from the misuse of mock objects. Google's codebase has suffered so badly from an abuse of mocking frameworks that it has led some engineers to declare "no more mocks!" Although that is a strong statement, understanding the limitations of mock objects can help you avoid misusing them.

NOTE

For more information on working effectively with mock objects, see Chapter 13.

In addition to the friction caused by brittle tests, a larger suite of tests will be slower to run. The slower a test suite, the less frequently it will be run, and the less benefit it provides. We use a number of techniques to speed up our test suite, including parallelizing execution and using faster hardware. However, these kinds of tricks are eventually swamped by a large number of individually slow test cases.

Tests can become slow for many reasons like booting significant portions of a system, firing up an emulator before execution, processing large datasets, or waiting for disparate systems to synchronize. Tests often start fast enough but slow down as the system grow. For example, maybe you have an integration test exercising a single dependency that takes five seconds to respond, but over the years you grow to depend on a dozen services and now the same tests take five minutes.

Tests can also become slow due to unnecessary speed limits introduced by functions like `sleep()` and `setTimeout()`. Calls to these functions are often used as naive heuristics before checking the result of non-deterministic behavior. Sleeping for half a second here or there doesn't seem too dangerous at first, however if a "wait-and-check" is embedded in a widely used utility, pretty soon you have added minutes of idle time to every run of your test suite. A better solution is to actively poll for a state transition with a frequency closer to microseconds. You can combine this with a timeout value in case a test fails to reach a stable state.

Failing to keep a test suite deterministic and fast ensures it will become roadblock to productivity. At Google, engineers who encounter these tests have found ways to work around slowdowns, with some going as far as to skip the tests entirely when submitting changes. Obviously, this is a risky practice and should be discouraged, but if a test suite is causing more harm than good, eventually engineers will find a way to get their job done, tests or no tests.

The secret to living with a large test suite is to treat it with respect. Incentivize engineers to care about their tests; reward them as much for having rock-solid tests as you would for having a great feature launch. Set appropriate performance goals and refactor slow or marginal tests. Basically, treat your tests like production code. When simple changes begin taking nontrivial time, spend effort making your tests less brittle.

In addition to developing the proper culture, invest in your testing infrastructure by developing linters, documentation, or other assistance that makes it more difficult to write bad tests. Reduce the number of frameworks and tools you need to support to increase the efficiency of the time you invest to improve things.[8] If you don't invest in making it easy to manage your tests, eventually engineers will decide it isn't worth having them at all.

History of Testing at Google

Now that we've discussed how Google approaches testing, it might be enlightening to learn how we got here. As mentioned previously, Google's engineers didn't always embrace the value of automated testing. In fact, until 2005, testing was closer to a curiosity than a disciplined practice. Most of the testing was done manually, if it was done at all. However, from 2005 to 2006 a testing revolution occurred and changed the way we approach software engineering. Its effects continue to reverberate within the company to this day.

The experience of the GWS project, which we discussed at the opening of this chapter, acted as a catalyst. It made it clear how powerful automated testing could be. Following the improvements to GWS in 2005, the practices began spreading across the entire company. The tooling was primitive. However, the volunteers, who came to be known as the Testing Grouplet, didn't let that slow them down.

Three key initiatives helped usher automated testing into the company's consciousness: Orientation Classes, the Test Certified program, and Testing on the Toilet. Each one had influence in a completely different way, and together they reshaped Google's engineering culture.

Orientation Classes

Even though much of the early engineering staff at Google eschewed testing, the pioneers of automated testing at Google knew that at the rate the company was growing, new engineers would quickly outnumber existing

team members. If they could reach all the new hires in the company, it could be an extremely effective avenue for introducing cultural change. Fortunately, there was, and still is, a single choke point that all new engineering hires pass through: orientation.

Most of Google's early orientation program concerned things like medical benefits and how Google Search worked, but starting in 2005 it also began including an hour-long discussion of the value of automated testing.[9] The class covered the various benefits of testing such as increased productivity, better documentation, and support for refactoring. It also covered how to write a good test. For many Nooglers (new Googlers) at the time, such a class was their first exposure to this material. Most important, all of these ideas were presented as though they were standard practice at the company. The new hires had no idea that they were being used as trojan horses to sneak this idea into their unsuspecting teams.

As Nooglers joined their teams following orientation, they began writing tests and questioning those on the team who didn't. Within only a year or two, the population of engineers who had been taught testing outnumbered the pretesting culture engineers. As a result, many new projects started off on the right foot.

Testing has now become more widely practiced in the industry so most new hires arrive with the expectations of automated testing firmly in place. Nonetheless, orientation classes continue to set expectations about testing and connect what Nooglers know about testing outside of Google to the challenges of doing so in our very large and very complex codebase.

Test Certified

Initially, the larger and more complex parts of our codebase appeared resistant to good testing practices. Some projects had such poor code quality that they were almost impossible to test. To give projects a clear path forward, the Testing Grouplet devised a certification program that they called Test Certified. Test Certified aimed to give teams a way to understand the maturity of their testing processes, and more critically, cookbook instructions on how to improve it.

The program was organized into five levels, and each level required some concrete actions to improve the test hygiene on the team. The levels were designed in such a way that each step up could be accomplished within a quarter, which made it a convenient fit for Google's internal planning cadence.

Test Certified Level 1 covered the basics: set up a continuous build; start tracking code coverage; classify all your tests as small, medium, or large; identify (but don't necessarily fix) flaky tests; and create a set of fast (not necessarily comprehensive) tests that can be run quickly. Each subsequent level added more challenges like "no releases with broken tests" or "remove all nondeterministic tests." By Level 5, all tests were automated, fast tests were running before every commit, all nondeterminism had been removed, and every behavior was covered. An internal dashboard applied social pressure by showing the level of every team. It wasn't long before teams were competing with one another to climb the ladder.

By the time the Test Certified program was replaced by an automated approach in 2015 (more on pH later) it had helped more than 1,500 projects improve their testing culture.

Testing on the Toilet

Of all the methods the Testing Grouplet used to try to improve testing at Google, perhaps none was more off-beat than Testing on the Toilet (TotT). The goal of TotT was fairly simple: actively raise awareness about testing across the entire company. The question is, what's the best way to do that in a company with employees scattered around the world?

The Testing Grouplet considered the idea of a regular email newsletter, but given the heavy volume of email everyone deals with at Google, it was likely to become lost in the noise. After a little bit of brainstorming, someone proposed the idea of posting flyers in the restroom stalls as a joke. We quickly recognized the genius in it: the bathroom is one place that everyone must visit at least once each day, no matter what. Joke or not, the idea was cheap enough to implement that it had to be tried.

In April 2006, a short writeup covering how to improve testing in Python appeared in restroom stalls across Google. This first episode was posted by a small band of volunteers. To say the reaction was polarized is an understatement; some saw it as an invasion of personal space, and they objected strongly. Mailing lists lit up with complaints, but the TotT creators were content: the people complaining were still talking about testing.

Ultimately, the uproar subsided and TotT quickly became a staple of Google culture. To date, engineers from across the company have produced several hundred episodes, covering almost every aspect of testing imaginable (in addition to a variety of other technical topics). New episodes are eagerly anticipated and some engineers even volunteer to post the episodes around

their own buildings. We intentionally limit each episode to exactly one page, challenging authors to focus on the most important and actionable advice. A good episode contains something an engineer can take back to the desk immediately and try.

Ironically for a publication that appears in one of the more private locations, TotT has had an outsized public impact. Most external visitors see an episode at some point in their visit, and such encounters often lead to funny conversations about how Googlers always seem to be thinking about code. Additionally, TotT episodes make great blog posts, something the original TotT authors recognized early on. They began publishing lightly edited versions publicly, helping to share our experience with the industry at large.

Despite starting as a joke, TotT has had the longest run, and the most profound impact of any of the testing initiatives started by the Testing Grouplet.

Testing Culture Today

Testing culture at Google today has come a long way from 2005. Nooglers still attend orientation classes on testing, and TotT continues to be distributed almost weekly. However, the expectations of testing have more deeply embedded themselves in the daily developer workflow.

Every code change at Google is required to go through code review. And every change is expected to include both the feature code and tests. Reviewers are expected to review the quality and correctness of both. In fact, it is perfectly reasonable to block a change if it is missing tests.

As a replacement for Test Certified, one of our engineering productivity teams recently launched a tool called Project Health (pH). The pH tool continuously gathers dozens of metrics on the health of a project, including test coverage and test latency, and makes them available internally. pH is measured on a scale of one (worst) to five (best). A pH-1 project is seen as a problem for the team to address. Almost every team that runs a continuous build automatically get a pH score.

Over time, testing has become an integral part of Google's engineering culture. We have myriad ways to reinforce its value to engineers across the company. Through a combination of training, gentle nudges, mentorship, and, yes, even a little friendly competition, we have created the clear expectation that testing is everyone's job.

Why didn't we start by mandating the writing of tests?

The Testing Grouplet had considered asking for a testing mandate from senior leadership but quickly decided against it. Any mandate on how to develop code would be seriously counter to Google culture and likely slow the progress, independent of the idea being mandated. The belief was that successful ideas would spread, so the focus became demonstrating success.

If engineers were deciding to write tests on their own, it meant that they had fully accepted the idea and were likely to keep doing the right thing—even if no one was compelling them to.

The Limits of Automated Testing

Automated testing is not suitable for all testing tasks. For example, testing the quality of search results often involves human judgement. We conduct targeted, internal, studies using Search Quality Raters who execute real queries and record their impressions. Similarly, it is difficult to capture the nuances of audio and video quality in an automated test, so we often use human judgement to evaluate the performance of telephony or video-calling systems.

In addition to qualitative judgements, there are certain creative assessments for which humans excel. For example, searching for complex security vulnerabilities is something that humans do better than automated systems. After a human has discovered and understood a flaw, it can be added to an automated security testing system like Google's Cloud Security Scanner where it can be run continuously and at scale.

A more generalized term for this technique is Exploratory Testing. Exploratory Testing is a fundamentally creative endeavor in which someone treats the application under test as a puzzle to be broken, maybe by executing an unexpected set of steps or by inserting unexpected data. When conducting an exploratory test, the specific problems to be found are unknown at the start. They are gradually uncovered by probing commonly overlooked code paths or unusual responses from the application. As with the detection of security vulnerabilities, as soon as an exploratory test discovers an issue, an automated test should be added to prevent future regressions.

Using automated testing to cover well-understood behaviors enables the expensive and qualitative efforts of human testers to focus on the parts of

your products for which they can provide the most value—and avoid boring them to tears in the process.

Conclusion

The adoption of developer-driven automated testing has been one of the most transformational software engineering practices at Google. It has enabled us to build larger systems with larger teams, faster than we ever thought possible. It has helped us keep up with the increasing pace of technological change. Over the past 15 years, we have successfully transformed our engineering culture to elevate testing into a cultural norm. Despite the company growing by a factor of almost 100 times since the journey began, our commitment to quality and testing is stronger today than it has ever been.

This chapter has been written to help orient you to how Google thinks about testing. In the next few chapters we are going to dive even deeper into some key topics that have helped shape our understanding of what it means to write good, stable, and reliable tests. We will discuss the what, why, and how of unit tests, the most common kind of test at Google. We will wade into the debate on how to effectively use test doubles in tests through techniques such as faking, stubbing, and interaction testing. Finally, we will discuss the challenges with testing larger and more complex systems, like many of those we have at Google.

At the conclusion of these three chapters you should have a much deeper and clearer picture of the testing strategies we use and, more important, why we use them.

TL;DRs

- Testing is as much about catching bugs as alerting you to changes
- For tests to scale, they must be automated
- A balanced test suite is necessary for maintaining healthy test coverage
- "If you liked it, you should have put a test on it"
- Changing the testing culture in organizations take time

1 "Defect Prevention: Reducing Costs and Enhancing Quality"

2 "Failure at Dhahran"

3 Getting the behavior right across different browsers and languages is a different story! But, ideally, the end-user experience should be the same for everyone.

4 Technically we have four sizes of test at Google: small, medium, large, and *enormous*. The internal difference between medium and large is actually subtle and historical; so, in this book, most descriptions of large actually apply to our notion of enormous.

5 There is a little wiggle room in this policy. Tests are allowed to access a filesystem if they use a hermetic, in-memory implementation.

6 Succeeding with Agile, by Mike Cohn, 2009.

7 Keep in mind that there are different kinds of coverage (line, path, branch, etc.) and each says something different about which code has been tested. In this simple example, line coverage is being used.

8 Each supported language at Google has one standard test framework and one standard mocking/stubbing library. One set of infrastructure runs most tests in all languages across the entire codebase.

9 This class was so successful that an updated version is still taught today. In fact, it is one of the longest-running orientation classes in the company's history.

Chapter 12. Unit Testing

Written by Erik Kueffler

Edited by Tom Manshreck

The previous chapter introduced two of the main axes along which Google classifies tests: *size* and *scope*. To recap, size refers to the resources consumed by a test and what it is allowed to do, and scope refers to how much code a test is intended to validate. Though Google has clear definitions for test size, scope tends to be a little fuzzier. We use the term *unit test* to refer to tests of relatively narrow scope, such as of a single class or method. Unit tests are usually small in size, but this isn't always the case.

After preventing bugs, the most important purpose of a test is to improve engineers' productivity. Compared to broader-scoped tests, unit tests have many properties that make them an excellent way to optimize productivity:

- They tend to be small according to Google's definitions of test size. Small tests are fast and deterministic, allowing developers to run them frequently as part of their workflow and get immediate feedback.
- They tend to be easy to write at the same time as the code they're testing, allowing engineers to focus their tests on the code they're working on without having to set up and understand a larger system.
- They promote high levels of test coverage because they are quick and easy to write. High test coverage allows engineers to make changes with confidence that they aren't breaking anything.
- They tend to make it easy to understand what's wrong when they fail because each test is conceptually simple and focused on a particular part of the system.
- They can serve as documentation and examples, showing engineers how to use the part of the system being tested and how that system is intended to work.

Due to their many advantages, most tests written at Google are unit tests, and as a rule of thumb, we encourage engineers to aim for a mix of about 80% unit tests and 20% broader-scoped tests. This advice, coupled with the ease of writing unit tests and the speed with which they run, means that engineers run

a *lot* of unit tests—it's not at all unusual for an engineer to execute thousands of unit tests (directly or indirectly) during the average workday.

Because they make up such a big part of engineers' lives, Google puts a lot of focus on *test maintainability*. Maintainable tests are ones that "just work": after writing them, engineers don't need to think about them again until they fail, and those failures indicate real bugs with clear causes. The bulk of this chapter focuses on exploring the idea of maintainability and techniques for achieving it.

The Importance of Maintainability

Imagine this scenario: Mary wants to add a simple new feature to the product and is able to implement it quickly, perhaps requiring only a couple dozen lines of code. But when she goes to check in her change, she gets a screen full of errors back from the automated testing system. She spends the rest of the day going through those failures one by one. In each case, the change introduced no actual bug, but broke some of the assumptions that the test made about the internal structure of the code, requiring those tests to be updated. Often, she has difficulty figuring out what the tests were trying to do in the first place, and the hacks she adds to fix them make those tests even more difficult to understand in the future. Ultimately, what should have been a quick job ends up taking hours or even days of busywork, killing Mary's productivity and sapping her morale.

Here, testing had the opposite of its intended effect by draining productivity rather than improving it while not meaningfully increasing the quality of the code under test. This scenario is far too common, and Google engineers struggle with it every day. There's no magic bullet, but many engineers at Google have been working to develop sets of patterns and practices to alleviate these problems, which we encourage the rest of the company to follow.

The problems Mary ran into weren't her fault, and there was nothing she could have done to avoid them: bad tests must be fixed before they are checked in, lest they impose a drag on future engineers. Broadly speaking, the issues she encountered fall into two categories. First, the tests she was working with were *brittle*: they broke in response to a harmless and unrelated change that introduced no real bugs. Second, the tests were *unclear*: after they were failing, it was difficult to determine what was wrong, how to fix it, and what those tests were supposed to be doing in the first place.

Preventing Brittle Tests

As just defined, a brittle test is one that fails in the face of an unrelated change to production code that does not introduce any real bugs.[1] Such tests must be diagnosed and fixed by engineers as part of their work. In small codebases with only a few engineers, having to tweak a few tests for every change might not be a big problem. But if a team regularly writes brittle tests, test maintenance will inevitably consume a larger and larger proportion of the team's time as they are forced to comb through an increasing number of failures in an ever-growing test suite. If a set of tests need to be manually tweaked by engineers for each change, calling it an "automated test suite" is a bit of a stretch!

Brittle tests cause pain in codebases of any size, but they become particularly acute at Google's scale. An individual engineer might easily run thousands of tests in a single day during the course of their work, and a single large-scale change (see Chapter 22) can trigger hundreds of thousands of tests. At this scale, spurious breakages that affect even a small percentage of tests can waste huge amounts of engineering time. Teams at Google vary quite a bit in terms of how brittle their test suites are, but we've identified a few practices and patterns that tend to make tests more robust to change.

Strive for Unchanging Tests

Before talking about patterns for avoiding brittle tests, we need to answer a question: just how often should we expect to need to change a test after writing it? Any time spent updating old tests is time that can't be spent on more valuable work. Therefore, *the ideal test is unchanging*: after it's written, it never needs to change unless the requirements of the system under test change.

What does this look like in practice? We need to think about the kinds of changes that engineers make to production code and how we should expect tests to respond to those changes. Fundamentally, there are four kinds of changes:

Pure refactorings

> When an engineer refactors the internals of a system without modifying its interface, whether for performance, clarity, or any other reason, the system's tests shouldn't need to change. The role of tests in this case is to ensure that the refactoring didn't change the system's

behavior. Tests that need to be changed during a refactoring indicate that either the change is affecting the system's behavior and isn't a pure refactoring, or that the tests were not written at an appropriate level of abstraction. Google's reliance on large-scale changes (described in Chapter 22) to do such refactorings makes this case particularly important for us.

New features

When an engineer adds new features or behaviors to an existing system, the system's existing behaviors should remain unaffected. The engineer must write new tests to cover the new behaviors, but they shouldn't need to change any existing tests. As with refactorings, a change to existing tests when adding new features suggest unintended consequences of that feature or inappropriate tests.

Bug fixes

Fixing a bug is much like adding a new feature: the presence of the bug suggests that a case was missing from the initial test suite, and the bug fix should include that missing test case. Again, bug fixes typically shouldn't require updates to existing tests.

Behavior changes

Changing a system's existing behavior is the one case when we expect to have to make updates to the system's existing tests. Note that such changes tend to be significantly more expensive than the other three types. A system's users are likely to rely on its current behavior, and changes to that behavior require coordination with those users to avoid confusion or breakages. Changing a test in this case indicates that we're breaking an explicit contract of the system, whereas changes in the previous cases indicate that we're breaking an unintended contract. Low-level libraries will often invest significant effort in avoiding the need to ever make a behavior change so as not to break their users.

The takeaway is that after you write a test, you shouldn't need to touch that test again as you refactor the system, fix bugs, or add new features. This understanding is what makes it possible to work with a system at scale: expanding it requires writing only a small number of new tests related to the change you're making rather than potentially having to touch every test that

has ever been written against the system. Only breaking changes in a system's behavior should require going back to change its tests, and in such situations, the cost of updating those tests tends to be small relative to the cost of updating all of the system's users.

Test via Public APIs

Now that we understand our goal, let's look at some practices for making sure that tests don't need to change unless the requirements of the system being tested change. By far the most important way to ensure this is to write tests that invoke the system being tested in the same way its users would; that is, make calls against its public API rather than its implementation details.[2] If tests work the same way as the system's users, by definition, change that breaks a test might also break a user. As an additional bonus, such tests can serve as useful examples and documentation for users.

Consider Example 12-1, which validates a transaction and saves it to a database.

Example 12-1. A transaction API

```
public void processTransaction(Transaction transaction) {
  if (isValid(transaction)) {
    saveToDatabase(transaction);
  }
}

private boolean isValid(Transaction t) {
  return t.getAmount() < t.getSender().getBalance();
}

private void saveToDatabase(Transaction t) {
  String s = t.getSender() + "," + t.getRecipient() + "," +
t.getAmount();
  database.put(t.getId(), s);
}

public void setAccountBalance(String accountName, int balance) {
  // Write the balance to the database directly
}

public void getAccountBalance(String accountName) {
  // Read transactions from the database to determine the
account balance
}
```

A tempting way to test this code would be to remove the "private" visibility modifiers and test the implementation logic directly, as demonstrated in Example 12-2.

Example 12-2. A naive test of a transaction API's implementation

```
@Test
public void emptyAccountShouldNotBeValid() {

assertThat(processor.isValid(newTransaction().setSender(EMPTY_AC
COUNT)))
      .isFalse();
}

@Test
public void shouldSaveSerializedData() {
  processor.saveToDatabase(newTransaction()
      .setId(123)
      .setSender("me")
      .setRecipient("you")
      .setAmount(100));
  assertThat(database.get(123)).isEqualTo("me,you,100");
}
```

This test interacts with the transaction processor in a much different way than its real users would: it peers into the system's internal state and calls methods that aren't publicly exposed as part of the system's API. As a result, the test is brittle, and almost any refactoring of the system under test (such as renaming its methods, factoring them out into a helper class, or changing the serialization format) would cause the test to break, even if such a change would be invisible to the class's real users.

Instead, the same test coverage can be achieved by testing only against the class's public API, as shown in Example 12-3.[3]

Example 12-3. Testing the public API

```
@Test
public void shouldTransferFunds() {
  processor.setAccountBalance("me", 150);
  processor.setAccountBalance("you", 20);

  processor.processTransaction(newTransaction()
      .setSender("me")
      .setRecipient("you")
      .setAmount(100));

  assertThat(processor.getAccountBalance("me")).isEqualTo(50);
  assertThat(processor.getAccountBalance("you")).isEqualTo(120);
}

@Test
public void shouldNotPerformInvalidTransactions() {
  processor.setAccountBalance("me", 50);
  processor.setAccountBalance("you", 20);
```

```
  processor.processTransaction(newTransaction()
      .setSender("me")
      .setRecipient("you")
      .setAmount(100));

  assertThat(processor.getAccountBalance("me")).isEqualTo(50);
  assertThat(processor.getAccountBalance("you")).isEqualTo(20);
}
```

Tests using only public APIs are, by definition, accessing the system under test in the same manner that its users would. Such tests are more realistic and less brittle because they form explicit contracts: if such a test breaks, it implies that an existing user of the system will also be broken. Testing only these contracts means that you're free to do whatever internal refactoring of the system you want without having to worry about making tedious changes to tests.

It's not always clear what constitutes a "public API," and the question really gets to the heart of what a "unit" is in unit testing. Units can be as small as an individual function or as broad as a set of several related packages/modules. When we say "public API" in this context, we're really talking about the API exposed by that unit to third parties outside of the team that owns the code. This doesn't always align with the notion of visibility provided by some programming languages; for example, classes in Java might define themselves as "public" to be accessible by other packages in the same unit but are not intended for use by other parties outside of the unit. Some languages like Python have no built-in notion of visibility (often relying on conventions like prefixing private method names with underscores), and build systems like Bazel can further restrict who is allowed to depend on APIs declared public by the programming language.

Defining an appropriate scope for a unit and hence what should be considered the public API is more art than science, but here are some rules of thumb:

- If a method or class exists only to support one or two other classes (i.e., it is a "helper class"), it probably shouldn't be considered its own unit, and its functionality should be tested through those classes instead of directly.
- If a package or class is designed to be accessible by anyone without having to consult with its owners, it almost certainly constitutes a unit that should be tested directly, where its tests access the unit in the same way that the users would.

- If a package or class can be accessed only by the people who own it, but it is designed to provide a general piece of functionality useful in a range of contexts (i.e., it is a "support library"), it should also be considered a unit and tested directly. This will usually create some redundancy in testing given that the support library's code will be covered both by its own tests and the tests of its users. However, such redundancy can be valuable: without it, a gap in test coverage could be introduced if one of the library's users (and its tests) were ever removed.

At Google, we've found that engineers sometimes need to be persuaded that testing via public APIs is better than testing against implementation details. The reluctance is understandable because it's often much easier to write tests focused on the piece of code you just wrote rather than figuring out how that code affects the system as a whole. Nevertheless, we have found it valuable to encourage such practices, as the extra upfront effort pays for itself many times over in reduced maintenance burden. Testing against public APIs won't completely prevent brittleness, but it's the most important thing you can do to ensure that your tests fail only in the event of meaningful changes to your system.

Test State, Not Interactions

Another way that tests commonly depend on implementation details involves not which methods of the system the test calls, but in how the results of those calls are verified. In general, there are two ways to verify that a system under test behaves as expected. With *state testing*, you observe the system itself to see what it looks like after invoking with it. With *interaction testing*, you instead check that the system took an expected sequence of actions on its collaborators in response to invoking it.[4] Many tests will perform a combination of state and interaction validation.

Interaction tests tend to be more brittle than state tests for the same reason that it's more brittle to test a private method than to test a public method: interaction tests check *how* a system arrived at its result, whereas usually you should care only *what* the result is. Example 12-4 illustrates a test that uses a test double (explained further in Chapter 13) to verify how a system interacts with a database:

Example 12-4. A brittle interaction test

```
@Test
public void shouldWriteToDatabase() {
  accounts.createUser("foobar");
```

```
  verify(database).put("foobar");
}
```

The test verifies that a specific call was made against a database API, but there are a couple different ways it could go wrong:

- If a bug in the system under test causes the record to be deleted from the database shortly after it was written, the test will pass even though we would have wanted it to fail.
- If the system under test is refactored to call a slightly different API to write an equivalent record, the test will fail even though we would have wanted it to pass.

It's much less brittle to directly test against the state of the system, as demonstrated in Example 12-5.

Example 12-5. Testing against state

```
@Test
public void shouldCreateUsers() {
  accounts.createUser("foobar");
  assertThat(accounts.getUser("foobar")).isNotNull();
}
```

This test more accurately expresses what we care about: the state of the system under test after interacting with it.

The most common reason for problematic interaction tests is an over-reliance on mocking frameworks. These frameworks make it easy to create test doubles that record and verify every call made against them, and to use those doubles in place of real objects in tests. This strategy leads directly to brittle interaction tests, and so we tend to prefer the use of real objects in favor of mocked objects, as long as the real objects are fast and deterministic.

NOTE

For a more extensive discussion of test doubles and mocking frameworks, when they should be used, and safer alternatives, see Chapter 13.

Writing Clear Tests

Sooner or later, even if we've completely avoided brittleness, our tests will fail. Failure is a good thing—test failures provide useful signals to engineers,

and are one of the main ways that a unit test provides value. Test failures happen for one of two reasons:[5]

- The system under test has a problem or is incomplete. This result is exactly what tests are designed for: alerting you of bugs so that you can fix them.
- The test itself is flawed. In this case, nothing is wrong with the system under test, but the test was specified incorrectly. If this was an existing test rather than one that you just wrote, this means that the test is brittle. The previous section discussed how to avoid brittle tests, but it's rarely possible to eliminate them entirely.

When a test fails, an engineer's first job is to identify which of these cases the failure falls into and then to diagnose the actual problem. The speed at which the engineer can do so depends on the test's *clarity*. A clear test is one whose purpose for existing and reason for failing is immediately clear to the engineer diagnosing a failure. Tests fail to achieve clarity when their reasons for failure aren't obvious or when it's difficult to figure out why they were originally written. Clear tests also bring other benefits, such as documenting the system under test and more easily serving as a basis for new tests.

Test clarity becomes significant over time. Tests will often outlast the engineers who wrote them, and the requirements and understanding of a system will shift subtly as it ages. It's entirely possible that a failing test might have been written years ago by an engineer no longer on the team, leaving no way to figure out its purpose or how to fix it. This stands in contrast with unclear production code, whose purpose you can usually determine with enough effort by looking at what calls it and what breaks when it's removed. With an unclear test, you might never understand its purpose, since removing the test will have no effect other than (potentially) introducing a subtle hole in test coverage.

In the worst case, these obscure tests just end up getting deleted when engineers can't figure out how to fix them. Not only does removing such tests introduce a hole in test coverage, but it also indicates that the test has been providing zero value for perhaps the entire period it has existed (which could have been years).

For a test suite to scale and be useful over time, it's important that each individual test in that suite be as clear as possible. This section explores techniques and ways of thinking about tests to achieve clarity.

Make Your Tests Complete and Concise

Two high-level properties that help tests achieve clarity are completeness and conciseness. A test is *complete* when its body contains all of the information a reader needs in order to understand how it arrives at its result. A test is *concise* when it contains no other distracting or irrelevant information. Example 12-6 shows a test that is neither complete nor concise:

Example 12-6. An incomplete and cluttered test

```
@Test
public void shouldPerformAddition() {
  Calculator calculator = new Calculator(new RoundingStrategy(),
      "unused", ENABLE_COSINE_FEATURE, 0.01, calculusEngine,
false);
  int result = calculator.calculate(newTestCalculation());
  assertThat(result).isEqualTo(5); // Where did this number come
from?
}
```

The test is passing a lot of irrelevant information into the constructor, and the actual important parts of the test are hidden inside of a helper method. The test can be made more complete by clarifying the inputs of the helper method, and more concise by using another helper to hide the irrelevant details of constructing the calculator, as illustrated in Example 12-7.

Example 12-7. A complete, concise test

```
@Test
public void shouldPerformAddition() {
  Calculator calculator = newCalculator();
  int result = calculator.calculate(newCalculation(2,
Operation.PLUS, 3));
  assertThat(result).isEqualTo(5);
}
```

Ideas we discuss later, especially around code sharing, will tie back to completeness and conciseness. In particular, it can often be worth violating the DRY (Don't Repeat Yourself) principle if it leads to clearer tests. Remember: a *test's body should contain all of the information needed to understand it without containing any irrelevant or distracting information*.

Test Behaviors, Not Methods

The first instinct of many engineers is to try to match the structure of their tests to the structure of their code such that every production method has a corresponding test method. This pattern can be convenient at first, but over time it leads to problems: as the method being tested grows more complex, its

test also grows in complexity and becomes more difficult to reason about. For example, consider the snippet of code in Example 12-8, which displays the results of a transaction.

Example 12-8. A transaction snippet

```
public void displayTransactionResults(User user, Transaction
transaction) {
  ui.showMessage("You bought a " + transaction.getItemName());
  if (user.getBalance() < LOW_BALANCE_THRESHOLD) {
    ui.showMessage("Warning: your balance is low!");
  }
}
```

It wouldn't be uncommon to find a test covering both of the messages that might be shown by the method, as presented in Example 12-9.

Example 12-9. A method-driven test

```
@Test
public void testDisplayTransactionResults() {
  transactionProcessor.displayTransactionResults(
      newUserWithBalance(
          LOW_BALANCE_THRESHOLD.plus(dollars(2))),
      new Transaction("Some Item", dollars(3)));

  assertThat(ui.getText()).contains("You bought a Some Item");
  assertThat(ui.getText()).contains("your balance is low");
}
```

With such tests, it's likely that the test started out covering only the first method. Later, an engineer expanded the test when the second message was added (violating the idea of unchanging tests that we discussed earlier). This modification sets a bad precedent: as the method under test becomes more complex and implements more functionality, its unit test will become increasingly convoluted and grow more and more difficult to work with.

The problem is that framing tests around methods can naturally encourage unclear tests because a single method often does a few different things under the hood and might have several tricky edge and corner cases. There's a better way: rather than writing a test for each method, write a test for each *behavior*.[6, 7] A behavior is any guarantee that a system makes about how it will respond to a series of inputs while in a particular state.[8] Behaviors can often be expressed using the words "given," "when," and "then": "*Given* that a bank account is empty, *when* attempting to withdraw money from it, *then* the transaction is rejected." The mapping between methods and behaviors is many-to-many: most nontrivial methods implement multiple behaviors, and some behaviors rely on the interaction of multiple methods.

The previous example can be rewritten using behavior-driven tests, as presented in Example 12-10.

Example 12-10. A behavior-driven test

```
@Test
public void displayTransactionResults_showsItemName() {
  transactionProcessor.displayTransactionResults(
      new User(), new Transaction("Some Item"));
  assertThat(ui.getText()).contains("You bought a Some Item");
}

@Test
public void displayTransactionResults_showsLowBalanceWarning() {
  transactionProcessor.displayTransactionResults(
      newUserWithBalance(
          LOW_BALANCE_THRESHOLD.plus(dollars(2))),
      new Transaction("Some Item", dollars(3)));
  assertThat(ui.getText()).contains("your balance is low");
}
```

The extra boilerplate required to split apart the single test is more than worth it,[9] and the resulting tests are much clearer than the original test. Behavior-driven tests tend to be clearer than method-oriented tests for several reasons. First, they read more like natural language, allowing them to be naturally understood rather than requiring laborious mental parsing. Second, they more clearly express cause and effect because each test is more limited in scope. Finally, the fact that each test is short and descriptive makes it easier to see what functionality is already tested and encourages engineers to add new streamlined test methods instead of piling onto existing methods.

STRUCTURE TESTS TO EMPHASIZE BEHAVIORS

Thinking about tests as being coupled to behaviors instead of methods significantly affects how they should be structured. Remember that every behavior has three parts: a "given" component that defines how the system is set up, a "when" component that defines the action to be taken on the system, and a "then" component that validates the result.[10] Tests are clearest when this structure is explicit. Some frameworks like Cucumber and Spock directly bake in given/when/then. Other languages can use whitespace and optional comments to make the structure stand out, such as that shown in Example 12-11.

Example 12-11. A well-structured test

```
@Test
public void transferFundsShouldMoveMoneyBetweenAccounts() {
  // Given two accounts with initial balances of $150 and $20
  Account account1 = newAccountWithBalance(usd(150));
```

```
  Account account2 = newAccountWithBalance(usd(20));

  // When transferring $100 from the first to the second account
  bank.transferFunds(account1, account2, usd(100));

  // Then the new account balances should reflect the transfer
  assertThat(account1.getBalance()).isEqualTo(usd(50));
  assertThat(account2.getBalance()).isEqualTo(usd(120));
}
```

This level of description isn't always necessary in trivial tests, and it's usually sufficient to omit the comments and rely on whitespace to make the sections clear. However, explicit comments can make more sophisticated tests easier to understand. This pattern makes it possible to read tests at three levels of granularity:

1. A reader can start by looking at the test method name (discussed below) to get a rough description of the behavior being tested.
2. If that's not enough, the reader can look at the given/when/then comments for a formal description of the behavior.
3. Finally, a reader can look at the actual code to see precisely how that behavior is expressed.

This pattern is most commonly violated by interspersing assertions among multiple calls to the system under test (i.e., combining the "when" and "then" blocks). Merging the "then" and "when" blocks in this way can make the test less clear because it makes it difficult to distinguish the action being performed from the expected result.

When a test does want to validate each step in a multistep process, it's acceptable to define alternating sequences of when/then blocks. Long blocks can also be made more descriptive by splitting them up with the word "and." Example 12-12 shows what a relatively complex behavior-driven test might look like.

Example 12-12. Alternating when/then blocks within a test

```
@Test
public void shouldTimeOutConnections() {
  // Given two users
  User user1 = newUser();
  User user2 = newUser();

  // And an empty connection pool with a 10-minute timeout
  Pool pool = newPool(Duration.minutes(10));

  // When connecting both users to the pool
```

```
  pool.connect(user1);
  pool.connect(user2);

  // Then the pool should have two connections
  assertThat(pool.getConnections()).hasSize(2);

  // When waiting for 20 minutes
  clock.advance(Duration.minutes(20));

  // Then the pool should have no connections
  assertThat(pool.getConnections()).isEmpty();

  // And each user should be disconnected
  assertThat(user1.isConnected()).isFalse();
  assertThat(user2.isConnected()).isFalse();
}
```

When writing such tests, be careful to ensure that you're not inadvertently testing multiple behaviors at the same time. Each test should cover only a single behavior, and the vast majority of unit tests require only one "when" and one "then" block.

NAME TESTS AFTER THE BEHAVIOR BEING TESTED

Method-oriented tests are usually named after the method being tested (e.g., a test for the `updateBalance` method is usually called `testUpdateBalance`). With more focused behavior-driven tests, we have a lot more flexibility and the chance to convey useful information in the test's name. The test name is very important: it will often be the first or only token visible in failure reports, so it's your best opportunity to communicate the problem when the test breaks. It's also the most straightforward way to express the intent of the test.

A test's name should summarize the behavior it is testing. A good name describes both the actions that are being taken on a system and the expected outcome.[11] Test names will sometimes include additional information like the state of the system or its environment before taking action on it. Some languages and frameworks make this easier than others by allowing tests to be nested within one another and named using strings, such as in Example 12-13, which uses Jasmine.

Example 12-13. Some sample nested naming patterns

```
describe("multiplication", function() {
  describe("with a positive number", function() {
    var positiveNumber = 10;
    it("is positive with another positive number", function() {
      expect(positiveNumber * 10).toBeGreaterThan(0);
    });
    it("is negative with a negative number", function() {
```

```
      expect(positiveNumber * -10).toBeLessThan(0);
    });
  });
  describe("with a negative number", function() {
    var negativeNumber = 10;
    it("is negative with a positive number", function() {
      expect(negativeNumber * 10).toBeLessThan(0);
    });
    it("is positive with another negative number", function() {
      expect(negativeNumber * -10).toBeGreaterThan(0);
    });
  });
});
```

Other languages require us to encode all of this information in a method name, leading to method naming patterns like that shown in Example 12-14.

Example 12-14. Some sample method naming patterns

```
multiplyingTwoPositiveNumbersShouldReturnAPositiveNumber
multiply_postiveAndNegative_returnsNegative
divide_byZero_throwsException
```

Names like this are much more verbose than we'd normally want to write for methods in production code, but the use case is different: we never need to write code that calls these, and their names frequently need to be read by humans in reports. Hence, the extra verbosity is warranted.

Many different naming strategies are acceptable so long as they're used consistently within a single test class. A good trick if you're stuck is to try starting the test name with the word "should." When taken with the name of the class being tested, this naming scheme allows the test name to be read as a sentence. For example, a test of a `BankAccount` class named `shouldNotAllowWithdrawalsWhenBalanceIsEmpty` can be read as "BankAccount should not allow withdrawals when balance is empty." By reading the names of all the test methods in a suite, you should get a good sense of the behaviors implemented by the system under test. Such names also help ensure that the test stays focused on a single behavior: if you need to use the word "and" in a test name, there's a good chance that you're actually testing multiple behaviors and should be writing multiple tests!

Don't Put Logic in Tests

Clear tests are trivially correct upon inspection; that is, it is obvious that a test is doing the correct thing just from glancing at it. This is possible in test code because each test needs to handle only a particular set of inputs, whereas production code must be generalized to handle any input. For production

code, we're able to write tests that ensure complex logic is correct. But test code doesn't have that luxury—if you feel like you need to write a test to verify your test, something has gone wrong!

Complexity is most often introduced in the form of *logic*. Logic is defined via the imperative parts of programming languages such as operators, loops, and conditionals. When a piece of code contains logic, you need to do a bit of mental computation to determine its result instead of just reading it off of the screen. It doesn't take much logic to make a test more difficult to reason about. For example, does the test in Example 12-15 look correct to you?[12]

Example 12-15. Logic concealing a bug

```
@Test
public void shouldNavigateToAlbumsPage() {
  String baseUrl = "http://photos.google.com/";
  Navigator nav = new Navigator(baseUrl);
  nav.goToAlbumPage();
  assertThat(nav.getCurrentUrl()).isEqualTo(baseUrl +
"/albums");
}
```

There's not much logic here: really just one string concatenation. But if we simplify the test by removing that one bit of logic, a bug immediately becomes clear, as demonstrated in Example 12-16.

Example 12-16. A test without logic reveals the bug

```
@Test
public void shouldNavigateToPhotosPage() {
  Navigator nav = new Navigator("http://photos.google.com/");
  nav.goToPhotosPage();
  assertThat(nav.getCurrentUrl())
      .isEqualTo("http://photos.google.com//albums"); // Oops!
}
```

When the whole string is written out, we can see right away that we're expecting two slashes in the URL instead of just one. If the production code made a similar mistake, this test would fail to detect a bug. Duplicating the base URL was a small price to pay for making the test more descriptive and meaningful (see the discussion of DAMP versus DRY tests later in this chapter).

If humans are bad at spotting bugs from string concatenation, we're even worse at spotting bugs that come from more sophisticated programming constructs like loops and conditionals. The lesson is clear: in test code, stick to straight-line code over clever logic, and consider tolerating some

duplication when it makes the test more descriptive and meaningful. We'll discuss ideas around duplication and code sharing later in this chapter.

Write Clear Failure Messages

One last aspect of clarity has to do not with how a test is written, but with what an engineer sees when it fails. In an ideal world, an engineer could diagnose a problem just from reading its failure message in a log or report without ever having to look at the test itself. A good failure message contains much the same information as the test's name: it should clearly express the desired outcome, the actual outcome, and any relevant parameters.

Here's an example of a bad failure message:

```
Test failed: account is closed
```

Did the test fail because the account was closed, or was the account expected to be closed and the test failed because it wasn't? A better failure message clearly distinguishes the expected from the actual state and gives more context about the result:

```
Expected an account in state CLOSED, but got account <{name:
"my-account", state: "OPEN"}
```

Good libraries can help make it easier to write useful failure messages. Consider the assertions in Example 12-17 in a Java test, the first of which uses classical JUnit asserts, and the second of which uses Truth, an assertion library developed by Google:

Example 12-17. An assertion using the Truth library

```
Set<String> colors = ImmutableSet.of("red", "green", "blue");
assertTrue(colors.contains("orange"));  // JUnit
assertThat(colors).contains("orange");  // Truth
```

Because the first assertion only receives a Boolean value, it is only able to give a generic error message like "expected <true> but was <false>," which isn't very informative in a failing test output. Because the second assertion explicitly receives the subject of the assertion, it is able to give a much more useful error message:[13] "AssertionError: <[red, green, blue]> should have contained <orange>."

Not all languages have such helpers available, but it should always be possible to manually specify the important information in the failure

message. For example, test assertions in Go conventionally look like Example 12-18.

Example 12-18. A test assertion in Go

```
result := Add(2, 3)
if result != 5 {
  t.Errorf("Add(2, 3) = %v, want %v", result, 5)
}
```

Tests and Code Sharing: DAMP, Not DRY

One final aspect of writing clear tests and avoiding brittleness has to do with code sharing. Most software attempts to achieve a principle called DRY—"Don't Repeat Yourself." DRY states that software is easier to maintain if every concept is canonically represented in one place and code duplication is kept to a minimum. This approach is especially valuable in making changes easier because an engineer needs to update only one piece of code rather than tracking down multiple references. The downside to such consolidation is that it can make code unclear, requiring readers to follow chains of references to understand what the code is doing.

In normal production code, that downside is usually a small price to pay for making code easier to change and work with. But this cost/benefit analysis plays out a little differently in the context of test code. Good tests are designed to be stable, and in fact you usually *want* them to break when the system being tested changes. So DRY doesn't have quite as much benefit when it comes to test code. At the same time, the costs of complexity are greater for tests: production code has the benefit of a test suite to ensure that it keeps working as it becomes complex, whereas tests must stand by themselves, risking bugs if they aren't self-evidently correct. As mentioned earlier, something has gone wrong if tests start becoming complex enough that it feels like they need their own tests to ensure that they're working properly.

Instead of being completely DRY, test code should often strive to be DAMP; that is, to promote "Descriptive And Meaningful Phrases." A little bit of duplication is okay in tests so long as that duplication makes the test simpler and clearer. To illustrate, Example 12-19 presents some tests that are far too DRY.

Example 12-19. A test that is too DRY

```
@Test
public void shouldAllowMultipleUsers() {
```

```
  List<User> users = createUsers(false, false);
  Forum forum = createForumAndRegisterUsers(users);
  validateForumAndUsers(forum, users);
}

@Test
public void shouldNotAllowBannedUsers() {
  List<User> users = createUsers(true);
  Forum forum = createForumAndRegisterUsers(users);
  validateForumAndUsers(forum, users);
}

// Lots more tests...

private static List<User> createUsers(boolean... banned) {
  List<User> users = new ArrayList<>();
  for (boolean isBanned : banned) {
    users.add(newUser()
        .setState(isBanned ? State.BANNED : State.NORMAL)
        .build());
  }
  return users;
}

private static Forum createForumAndRegisterUsers(List<User>
users) {
  Forum forum = new Forum();
  for (User user : users) {
    try {
      forum.register(user);
    } catch(BannedUserException ignored) {}
  }
  return forum;
}

private static void validateForumAndUsers(Forum forum,
List<User> users) {
  assertThat(forum.isReachable()).isTrue();
  for (User user : users) {
    assertThat(forum.hasRegisteredUser(user))
        .isEqualTo(user.getState() == State.BANNED);
  }
}
```

The problems in this code should be apparent based on the previous discussion of clarity. For one, although the test bodies are very concise, they are not complete: important details are hidden away in helper methods that the reader can't see without having to scroll to a completely different part of the file. Those helpers are also full of logic that makes them more difficult to verify at a glance (did you spot the bug?). The test becomes much clearer when it's rewritten to use DAMP, as shown in Example 12-20.

Example 12-20. Tests should be DAMP

```
@Test
public void shouldAllowMultipleUsers() {
  User user1 = newUser().setState(State.NORMAL).build();
  User user2 = newUser().setState(State.NORMAL).build();

  Forum forum = new Forum();
  forum.register(user1);
  forum.register(user2);

  assertThat(forum.hasRegisteredUser(user1)).isTrue();
  assertThat(forum.hasRegisteredUser(user2)).isTrue();
}

@Test
public void shouldNotRegisterBannedUsers() {
  User user = newUser().setState(State.BANNED).build();

  Forum forum = new Forum();
  try {
    forum.register(user);
  } catch(BannedUserException ignored) {}

  assertThat(forum.hasRegisteredUser(user)).isFalse();
}
```

These tests have more duplication, and the test bodies are a bit longer, but the extra verbosity is worth it. Each individual test is far more meaningful and can be understood entirely without leaving the test body. A reader of these tests can feel confident that the tests do what they claim to do and aren't hiding any bugs.

DAMP is not a replacement for DRY; it is complementary to it. Helper methods and test infrastructure can still help make tests clearer by making them more concise, factoring out repetitive steps whose details aren't relevant to the particular behavior being tested. The important point is that such refactoring should be done with an eye toward making tests more descriptive and meaningful, and not solely in the name of reducing repetition. The rest of this section will explore common patterns for sharing code across tests.

Shared Values

Many tests are structured by defining a set of shared values to be used by tests and then by defining the tests that cover various cases for how these values interact. Example 12-21 illustrates what such tests look like:

Example 12-21. Shared values with ambiguous names

```
private static final Account ACCOUNT_1 = Account.newBuilder()
```

```
    .setState(AccountState.OPEN).setBalance(50).build();

private static final Account ACCOUNT_2 = Account.newBuilder()
    .setState(AccountState.CLOSED).setBalance(0).build();

private static final Item ITEM = Item.newBuilder()
    .setName("Cheeseburger").setPrice(100).build();

// Hundreds of lines of other tests...

@Test
public void canBuyItem_returnsFalseForClosedAccounts() {
  assertThat(store.canBuyItem(ITEM, ACCOUNT_1)).isFalse();
}

@Test
public void canBuyItem_returnsFalseWhenBalanceInsufficient() {
  assertThat(store.canBuyItem(ITEM, ACCOUNT_2)).isFalse();
}
```

This strategy can make tests very concise, but it causes problems as the test suite grows. For one, it can be difficult to understand why a particular value was chosen for a test. In Example 12-21, the test names fortunately clarify which scenarios are being tested, but you still need to scroll up to the definitions to confirm that `ACCOUNT_1` and `ACCOUNT_2` are appropriate for those scenarios. More descriptive constant names (e.g., `CLOSED_ACCOUNT` and `ACCOUNT_WITH_LOW_BALANCE`) help a bit, but they still make it more difficult to see the exact details of the value being tested, and the ease of reusing these values can encourage engineers to do so even when the name doesn't exactly describe what the test needs.

Engineers are usually drawn to using shared constants because constructing individual values in each test can be verbose. A better way to accomplish this goal is to construct data using helper methods[14] (see Example 12-22) that require the test author to specify only values they care about, and setting reasonable defaults[15] for all other values. This construction is trivial to do in languages that support named parameters, but languages without named parameters can use constructs such as the *Builder* pattern to emulate them (often with the assistance of tools such as AutoValue):

Example 12-22. Shared values using helper methods

```
# A helper method wraps a constructor by defining arbitrary
defaults for
# each of its parameters.
def newContact(
    firstName="Grace", lastName="Hopper", phoneNumber="555-123-
4567"):
  return Contact(firstName, lastName, phoneNumber)
```

```
# Tests call the helper, specifying values for only the
parameters that they
# care about.
def test_fullNameShouldCombineFirstAndLastNames(self):
  def contact = newContact(firstName="Ada", lastName="Lovelace")
  self.assertEqual(contact.fullName(), "Ada Lovelace")

// Languages like Java that don't support named parameters can
emulate them
// by returning a mutable "builder" object that represents the
value under
// construction.
private static Contact.Builder newContact() {
  return Contact.newBuilder()
    .setFirstName("Grace")
    .setLastName("Hopper")
    .setPhoneNumber("555-123-4567");
}

// Tests then call methods on the builder to overwrite only the
parameters
// that they care about, then call build() to get a real value
out of the
// builder.
@Test
public void fullNameShouldCombineFirstAndLastNames() {
  Contact contact = newContact()
      .setFirstName("Ada")
      .setLastName("Lovelace")
      .build();
  assertThat(contact.getFullName()).isEqualTo("Ada Lovelace");
}
```

Using helper methods to construct these values allows each test to create the exact values it needs without having to worry about specifying irrelevant information or conflicting with other tests.

Shared Setup

A related way that tests shared code is via setup/initialization logic. Many test frameworks allow engineers to define methods to execute before each test in a suite is run. Used appropriately, these methods can make tests clearer and more concise by obviating the repetition of tedious and irrelevant initialization logic. Used inappropriately, these methods can harm a test's completeness by hiding important details in a separate initialization method.

The best use case for setup methods is to construct the object under tests and its collaborators. This is useful when the majority of tests don't care about the

specific arguments used to construct those objects and can let them stay in their default states. The same idea also applies to stubbing return values for test doubles, which is a concept that we explore in more detail in Chapter 13.

One risk in using setup methods is that they can lead to unclear tests if those tests begin to depend on the particular values used in setup. For example, the test in Example 12-23 seems incomplete because a reader of the test needs to go hunting to discover where the string "Donald Knuth" came from.

Example 12-23. Dependencies on values in setup methods

```
private NameService nameService;
private UserStore userStore;

@Before
public void setUp() {
  nameService = new NameService();
  nameService.set("user1", "Donald Knuth");
  userStore = new UserStore(nameService);
}

// [... hundreds of lines of tests ...]

@Test
public void shouldReturnNameFromService() {
  UserDetails user = userStore.get("user1");
  assertThat(user.getName()).isEqualTo("Donald Knuth");
}
```

Tests like these that explicitly care about particular values should state those values directly, overriding the default defined in the setup method if need be. The resulting test contains slightly more repetition, as shown in Example 12-24, but the result is far more descriptive and meaningful.

Example 12-24. Overriding values in setup mMethods

```
private NameService nameService;
private UserStore userStore;

@Before
public void setUp() {
  nameService = new NameService();
  nameService.set("user1", "Donald Knuth");
  userStore = new UserStore(nameService);
}

@Test
public void shouldReturnNameFromService() {
  nameService.set("user1", "Margaret Hamilton");
  UserDetails user = userStore.get("user1");
  assertThat(user.getName()).isEqualTo("Margaret Hamilton");
}
```

Shared Helpers and Validation

The last common way that code is shared across tests is via "helper methods" called from the body of the test methods. We already discussed how helper methods can be a useful way for concisely constructing test values—this usage is warranted, but other types of helper methods can be dangerous.

One common type of helper is a method that performs a common set of assertions against a system under test. The extreme example is a `validate` method called at the end of every test method, which performs a set of fixed checks against the system under test. Such a validation strategy can be a bad habit to get into because tests using this approach are less behavior driven. With such tests, it is much more difficult to determine the intent of any particular test and to infer what exact case the author had in mind when writing it. When bugs are introduced, this strategy can also make them more difficult to localize because they will frequently cause a large number of tests to start failing.

More focused validation methods can still be useful, however. The best validation helper methods assert a *single conceptual fact* about their inputs, in contrast to general-purpose validation methods that cover a range of conditions. Such methods can be particularly helpful when the condition that they are validating is conceptually simple but requires looping or conditional logic to implement that would reduce clarity were it included in the body of a test method. For example, the helper method in Example 12-25 might be useful in a test covering several different cases around account access.

Example 12-25. A conceptually simple test

```
private void assertUserHasAccessToAccount(User user, Account
account) {
  for (long userId : account.getUsersWithAccess()) {
    if (user.getId() == userId) {
      return;
    }
  }
  fail(user.getName() + " cannot access " + account.getName());
}
```

Defining Test Infrastructure

The techniques we've discussed so far cover sharing code across methods in a single test class or suite. Sometimes, it can also be valuable to share code across multiple test suites. We refer to this sort of code as *test infrastructure*. Though it is usually more valuable in integration or end-to-end tests,

carefully designed test infrastructure can make unit tests much easier to write in some circumstances.

Custom test infrastructure must be approached more carefully than the code sharing that happens within a single test suite. In many ways, test infrastructure code is more similar to production code than it is to other test code given that it can have many callers that depend on it and can be difficult to change without introducing breakages. Most engineers aren't expected to make changes to the common test infrastructure while testing their own features. Test infrastructure needs to be treated as its own separate product, and accordingly *test infrastructure must always have its own tests*.

Of course, most of the test infrastructure that most engineers use comes in the form of well-known third-party libraries like JUnit. A huge number of such libraries are available, and standardizing on them within an organization should happen as early and universally as possible. For example, Google many years ago mandated Mockito as the only mocking framework that should be used in new Java tests and banned new tests from using other mocking frameworks. This edict produced some grumbling at the time from people comfortable with other frameworks, but today it's universally seen as a good move that made our tests easier to understand and work with.

Conclusion

Unit tests are one of the most powerful tools that we as software engineers have to make sure that our systems keep working over time in the face of unanticipated changes. But with great power comes great responsibility, and careless use of unit testing can result in a system that requires much more effort to maintain and takes much more effort to change without actually improving our confidence in said system.

Unit tests at Google are far from perfect, but we've found tests that follow the practices outlined in this chapter to be orders of magnitude more valuable than those that don't. We hope they'll help you to improve the quality of your own tests!

TL;DRs

- Strive for unchanging tests.
- Test via public APIs.

- Test state, not interactions.
- Make your tests complete and concise.
- Test behaviors, not methods.
- Structure tests to emphasize behaviors.
- Name tests after the behavior being tested.
- Don't put logic in tests.
- Write clear failure messages.
- Follow DAMP over DRY when sharing code for tests.

1 Note that this is slightly different from a *flaky test*, which fails nondeterministically without any change to production code.

2 *https://testing.googleblog.com/2015/01/testing-on-toilet-prefer-testing-public.html*

3 This is sometimes called the "Use the front door first principle."

4 *https://testing.googleblog.com/2013/03/testing-on-toilet-testing-state-vs.html*

5 These are also the same two reasons that a test can be "flaky." Either the system under test has a nondeterministic fault, or the test is flawed such that it sometimes fails when it should pass.

6 *https://testing.googleblog.com/2014/04/testing-on-toilet-test-behaviors-not.html*

7 *https://dannorth.net/introducing-bdd/*

8 Furthermore, a *feature* (in the product sense of the word) can be expressed as a collection of behaviors.

9 *https://testing.googleblog.com/2018/06/testing-on-toilet-keep-tests-focused.html*

10 These components are sometimes referred to as "arrange," "act," and "assert."

11 *https://testing.googleblog.com/2014/10/testing-on-toilet-writing-descriptive.html*

12 *https://testing.googleblog.com/2014/07/testing-on-toilet-dont-put-logic-in.html*

13 *https://testing.googleblog.com/2014/12/testing-on-toilet-truth-fluent.html*

14 *https://testing.googleblog.com/2018/02/testing-on-toilet-cleanly-create-test.html*

15 In many cases it can even be useful to slightly randomize the default values returned for fields that aren't explicitly set. This helps to ensure that two different instances won't accidentally compare as equal, and makes it more difficult for engineers to hardcode dependencies on the defaults.

Chapter 13. Test Doubles

Written by Andrew Trenk and Dillon Bly

Edited by Tom Manshreck

Unit tests are a critical tool for keeping developers productive and reducing defects in code. Although they can be easy to write for simple code, writing them becomes difficult as code becomes more complex.

For example, imagine trying to write a test for a function that sends a request to an external server and then stores the response in a database. Writing a handful of tests might be doable with some effort. But if you need to write hundreds or thousands of tests like this, your test suite will likely take hours to run, and could become flaky due to issues like random network failures or tests overwriting one another's data.

Test doubles come in handy in such cases. A *test double* is an object or function that can stand in for a real implementation in a test, similar to how a stunt double can stand in for an actor in a movie. The use of test doubles is often referred to as *mocking*, but we avoid that term in this chapter because, as we'll see, that term is also used to refer to more-specific aspects of test doubles.

Perhaps the most obvious type of test double is a simpler implementation of an object that behaves similarly to the real implementation, such as an in-memory database. Other types of test doubles can make it possible to validate specific details of your system, such as by making it easy to trigger a rare error condition, or ensuring a heavyweight function is called without actually executing the function's implementation.

The previous two chapters introduced the concept of *small tests* and discussed why they should comprise the majority of tests in a test suite. However, production code often doesn't fit within the constraints of small tests due to communication across multiple processes or machines. Test doubles can be much more lightweight than real implementations, allowing you to write many small tests that execute quickly and are not flaky.

The Impact of Test Doubles on Software Development

The use of test doubles introduces a few complications to software development that require some trade-offs to be made. The concepts introduced here are discussed in more depth throughout this chapter:

Testability

To use test doubles, a codebase needs to be designed to be *testable*—it should be possible for tests to swap out real implementations with test doubles. For example, code that calls a database needs to be flexible enough to be able to use a test double in place of a real database. If the codebase isn't designed with testing in mind and you later decide that tests are needed, it can require a major commitment to refactor the code to support the use of test doubles.

Applicability

Although proper application of test doubles can provide a powerful boost to engineering velocity, their improper use can lead to tests that are brittle, complex, and less effective. These downsides are magnified when test doubles are used improperly across a large codebase, potentially resulting in major losses in productivity for engineers. In many cases, test doubles are not suitable and engineers should prefer to use real implementations, instead.

Fidelity

Fidelity refers to how closely the behavior of a test double resembles the behavior of the real implementation that it's replacing. If the behavior of a test double significantly differs from the real implementation, tests that use the test double likely wouldn't provide much value—for example, imagine trying to write a test with a test double for a database that ignores any data added to the database and always returns empty results. But perfect fidelity might not be feasible; test doubles often need to be vastly simpler than the real implementation in order to be suitable for use in tests. In many situations, it is appropriate to use a test double even without perfect fidelity. Unit tests that use test doubles often need to be supplemented by larger-scope tests that exercise the real implementation.

Test Doubles at Google

At Google, we've seen countless examples of the benefits to productivity and software quality that test doubles can bring to a codebase, as well as the negative impact they can cause when used improperly. The practices we follow at Google have evolved over time based on these experiences. Historically, we had few guidelines on how to effectively use test doubles, but best practices evolved as we saw common patterns and antipatterns arise in many teams' codebases.

One lesson we learned the hard way is the danger of overusing mocking frameworks, which allow you to easily create test doubles (we will discuss mocking frameworks in more detail later in this chapter). When mocking frameworks first came into use at Google, they seemed like a hammer fit for every nail—they made it very easy to write highly focused tests against isolated pieces of code without having to worry about how to construct the dependencies of that code. It wasn't until several years and countless tests later that we began to realize the cost of such tests: though these tests were easy to write, we suffered greatly given that they required constant effort to maintain while rarely finding bugs. The pendulum at Google has now begun swinging in the other direction, with many engineers avoiding mocking frameworks in favor of writing more-realistic tests.

Even though the practices discussed in this chapter are generally agreed upon at Google, the actual application of them varies widely from team to team. This variance stems from engineers having inconsistent knowledge of these practices, inertia in an existing codebase that doesn't conform to these practices, or teams doing what is easiest for the short term without thinking about the long-term implications.

Basic Concepts

Before we dive into how to effectively use test doubles, let's cover some of the basic concepts related to them. These build the foundation for best practices that we will discuss later in this chapter.

An Example Test Double

Imagine an ecommerce site which needs to process credit card payments. At its core, it might have something like the code shown in Example 13-1.

Example 13-1. A credit card service

```
class PaymentProcessor {
  private CreditCardService creditCardService;
  ...
  boolean makePayment(CreditCard creditCard, Money amount) {
    if (creditCard.isExpired()) { return false; }
    boolean success =
        creditCardService.chargeCreditCard(creditCard, amount);
    return success;
  }
}
```

It would be infeasible to use a real credit card service in a test (imagine all the transaction fees from running the test!), but a test double could be used in its place to *simulate* the behavior of the real system. The code in Example 13-2 shows an extremely simple test double.

Example 13-2. A trivial test double

```
class TestDoubleCreditCardService implements CreditCardService {
 @Override
 public boolean chargeCreditCard(CreditCard creditCard, Money
amount) {
   return true;
 }
}
```

Although this test double doesn't look very useful, using it in a test still allows us to test some of the logic in the `makePayment()` method. For example, in Example 13-3, we can validate that the method behaves properly when the credit card is expired because the code path that the test exercises doesn't rely on the behavior of the credit card service.

Example 13-3. Using the test double

```
@Test public void cardIsExpired_returnFalse() {
  boolean success = paymentProcessor.makePayment(EXPIRED_CARD,
AMOUNT);
  assertThat(success).isFalse();
}
```

The following sections in this chapter will discuss how to make use of test doubles in more-complex situations than this one.

Seams

Code is said to be *testable* if it is written in a way that makes it possible to write unit tests for the code. A *seam* is a way to make code testable by allowing for the use of test doubles—it makes it possible to use different

dependencies for the system under test rather than the dependencies used in a production environment.

Dependency injection is a common technique for introducing seams. In short, when a class utilizes dependency injection, any classes it needs to use (i.e., the class's *dependencies*) are passed to it rather than instantiated directly, making it possible for these dependencies to be substituted in tests.

Example 13-4 shows an example of dependency injection. Rather than the constructor creating an instance of `CreditCardService`, it accepts an instance as a parameter.

Example 13-4. Dependency injection

```
class PaymentProcessor {
  private CreditCardService creditCardService;

  PaymentProcessor(CreditCardService creditCardService) {
    this.creditCardService = creditCardService;
  }
  ...
}
```

The code that calls this constructor is responsible for creating an appropriate `CreditCardService` instance. Whereas the production code can pass in an implementation of `CreditCardService` that communicates with an external server, the test can pass in a test double, as demonstrated in Example 13-5.

Example 13-5. Passing in a test double

```
PaymentProcessor paymentProcessor =
    new PaymentProcessor(new TestDoubleCreditCardService());
```

To reduce boilerplate associated with manually specifying constructors, automated dependency injection frameworks can be used for constructing object graphs automatically. At Google, Guice and Dagger are automated dependency injection frameworks that are commonly used for Java code.

With dynamically typed languages such as Python or JavaScript, it is possible to dynamically replace individual functions or object methods. Dependency injection is less important in these languages because this capability makes it possible to use real implementations of dependencies in tests while only overriding functions or methods of the dependency that are unsuitable for tests.

Writing testable code requires an upfront investment. It is especially critical early in the lifetime of a codebase because the later testability is taken into account, the more difficult it is to apply to a codebase. Code written without

testing in mind typically needs to be refactored or rewritten before you can add appropriate tests.

Mocking Frameworks

A *mocking framework* is a software library that makes it easier to create test doubles within tests; it allows you to replace an object with a *mock*, which is a test double whose behavior is specified inline in a test. The use of mocking frameworks reduces boilerplate because you don't need to define a new class each time you need a test double.

Example 13-6 demonstrates the use of Mockito, a mocking framework for Java. Mockito creates a test double for `CreditCardService` and instructs it to return a specific value.

Example 13-6. Mocking frameworks

```
class PaymentProcessorTest {
  ...
  PaymentProcessor paymentProcessor;

  // Create a test double of CreditCardService with just one
line of code.
  @Mock CreditCardService mockCreditCardService;
  @Before public void setUp() {
    // Pass in the test double to the system under test.
    paymentProcessor = new
PaymentProcessor(mockCreditCardService);
  }
  @Test public void chargeCreditCardFails_returnFalse() {
    // Give some behavior to the test double: it will return
false
    // anytime the chargeCreditCard() method is called. The
usage of
    // "any()" for the method's arguments tells the test double
to
    // return false regardless of which arguments are passed.
    when(mockCreditCardService.chargeCreditCard(any(), any())
        .thenReturn(false);
    boolean success = paymentProcessor.makePayment(CREDIT_CARD,
AMOUNT);
    assertThat(success).isFalse();
  }
}
```

Mocking frameworks exist for most major programming languages. At Google, we use Mockito for Java, googlemock for C++, and unittest.mock for Python.

Although mocking frameworks facilitate easier usage of test doubles, they come with some significant caveats given that their overuse will often make a codebase more difficult to maintain. We cover some of these problems later in this chapter.

Techniques for Using Test Doubles

There are three primary techniques for using test doubles. This section presents a brief introduction to these techniques to give you a quick overview of what they are and how they differ. Later sections in this chapter go into more details on how to effectively apply them.

An engineer who is aware of the distinctions between these techniques is more likely to know the appropriate technique to use when faced with the need to use a test double.

Faking

A *fake* is a lightweight implementation of an API that behaves similar to the real implementation but isn't suitable for production; for example, an in-memory database. Example 13-7 presents an example of faking.

Example 13-7. A simple fake

```
// Creating the fake is fast and easy.
AuthorizationService fakeAuthorizationService =
    new FakeAuthorizationService();
AccessManager accessManager = new
AccessManager(fakeAuthorizationService):
// Unknown user IDs shouldn't have access.
assertFalse(accessManager.userHasAccess(USER_ID));
// The user ID should have access after it is added to
// the authorization service.
fakeAuthorizationService.addAuthorizedUser(new User(USER_ID));
assertThat(accessManager.userHasAccess(USER_ID)).isTrue();
```

Using a fake is often the ideal technique when you need to use a test double, but a fake might not exist for an object you need to use in a test, and writing one can be challenging because you need to ensure that it has similar behavior to the real implementation, now and in the future.

Stubbing

Stubbing is the process of giving behavior to a function that otherwise has no behavior on its own—you specify to the function exactly what values to return (that is, you *stub* the return values).

Example 13-8 illustrates stubbing. The `when(...).thenReturn(...)` method calls from the Mockito mocking framework specify the behavior of the `lookupUser()` method.

Example 13-8. Stubbing

```
// Pass in a test double that was created by a mocking
framework.
AccessManager accessManager = new
AccessManager(mockAuthorizationService):
// The user ID shouldn't have access if null is returned.
when(mockAuthorizationService.lookupUser(USER_ID)).thenReturn(nu
ll);
assertThat(accessManager.userHasAccess(USER_ID)).isFalse();
// The user ID should have access if a non-null value is
returned.
when(mockAuthorizationService.lookupUser(USER_ID)).thenReturn(US
ER);
assertThat(accessManager.userHasAccess(USER_ID)).isTrue();
```

Stubbing is typically done through mocking frameworks to reduce boilerplate that would otherwise be needed for manually creating new classes that hardcode return values.

Although stubbing can be a quick and simple technique to apply, it has limitations, which we'll discuss later in this chapter.

Interaction Testing

Interaction testing is a way to validate *how* a function is called without actually calling the implementation of the function. A test should fail if a function isn't called the correct way; for example, if the function isn't called at all, it's called too many times, or it's called with the wrong arguments.

Example 13-9 presents an instance of interaction testing. The `verify(...)` method from the Mockito mocking framework is used to validate that `lookupUser()` is called as expected.

Example 13-9. Interaction testing

```
// Pass in a test double that was created by a mocking
framework.
```

```
AccessManager accessManager = new
AccessManager(mockAuthorizationService);
accessManager.userHasAccess(USER_ID);
// The test will fail if accessManager.userHasAccess(USER_ID)
didn't call
// mockAuthorizationService.lookupUser(USER_ID).
verify(mockAuthorizationService).lookupUser(USER_ID);
```

Similar to stubbing, interaction testing is typically done through mocking frameworks. This reduces boilerplate compared to manually creating new classes that contain code to keep track of how often a function is called and which arguments were passed in.

Interaction testing is sometimes called *mocking*. We avoid this terminology in this chapter because it can be confused with mocking frameworks, which can be used for stubbing as well as for interaction testing.

As discussed later in this chapter, interaction testing is useful in certain situations but should be avoided when possible because overuse can easily result in brittle tests.

Real Implementations

Although test doubles can be invaluable testing tools, our first choice for tests is to use the real implementations of the system under test's dependencies; that is, the same implementations that are used in production code. Tests have higher fidelity when they execute code as it will be executed in production, and using real implementations helps accomplish this.

At Google, the preference for real implementations developed over time as we saw that overuse of mocking frameworks had a tendency to pollute tests with repetitive code that got out of sync with the real implementation and made refactoring difficult. We'll look at this topic in more detail later in this chapter.

Preferring real implementations in tests is known as *classical testing*. There is also a style of testing known as *mockist testing*, in which the preference is to use mocking frameworks instead of real implementations. Even though some people in software industry practice mockist testing (including the creators of the first mocking frameworks), at Google we have found that this style of testing is difficult to scale. It requires engineers to follow strict guidelines when designing the system under test, and the default behavior of most

engineers at Google has been to write code in a way that is more suitable for the classical testing style.

Prefer Realism Over Isolation

Using real implementations for dependencies makes the system under test more realistic given that all code in these real implementations will be executed in the test. In contrast, a test that utilizes test doubles isolates the system under test from its dependencies so that the test does not execute code in the dependencies of the system under test.

We prefer realistic tests because they give more confidence that the system under test is working properly. If unit tests rely too much on test doubles, an engineer might need to run integration tests or manually verify that their feature is working as expected in order to gain this same level of confidence. Carrying out these extra tasks can slow down development and can even allow bugs to slip through if engineers skip these tasks entirely when they are too time consuming to carry out compared to running unit tests.

Replacing all dependencies of a class with test doubles arbitrarily isolates the system under test to the implementation that the author happens to put directly into the class and excludes implementation that happens to be in different classes. However, a good test should be independent of implementation—it should be written in terms of the API being tested rather than in terms of how the implementation is structured.

Using real implementations can cause your test to fail if there is a bug in the real implementation. This is good! You *want* your tests to fail in such cases because it indicates that your code won't work properly in production. Sometimes, a bug in a real implementation can cause a cascade of test failures because other tests that use the real implementation might fail, too. But with good developer tools, such as a Continuous Integration (CI) system, it is usually easy to track down the change that caused the failure.

@DONOTMOCK

At Google, we've seen enough tests that overrely on mocking frameworks to motivate the creation of the `@DoNotMock` annotation in Java, which is available as part of the ErrorProne static analysis tool. This annotation is a way for API owners to declare, "this type should not be mocked because better alternatives exist."

If an engineer attempts to use a mocking framework to create an instance of a class or interface that has been annotated as `@DoNotMock`, as demonstrated in Example 13-10, they will see an error directing them to use a more suitable test strategy, such as a real

implementation or a fake. This annotation is most commonly used for value objects that are simple enough to use as-is, as well as for APIs that have well-engineered fakes available.

Example 13-10. The @DoNotMock Annotation

```
@DoNotMock("Use SimpleQuery.create() instead of mocking.")
public abstract class Query {
  public abstract String getQueryValue();
}
```

Why would an API owner care? In short, it severely constrains the API owner's ability to make changes to their implementation over time. As we'll explore later in the chapter, every time a mocking framework is used for stubbing or interaction testing, it duplicates behavior provided by the API.

When the API owner wants to change their API, they might find that it has been mocked thousands or even tens of thousands of times throughout Google's codebase! These test doubles are very likely to exhibit behavior that violates the API contract of the type being mocked—for instance, returning null for a method that can never return null. Had the tests used the real implementation or a fake, the API owner could make changes to their implementation without first fixing thousands of flawed tests.

How to Decide When to Use a Real Implementation

A real implementation is preferred if it is fast, deterministic, and has simple dependencies. For example, a real implementation should be used for a *value object*. Examples include an amount of money, a date, a geographical address, or a collection class such as a list or a map.

However, for more complex code, using a real implementation often isn't feasible. There might not be an exact answer on when to use a real implementation or a test double given that there are trade-offs to be made, so you need to take the following considerations into account.

EXECUTION TIME

One of the most important qualities of unit tests is that they should be fast—you want to be able to continually run them during development so that you can get quick feedback on whether your code is working (and you also want them to finish quickly when run in a CI system). As a result, a test double can be very useful when the real implementation is slow.

How slow is too slow for a unit test? If a real implementation added one millisecond to the running time of each individual test case, few people

would classify it as slow. But what if it added 10 milliseconds, 100 milliseconds, 1 second, and so on?

There is no exact answer here—it can depend on whether engineers feel a loss in productivity, and how many tests are using the real implementation (one second extra per test case may be reasonable if there are five test cases, but not if there are 500). For borderline situations, it is often simpler to use a real implementation until it becomes too slow to use, at which point the tests can be updated to use a test double, instead.

Parellelization of tests can also help reduce execution time. At Google, our test infrastructure makes it trivial to split up tests in a test suite to be executed across multiple servers. This increases the cost of CPU time, but it can provide a large savings in developer time. We discuss this more in Chapter 18.

Another trade-off to be aware of: using a real implementation can result in increased build times given that the tests need to build the real implementation as well as all of its dependencies. Using a highly scalable build system like Bazel can help because it caches unchanged build artifacts.

DETERMINISM

A test is *deterministic* if, for a given version of the system under test, running the test always results in the same outcome; that is., the test either always passes or always fails. In contrast, a test is *nondeterministic* if its outcome can change, even if the system under test remains unchanged.

Nondeterminism in tests can lead to flakiness—tests can occasionally fail even when there are no changes to the system under test. As discussed in Chapter 11, flakiness harms the health of a test suite if developers start to distrust the results of the test and ignore failures. If use of a real implementation rarely causes flakiness, it might not warrant a response, because there is little disruption to engineers. But if flakiness happens often, it might be time to replace a real implementation with a test double because doing so will improve the fidelity of the test.

A real implementation can be much more complex compared to a test double, which increases the likelihood that it will be nondeterministic. For example, a real implementation that utilizes multithreading might occasionally cause a test to fail if the output of the system under test differs depending on the order in which the threads are executed.

A common cause of nondeterminism is code that is not hermetic; that is, it has dependencies on external services that are outside the control of a test. For example, a test that tries to read the contents of a web page from an HTTP server might fail if the server is overloaded or if the web page contents change. Instead, a test double should be used to prevent the test from depending on an external server. If using a test double is not feasible, another option is to use a hermetic instance of a server, which has its life cycle controlled by the test. Hermetic instances are discussed in more detail in the next chapter.

Another example of nondeterminism is code that relies on the system clock given that the output of the system under test can differ depending on the current time. Instead of relying on the system clock, a test can use a test double that hardcodes a specific time.

DEPENDENCY CONSTRUCTION

When using a real implementation, you need to construct all of its dependencies. For example, an object needs its entire dependency tree to be constructed: all objects that it depends on, all objects that these dependent objects depend on, and so on. A test double often has no dependencies, so constructing a test double can be much simpler compared to constructing a real implementation.

As an extreme example, imagine trying to create the object in the code snippet that follows in a test. It would be time consuming to determine how to construct each individual object. Tests will also require constant maintenance because they need to be updated when the signature of these objects' constructors is modified:

```
Foo foo = new Foo(new A(new B(new C()), new D()), new E(),
..., new Z());
```

It can be tempting to instead use a test double because constructing one can be trivial. For example, this is all it takes to construct a test double when using the Mockito mocking framework:

```
@Mock Foo mockFoo;
```

Although creating this test double is much simpler, there are significant benefits to using the real implementation, as discussed earlier in this section. There are also often significant downsides to overusing test doubles in this

way, which we look at later in this chapter. So, a trade-off needs to be made when considering whether to use a real implementation or a test double.

Rather than manually constructing the object in tests, the ideal solution is to use the same object construction code that is used in the production code, such as a factory method or automated dependency injection. To support the use case for tests, the object construction code needs to be flexible enough to be able to use test doubles rather than hardcoding the implementations that will be used for production.

Faking

If using a real implementation is not feasible within a test, the best option is often to use a fake in its place. A fake is preferred over other test double techniques because it behaves similarly to the real implementation: the system under test shouldn't even be able to tell whether it is interacting with a real implementation or a fake. Example 13-11 illustrates a fake file system.

Example 13-11. A fake file system

```
// This fake implements the FileSystem interface. This interface
is also
// used by the real implementation.
public class FakeFileSystem implements FileSystem {
  // Stores a map of file name to file contents. The files are
stored in
  // memory instead of on disk since tests shouldn't need to do
disk I/O.
  private Map<String, String> files = new HashMap<>();
  @Override
  public void writeFile(String fileName, String contents) {
    // Add the file name and contents to the map.
    files.add(fileName, contents);
  }
  @Override
  public String readFile(String fileName) {
    String contents = files.get(fileName);
    // The real implementation will throw this exception if the
    // file isn't found, so the fake must throw it too.
    if (contents == null) { throw new
FileNotFoundException(fileName); }
    return contents;
  }
}
```

Why Are Fakes Important?

Fakes can be a powerful tool for testing: they execute quickly and allow you to effectively test your code without the drawbacks of using real implementations.

A single fake has the power to radically improve the testing experience of an API. If you scale that to a large number of fakes for all sorts of APIs, fakes can provide an enormous boost to engineering velocity across a software organization.

At the other end of the spectrum, in a software organization where fakes are rare, velocity will be slower because engineers can end up struggling with using real implementations that lead to slow and flaky tests. Or engineers might resort to other test double techniques such as stubbing or interaction testing, which as we'll examine later in this chapter, can result in tests that are unclear, brittle, and less effective.

When Should Fakes Be Written?

A fake requires more effort and more domain experience to create because it needs to behave similarly to the real implementation. A fake also requires maintenance: whenever the behavior of the real implementation changes, the fake must also be updated to match this behavior. Because of this, the team that owns the real implementation should write and maintain a fake.

If a team is considering writing a fake, a trade-off needs to be made on whether the productivity improvements that will result from the use of the fake outweigh the costs of writing and maintaining it. If there are only a handful of users, it might not be worth their time, whereas if there are hundreds of users, it can result in an obvious productivity improvement.

To reduce the number of fakes that need to be maintained, a fake should typically be created only at the root of the code that isn't feasible for use in tests. For example, if a database can't be used in tests, a fake should exist for the database API itself rather than for each class that calls the database API.

Maintaining a fake can be burdensome if its implementation needs to be duplicated across programming languages, such as for a service that has client libraries that allow the service to be invoked from different languages. One solution for this case is to create a single fake service implementation and have tests configure the client libraries to send requests to this fake service. This approach is more heavyweight compared to having the fake

written entirely in memory because it requires the test to communicate across processes. However, it can be a reasonable trade-off to make, as long as the tests can still execute quickly.

The Fidelity of Fakes

Perhaps the most important concept surrounding the creation of fakes is *fidelity; in other words*, how closely the behavior of a fake matches the behavior of the real implementation. If the behavior of a fake doesn't match the behavior of the real implementation, a test using that fake is not useful—a test might pass when the fake is used, but this same code path might not work properly in the real implementation.

Perfect fidelity is not always feasible. After all, the fake was necessary because the real implementation wasn't suitable in one way or another. For example, a fake database would usually not have fidelity to a real database in terms of hard-drive storage, because the fake would store everything in memory.

Primarily, however, a fake should maintain fidelity to the API contracts of the real implementation. For any given input to an API, a fake should return the same output and perform the same state changes of its corresponding real implementation. For example, for a real implementation of `database.save(itemId)`, if an item is successfully saved when its ID does not yet exist but an error is produced when the ID already exists, the fake must conform to this same behavior.

One way to think about this is that the fake must have perfect fidelity to the real implementation, but *only from the perspective of the test*. For example, a fake for a hashing API doesn't need to guarantee that the hash value for a given input is exactly the same as the hash value that is generated by the real implementation—tests likely don't care about the specific hash value, only that the hash value is unique for a given input. If the contract of the hashing API doesn't make guarantees of what specific hash values will be returned, the fake is still conforming to the contract even if it doesn't have perfect fidelity to the real implementation.

Other examples where perfect fidelity typically might not be useful for fakes include latency and resource consumption. However, you cannot use a fake if you need to explicitly test for these constraints (e.g., a performance test that verifies the latency of a function call), so you would need to resort to other mechanisms such as by using a real implementation instead of a fake.

A fake might not need to have 100% of the functionality of its corresponding real implementation, especially if such behavior is not needed by most tests (e.g., error handling code for rare edge cases). It is best to have the fake fail fast in this case; for example, raise an error if an unsupported code path is executed. This failure communicates to the engineer that the fake is not appropriate in this situation.

Fakes Should Be Tested

A fake must have its *own* tests to ensure that it conforms to the API of its corresponding real implementation. A fake without tests might initially provide realistic behavior, but without tests, this behavior can diverge over time as the real implementation evolves.

One approach to writing tests for fakes involves writing tests against the API's public interface and running those tests against both the real implementation and the fake (these are known as *contract tests*). The tests that run against the real implementation will likely be slower, but their downside is minimized because they need to be run only by the owners of the fake.

What to Do If a Fake Is Not Available

If a fake is not available, first ask the owners of the API to create one. The owners might not be familiar with the concept of fakes, or they might not realize the benefit they provide to users of an API.

If the owners of an API are unwilling or unable to create a fake, you might be able to write your own. One way to do this is to wrap all calls to the API in a single class and then create a fake version of the class that doesn't talk to the API. Doing this can also be much simpler than creating a fake for the entire API because often you'll need to use only a subset of the API's behavior anyway. At Google, some teams have even contributed their fake to the owners of the API, which has allowed other teams to benefit from the fake.

Finally, you could decide to settle on using a real implementation (and deal with the trade-offs of real implementations that are mentioned earlier in this chapter), or resort to other test double techniques (and deal with the trade-offs that we will mention later in this chapter).

In some cases, you can think of a fake as an optimization: if tests are too slow using a real implementation, you can create a fake to make them run faster. But if the speedup from a fake doesn't outweigh the work it would take to

create and maintain the fake, it would be better to stick with using the real implementation.

Stubbing

As discussed earlier in this chapter, stubbing is a way for a test to hardcode behavior for a function that otherwise has no behavior on its own. It is often a quick and easy way to replace a real implementation in a test. For example, the code in Example 13-12 uses stubbing to simulate the response from a credit card server:

Example 13-12. Using stubbing to simulate responses

```
@Test public void getTransactionCount() {
  transactionCounter = new
TransactionCounter(mockCreditCardServer);
  // Use stubbing to return three transactions.
  when(mockCreditCardServer.getTransactions()).thenReturn(
      newList(TRANSACTION_1, TRANSACTION_2, TRANSACTION_3));

assertThat(transactionCounter.getTransactionCount()).isEqualTo(3
);
}
```

The Dangers of Overusing Stubbing

Because stubbing is so easy to apply in tests, it can be tempting to use this technique anytime it's not trivial to use a real implementation. However, overuse of stubbing can result in major losses in productivity for engineers that need to maintain these tests:

TESTS BECOME UNCLEAR

Stubbing involves writing extra code to define the behavior of the functions being stubbed. Having this extra code detracts from the intent of the test, and this code can be difficult to understand if you're not familiar with the implementation of the system under test.

A key sign that stubbing isn't appropriate for a test is if you find yourself mentally stepping through the system under test in order to understand why certain functions in the test are stubbed.

TESTS BECOME BRITTLE

Stubbing leaks implementation details of your code into your test. When implementation details in your production code change, you'll need to update

your tests to reflect these changes. Ideally, a good test should need to change only if user-facing behavior of an API changes; it should remain unaffected by changes to the API's implementation.

TESTS BECOME LESS EFFECTIVE

With stubbing, there is no way to ensure the function being stubbed behaves like the real implementation, such as in a statement like that shown in the following snippet that hardcodes part of the contract of the `add()` method (*"If 1 and 2 are passed in, 3 will be returned"*):

```
  when(stubCalculator.add(1, 2)).thenReturn(3);
```

Stubbing is a poor choice if the system under test depends on the real implementation's contract, because you will be forced to duplicate the details of the contract, and there is no way to guarantee that the contract is correct (i.e., that the stubbed function has fidelity to the real implementation).

Additionally, with stubbing there is no way to store state, which can make it difficult to test certain aspects of your code. For example, if you call `database.save(item)` on either a real implementation or a fake, you might be able to retrieve the item by calling `database.get(item.id())` given that both of these calls are accessing internal state, but with stubbing there is no way to do this.

AN EXAMPLE OF OVERUSING STUBBING

Example 13-13 illustrates a test that overuses stubbing.

Example 13-13. Overuse of stubbing

```
@Test public void creditCardIsCharged() {
  // Pass in test doubles that were created by a mocking
framework.
  paymentProcessor =
      new PaymentProcessor(mockCreditCardServer,
mockTransactionProcessor);
  // Set up stubbing for these test doubles.

when(mockCreditCardServer.isServerAvailable()).thenReturn(true);

when(mockTransactionProcessor.beginTransaction()).thenReturn(tra
nsaction);

when(mockCreditCardServer.initTransaction(transaction)).thenRetu
rn(true);
  when(mockCreditCardServer.pay(transaction, creditCard, 500))
```

```
      .thenReturn(false);

when(mockTransactionProcessor.endTransaction()).thenReturn(true)
;
  // Call the system under test.
  paymentProcessor.processPayment(creditCard,
Money.dollars(500));
  // There is no way to tell if the pay() method actually
carried out the
  // transaction, so the only thing the test can do is verify
that the
  // pay() method was called.
  verify(mockCreditCardServer).pay(transaction, creditCard,
500);
}
```

Example 13-14 rewrites the same test but avoids using stubbing. Notice how the test is shorter, and that implementation details (such as how the transaction processor is used) are not exposed in the test. No special setup is needed because the credit card server knows how to behave.

Example 13-14. Refactoring a test to avoid stubbing

```
@Test public void creditCardIsCharged() {
  paymentProcessor =
      new PaymentProcessor(creditCardServer,
transactionProcessor);
  // Call the system under test.
  paymentProcessor.processPayment(creditCard,
Money.dollars(500));
  // Query the credit card server state to see if the payment
went through.
  assertThat(creditCardServer.getMostRecentCharge(creditCard))
      .isEqualTo(500);
}
```

We obviously don't want such a test to talk to an external credit card server, so a fake credit card server would be more suitable. If a fake isn't available, another option is to use a real implementation that talks to a hermetic credit card server, although this will increase the execution time of the tests. (We explore hermetic servers in the next chapter.)

When Is Stubbing Appropriate?

Rather than a catch-all replacement for a real implementation, stubbing is appropriate when you need a function to return a specific value to get the system under test into a certain state, such as Example 13-12 that requires the system under test to return a non-empty list of transactions. Because a function's behavior is defined inline in the test, stubbing can simulate a wide

variety of return values or errors that might not be possible to trigger from a real implementation or a fake.

To ensure its purpose is clear, each stubbed function should have a direct relationship with the test's assertions. As a result, a test typically should stub out a small number of functions because stubbing out many functions can lead to tests that are less clear. A test that requires many functions to be stubbed can be a sign that stubbing is being overused, or that the system under test is too complex and should be refactored.

Note that even when stubbing is appropriate, real implementations or fakes are still preferred because they don't expose implementation details and they give you more guarantees about the correctness of the code compared to stubbing. But stubbing can be a reasonable technique to use, as long as its usage is constrained so that tests don't become overly complex.

Interaction Testing

As discussed earlier in this chapter, interaction testing is a way to validate how a function is called without actually calling the implementation of the function.

Mocking frameworks make it easy to perform interaction testing. However, to keep tests useful, readable, and resilient to change, it's important to perform interaction testing only when necessary.

Prefer State Testing Over Interaction Testing

In contrast to interaction testing, it is preferred to test code through *state testing*.

With state testing, you call the system under test and validate that either the correct value was returned or that some other state in the system under test was properly changed. Example 13-15 presents an example of state testing.

Example 13-15. State testing

```
@Test public void sortNumbers() {
  NumberSorter numberSorter = new NumberSorter(quicksort,
bubbleSort);
  // Call the system under test.
  List sortedList = numberSorter.sortNumbers(newList(3, 1, 2));
  // Validate that the returned list is sorted. It doesn't
matter which
```

```
  // sorting algorithm is used, as long as the right result was
returned.
  assertThat(sortedList).isEqualTo(newList(1, 2, 3));
}
```

Example 13-16 illustrates a similar test scenario but instead uses interaction testing. Note how it's impossible for this test to determine that the numbers are actually sorted, because the test doubles don't know how to sort the numbers—all it can tell you is that the system under test tried to sort the numbers.

Example 13-16. Interaction testing

```
@Test public void sortNumbers_quicksortIsUsed() {
  // Pass in test doubles that were created by a mocking
framework.
  NumberSorter numberSorter =
      new NumberSorter(mockQuicksort, mockBubbleSort);

  // Call the system under test.
  numberSorter.sortNumbers(newList(3, 1, 2));

  // Validate that numberSorter.sortNumbers() used quicksort.
The test
  // will fail if mockQuicksort.sort() is never called (e.g., if
  // mockBubbleSort is used) or if it's called with the wrong
arguments.
  verify(mockQuicksort).sort(newList(3, 1, 2));
}
```

At Google, we've found that emphasizing state testing is more scalable; it reduces test brittleness, making it easier to change and maintain code over time.

The primary issue with interaction testing is that it can't tell you that the system under test is working properly; it can only validate that certain functions are called as expected. It requires you to make an assumption about the behavior of the code; for example, "*If* `database.save(item)` *is called, we assume the item will be saved to the database.*" State testing is preferred because it actually validates this assumption (such as by saving an item to a database and then querying the database to validate that the item exists),

Another downside of interaction testing is that it utilizes implementation details of the system under test—to validate that a function was called, you are exposing to the test that the system under test calls this function. Similar to stubbing, this extra code makes tests brittle because it leaks implementation details of your production code into tests. Some people at Google jokingly refer to tests that overuse interaction testing as *change-*

detector tests because they fail in response to any change to the production code, even if the behavior of the system under test remains unchanged.

When Is Interaction Testing Appropriate?

There are some cases for which interaction testing is warranted:

- You cannot perform state testing because you are unable to use a real implementation or a fake (e.g., if the real implementation is too slow and no fake exists). As a fallback, you can perform interaction testing to validate that certain functions are called. Although not ideal, this does provide some basic level of confidence that the system under test is working as expected.
- Differences in the number or order of calls to a function would cause undesired behavior. Interaction testing is useful because it could be difficult to validate this behavior with state testing. For example, if you expect a caching feature to reduce the number of calls to a database, you can verify that the database object is not accessed more times than expected. Using Mockito, the code might look similar to this:

```
verify(databaseReader, atMostOnce()).selectRecords().
```

Interaction testing is not a complete replacement for state testing. If you are not able to perform state testing in a unit test, strongly consider supplementing your test suite with larger-scoped tests that do perform state testing. For instance, if you have a unit test that validates usage of a database through interaction testing, consider adding an integration test that can perform state testing against a real database. Larger-scope testing is an important strategy for risk mitigation, and we discuss it in the next chapter.

Best Practices for Interaction Testing

When performing interaction testing, following these practices can reduce some of the impact of the aforementioned downsides.

PREFER TO PERFORM INTERACTION TESTING ONLY FOR STATE-CHANGING FUNCTIONS

When a system under test calls a function on a dependency, that call falls into one of two categories:

State-changing

Functions that have side effects on the world outside the system under test. Examples: `sendEmail()`, `saveRecord()`, `logAccess()`.

Non-state-changing

Functions that don't have side effects; they return information about the world outside the system under test and don't modify anything. Examples: `getUser()`, `findResults()`, `readFile()`.

In general, you should perform interaction testing only for functions that are state-changing. Performing interaction testing for non-state-changing functions is usually redundant given that the system under test will use the return value of the function to do other work that you can assert. The interaction itself is not an important detail for correctness, because it has no side effects.

Performing interacting testing for non-state-changing functions makes your test brittle because you'll need to update the test any time the pattern of interactions changes. It also makes the test less readable given that the additional assertions make it more difficult to determine which assertions are important for ensuring correctness of the code. By contrast, state-changing interactions represent something useful that your code is doing to change state somewhere else.

Example 13-17 demonstrates interaction testing on both state-changing and non-state-changing functions.

Example 13-17. State-changing and non-state-changing interactions

```
@Test public void grantUserPermission() {
  UserAuthorizer userAuthorizer =
      new UserAuthorizer(mockUserService,
mockPermissionDatabase);

when(mockPermissionService.getPermission(FAKE_USER)).thenReturn(
EMPTY);
  // Call the system under test.
  userAuthorizer.grantPermission(USER_ACCESS);
  // addPermission() is state-changing, so it is reasonable to
perform
  // interaction testing to validate that it was called.
  verify(mockPermissionDatabase).addPermission(FAKE_USER,
USER_ACCESS);
  // getPermission() is non-state-changing, so this line of code
isn't
  // needed. One clue that interaction testing may not be
needed:
  // getPermission() was already stubbed earlier in this test.
```

```
  verify(mockPermissionDatabase).getPermission(FAKE_USER);
}
```

AVOID OVERSPECIFICATION

In Chapter 12, we discuss why it is useful to test behaviors rather than methods. This means that a test method should focus on verifying one behavior of a method or class rather than trying to verify multiple behaviors in a single test.

When performing interaction testing, we should aim to apply the same principle by avoiding overspecifying which functions and arguments are validated. This leads to tests that are more concise and clearer. It also leads to tests that are resilient to changes made to behaviors that are outside the scope of each test, so fewer tests will fail if a change is made to a way a function is called.

Example 13-18 illustrates interaction testing with overspecification. The intention of the test is to validate that the user's name is included in the greeting prompt, but the test will fail if unrelated behavior is changed:

Example 13-18. Overspecified interaction tests

```
@Test public void displayGreeting_renderUserName() {
  when(mockUserService.getUserName()).thenReturn("Fake User");
  userGreeter.displayGreeting(); // Call the system under test.
  // The test will fail if any of the arguments to setText() are
changed.
  verify(userPrompt).setText("Fake User", "Good morning!",
"Version 2.1");
  // The test will fail if setIcon() is not called, even though
this
  // behavior is incidental to the test since it is not related
to
  // validating the user name.
  verify(userPrompt).setIcon(IMAGE_SUNSHINE);
}
```

Example 13-19 illustrates interaction testing with more care in specifying relevant arguments and functions. The behaviors being tested are split into separate tests, and each test validates the minimum amount necessary for ensuring the behavior it is testing is correct.

Example 13-19. Well-specified interaction tests

```
@Test public void displayGreeting_renderUserName() {
  when(mockUserService.getUserName()).thenReturn("Fake User");
  userGreeter.displayGreeting(); // Call the system under test.
  verify(userPrompter).setText(eq("Fake User"), any(), any());
}
```

```
@Test public void
displayGreeting_timeIsMorning_useMorningSettings() {
  setTimeOfDay(TIME_MORNING);
  userGreeter.displayGreeting(); // Call the system under test.
  verify(userPrompt).setText(any(), eq("Good morning!"), any());
  verify(userPrompt).setIcon(IMAGE_SUNSHINE);
}
```

Conclusion

We've learned that test doubles are crucial to engineering velocity because they can help comprehensively test your code and ensure that your tests run fast. On the other hand, misusing them can be a major drain on productivity because they can lead to tests that are unclear, brittle, and less effective. This is why it's important for engineers to understand the best practices for how to effectively apply test doubles.

Although test doubles are great for working around dependencies that are difficult to use in tests, if you want to maximize confidence in your code, at some point you still want to exercise these dependencies in tests. The next chapter will cover larger-scope testing, for which these dependencies are used regardless of their suitability for unit tests; for example, even if they are slow or nondeterministic.

There is often no exact answer regarding whether to use a real implementation or a test double, or which test double technique to use. An engineer might need to make some trade-offs when deciding the proper approach for their use case.

TL;DRs

- A real implementation should be preferred over a test double.
- A fake is often the ideal solution if a real implementation can't be used in a test.
- Overuse of stubbing leads to tests that are unclear and brittle.
- Interaction testing should be avoided when possible: it leads to tests that are brittle because it exposes implementation details of the system under test.

Chapter 14. Larger Testing

Written by Joseph Graves

Edited by Tom Manshreck

In previous chapters, we have recounted how a testing culture was established at Google and how small unit tests became a fundamental part of the developer workflow. But what about other kinds of tests? It turns out that Google does indeed use many larger tests and these comprise a significant part of the risk mitigation strategy necessary for healthy software engineering. But these tests present additional challenges to ensure that they are valuable assets and not resource sinks. In this chapter, we'll discuss what we mean by "larger tests," when we execute them, and best practices for keeping them effective.

What are Larger Tests?

As mentioned previously, Google has specific notions of test size. Small tests are restricted to one thread, one process, one machine. Larger tests do not have the same restrictions. But Google also has notions of test *scope*. A unit test necessarily is of smaller scope than an integration test. And the largest-scoped tests (sometimes called end-to-end or system tests) typically involve several real dependencies and fewer test doubles.

Larger tests are many things that small tests are not. They are not bound by the same constraints; thus, they can exhibit the following characteristics:

- They may be slow. Our large tests have a default timeout of 15 minutes or 1 hour, but we also have tests that run for multiple hours or even days.
- They may be nonhermetic. Large tests may share resources with other tests and traffic.
- They may be nondeterministic. If a large test is nonhermetic it is almost impossible to guarantee determinism: other tests or user state may interfere with it.

So why have larger tests? Reflect back on your coding process. How do you confirm that the programs you write actually work? You might be writing

and running unit tests as you go, but do you find yourself running the actual binary and trying it out yourself? And when you share this code with others, how do they test it? By running your unit tests, or by trying it out themselves?

Also, how do you know that your code continues to work during upgrades? Suppose that you have a site that uses the Google Maps API and there's a new API version. Your unit tests likely won't help you to know whether there are any compatibility issues. You'd probably run it and try it out to see whether anything broke.

Unit tests can give you confidence about individual functions, objects, and modules, but large tests provide more confidence that the overall system works as intended. And having actual automated tests scales in ways that manual testing does not.

Fidelity

The primary reason larger tests exist is to address *fidelity*. Fidelity is the property by which a test is reflective of the real behavior of the system under test (SUT).

One way of envisioning fidelity is in terms of the environment. As Figure 14-1 illustrates, unit tests bundle a test and a small portion of code together as a runnable unit, which ensures the code is tested but is very different from how production code runs. Production itself is, naturally, the environment of highest fidelity in testing. There is also a spectrum of interim options. A key for larger tests is to find the proper fit, because increasing fidelity also comes with increasing costs and (in the case of production) increasing risk of failure.

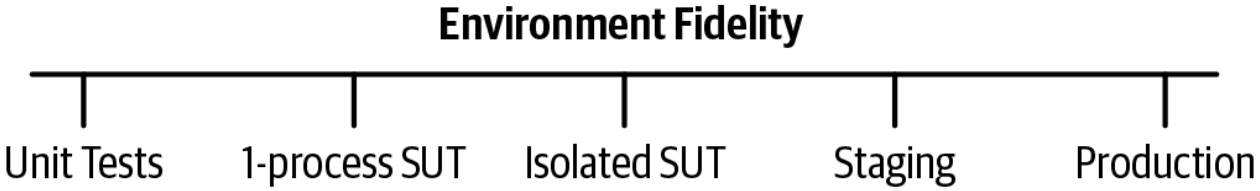

Figure 14-1. Scale of increasing fidelity

Tests can also be measured in terms of how faithful the test content is to reality. Many hand-crafted large tests are dismissed by engineers if the test data itself looks unrealistic. Test data copied from production is much more faithful to reality (having been captured that way) but a big challenge is how to create realistic test traffic *before* launching the new code. This is

particularly a problem in artificial intelligence (AI), for which the "seed" data often suffers from intrinsic bias. And, because most data for unit tests is handcrafted, it covers a narrow range of cases and tends to conform to the biases of the author. The uncovered scenarios missed by the data represent a fidelity gap in the tests.

Common Gaps in Unit Tests

Larger tests might also be necessary where smaller tests fail. The subsections that follow present some particular areas where unit tests do not provide good risk mitigation coverage.

UNFAITHFUL DOUBLES

A single unit test typically covers one class or module. Test doubles (as discussed in Chapter 13) are frequently used to eliminate heavyweight or hard-to-test dependencies. But when those dependencies are replaced, it becomes possible that the replacement and the doubled thing do not agree.

Almost all unit tests at Google are written by the same engineer who is writing the unit under test. When those unit tests need doubles and when the doubles used are mocks, it is the engineer writing the unit test defining the mock and its intended behavior. But that engineer usually did *not* write the thing being mocked and can be misinformed about its actual behavior. The relationship between the unit under test and a given peer is a behavioral contract, and if the engineer is mistaken about the actual behavior, the understanding of the contract is invalid.

Moreover, mocks become stale. If this mock-based unit test is not visible to the author of the real implementation and the real implementation changes, there is no signal that the test (and the code being tested) should be updated to keep up with the changes.

Note that, as mentioned in Chapter 13, if teams provide fakes for their own services, this concern is mostly alleviated.

CONFIGURATION ISSUES

Unit tests cover code within a given binary. But that binary is typically not completely self-sufficient in terms of how it is executed. Usually a binary has some kind of deployment configuration or starter script. Additionally, real end-user-serving production instances have their own configuration files or configuration databases.

If there are issues with these files or the compatibility between the state defined by these stores and the binary in question, these can lead to major user issues. Unit tests alone cannot verify this compatibility.[1] Incidentally, this is a good reason to ensure that your configuration is in version control as well as your code, because then changes to configuration can be identified as the source of bugs as opposed to introducing random external flakiness and can be built in to large tests.

At Google, configuration changes are the number one reason for our major outages. This is an area in which we have underperformed and has led to some of our most embarrassing bugs. For example, there was a global Google outage back in 2013 due to a bad network configuration push that was never tested. Configurations tend to be written in configuration languages, not production code languages. They also often have faster production rollout cycles than binaries and they can be more difficult to test. All of these lead to a higher likelihood of failure. But at least in this case (and others) configuration was version controlled and we could quickly identify the culprit and mitigate the issue.

ISSUES THAT ARISE UNDER LOAD

At Google, unit tests are intended to be small and fast because they need to fit into our standard test execution infrastructure and also be run many times as part of a frictionless developer workflow. But performance, load, and stress testing often require sending large volumes of traffic to a given binary. These volumes become difficult to test in the model of a typical unit test. And our large volumes are big, often thousands or millions of queries per second (in the case of ads real-time bidding[2])!

UNANTICIPATED BEHAVIORS, INPUTS, AND SIDE EFFECTS

Unit tests are limited by the imagination of the engineer writing them. That is, they can only test for anticipated behaviors and inputs. However, issues that users find with a product are mostly unanticipated (otherwise it would be unlikely that they would make it to end users as issues). This fact suggests that different test techniques are needed to test for unanticipated behaviors.

Hyrum's Law[3] is an important consideration here: even if we could test 100% for conformance to a strict specified contract, the effective user contract applies to all visible behaviors, not just a stated contract. It is unlikely that unit tests alone test for all visible behaviors that are not specified in the public API.

EMERGENT BEHAVIORS AND THE "VACUUM EFFECT"

Unit tests are limited to the scope that they cover (especially with the widespread use of test doubles), so if behavior changes in areas outside of this scope it cannot be detected. And because unit tests are designed to be fast and reliable, they deliberately eliminate the chaos of real dependencies, network, and data. A unit test is like a problem in theoretical physics: ensconced in a vacuum, neatly hidden from the mess of the real world, which is great for speed and reliability but misses certain defect categories.

Why Not Have Larger Tests?

In earlier chapters, we discussed many of the properties of a developer-friendly test. In particular it needs to be as follows:

Reliable

: It must not be flaky and it must provide a useful pass/fail signal.

Fast

: It needs to be fast enough to not interrupt the developer workflow.

Scalable

: Google needs to be able to run all such useful affected tests efficiently for presubmits and for postsubmits.

Good unit tests exhibit all of these properties. Larger tests often violate all of these constraints. For example, larger tests are often flakier because they use more infrastructure than does a small unit test. They are also often much slower, both to set up as well as to run. And they have trouble scaling because of the resource and time requirements but often also because they are not isolated—these tests can collide with one another.

Additionally, larger tests present two other challenges. First, there is a challenge of ownership. A unit test is clearly owned by the engineer (and team) who owns the unit. A larger test spans multiple units and thus can span multiple owners. This presents a long-term ownership challenge: who is responsible for maintaining the test and who is responsible for diagnosing issues when the test breaks? Without clear ownership a test rots.

The second challenge for larger tests is one of standardization (or the lack thereof). Unlike unit tests, larger tests suffer a lack of standardization in terms of the infrastructure and process by which these tests are written, run, and debugged. The approach to larger tests is a product of a system's

architectural decisions, thus introducing variance in the type of tests required. For example, the way we build and run A-B diff regression tests in Google Ads is completely different from the way such tests are built and run in Search backends, which is different again from Drive. They use different platforms, different languages, different infrastructures, different libraries, and competing testing frameworks.

This lack of standardization has a significant impact. Because larger tests have so many ways of being run, they often are skipped during large-scale changes. (See Chapter 22) The infrastructure does not have a standard way to run those tests, and asking the people executing LSCs to know the local particulars for testing on every team doesn't scale. Because larger tests differ in implementation from team to team, tests that actually test the integration between those teams require unifying incompatible infrastructures. And because of this lack of standardization, we cannot teach a single approach to Nooglers (new Googlers) or even more experienced engineers, which both perpetuates the situation and also leads to a lack of understanding about the motivations of such tests.

Larger Tests at Google

When we discussed the history of testing at Google earlier (see Chapter 11) we mentioned how Google Web Server (GWS) mandated automated tests in 2003 and how this was a watershed moment. However, we actually had automated tests in use before this point, but a common practice was using automated large and enormous tests. For example, AdWords created an end-to-end test back in 2001 to validate product scenarios. Similarly, in 2002, Search wrote a similar "regression test" for its indexing code, and AdSense (which had not even publicly launched yet) created its variation on the AdWords test.

Other "larger" testing patterns also existed circa 2002. The Google search frontend relied heavily on manual QA—manual versions of end-to-end test scenarios. And GMail got its version of a "local demo" environment—a script to bring up an end-to-end GMail environment locally with some generated test users and mail data for local manual testing.

When C/J Build (our first continuous build framework) launched, it did not distinguish between unit tests and other tests, but there were two critical developments that led to a split. First, Google focused on unit tests because we wanted to encourage the testing pyramid and to ensure the vast majority

of written tests were unit tests. Second, when TAP replaced C/J Build as our formal continuous build system, it was only able to do so for tests that met TAP's eligibility requirements: hermetic tests buildable at a single change that could run on our build/test cluster within a maximum time limit. Although most unit tests satisfied this requirement, larger tests mostly did not. However, this did not stop the need for other kinds of tests, and they have continued to fill the coverage gaps. C/J Build even stuck around for years specifically to handle these kinds of tests until newer systems replaced it.

Larger Tests and Time

Throughout this book, we have looked at the influence of time on software engineering, because Google has built software running for more than 20 years. How are larger tests influenced by the time dimension? We know that certain activities make more sense the longer the expected lifespan of code, and testing of various forms is an activity that makes sense at all levels, but the test types that are appropriate change over the expected lifetime of code.

As we pointed out before, unit tests begin to make sense for software with an expected lifespan from hours on up. At the minutes level (for small scripts), manual testing is most common, and the SUT usually runs locally, but the local demo likely *is* production, especially for one-off scripts, demos, or experiments. At longer lifespans, manual testing continues to exist but the SUTs usually diverge because the production instance is often cloud hosted instead of locally hosted.

The remaining larger tests all provide value for longer-lived software, but the main concern becomes the maintainability of such tests as time increases.

Incidentally, this time impact might be one reason for the development of the "ice cream cone" testing antipattern, as mentioned in the Chapter 11 and shown again in Figure 14-1.

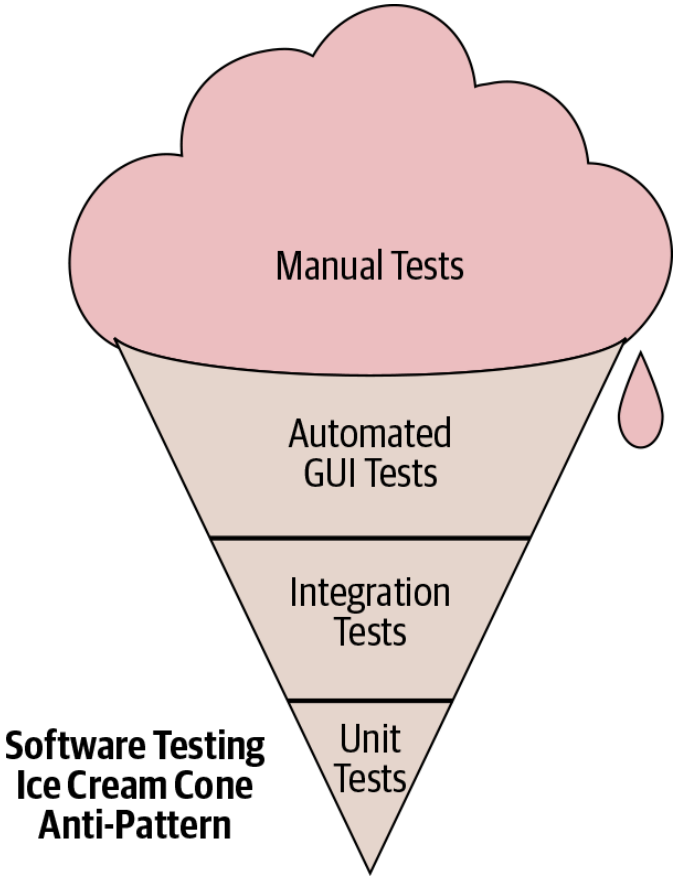

Figure 14-2. The ice cream cone testing antipattern

When development starts with manual testing (when engineers think that code is meant to last only for minutes), those manual tests accumulate and dominate the initial overall testing portfolio. For example, it's pretty typical to hack on a script or an app and test it out by running it, and then to continue to add features to it but continue to test it out by running it manually. This prototype eventually becomes functional and is shared with others but no automated tests actually exist for it.

Even worse, if the code is difficult to unit test (because of the way it was implemented in the first place) the only automated tests that can be written are end-to-end ones, and we have inadvertently created "legacy code" within days.

It is *critical* for longer-term health to move toward the test pyramid within the first few days of development, by building out unit tests and then to top it off after that point by introducing automated integration tests and moving away from manual end-to-end tests. We succeeded by making unit tests a requirement for submission, but covering the gap between unit tests and manual tests is necessary for long-term health.

Larger Tests at Google Scale

It would seem that larger tests should be more necessary and more appropriate at larger scales of software, but even though this is so, the complexity of authoring, running, maintaining, and debugging these tests increases with the growth in scale, even more so than with unit tests.

In a system composed of microservices or separate servers, the pattern of interconnections looks like a graph: let the number of nodes in that graph be our *N*. Every time a new node is added to this graph there is a multiplicative effect on the number of distinct execution paths through it.

Figure 14-3 depicts an imagined SUT: this system consists of a social network with users, a social graph, a stream of posts, and some ads mixed in. The ads are created by advertisers and served in the context of the social stream. This SUT alone consists of two groups of users, two UIs, three databases, an indexing pipeline, and six servers. There are 14 edges enumerated in the graph. Testing all of the end-to-end possibilities is already difficult. Imagine if we add more services, pipelines, and databases to this mix: photos and images, machine learning photo analysis, and so on?

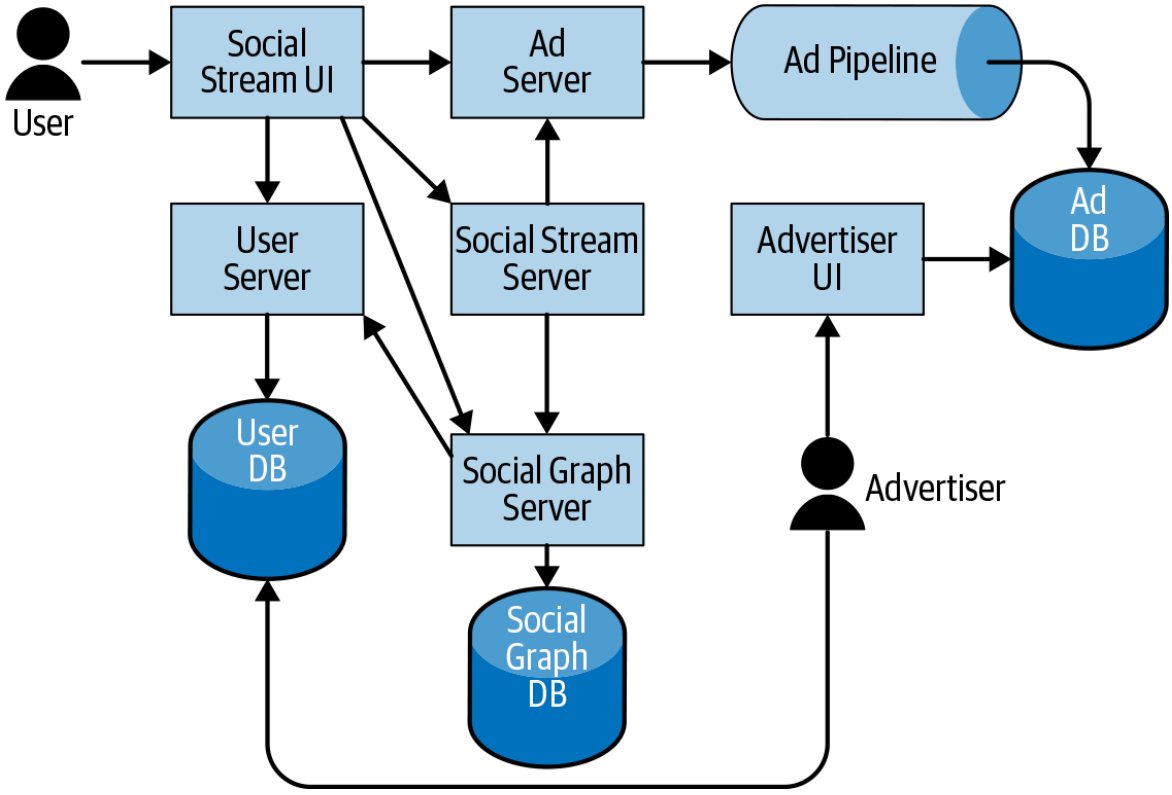

Figure 14-3. Example of a fairly small SUT: a social network with advertising

The rate of distinct scenarios to test in an end-to-end way can grow exponentially or combinitorically depending on the structure of the system

under test, and that growth does not scale. Therefore, as the system grows, we must find alternative larger testing strategies to keep things manageable.

However, the value of such tests also increases because of the decisions that were necessary to achieve this scale. This is an impact of fidelity: as we move toward larger-*N* layers of software, if the service doubles are lower fidelity (1-epsilon), the chance of bugs when putting it all together is exponential in *N*. Looking at this example SUT again, if we replace the user server and ad server with doubles and those doubles are low fidelity (e.g., 10% inaccurate), the likelihood of a bug is 99% ($1 - (0.1 * 0.1)$). And that's just with two low-fidelity doubles.

Therefore, it becomes critical to implement larger tests in ways that work well at this scale but maintain reasonably high fidelity.

TIP: "THE SMALLEST POSSIBLE TEST"

Even for integration tests, smaller is better—a handful of large tests is preferable to an enormous one. And, because the scope of a test is often coupled to the scope of the SUT, finding ways to make the SUT smaller help make the test smaller.

One way to achieve this test ratio when presented with a user journey which can require contributions from many internal systems is to "chain" tests, as illustrated in Figure 14-4, not specifically in their execution, but to create multiple smaller pairwise integration tests that represent the overall scenario. This is done by ensuring that the output of one test is used as the input to another test by persisting this output to a data repository.

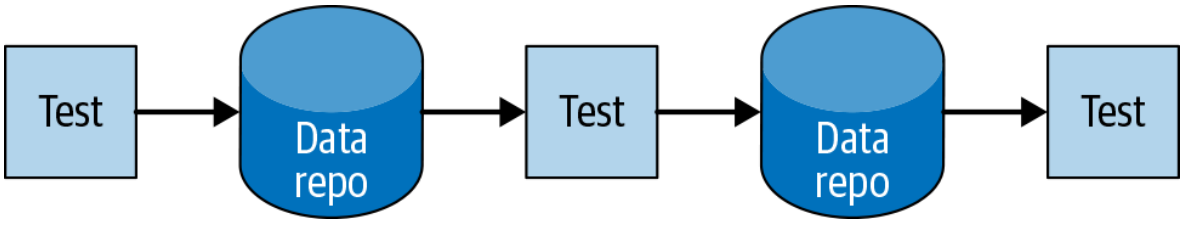

Figure 14-4. Chained tests

Structure of a Large Test

Although large tests are not bound by small test constraints and could conceivably consist of anything, most large tests exhibit common patterns. Large tests usually consist of a workflow with the following phases:

- Obtain a system under test
- Seed necessary test data

- Perform actions using the system under test
- Verify behaviors

The System Under Test

One key component of large tests is the aforementioned SUT (see Figure 14-5). A typical unit test focuses its attention on one class or module. Moreover, the test code runs in the same process (or Java Virtual Machine [JVM], in the Java case) as the code being tested. For larger tests, the SUT is often very different; one or more separate processes with test code often (but not always) in its own process.

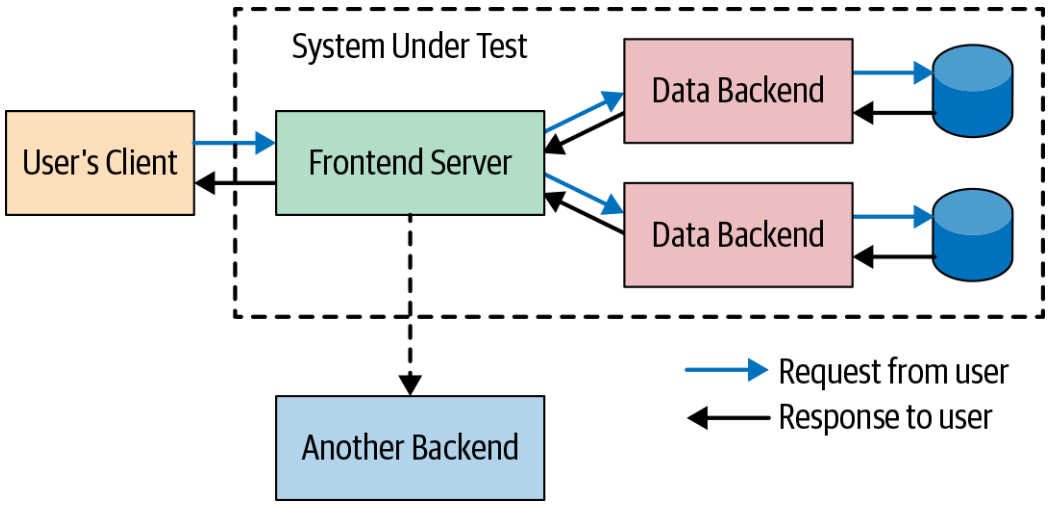

Figure 14-5. An example system under test (SUT)

At Google we use many different forms of SUTs and the scope of the SUT is one of the primary drivers of the scope of the large test itself (the larger the SUT, the larger the test). Each SUT form can be judged based on two primary factors:

Hermeticity

> This is the SUT's isolation from usages and interactions from other than the test in question. An SUT with high hermeticity will have the least exposure to sources of concurrency and infrastructure flakiness.

Fidelity

> The SUT's accuracy in reflecting the production system being tested. An SUT with high fidelity will consist of binaries that resemble the production versions (rely on similar configurations, use similar infrastructures, and have a similar overall topology).

Often these two factors are in direct conflict.

Following are some examples of SUTs:

Single-process SUT

The entire system under test is packaged into a single binary (even if in production these are multiple separate binaries). Additionally, the test code can be packaged into the same binary as the SUT. Such a test-SUT combination can be a "small" test if everything is single-threaded, but it is the least faithful to the production topology and configuration.

Single-machine SUT

The system under test consists of one or more separate binaries (same as production) and the test is its own binary. But everything runs on one machine. This is used for "medium" tests. Ideally, we use the production launch configuration of each binary when running those binaries locally for increased fidelity.

Multimachine SUT

The system under test is distributed across multiple machines (much like a production cloud deployment). This is even higher fidelity than the single-machine SUT, but its use makes tests "large" size and the combination is susceptible to increased network and machine flakiness.

Shared environments (staging and production)

Instead of running a standalone SUT, the test just uses a shared environment. This has the lowest cost because these shared environments usually already exist, but the test might conflict with other simultaneous uses and one must wait for the code to be pushed to those environments. Production also increases the risk of end-user impact.

Hybrids

Some SUTs represent a mix: it might be possible to run some of the SUT but have it interact with a shared environment. Usually the thing being tested is explicitly run but its backends are shared. For a company as expansive as Google, it is practically impossible to run multiple copies of all of Google's interconnected services, so some hybridization is required.

THE BENEFITS OF HERMETIC SUTS

The SUT in a large test can be a major source of both unreliability and long turnaround time. For example, an in-production test uses the actual production system deployment. As mentioned earlier, this is popular because there is no extra overhead cost for the environment, but production tests cannot be run until the code reaches that environment, which means those tests cannot themselves block the release of the code to that environment—the SUT is too late, essentially.

The most common first alternative is to create a giant shared staging environment and to run tests there. This is usually done as part of some release promotion process, but it again limits test execution to only when the code is available. As an alternative, some teams will allow engineers to "reserve" time in the staging environment and to use that time window to deploy pending code and to run tests, but this does not scale with a growing number of engineers or a growing number of services, because the environment, its number of users, and the likelihood of user conflicts all quickly grow.

The next step is to support cloud-isolated or machine-hermetic SUTs. Such an environment improves the situation by avoiding the conflicts and reservation requirements for code release.

CALLOUT: RISKS OF TESTING IN PRODUCTION AND WEBDRIVER TORSO

We mentioned that testing in production can be risky. One humorous episode resulting from testing in production was known as the Webdriver Torso incident. We needed a way to verify that video rendering in YouTube production was working properly and so created automated scripts to generate test videos, upload them, and verify the quality of the upload. This was done in a Google-owned YouTube channel called Webdriver Torso. But this channel was public, as were most of the videos.

Subsequently, this channel was publicized in an article at Wired, which led to its spread throughout the media and subsequent efforts to solve the mystery. Finally, a blogger tied everything back to Google. Eventually, we came clean by having a bit of fun with it, including a Rickroll and an Easter Egg, so everything worked out well. But we do need to think about the possibility of end user discovery of any test data we include in production and be prepared for it.

REDUCING THE SIZE OF YOUR SUT AT PROBLEM BOUNDARIES

There are particularly painful testing boundaries that might be worth avoiding. Tests that involve both frontends and backends become painful because user interface (UI) tests are notoriously unreliable and costly:

- UIs often change in look-and-feel ways that make UI tests brittle but do not actually impact the underlying behavior.
- UIs often have asynchronous behaviors that are difficult to test.

Although it is useful to have end-to-end tests of a UI of a service all the way to its backend, these tests have a multiplicative maintenance cost for both the UI and the backends. Instead, if the backend provides a public API, it is often easier to split the tests into connected tests at the UI/API boundary and to use the public API to drive the end-to-end tests. This is true whether the UI is a browser, command-line interface (CLI), desktop app, and mobile app.

Another special boundary is for third-party dependencies. Third-party systems might not have a public shared environment for testing, and in some cases, there is a cost with sending traffic to a third party. Therefore, it is not recommended to have automated tests use a real third-party API and that dependency is an important seam at which to split tests.

To address this issue of size, we have made this SUT smaller by replacing its databases with in-memory databases and removing one of the servers outside the scope of the SUT that we actually care about, as shown in Figure 14-6. This SUT is more likely to fit on a single machine.

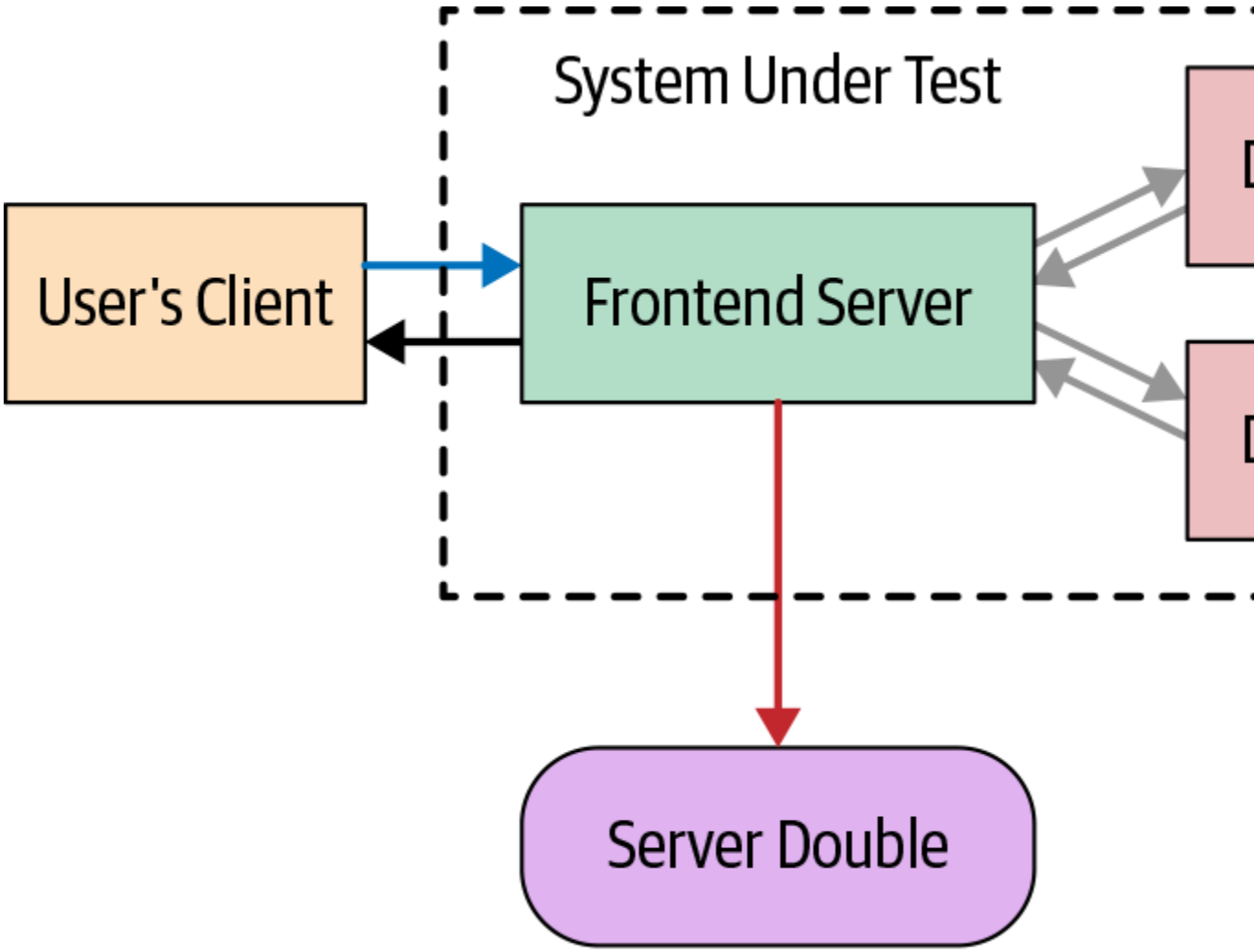

Figure 14-6. A reduced-size SUT

The key is to identify trade-offs between fidelity and cost/reliability, and to identify reasonable boundaries. If we can run a handful of binaries and a test and pack it all into the same machines that do our regular compiles, links, and unit test executions, we have the easiest and most stable "integration" tests for our engineers.

CALLOUT: RECORD-REPLAY PROXIES

In the previous chapter, we discussed test doubles and approaches that can be used to decouple the class under test from its difficult-to-test dependencies. We can also double entire servers and processes by using a mock, stub, or fake server or process with the equivalent API. However, there is no guarantee that the test double used actually conforms to the contract of the real thing that it is replacing.

One way of dealing with an SUT's dependent but subsidiary services is to use a test double, but how does one know that the double reflects the dependency's actual behavior? A growing approach outside of Google is to use a framework for consumer-driven contract tests. These are tests that define a contract for both the client and the provider of the service and this contract can drive automated tests. That is, a client defines a mock of the service saying that, for these input arguments I get a particular output. Then, the real service uses this input/output pair in a real test to ensure that it produces that output given those inputs. Two public tools for consumer-driven contract testing are Pact Contract Testing and Spring Cloud Contracts. Google's heavy dependency on protocol buffers means that we don't use these internally.

At Google, we do something a little bit different. Our most popular approach (for which there is a public API) is to use a larger test to generate a smaller one by recording the traffic to those external services when running the larger test and replaying it when running smaller tests. The larger, or "Record Mode" test runs continuously on post-submit but its primary purpose is to generate these traffic logs (it must pass, however, for the logs to be generated). The smaller, or "Replay Mode" test is used during development, and presubmit testing.

One of the interesting aspects of how record-replay works is that, because of nondeterminism, requests must be matched via a matcher to determine which response to replay. This makes them very similar to stubs and mocks in that argument matching is used to determine the resulting behavior.

What happens for new tests or tests where the client behavior changes significantly? In these cases, a request might no longer match what is in the recorded traffic file, so the test cannot pass in Replay mode. In that circumstance, the engineer must run the test in Record mode to generate new traffic, so it is important to make running Record tests easy, fast, and stable.

Test Data

A test needs data, and a large test needs two different kinds of data:

Seeded data

> Data preinitialized into the system under test reflecting the state of the SUT at the inception of the test

Test traffic

Data sent to the system under test by the test itself during its execution

Because of the notion of the separate and larger SUT, the work to seed the SUT state is often orders of magnitude more complex than the setup work done in a unit test. For example:

Domain data

Some databases contain data prepopulated into tables and used as configuration for the environment. Actual service binaries using such a database may fail on startup if domain data is not provided.

Realistic baseline

For an SUT to be perceived as realistic, it might require a realistic set of base data at startup, both in terms of quality and quantity. For example, large tests of a social network likely need a realistic social graph as the base state for tests: enough test users with realistic profiles as well as enough interconnections between those users must exist for the testing to be accepted.

Seeding APIs

The APIs by which data is seeded may be complex. It might be possible to directly write to a datastore, but doing so might bypass triggers and checks performed by the actual binaries that perform the writes.

Data can be generated in different ways, such as the following:

Handcrafted data

Like for smaller tests, we can create test data for larger tests by hand. But it might require more work to set up data for multiple services in a large SUT, and we might need to create a lot of data for larger tests.

Copied data

We can copy data, typically from production. For example, we might test a map of Earth by starting with a copy of our production map data to provide a baseline and then test our changes to it.

Sampled data

Copying data can provide too much data to reasonably work with. Sampling data can reduce the volume, thus reducing test time and making it easier to reason about. “Smart sampling” consists of

techniques to copy the minimum data necessary to achieve maximum coverage.

Verification

After an SUT is running and traffic is sent to it, we must still verify the behavior. There are a few different ways to do this:

Manual

Much like when you try out your binary locally, manual verification uses humans to interact with an SUT to determine whether it functions correctly. This verification can consist of testing for regressions by performing actions as defined on a consistent test plan or it can be exploratory, working a way through different interaction paths to identify possible new failures.

Note that manual regression testing does not scale sublinearly: the larger a system grows and the more journeys through it there are, the more human time is needed to manually test.

Assertions

Much like with unit tests, these are explicit checks about the intended behavior of the system. For example, for an integration test of Google search of `xyzzy`, an assertion might be as follows:

```
assertThat(response.Contains("Colossal Cave"))
```

A/B comparison (differential)

Instead of defining explicit assertions, A/B testing involves running two copies of the SUT, sending the same data, and comparing the output. The intended behavior is not explicitly defined: a human must manually go through the differences to ensure any changes are intended.

Types of Larger Tests

We can now combine these different approaches to the SUT, data, and assertions to create different kinds of large tests. Each test then has different properties as to which risks it mitigates; how much toil is required to write, maintain, and debug it; and how much it costs in terms of resources to run.

What follows is a list of different kinds of large tests that we use at Google, how they are composed, what purpose they serve, and what their limitations are:

- Functional testing of one or more binaries
- Browser and device testing
- Performance, load, and stress testing
- Deployment configuration testing
- Exploratory testing
- A/B Diff (regression) testing
- User acceptance testing (UAT)
- Probers and canary analysis
- Disaster recovery and chaos engineering
- User evaluation

Given such a wide number of combinations and thus a wide range of tests, how do we manage what to do and when? Part of designing software is drafting the test plan, and a key part of the test plan is a strategic outline of what types of testing are needed and how much of each. This test strategy identifies the primary risk vectors and identifies the necessary testing approaches to mitigate those risk vectors.

At Google, we have a specialized engineering role of "Test Engineer," and one of the things we look for in a good test engineer is the ability to outline a test strategy for our products.

Functional Testing of One or More Interacting Binaries

Tests of these type have the following characteristics:

- SUT: single-machine hermetic or cloud-deployed isolated
- Data: handcrafted
- Verification: assertions

As we have seen so far, unit tests are not capable of testing a complex system with true fidelity, simply because they are packaged in a different way than the real code is packaged. Many functional testing scenarios interact with a

given binary differently than with classes inside that binary, and these functional tests require separate SUTs and thus are canonical larger tests.

Testing the interactions of multiple binaries is, unsurprisingly, even more complicated than testing a single binary. A common use case is within microservices environments when services are deployed as many separate binaries. In this case, a functional test can cover the real interactions between the binaries by bringing up an SUT composed of all the relevant binaries and by interacting with it through a published API.

Browser and Device Testing

Testing web UIs and mobile applications is a special case of functional testing of one or more interacting binaries. It is possible to unit test the underlying code, but for the end users the public API is the application itself. Having tests that interact with the application as a third party through its frontend provides an extra layer of coverage.

Performance, Load, and Stress testing

Tests of these type have the following characteristics:

- SUT: cloud-deployed isolated
- Data: hand-crafted or multiplexed-from production
- Verification: diff (performance metrics)

Although it is possible to test a small unit in terms of performance, load, and stress, often such tests require sending simultaneous traffic to an external API. That definition implies that such tests are multithreaded tests that usually test at the scope of a binary under test. However, these tests are critical for ensuring that there is no degradation in performance between versions and that the system can handle expected spikes in traffic.

As the scale of the load test grows, the scope of the input data also grows, and it eventually becomes difficult to generate the scale of load required to trigger bugs under load. Load and stress handling are “highly emergent” properties of a system; that is, these complex behaviors belong to the overall system but not the individual members. Therefore, it is important to make these tests look as close to production as possible. Each SUT requires resources akin to what production requires and it becomes difficult to mitigate noise from the production topology.

One area of research for eliminating noise in performance tests is in modifying the deployment topology—how the various binaries are distributed across a network of machines. The machine running a binary can affect the performance characteristics; thus, if in a performance diff test the base version runs on a fast machine (or one with fast network) and the new version on a slow one, it can appear like a performance regression. This characteristic implies that the optimal deployment is to run both versions on the same machine. If a single machine cannot fit both versions of the binary, an alternative is to calibrate by performing multiple runs and removing peaks and valleys.

Deployment Configuration Testing

Tests of these type have the following characteristics:

- SUT: single-machine hermetic or cloud-deployed isolated
- Data: none
- Verification: assertions (doesn't crash)

Many times, it is not the code that is the source of defects but instead configuration: data files, databases, option definitions, and so on. Larger tests can test the integration of the SUT with its configuration files because these configuration files are read during the launch of the given binary.

Such a test is really a smoke test of the SUT without needing much in the way of additional data or verification. If the SUT starts successfully, the test passes. If not, the test fails.

Exploratory Testing

Tests of these type have the following characteristics:

- SUT: production or shared staging
- Data: production or a known test universe
- Verification: manual

Exploratory testing[4] is a form of manual testing that focuses not on looking for behavioral regressions by repeating known test flows, but on looking for questionable behavior by trying out new user scenarios. Trained users/testers interact with a product through its public APIs looking for new paths through

the system and for which behavior deviates from either expected or intuitive behavior or if there are security vulnerabilities.

Exploratory testing is useful for both new and launched systems to uncover unanticipated behaviors and side effects. By having testers follow different reachable paths through the system we can increase the system coverage and, when these testers identify bugs, capture new automated functional tests. In a sense, this is a bit like a manual "fuzz testing" version of functional integration testing.

LIMITATIONS

Manual testing does not scale sublinearly; that is, it requires human time to perform the manual tests. Any defects found by exploratory tests should be replicated with an automated test that can run much more frequently.

BUG BASHES

One common approach we use for manual exploratory testing is the bug bash. A team of engineers and related personnel (managers, product managers, test engineers, anyone with familiarity with the product) schedules a "meeting," but at this session, everyone involved manually tests the product. There can be some published guidelines as to particular focus areas for the bug bash and/or starting points for using the system, but the goal is to provide enough interaction variety to document questionable product behaviors and outright bugs.

A/B Diff Regression Testing

Tests of these type have the following characteristics:

- SUT: two cloud-deployed isolated environments
- Data: usually multiplexed from production or sampled
- Verification: A/B diff comparison

Unit tests cover expected behavior paths for a small section of code. But it is impossible to predict many of the possible failure modes for a given publicly facing product. Additionally, as Hyrum's Law states, the actual public API is not the declared one but all user-visible aspects of a product. Given those two properties, it is no surprise that A/B diff tests are possibly the most common form of larger testing at Google. This approach conceptually dates back to

1998. At Google, we have been running tests based on this model since 2001 for most of our products, starting with Ads, Search, and Maps.

A/B diff tests operate by sending traffic to a public API and comparing the responses between old and new versions (especially during migrations). Any deviations in behavior must be reconciled as either anticipated or unanticipated (regressions). In this case, the SUT is composed of two sets of real binaries: one running at the candidate version and the other running at the base version. A third binary sends traffic and compares the results.

There are other variants. We use A-A testing (comparing a system to itself) to identify nondeterministic behavior, noise, and flakiness, and to help remove those from A-B diffs. We also occasionally use A-B-C testing, comparing the last production version, the baseline build, and a pending change, to make it easy at one glance to see not only the impact of an immediate change, but also the accumulated impacts of what would be the next-to-release version.

A/B diff tests are a cheap but automatable way to detect unanticipated side effects for any launched system.

LIMITATIONS

Diff testing does introduce a few challenges to solve:

Approval

> Someone must understand the results enough to know whether any differences are expected. Unlike a typical test, it is not clear whether diffs are a good or bad thing (or whether the baseline version is actually even valid), and so there is often a manual step in the process.

Noise

> For a diff test, anything that introduces unanticipated noise into the results leads to more manual investigation of the results. It becomes necessary to remediate noise, and this is a large source of complexity in building a good diff test.

Coverage

> Generating enough useful traffic for a diff test can be a challenging problem. The test data must cover enough scenarios to identify corner-case differences, but it is difficult to manually curate such data.

Set-up

Configuring and maintaining one SUT is fairly challenging. Creating two at a time can double the complexity, especially if these share interdependencies.

UAT

Tests of these type have the following characteristics:

- SUT: machine-hermetic or Cloud-deployed isolated
- Data: handcrafted
- Verification: assertions

A key aspect of unit tests is that they are written by the developer writing the code under test. But that makes it quite likely that misunderstandings about the *intended* behavior of a product are reflected not only in the code, but also the unit tests. Such unit tests verify that code is "Working as implemented" instead of "Working as intended."

For cases in which there is either a specific end customer or a customer proxy (a customer committee or even a product manager), UATs are automated tests that exercise the product through public APIs to ensure the overall behavior for specific user journeys is as intended. Multiple public frameworks exist (e.g., Cucumber and RSpec) to make such tests writable/readable in a user-friendly language, often in the context of "runnable specifications."

Google does not actually do a lot of automated UAT and does not use specification languages very much. Many of Google's products historically have been created by the software engineers themselves. There has been little need for runnable specification languages because those defining the intended product behavior are often fluent in the actual coding languages themselves.

Probers and Canary Analysis

Tests of these type have the following characteristics:

- SUT: production
- Data: production
- Verification: assertions and A/B diff (of metrics)

Probers and canary analysis are ways to ensure that the production environment itself is healthy. In these respects, they are a form of production monitoring, but they are structurally very similar to other large tests.

Probers are functional tests that run encoded assertions against the production environment. Usually these tests perform well-known and deterministic read-only actions so that the assertions hold even though the production data changes over time. For example, a prober might perform a Google search at www.google.com and verify that a result is returned, but not actually verify the contents of the result. In that respect they are "smoke tests" of the production system, but they provide early detection of major issues.

Canary analysis is similar, except that it focuses on when a release is being pushed to the production environment. If the release is staged over time, we can run both prober assertions targeting the upgraded (canary) services as well as compare health metrics of both the canary and baseline parts of production and make sure that they are not out of line.

Probers should be used in any live system. If the production rollout process includes a phase in which the binary is deployed to a limited subset of the production machines (a canary phase), canary analysis should be used during that procedure.

LIMITATIONS

Any issues caught at this point in time (in production) are already affecting end users.

If a prober performs a mutable (write) action, it will modify the state of production. This could lead to one of three outcomes: nondeterminism and failure of the assertions, failure of the ability to write in the future, or user-visible side effects.

Disaster Recovery and Chaos Engineering

Tests of these type have the following characteristics:

- SUT: production
- Data: production and user-crafted (fault injection)
- Verification: manual and A/B diff (metrics)

These test how well your systems will react to unexpected changes or failures.

For years Google has run an annual war game called DiRT (Disaster Recovery Testing) during which faults are injected into our infrastructure at a nearly planetary scale. We simulate everything from datacenter fires to malicious attacks. In one memorable case we simulated an earthquake that completely isolated our headquarters in Mountain View, California, from the rest of the company. Doing so exposed not only technical shortcomings but also revealed the challenge of running a company when all the key decision makers were unreachable.[5]

The impacts of DiRT tests require a lot of coordination across the company, by contrast chaos engineering is more of a "continuous testing" for your technical infrastructure. Made popular by Netflix,[6] chaos engineering involves writing programs that continuously introduce a background level of faults into your systems and see what happens. Some of the faults can be quite large but in most cases chaos testing tools are designed to restore functionality before things get out of hand. The goal of chaos engineering is to help teams break assumptions of stability and reliability and help them grapple with the challenges of building resiliency in. Today teams at Google perform thousands of chaos tests each week using our own home-grown system called Catzilla.

These kinds of fault and negative tests make sense for live production systems that have enough theoretical fault tolerance to support them and for which the costs and risks of the tests themselves are affordable.

LIMITATIONS

Any issues caught at this point in time (in production) are already affecting end users.

DiRT is quite expensive to run and therefore we run a coordinated exercise on an infrequent scale. When we create this level of outage we actually cause pain and negatively impact employee performance.

If a prober performs a mutable (write) action, it will modify the state of production. This could lead to either nondeterminism and failure of the assertions, failure of the ability to write in the future, or user-visible side effects.

User Evaluation

Tests of these type have the following characteristics:

- SUT: production
- Data: production
- Verification: manual and A/B diffs (of metrics)

Production-based testing makes it possible to collect a lot of data about user behavior. We have a few different ways to collect metrics about the popularity of and issues with upcoming features, which provides us with an alternative to UAT:

Dogfooding

> It's possible using limited rollouts and experiments to make features in production available to a subset of users. We do this with our own staff sometimes (eat our own dogfood) and give us valuable feedback in the real deployment environment.

Experimentation

> A new behavior is made available as an experiment to a subset of users without their knowing. Then, the experiment group is compared to the control group at an aggregate level in terms of some desired metric. For example, in YouTube, we had a limited experiment changing the way video upvotes worked (eliminating the downvote), and only a portion of the user base saw this change.
>
> This is a massively important approach for Google. One of the first stories a Noogler hears upon joining the company is about the time Google launched an experiment changing the background shading color for AdWords ads in Google search and noticed a significant increase in ad clicks for users in the experimental group versus the control group.

Rater evaluation

> Human raters are presented with results for a given operation and choose which one is "better" and why. This feedback is then used to determine whether a given change is positive, neutral, or negative. For example, Google has historically used rater evaluation for search queries (we have published the guidelines we give our raters). In some

cases, the feedback from this ratings data can help determine launch go/no-go for algorithm changes. Rater evaluation is critical for nondeterministic systems like machine learning systems for which there is no clear correct answer, only a notion of better or worse.

Large Tests and the Developer Workflow

We've talked about what large tests are, why to have them, when to have them, and how much to have, but we have not said much about the who. Who writes the tests? Who runs the tests and investigates the failures? Who owns the tests? And how do we make this tolerable?

Although standard unit test infrastructure might not apply, it is still critical to integrate larger tests into the developer workflow. One way of doing this is to ensure that automated mechanisms for presubmit and postsubmit execution exist, even if these are different mechanisms than the unit test ones. At Google, many of these large tests do not belong in TAP. They are nonhermetic, too flaky, and/or too resource intensive. But we still need to keep them from breaking or else they provide no signal and become too difficult to triage. What we do, then, is to have a separate post-submit continuous build for these. We also encourage running these tests presubmit, because that provides feedback directly to the author.

A/B diff tests that require manual blessing of diffs can also be incorporated into such a workflow. For presubmit, it can be a code-review requirement to approve any diffs in the UI before approving the change. One such test we have files release-blocking bugs automatically if code is submitted with unresolved diffs.

In some cases, tests are so large or painful that presubmit execution adds to much developer friction. These tests still run postsubmit and are also run as part of the release process. The drawback to not running these presubmit is that the taint makes it into the monorepo and we need to identify the culprit change to roll it back. But we need to make the trade-off between developer pain and the incurred change latency and the reliability of the continuous build.

Authoring Large Tests

Although the structure of large tests is fairly standard, there is still a challenge with creating such a test, especially if it is the first time someone on the team has done so.

The best way to make it possible to write such tests is to have clear libraries, documentation, and examples. Unit tests are easy to write because of native language support (JUnit was once esoteric but is now mainstream). We reuse these assertion libraries for functional integration tests, but we also have created over time libraries for interacting with SUTs, for running A/B diffs, for seeding test data, and for orchestrating test workflows.

Larger tests are more expensive to maintain, in both resources and human time, but not all large tests are created equal. One reason that A/B diff tests are popular is that they have less human cost in maintaining the verification step. Similarly, production SUTs have less maintenance cost than isolated hermetic SUTs. And because all of this authored infrastructure and code must be maintained, the cost savings can compound.

However, this cost must be looked at holistically. If the cost of manually reconciling diffs or of supporting and safeguarding production testing outweighs the savings, it becomes ineffective.

Running Large Tests

We mentioned above how our larger tests don't fit in TAP and so we have alternate continuous builds and presubmits for them. One of the initial challenges for our engineers is how to even run nonstandard tests and how to iterate on them.

As much as possible we have tried to make our larger tests run in ways familiar for our engineers. Our presubmit infrastructure puts a common API in front of running both these tests and running TAP tests, and our code review infrastructure shows both sets of results. But many large tests are bespoke and thus need specific documentation for how to run them on demand. This can be a source of frustration for unfamiliar engineers.

SPEEDING UP TESTS

Engineers don't wait for slow tests. The slower a test is, the less frequently an engineer will run it, and the longer the wait after a failure until it is passing again.

The best way to speed up a test is often to reduce its scope or to split a large test into two smaller tests that can run in parallel. But there are some other tricks that you can do to speed up larger tests.

Some naive tests will use time-based sleeps to wait for nondeterministic action to occur, and this is quite common in larger tests. However, these tests do not have thread limitations and real production users want to wait as little as possible, so it is best for tests to react the way real production users would. Approaches include the following:

- Polling for a state transition repeatedly over a time window for an event to complete with a frequency closer to microseconds. You can combine this with a timeout value in case a test fails to reach a stable state.
- Implementing an event handler.
- Subscribing to a notification system for an event completion.

Note that tests that rely on sleeps and timeouts will all start failing when the fleet running those tests becomes overloaded, which spirals because those tests need to be rerun more often, increasing the load further.

Lower internal system timeouts and delays: A production system usually is configured assuming a distributed deployment topology, but an SUT might be deployed on a single machine (or at least a cluster of colocated machines). If there are hardcoded timeouts or (especially) sleep statements in the production code to account for production system delay, these should be made tunable and reduced when running tests.

Optimize test build time: one downside of our monorepo is that all of the dependencies for a large test are built and provided as inputs, but this might not be necessary for some larger tests. If the SUT is composed of a core part that is truly the focus of the test and some other necessary peer binary dependencies, it might be possible to use prebuilt versions of those other binaries at a known good version. Our build system (based on the monorepo) does not support this model easily, but the approach is actually more reflective of production in which different services release at different versions.

DRIVING OUT FLAKINESS

Flakiness is bad enough for unit tests, but for larger tests it can make them unusable. A team should view eliminating flakiness of such tests as a high priority. But how can flakiness be removed from such tests?

Minimizing flakiness starts with reducing the scope of the test—a hermetic SUT will not be at risk of the kinds of multiuser and real-world flakiness of production or a shared staging environment, and a single-machine hermetic SUT will not have the network and deployment flakiness issues of a distributed SUT. But you can mitigate other flakiness issues through test design and implementation and other techniques. In some cases, you will need to balance these with test speed.

Just as making tests reactive or event driven can speed them up, it can also remove flakiness. Timed sleeps require timeout maintenance and these timeouts can be embedded in the test code. Increasing internal system timeouts can reduce flakiness, whereas reducing internal timeouts can lead to flakiness if the system behaves in a nondeterministic way. The key here is to identify a trade-off that defines both a tolerable system behavior for end users (e.g., our maximum allowable timeout is *n* seconds) but handles flaky test execution behaviors well.

A bigger problem with internal system timeouts is that exceeding them can lead to difficult errors to triage. A production system will often try to limit end-user exposure to catastrophic failure by handling possible internal system issues gracefully. For example, if Google cannot serve an ad in a given time limit, we don't return a 500, we just don't serve an ad. But this looks to a test runner as if the ad serving code might be broken when there is just a flaky timeout issue. It's important to make the failure mode obvious in this case and to make it easy to tune such internal timeouts for test scenarios.

MAKING TESTS UNDERSTANDABLE

A specific case for which it can be difficult to integrate tests into the developer workflow is when those tests produce results that are unintelligible to the engineer running the tests. Even unit tests can produce some confusion—if my change breaks your test, it can be difficult to understand why if I am generally unfamiliar with your code—but for larger tests, such confusion can be insurmountable. Tests that are assertive must provide a clear pass/fail signal and must provide meaningful error output to help triage the source of failure. Tests that require human investigation, like A/B diff tests, require special handling to be meaningful or else risk being skipped during presubmit.

How does this work in practice? A good large test that fails should do the following:

Have a message that clearly identifies what the failure is

The worst-case scenario is to have an error that just says "Assertion failed" and a stack trace. A good error anticipates the test runner's unfamiliarity with the code and provides a message that gives context: "In test_ReturnsOneFullPageOfSearchResultsForAPopularQuery, expected 10 search results but got 1." For a performance or A/B diff test that fails, there should be a clear explanation in the output of what is being measured and why the behavior is considered suspect.

Minimize the effort necessary to identify the root cause of the discrepancy

A stack trace is not useful for larger tests because the call chain can span multiple process boundaries. Instead, it's necessary to produce a trace across the call chain or to invest in automation that can narrow down the culprit. The test should produce some kind of artifact to this effect. For example, Dapper is a framework used by Google to associate a single request ID with all the requests in an RPC call chain and all of the associated logs for that request can be correlated by that ID to facilitate tracing.

Provide support and contact information.

It should be easy for the test runner to get help by making the owners and supporters of the test easy to contact.

Owning Large Tests

Larger tests must have documented owners—engineers who can adequately review changes to the test and who can be counted on to provide support in the case of test failures. Without proper ownership, a test can fall victim to the following:

- It becomes more difficult for contributors to modify and update the test
- It takes longer to resolve test failures

And the test rots.

Integration tests of components within a particular project should be owned by the project lead. Feature-focused tests (tests that cover a particular business feature across a set of services) should be owned by a "feature owner"; in some cases this owner might be a software engineer responsible for the feature implementation end to end; in other cases it might be a product manager or a "test engineer" who owns the description of the business scenario. Whoever owns the test must be empowered to ensure its overall

health and must have both the ability to support its maintenance and the incentives to do so.

It is possible to build automation around test owners if this information is recorded in a structured way. Some approaches that we use include the following:

Regular code ownership

> In many cases a larger test is a standalone code artifact that lives in a particular location in our codebase. In that case, we can use the OWNERS (Chapter 9) information already present in the monorepo to hint to automation that the owner(s) of a particular test are the owners of the test code.

Per-test annotations

> In some cases, multiple test methods can be added to a single test class or module and each of these test methods can have a different feature owner. We use per-language structured annotations to document the test owner in each of these cases so that if a particular test method fails, we can identify the owner to contact.

Conclusion

A comprehensive test suite requires larger tests, both to ensure that tests match the fidelity of the system under test, and to address issues that unit tests cannot adequately cover. Because such tests are necessarily more complex and slower to run, care must be taken to ensure such larger tests are properly owned, well maintained, and run when necessary (such as before deployments to production). Overall, such larger tests must still be made as small as possible (while still retaining fidelity) to avoid developer friction. A comprehensive test strategy that identifies the risks of a system, and the larger tests that address them, is necessary for most software projects.

TL;DRs

- Larger tests cover things unit tests cannot.
- Large tests are composed of a System Under Test, Data, Action, and Verification.

- A good design includes a test strategy that identifies risks and larger tests that mitigate them.
- Extra effort must be made with larger tests to keep them from creating friction in the developer workflow.

1 See "Continuous Delivery" and Chapter 25 for more information.

2 *http://www0.cs.ucl.ac.uk/staff/w.zhang/rtb-papers/turn-throatling.pdf*

3 *http://hyrumslaw.com*

4 Whittaker, James A. 2009. "Exploratory Software Testing." Addison-Wesley Professional.

5 During this test, almost no one could get anything done, so many people gave up on work and went to one of our many cafes, and in doing so, we ended up creating a DDoS attack on our cafe teams!

6 "Open-sourcing Netflix's chaos generator, Chaos Monkey"

Chapter 15. Deprecation

Written by Hyrum Wright

Edited by Tom Manshreck

I love deadlines. I like the whooshing sound they make as they fly by.

Douglas Adams

All systems age. Even though software is a digital asset and the physical bits themselves don't degrade, new technologies, libraries, techniques, languages, and other environmental changes over time render existing systems obsolete. Old systems require continued maintenance, esoteric expertise, and generally more work as they diverge from the surrounding ecosystem. It's often better to invest effort in turning off obsolete systems, rather than letting them lumber along indefinitely alongside the systems that replace them. But the number of obsolete systems still running suggests that in practice doing so is not trivial. We refer to the process of orderly migration away from and eventual removal of obsolete systems as *deprecation*.

Deprecation is yet another topic that more accurately belongs to the discipline of *software engineering* than *programming* because it requires thinking about how to manage a system over time. For long-running software ecosystems, planning for and executing deprecation correctly reduces resource costs and improves velocity by removing the redundancy and complexity that builds up in a system over time. On the other hand, poorly deprecated systems may cost more than leaving them alone. While deprecating systems requires additional effort, it's possible to plan for deprecation during the design of the system so that it's easier to eventually decommission and remove it. Deprecations can affect systems ranging from individual function calls to entire software stacks. For concreteness, much of what follows focuses on code-level deprecations.

Unlike with most of the other topics we have discussed in this book, Google is still learning how best to deprecate and remove software systems. This chapter describes the lessons we've learned as we've deprecated large and heavily used internal systems. Sometimes, it works as expected, and sometimes it doesn't, but the general problem of removing obsolete systems remains a difficult and evolving concern in the industry.

This chapter primarily deals with deprecating technical systems, not end-user products. The distinction is somewhat arbitrary given that an external-facing API is just another sort of product, and an internal API may have consumers that consider themselves end users. Although many of the principles apply to turning down a public product, we concern ourselves here with the technical and policy aspects of deprecating and removing obsolete systems where the system owner has visibility into its use.

Why Deprecate?

Our discussion of deprecation begins from the fundamental premise that *code is a liability, not an asset*. After all, if code were an asset, why should we even bother spending time trying to turn down and remove obsolete systems? Code has costs, some of which are borne in the process of creating a system, but many other costs are borne as a system is maintained across its lifetime. These ongoing costs, such as the operational resources required to keep a system running or the effort to continually update its codebase as surrounding ecosystems evolve, mean that it's worth evaluating the trade-offs between keeping an aging system running or working to turn it down.

The age of a system alone doesn't justify its deprecation. A system could be finely crafted over several years to be the epitome of software form and function. Some software systems, such as the LaTeX typesetting system, have been improved over the course of decades, and even though changes still happen, they are few and far between. Just because something is old, it does not follow that it is obsolete.

Deprecation is best suited for systems that are demonstrably obsolete and a replacement exists that provides comparable functionality. The new system might use resources more efficiently, have better security properties, be built in a more sustainable fashion, or just fix bugs. Having two systems to accomplish the same thing might not seem like a pressing problem, but over time, the costs of maintaining them both can grow substantially. Users may need to use the new system, but still have dependencies that use the obsolete one.

The two systems might need to interface with each other, requiring complicated transformation code. As both systems evolve, they may come to depend on each other, making eventual removal of either more difficult. In the long run, we've discovered that having multiple systems performing the same function also impedes the evolution of the newer system because it is

still expected to maintain compatibility with the old one. Spending the effort to remove the old system can pay off as the replacement system can now evolve more quickly.

Earlier we made the assertion that "code is a liability, not an asset." If that is true, why have we spent most of this book discussing the most efficient way to build software systems that can live for decades? Why put all that effort into creating more code when it's simply going to end up on the liability side of the balance sheet?

Code *itself* doesn't bring value: it is the *functionality* that it provides that brings value. That functionality is an asset if it meets a user need: the code that implements this functionality is simply a means to that end. If we could get the same functionality from a single line of maintainable, understandable code, as 10,000 lines of convoluted spaghetti code, we would prefer the former. Code itself carries a cost, and the simpler the code that needs to be to maintained the same amount of functionality, the better.

Instead of focusing on how much code we can produce, or how large is our codebase, we should instead focus on how much functionality it can deliver per unit of code, and try to maximize that metric. One of the easiest ways to do so isn't writing more code and hoping to get more functionality, it's removing excess code and systems that are no longer needed. Deprecation policies and procedures make this possible.

Even though deprecation is useful, we've learned at Google that organizations have a limit on the amount of deprecation work that is reasonable to undergo simultaneously, from the aspect of the teams doing the deprecation as well as the customers of those teams. For example, although everybody appreciates having freshly paved roads, if the public works department decided to close down *every* road for paving simultaneously, nobody would go anywhere. By focusing their efforts, paving crews can get specific jobs done faster while also allowing other traffic to make progress. Likewise, it's important to choose deprecation projects with care and then commit to following through on finishing them.

Why Is Deprecation So Hard?

We've mentioned Hyrum's Law elsewhere in this book, but it's worth repeating its applicability here: the more users of a system, the higher the probability that users are using it in unexpected and unforeseen ways, and the harder it will be to deprecate and remove such a system. Their usage just "happens to work" instead of being "guaranteed to work." In this context, removing a system can be thought of as the ultimate change: we aren't just changing behavior, we are removing that behavior completely! This kind of radical alteration will shake loose a number of unexpected dependents.

To further complicate matters, deprecation usually isn't an option until a newer system is available which provides the same (or better!) functionality. The new system might be better, but it is also different: after all, if it were exactly the same as the obsolete system, it wouldn't provide any benefit to users who migrate to it (though it might benefit the team operating it). This functional difference means a one-to-one match between the old system and the new system is rare, and every use of the old system must be evaluated in the context of the new one.

Another surprising reluctance to deprecate is emotional attachment to old systems, particularly those that the deprecator had a hand in helping to create. An example of this change aversion happens when systematically removing old code at Google: we've occasionally encountered resistance of the form "I like this code!" It can be difficult to convince engineers to tear down something they've spent years building. This is an understandable response, but ultimately self-defeating: if a system is obsolete, it has a net cost on the organization and should be removed. One of the ways we've addressed concerns about keeping old code within Google is by ensuring that the source code repository isn't just searchable at trunk, but also historically. Even code that has been removed can be found again (see Chapter 17).

There's an old joke within Google that there are two ways of doing things: the one that's deprecated, and the one that's not-yet-ready. This is usually the result of a new solution being "almost" done and is the unfortunate reality of working in a technological environment that is complex and fast paced.

Google engineers have become used to working in this environment, but it can still be disconcerting. Good documentation, plenty of signposts, and teams of experts helping with the deprecation and migration process all make it easier to know whether you should be using the old thing, with all its warts, or the new one, with all its uncertainties.

Finally, funding and executing deprecation efforts can be difficult politically; staffing a team and spending time removing obsolete systems costs real money, whereas the costs of doing nothing and letting the system lumber along unattended are not readily observable. It can be difficult to convince the relevant stakeholders that deprecation efforts are worthwhile, particularly if they negatively impact new feature development. Research techniques, such as those described in the XREF(Measuring Success), can provide concrete evidence that a deprecation is worthwhile.

Given the difficulty in deprecating and removing obsolete software systems, it is often easier for users to evolve a system *in situ*, rather than completely replacing it. Incrementality doesn't avoid the deprecation process altogether,

but it does break it down into smaller, more manageable chunks that can yield incremental benefits. Within Google, we've observed that migrating to entirely new systems is *extremely* expensive, and the costs are frequently underestimated. Incremental deprecation efforts accomplished by in-place refactoring can keep existing systems running while making it easier to deliver value to users.

Deprecation During Design

Like many engineering activities, deprecation of a software system can be planned as those systems are first built. Choices of programming language, software architecture, team composition and even company policy and culture all impact how easy it will be to eventually remove a system after it has reached the end of its useful life.

The concept of designing systems so that they can eventually be deprecated might be radical in software engineering, but it is common in other engineering disciplines. Consider the example of a nuclear power plant, which is an extremely complex piece of engineering. As part of the design of a nuclear power station, its eventual decommissioning after a lifetime of productive service must be taken into account, even going so far as to allocate funds for this purpose.[1] Many of the design choices in building a nuclear power plant are affected when engineers know that it will eventually need to be decommissioned.

Unfortunately, software systems are rarely so thoughtfully designed. Many software engineers are attracted to the task of building and launching new systems, not maintaining existing ones. The corporate culture of many companies, including Google, emphasizes building and shipping new products quickly, which often provides a disincentive for designing with deprecation in mind from the beginning. And in spite of the popular notion of software engineers as data-driven automata, it can be psychologically difficult to plan for the eventual demise of the creations we are working so hard to build.

So, what kinds of considerations should we think about when designing systems that we can more easily deprecate in the future? Here are a couple of the questions we encourage engineering teams at Google to ask:

- How easy will it be for my consumers to migrate from my product to a potential replacement?
- How can I replace parts of my system incrementally?

Many of these questions relate to how a system provides and consumes dependencies. For a more thorough discussion of how we manage these dependencies, see Chapter 16.

Finally, we should point out that the decision as to whether to support a project long term is made when an organization first decides to build the project. After a software system exists, the only remaining options are support it, carefully deprecate it, or let it stop functioning when some external event causes it to break. These are all valid options, and the trade-offs between them will be organization specific. A new startup with a single project will unceremoniously kill it when the company goes bankrupt, but a large company will need to think more closely about the impact across its portfolio and reputation as they consider removing old projects. As mentioned earlier, Google is still learning how best to make these trade-offs with our own internal and external products.

In short, don't start projects that your organization isn't committed to support for the expected lifespan of the organization. Even if the organization chooses to deprecate and remove the project, there will still be costs, but they can be mitigated through planning and investments in tools and policy.

Types of Deprecation

Deprecation isn't a single kind of process, but a continuum of them, ranging from "we'll turn this off someday, we hope" to "this system is going away tomorrow, customers better be ready for that." Broadly speaking, we divide this continuum into two separate areas: advisory and compulsory.

Advisory Deprecation

Advisory deprecations are those that don't have a deadline and aren't high priority for the organization (and for which the company isn't willing to dedicate resources). These could also be labeled *aspirational* deprecations: the team knows the system has been replaced, and although they hope clients will eventually migrate to the new system, they don't have imminent plans to either provide support to help move clients or delete the old system. This kind of deprecation often lacks enforcement: we hope that clients move, but can't force them to. As our friends in SRE will readily tell you: "Hope is not a strategy."

Advisory deprecations are a good tool for advertising the existence of a new system and encouraging early adopting users to start trying it out. Such a new

system should *not* be considered in a beta period: it should be ready for production uses and loads, and should be prepared to support new users indefinitely. Of course, any new system is going to experience growing pains, but after the old system has been deprecated in any way, the new system will become a critical piece of the organization's infrastructure.

One scenario we've seen at Google in which advisory deprecations have strong benefits is when the new system offers compelling benefits to its users. In these cases, simply notifying users of this new system and providing them self-service tools to migrate to it often encourages adoption. However, the benefits cannot be simply incremental: they must be transformative. Users will be hesitant to migrate on their own for marginal benefits, and even new systems with vast improvements will not gain full adoption using only advisory deprecation efforts.

Advisory deprecation allows system authors to nudge users in the desired direction, but they should not be counted on to do the majority of migration work. It is often tempting to simply put a deprecation warning on an old system and walk away without any further effort. Our experience at Google has been that this can lead to (slightly) fewer new uses of an obsolete system, but it rarely leads to teams actively migrating away from it. Existing uses of the old system exert a sort of conceptual (or technical) pull toward it: comparatively many uses of the old system will tend to pick up a large share of new uses, no matter how much we say, "Please use the new system." The old system will continue to require maintenance and other resources unless its users are more actively encouraged to migrate.

Compulsory Deprecation

This active encouragement comes in the form of *compulsory* deprecation. This kind of deprecation usually comes with a deadline for removal of the obsolete system: if users continue to depend on it beyond that date, they will find their own systems no longer work.

Counterintuitively, the best way for compulsory deprecation efforts to scale is by localizing the expertise of migrating users to within a single team of experts—usually the team responsible for removing the old system entirely. This team has incentives to help others migrate from the obsolete system and can develop experience and tools that can then be used across the organization. Many of these migrations can be effected using the same tools discussed as part of the Chapter 22.

For compulsory deprecation to actually work, its schedule needs to have an enforcement mechanism. This does not imply that the schedule can't change, but empower the team running the deprecation process to break noncompliant users after they have been sufficiently warned through efforts to migrate them. Without this power, it becomes easy for customer teams to ignore deprecation work in favor of features or other more pressing work.

At the same time, compulsory deprecations without staffing to do the work can come across to customer teams as mean spirited, which usually impedes completing the deprecation. Customers simply see such deprecation work as an unfunded mandate, requiring them to push aside their own priorities to do work just to keep their services running. This feels much like the "running to stay in place" phenomenon and creates friction between infrastructure maintainers and their customers. It's for this reason that we strongly advocate that compulsory deprecations are actively staffed by a specialized team through completion.

It's also worth noting that even with the force of policy behind them, compulsory deprecations can still face political hurdles. Imagine trying to enforce a compulsory deprecation effort when the last remaining user of the old system is a critical piece of infrastructure your entire organization depends on. How willing would you be to break that infrastructure—and everybody that depends on it transitively—just for the sake of making an arbitrary deadline? It is hard to believe the deprecation is really compulsory if that team can veto its progress.

Google's monolithic repository and dependency graph gives us tremendous insight into how systems are used across our ecosystem. Even so, some teams might not even know they have a dependency on an obsolete system, and it can be difficult to discover these dependencies analytically. It's also possible to find them dynamically through tests of increasing frequency and duration during which the old system is turned off temporarily. These intentional changes provide a mechanism for discovering unintended dependencies by seeing what breaks, thus alerting teams to a need to prepare for the upcoming deadline. With Google, we occasionally change the name of implementation-only symbols to see which users are depending on them unaware.

Frequently at Google when a system is slated for deprecation and removal, the team will announce planned outages of increasing duration in the months and weeks prior to the turndown. Similar to Google's Disaster Recovery Testing (DiRT) exercises, these events often discover unknown dependencies between running systems. This incremental approach allows those dependent

teams to discover and then plan for the system's eventual removal, or even work with the deprecating team to adjust their timeline. (The same principles also apply for static code dependencies, but the semantic information provided by static analysis tools is often sufficient to detect all the dependencies of the obsolete system.)

Deprecation Warnings

For both advisory and compulsory deprecations, it is often useful to have a programmatic way of marking systems as deprecated, so that users are warned about their use and encouraged to move away. It's often tempting to just mark something as deprecated and hope its uses eventually disappear, but remember "hope is not a strategy." Deprecation warnings can help prevent new uses, but rarely lead to migration of existing systems.

What usually happens in practice is that these warnings accumulate over time. If they are used in a transitive context (for example, library A depends on library B, which depends on library C, and C issues a warning which shows up when A is built), these warnings can soon overwhelm users of a system to the point where they ignore them altogether. In health care, this phenomenon is known as "alert fatigue."

Any deprecation warning issued to a user needs to have two properties: actionability and relevance. A warning is *actionable* if the user can use the warning to actually perform some relevant action, not just in theory, but in practical terms, given the expertise in that problem area that we expect for an average engineer. For example, a tool might warn that a call to a given function should be replaced with a call to its updated counterpart, or an email might outline the steps required to move data from an old system to a new one. In each case, the warning provided the next steps that an engineer can perform to no longer depend on the deprecated system.[2]

A warning can be actionable, but still be annoying. To be useful, a deprecation warning should also be *relevant*. A warning is relevant if it surfaces at a time when a user actually performs the indicated action. Warning about the use of a deprecated function is best done while the engineer is writing code that uses that function, not after it has been checked into the repository for several weeks. Likewise, an email for data migration is best sent several months before the old system is removed rather than as an afterthought a weekend before the removal occurs.

It's important to resist the urge to put deprecation warnings on everything possible. Warnings themselves are not bad, but naïve tooling often produces

a quantity of warning messages that can overwhelm the unsuspecting engineer. Within Google, we are very liberal with marking old functions as deprecated but leverage tooling such as ErrorProne or clang-tidy to ensure that warnings are surfaced in targeted ways. As discussed in Chapter 20, we limit these warnings to newly changed lines as a way to warn people about new uses of the deprecated symbol. Much more intrusive warnings, such as for deprecated targets in the dependency graph, are added only for compulsory deprecations, and the team is actively moving users away. In either case, tooling plays an important role in surfacing the appropriate information to the appropriate people at the proper time, allowing more warnings to be added without fatiguing the user.

Managing the Deprecation Process

Although they can feel like different kinds of projects because we're deconstructing a system rather than building it, deprecation projects are similar to other software engineering projects in the way they are managed and run. We won't spend too much effort going over similarities between those management efforts, but it's worth pointing out the ways in which they differ.

Process Owners

We've learned at Google that without explicit owners, a deprecation process is unlikely to make meaningful progress, no matter how many warnings and alerts a system might generate. Having explicit project owners who are tasked with managing and running the deprecation process might seem like a poor use of resources, but the alternatives are even worse: don't ever deprecate anything; or delegate deprecation efforts to the users of the system. The second case becomes simply an advisory deprecation, which will never organically finish, and the first is a commitment to maintain every old system *ad infinitum*. Centralizing deprecation efforts helps better assure that expertise actually *reduces* costs by making them more transparent.

Abandoned projects often present a problem when establishing ownership and aligning incentives. Every organization of reasonable size has projects that are still actively used but which nobody clearly owns or maintains, and Google is no exception. Projects sometimes enter this state because they are deprecated: the original owners have moved on to a successor project, leaving the obsolete one chugging along in the basement, still a dependency of a critical project, and hoping it just fades away eventually.

Such projects are unlikely to fade away on their own. In spite of our best hopes, we've found that these projects still require deprecation experts to remove them and prevent their failure at inopportune times. These teams should have removal as their primary goal, not just a side project of some other work. In the case of competing priorities, deprecation work will almost always be perceived as having a lower priority and rarely receive the attention it needs. These sorts of important-not-urgent cleanup tasks are a great use of 20% time and provide engineers a great way to exposure to other parts of the codebase.

Milestones

When building a new system, project milestones are generally pretty clear: "Launch the frobnazzer features by next quarter." Following incremental development practices, teams build and deliver functionality incrementally to users, who get a win whenever they take advantage of a new feature. The end goal might be to launch the entire system, but incremental milestones help give the team a sense of progress and ensure they don't need to wait until the end of the process to generate value for the organization.

In contrast, it can often feel that the only milestone of a deprecation process is removing the obsolete system entirely. The team can feel they haven't made any progress until they've turned out the lights and gone home. Although this might be the most meaningful step for the team, if it has done its job correctly, it's often the least noticed by anyone external to the team, because by that point, the obsolete system no longer has any users. Deprecation project managers should resist the temptation to make this the only measurable milestone, particularly given that it might not even happen in all deprecation projects.

Similar to building a new system, managing a team working on deprecation should involve concrete incremental milestones, which are measurable and deliver value to users. The metrics used to evaluate the progress of the deprecation will be different, but it is still good for morale to celebrate incremental achievements in the deprecation process. We have found it useful to recognize appropriate incremental milestones, such as deleting a key subcomponent, just as we'd recognize accomplishments in building a new product.

Deprecation Tooling

Much of the tooling used to manage the deprecation process is discussed in depth elsewhere in this book, such as the large-scale change (LSC) process (Chapter 22) or our code review tools (Chapter 19). Rather than talk about the specifics of the tools, we'll briefly outline how those tools are useful when managing the deprecation of an obsolete system. These tools can be categorized as discovery, migration, and backsliding prevention tooling.

DISCOVERY

During the early stages of a deprecation process, and in fact during the entire process, it is useful to know *how* and *by whom* an obsolete system is being used. Much of the initial work of deprecation is determining who is using the old system—and in which unanticipated ways. Depending on the kinds of use, this process may require revisiting the deprecation decision once new information is learned. We also use these tools throughout the deprecation process to understand how the effort is progressing.

Within Google, we use tools like Code Search (see Chapter 17) and Kythe (see Chapter 23) to statically determine which customers use a given library, and often to sample existing usage to see what sorts of behaviors customers are unexpectedly depending on. Because runtime dependencies generally require some static library or thin client use, this technique yields much of the information needed to start and run a deprecation process. Logging and runtime sampling in production help discover issues with dynamic dependencies.

Finally, we treat our global test suite as an oracle to determine whether all references to an old symbol have been removed. As discussed in Chapter 11, tests are a mechanism of preventing unwanted behavioral changes to a system as the ecosystem evolves. Deprecation is a large part of that evolution, and customers are responsible for having sufficient testing to ensure that the removal of an obsolete system will not harm them.

MIGRATION

Much of the work of doing deprecation effort at Google is achieved by using the same set of code generation and review tooling we mentioned earlier. The LSC process and tooling are particularly useful in managing the large effort of actually updating the codebase to refer to new libraries or runtime services.

PREVENTING BACKSLIDING

Finally, an often-overlooked piece of deprecation infrastructure is tooling for preventing the addition of new uses of the very thing being actively removed. Even for advisory deprecations, it is useful to warn users to shy away from a deprecated system in favor of a new one when they are writing new code. Without backsliding prevention, deprecation can become a game of whack-a-mole in which users constantly add new uses of a system with which they are familiar (or find examples of elsewhere in the codebase), and the deprecation team constantly migrates these new uses. This process is both counterproductive and demoralizing.

To prevent deprecation backsliding on a micro level, we use the Tricorder static analysis framework to notify users that they are adding calls into a deprecated system and give them feedback on the appropriate replacement. Owners of deprecated systems can add compiler annotations to deprecated symbols (such as the `@deprecated` Java annotation) and Tricorder surfaces new uses of these symbols at review time. These annotations give control over messaging to the teams which own the deprecated system, while at the same time automatically alerting the change author. In limited cases, the tooling also suggests a push-button fix to migrate to the suggested replacement.

On a macro level, we use visibility whitelists in our build system to ensure that new dependencies are not introduced to the deprecated system. Automated tooling periodically examines these whitelists and prunes them as dependent systems are migrated away from the obsolete system.

Conclusion

Deprecation can feel like the dirty work of cleaning up the street after the circus parade has just passed through town, yet these efforts improve the overall software ecosystem by reducing maintenance overhead and cognitive burden of engineers. Scalably maintaining complex software systems over time is more than just building and running software: we must also be able to remove systems that are obsolete or otherwise unused.

A complete deprecation process involves successfully managing social and technical challenges through policy and tooling. Deprecating in an organized and well-managed fashion is often overlooked as a source of benefit to an organization, but is essential for its long-term sustainability.

TL;DRs

- Software systems have continuing maintenance costs that should be weighed against the costs of removing them.
- Removing things is often more difficult than building them to begin with because existing users are often using the system beyond its original design.
- Evolving a system in place is usually cheaper than replacing it with a new one, when turndown costs are included.
- It is difficult to honestly evaluate the costs involved in deciding whether to deprecate: aside from the direct maintenance costs involved in keeping the old system around, there are ecosystem costs involved in having multiple similar systems to choose between and that might need to interoperate. The old system might implicitly be a drag on feature development for the new. These ecosystem costs are diffuse and difficult to measure. Deprecation and removal costs are often similarly diffuse.

1 "Design and Construction of Nuclear Power Plants to Facilitate Decommissioning." IAEA technical Report Series No. 382.

2 See *https://abseil.io/docs/cpp/tools/api-upgrades* for an example.

Part IV. Tools

Chapter 16. Version Control and Branch Management

Written by Titus Winters

Edited by Lisa Carey

Perhaps no software engineering tool is quite as universally adopted throughout the industry as version control. One can hardly imagine any software organization larger than a few people that doesn't rely on a formal Version Control System (VCS) to manage its source code and coordinate activities between engineers.

In this chapter, we're going to look at why the use of version control has become such an unambiguous norm in software engineering, and we describe the various possible approaches to version control and branch management, including how we do it at scale across all of Google. We'll also examine the pros and cons of various approaches; although we believe everyone should use version control, some version control policies and processes might work better for your organization (or in general) than others. In particular, we find "trunk-based development" as popularized by DevOps[1] (one repository, no dev branches) to be a particularly scalable policy approach, and we'll provide some suggestions as to why that is.

What Is Version Control?

NOTE

This section might be a little basic for many readers: use of version control is, after all, fairly ubiquitous. If you want to skip ahead, we suggest jumping to the section ""Source of Truth""

A VCS is a system that tracks revisions (versions) of files over time. A VCS maintains some metadata about the set of files being managed, and collectively a copy of the files and metadata is called a repository[2] (repo for short). A VCS helps coordinate the activities of teams by allowing multiple developers to work on the same set of files simultaneously. Early VCSs did this by granting one person at a time the right to edit a file—that style of locking is enough to establish sequencing (an agreed upon "which is newer," an important feature of VCS). More advanced systems ensure that changes to

a *collection* of files submitted at once are treated as a single unit (*atomicity* when a logical change touches multiple files). Systems like CVS (a popular VCS from the 90s) that didn't have this atomicity for a commit were subject to corruption and lost changes. Ensuring atomicity removes the chance of previous changes being overwritten unintentionally, but requires tracking which version was last synced to—at commit time, the commit is rejected if any file in the commit has been modified at head since the last time the local developer synced. Especially in such a change-tracking VCS, a developer's working copy of the managed files will therefore need metadata of its own. Depending on the design of the VCS, this copy of the repository can be a repository itself, or might contain a reduced amount of metadata—such a reduced copy is usually a "client" or "workspace."

This seems like a lot of complexity: why is a VCS necessary? What is it about this sort of tool that has allowed it to become one of the few nearly universal tools for software development and software engineering?

Imagine for a moment working without a VCS. For a (very) small group of distributed developers working on a project of limited scope without any understanding of version control, the simplest and lowest-infrastructure solution is to just pass copies of the project back and forth. This works best when edits are nonsimultaneous (people are working in different time zones, or at least with different working hours). If there's any chance for people to not know which version is the most current, we immediately have an annoying problem: tracking which version is the most up to date. Anyone who has attempted to collaborate in a non-networked environment will likely recall the horrors of copying back-and-forth files named *Presentation v5 - final - redlines - Josh's version v2*. And as we shall see, when there isn't a single agreed-upon source of truth, collaboration becomes high friction and error prone.

Introducing shared storage requires slightly more infrastructure (getting access to shared storage), but provides an easy and obvious solution. Coordinating work in a shared drive might suffice for a while, with a small enough number of people but still requires out-of-band collaboration to avoid overwriting one another's work. Further, working directly in that shared storage means that any development task that doesn't keep the build working continuously will begin to impede everyone on the team—if I'm making a change to some part of this system at the same time that you kick off a build, your build won't work. Obviously, this doesn't scale well.

In practice, lack of file locking and lack of merge tracking will inevitably lead to collisions and work being overwritten. Such a system is very likely to introduce out-of-band coordination to decide who is working on any given file. If that file-locking is encoded in software, we've begun reinventing an early-generation version control like RCS (among others). After you realize that granting write permissions a file at a time is too coarse grained and you begin wanting line-level tracking—we're definitely reinventing version control. It seems nearly inevitable that we'll want some structured mechanism to govern these collaborations. Because we seem to just be reinventing the wheel in this hypothetical, we might as well use an off-the-shelf tool.

Why Is Version Control Important?

While version control is practically ubiquitous now, this was not always the case. The very first VCSs date back to the 1970s (SCCS) and 1980s (RCS)—many years later than the first references to software engineering as a distinct discipline. Teams participated in "the multiperson development of multiversion software" before the industry had any formal notion of version control. Version control evolved as a response to the novel challenges of digital collaboration. It took decades of evolution and dissemination for reliable, consistent use of version control to evolve into the norm that it is today.[3] So how did it become so important, and, given that it seems like a self-evident solution, why might anyone resist the idea of VCS?

Recall from that software engineering is programming integrated over time; we're drawing a distinction (in dimensionality) between the instantaneous production of source code and the act of maintaining that product over time. That basic distinction goes a long way to explaining the importance of, and hesitation toward, VCS: at the most fundamental level, version control is the engineer's primary tool for managing the interplay between raw source and time. We can conceptualize VCS as a way to extend a standard filesystem. A filesystem is a mapping from filename to contents. A VCS extends that to provide a mapping from (filename, time) to contents, along with the metadata necessary to track last sync points and audit history. Version control makes the consideration of time an explicit part of the operation: unnecessary in a programming task, critical in a software engineering task. In most cases, a VCS also allows for an extra input to that mapping (a branch name) to allow for parallel mappings; thus:

```
VCS(filename, time, branch) => file contents
```

In the default usage, that branch input will have a commonly understood default: we call that "head," "default," or "trunk" to denote main branch.

The (minor) remaining hesitation toward consistent use of version control comes almost directly from conflating programming and software engineering—we teach programming, we train programmers, we interview for jobs based on programming problems and techniques. It's perfectly reasonable for a new hire, even at a place like Google, to have little or no experience with code that is worked on by more than one person or for more than a couple weeks. Given that experience and understanding of the problem, version control seems like an alien solution. Version control is solving a problem that our new-hire hasn't necessarily experienced: an "undo," not for a single file but for an entire project, adding a lot of complexity for sometimes non-obvious benefits.

In some software groups, the same result plays out when management views the job of the techies as "software development" (sit down and write code) rather than "software engineering" (produce code, keep it working and useful for some extended period). With a mental model of programming as the primary task, and little understanding of the interplay between code and the passage of time, it's easy to see something described as "go back to a previous version to undo a mistake" as a weird, high-overhead luxury.

In addition to allowing separate storage and reference to versions over time, version control helps us bridge the gap between single-developer and multideveloper processes. In practical terms, this is why version control is so critical to software engineering, because it allows us to scale up teams and organizations, even though we use it only infrequently as an "undo" button. Development is inherently a branch-and-merge process, both when coordinating between multiple developers or a single developer at different points in time. A VCS removes the question of "which is more recent?" Use of modern version control automates error-prone operations like tracking which set of changes have been applied. Version control is how we coordinate between multiple developers and/or multiple points in time.

Because VCS has become so thoroughly embedded in the process of software engineering, even legal and regulatory practices have caught up. VCS allows a formal record of every change to every line of code, which is increasingly necessary for satisfying audit requirements. When mixing between in-house development and appropriate use of third-party sources, VCS helps track provenance and origination for every line of code.

In addition to the technical and regulatory aspects of tracking source over time and handling sync/branch/merge operations, version control triggers some nontechnical changes in behavior. The ritual of committing to version control and producing a commit log is a trigger for a moment of reflection: what have you accomplished since your last commit? Is the source in a state that you're happy with? The moment of introspection associated with committing, writing up a summary, and marking a task complete might have value on its own for many people. The start of the commit process is a perfect time to run through a checklist, run static analyses (see , check test coverage, run tests and dynamic analysis, and so on.

Like any process, version control comes with some overhead: someone must configure and manage your version control system, and individual developers must use it. But make no mistake about it: these can almost always be pretty cheap. Anecdotally, most experienced software engineers will instinctively use version control for any project that lasts more than a day or two, even for a single-developer project. The consistency of that result argues that the trade-off in terms of value (including risk reduction) versus overhead must be a pretty easy one. But we've promised to acknowledge that context matters and to encourage engineering leaders to think for themselves. It is always worth considering alternatives, even on something as fundamental as version control.

In truth, it's difficult to envision any task that can be considered modern software engineering that doesn't immediately adopt a VCS. Given that you understand the value and need for version control, you are likely now asking what type of version control you need.

Centralized VCS versus Distributed VCS

At the most simplistic level, all modern VCSs are equivalent to one another: so long as your system has a notion of atomically committing changes to a batch of files, everything else is just UI. You could build the same general semantics (not workflow) of any modern VCS out of another one and a pile of simple shell scripts. Thus, arguing about which VCS is "better" is primarily a matter of user experience—the core functionality is the same, the differences come in user experience, naming, edge-case features, and performance. Choosing a VCS is like choosing a filesystem format: when choosing among a modern-enough format, the differences are fairly minor, and the more important question by far is the content you fill that system with and the way you *use* it. However, major architectural differences in VCS can make configuration, policy, and scaling decisions easier or more difficult, so

it's important to be aware of the big architectural differences, chiefly the decision between centralized or decentralized.

CENTRALIZED VCS

In centralized VCS implementations, the model is one of a single central repository (likely stored on some shared compute resource for your organization). Although a developer can have files checked out and accessible on their local workstation, operations that interact on the version control status of those files need to be communicated to the central server (adding files, syncing, updating existing files, etc.). Any code that is committed by a developer is committed into that central repository. The first VCS implementations were all centralized VCSs.

Going back to the 1970s and early 1980s, we see that the earliest of these VCSs, such as RCS, focused on locking and preventing multiple simultaneous edits. You could copy the contents of a repository, but if you wanted to edit a file, you might need to acquire a lock, enforced by the VCS, to ensure that only you are making edits. When you've completed an edit, you release the lock. The model worked fine when any given change was a quick thing, or if there was rarely more than one person that wanted the lock for a file at any given time. Small edits like tweaking config files worked OK, as did working on a small team that either kept disjointed working hours or that rarely worked on overlapping files for extended periods. This sort of simplistic locking has inherent problems with scale: it can work fine for a few people, but has the potential to fall apart with larger groups if any of those locks become contended.[4]

As a response to this scaling problem, the VCSs that were popular through the 90s and early 2000s operated at a higher level. These more-modern centralized VCSs avoid the exclusive locking, but track which changes you've synced, requiring your edit to be based on the most-current version of every file in your commit. CVS wrapped and refined RCS by (mostly) operating on batches of files at a time and allowing multiple developers to check out a file at the same time: so long as your base version contained all of the changes in the repository, you're allowed to commit. Subversion advanced further by providing true atomicity for commits, version tracking, and better tracking for unusual operations (renames, use of symbolic links, etc.). The centralized repository/checked-out client model continues today within Subversion as well as most commercial VCSs.

DISTRIBUTED VCS

Starting in the mid-2000s, many popular VCSs follow the Distributed Version Control System (DVCS) paradigm, seen in systems like Git and Mercurial. The primary conceptual difference between DVCS and more traditional centralized VCS (Subversion, CVS) is the question: "Where can you commit?" or perhaps, "Which copies of these files count as a repository?"

A DVCS world does not enforce the constraint of a central repository: if you have a copy (clone, fork) of the repository, you have a repository that you can commit to as well as all of the metadata necessary to query for information about things like revision history. A standard workflow is to clone some existing repository, make some edits, commit them locally, and then push some set of commits to another repository, which may or may not be the original source of the clone. Any notion of centrality is purely conceptual, a matter of policy, not fundamental to the technology or the underlying protocols.

The DVCS model allows for better offline operation and collaboration without inherently declaring one particular repository to be the source of truth. One repository isn't necessary "ahead" or "behind," because changes aren't inherently projected into a linear timeline. However, considering common *usage*, both the centralized and DVCS models are largely interchangeable: whereas a centralized VCS provides a clearly defined central repository through technology, most DVCS ecosystems define a central repository for a project as a matter of policy. That is, most DVCS projects are built around one conceptual source of truth (a particular repository on GitHub, for instance). DVCS models tend to assume a more distributed use case, and have found particularly strong adoption in the open source world.

Generally speaking, the dominant source control system today is Git, which implements DVCS.[5] When in doubt, use that—there's some value in doing what everyone else does. If your use cases are expected to be unusual, gather some data and evaluate the trade-offs.

Google has a complex relationship with DVCS: our main repository is based on a (massive) custom in-house centralized VCS. There are periodic attempts to integrate more standard external options and to match the workflow that our engineers (especially Nooglers) have come to expect from external development. Unfortunately, those attempts to move toward more common tools like Git have been stymied by the sheer size of the codebase and

userbase, to say nothing of Hyrum's Law effects tying us to a particular VCS and interface for that VCS.[6] This is perhaps not surprising: most existing tools don't scale well with 50,000 engineers and tens of millions of commits.[7] The DVCS model, which often (but not always) includes transmission of history and metadata, requires a lot of data to spin up a repository to work out of.

In our workflow, centrality and in-the-cloud storage for the codebase seem to be critical to scaling. The DVCS model is built around the idea of downloading the entire codebase and having access to it locally. In practice, over time and as your organization scales up, any given developer is going to operate on a relatively smaller percentage of the files in a repository, and a small fraction of the versions of those files. As we grow (in file count and engineer count), that transmission becomes almost entirely waste. The only need for locality for most files occurs when building, but distributed (and reproducible) build systems seem to scale better for that task as well (see).

Source of Truth

Centralized VCSs (Subversion, CVS, Perforce, etc.) bake the source-of-truth notion into the very design of the system: whatever is most recently committed at trunk is the current version. When a developer goes to check out the project, by default that trunk version is what they will be presented with. Your changes are "done" when they have been recommitted on top of that version.

However, unlike centralized VCS, there is no *inherent* notion of which copy of the distributed repository is the single source of truth in DVCS systems In theory, it's possible to pass around commit tags and PRs with no centralization or coordination, allowing disparate branches of development to propagate unchecked, and thus risking a conceptual return to the world of *Presentation v5 - final - redlines - Josh's version v2*. Because of this, DVCS requires more explicit policy and norms than a centralized VCS does.

Well-managed projects using DVCS declare one specific branch in one specific repository to be the source of truth and thus avoid the more chaotic possibilities. We see this in practice with the spread of hosted DVCS solutions like GitHub or GitLab—users can clone and fork the repository for a project, but there is still a single primary repository: things are "done" when they are in the trunk branch on that repository.

It isn't an accident that centralization and Source of Truth has crept back into the usage even in a DVCS world. To help illustrate just how important this

Source of Truth idea is, let's imagine what happens when we don't have a clear source of truth.

SCENARIO: NO CLEAR SOURCE OF TRUTH

Imagine that your team adheres to the DVCS philosophy enough to avoid defining a specific branch+repository as the ultimate source of truth.

In some respects, this is reminiscent of the *Presentation v5 - final - redlines - Josh's version v2* model—after you pull from a teammate's repository, it isn't necessarily clear which changes are present and which are not. In some respects, it's better than that because the DVCS model tracks the merging of individual patches at a much finer granularity than those ad hoc naming schemes, but there's a difference between the DVCS knowing *which* changes are incorporated and every engineer being sure they have *all* the past/relevant changes represented.

Consider what it takes to ensure that a release build includes all of the features that have been developed by each developer for the past few weeks. What (noncentralized, scalable) mechanisms are there to do that? Can we design policies that are fundamentally better than having everyone sign off? Are there any that require only sublinear human effort as the team scales up? Is that going to continue working as the number of developers on the team scales up? As far as we can see: probably not. Without a central Source of Truth, someone is going to keep a list of which features are potentially ready to be included in the next release. Eventually that bookkeeping is reproducing the model of having a centralized Source of Truth.

Further imagine: when a new developer joins the team, where do they get a fresh known-good copy of the code?

DVCS enables a lot of great workflows and interesting usage models. But if you're concerned with finding a system that requires sublinear human effort to manage as the team grows, it's pretty important to have one repository (and one branch) actually defined to be the ultimate source of truth.

There is some relativity in that Source of Truth. That is, for a given project, that Source of Truth might be different for a different organization. This caveat is important: it's reasonable for engineers at Google or RedHat to have different Sources of Truth for Linux Kernel patches, still different than Linus (the Linux Kernel maintainer) himself would: DVCS works fine when organizations and their Sources of Truth are hierarchical (and invisible to those outside the organization)—that is perhaps the most practically useful

effect of the DVCS model. A RedHat engineer can commit to the local Source of Truth repository, and changes can be pushed from there upstream periodically, while Linus has a completely different notion of what is the Source of Truth. So long as there is no choice or uncertainty as to where a change should be pushed, we can avoid a large class of chaotic scaling problems in the DVCS model.

In all of this thinking, we're assigning special significance to the trunk branch. But of course, "trunk" in your VCS is only the technology default, and an organization can choose different policies on top of that. Perhaps the default branch has been abandoned and all work actually happens on some custom development branch—other than needing to provide a branch name in more operations, there's nothing inherently broken in that approach, it's just nonstandard. There's an (oft-unspoken) truth when discussing version control: the technology is only one part of it for any given organization, there is almost always an equal amount of policy and usage convention on top of that.

No topic in version control has more policy and convention than the discussion of how to use and manage branches. We look at branch management in more detail in the next section.

Version Control versus Dependency Management

There's a lot of conceptual similarity between discussions of version control policies and . The differences are primarily in two forms: VCS policies are largely about how you manage your own code, and are usually much finer grained. Dependency management is more challenging because we primarily focus on projects managed and controlled by other organizations, at a higher granularity, and these situations mean that you don't have perfect control. We'll discuss a lot more of these high-level issues later in the book.

Branch Management

Being able to track different revisions in Version Control opens up a variety of different approaches for how to manage those different versions. Collectively, these different approaches fall under the term *branch management*, in contrast to a single "trunk."

Work in Progress Is Akin to a Branch

Any discussion that an organization has about branch management policies ought to at least acknowledge that every piece of work-in-progress in the organization is equivalent to a branch. This is more explicitly the case with a DVCS in which developers are more likely to make numerous local staging commits before pushing back to the upstream Source of Truth. This is still true of centralized VCS: uncommitted local changes aren't conceptually different than committed changes on a branch, other than potentially being more difficult to find and diff against. Some centralized systems even make this explicit. For example, when using Perforce every change is given two revision numbers: one indicating the implicit branch point where the change was created, and one indicating where it was recommitted, as illustrated in Figure 16-1). Perforce users can query to see who has outstanding changes to a given file, inspect the pending changes in other users uncommitted changes, and more.

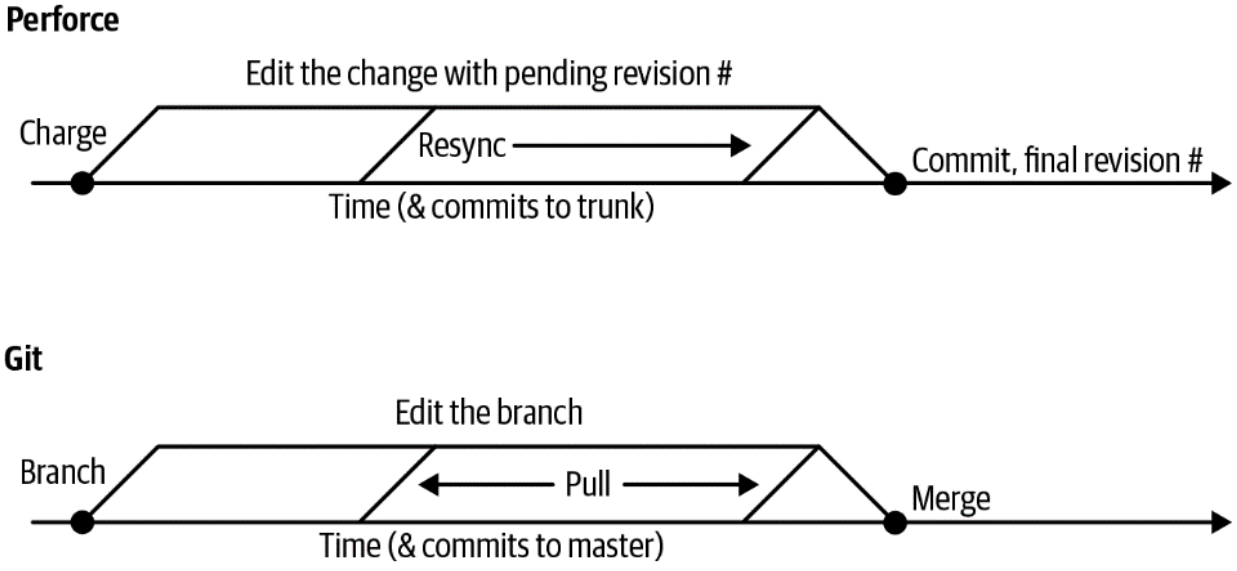

Figure 16-1. Two revision numbers in Perforce

This "uncommitted work is akin to a branch" idea is particularly relevant when thinking about refactoring tasks. Imagine a developer being told, "Go rename Widget to OldWidget." Depending on an organization's branch management policies and understanding, what counts as a branch, and which branches matter, this could have several interpretations:

- Rename Widget on the trunk branch in the Source of Truth repository
- Rename Widget on all branches in the Source of Truth repository
- Rename Widget on all branches in the Source of Truth repository, and find all devs with outstanding changes to files that reference Widget.

If we were to speculate, attempting to support that "rename this everywhere, even in outstanding changes" use case is part of why commercial centralized VCSs tend to track things like "Which engineers have this file open for editing?" (We don't think this is a scalable way to *perform* a refactoring task, but we understand the point of view.)

Dev Branches

In the age before consistent unit testing (see), when the introduction of any given change had a high risk of regressing functionality elsewhere in the system, it made sense to treat *trunk* specially. "We don't commit to trunk," your Tech Lead might say, "until new changes have gone through a full round of testing. Our team uses feature-specific development branches, instead."

A development branch (usually "dev branch") is a half-way point between "this is done but not committed" and "this is what new work is based on." The problem that these are attempting to solve (instability of the product) is a legitimate one—but one that we have found to be solved far better with more extensive use of tests, Continuous Integration (CI) (see), and quality enforcement practices like thorough code review.

We believe that a version control policy that makes extensive use of dev branches as a means toward product stability is inherently misguided. The same set of commits are going to be merged to trunk, eventually. Small merges are easier than big ones. Merges done by the engineer who authored those changes are easier than batching unrelated changes and merging later (which will happen eventually if a team is sharing a dev branch). If presubmit testing on the merge reveals any new problems, the same argument applies: it's easier to determine whose changes are responsible for a regression if there is only one engineer involved. Merging a large dev branch implies that more changes are happening in that test run, making failures more difficult to isolate. Triaging and root causing the problem is difficult; fixing it is even worse.

Beyond the lack of expertise and inherent problems in merging a single branch, there are significant scaling risks when relying on dev branches. This is a very common productivity drain for a software organization. When there are multiple branches being developed in isolation for long periods, coordinating merge operations becomes significantly more expensive (and possibly riskier) than they would be with trunk-based development.

HOW DID WE BECOME ADDICTED TO DEV BRANCHES?

It's easy to see how organizations fall into this trap: they see, "Merging this long-lived development branch reduced stability" and conclude, "Branch merges are risky." Rather than solve that with "Better testing" and "Don't use branch-based development strategies," they focus on slowing down and coordinating the symptom: the branch merges. Teams begin developing new branches based on other in-flight branches. Teams working on a long-lived dev branch might or might not regularly have that branch synched with the main development branch. As the organization scales up, the number of development branches grows, as well, and the more effort is placed on coordinating that branch merge strategy. Increasing effort is thrown at coordination of branch merges—a task that inherently doesn't scale. Some unlucky engineer becomes the Build Master/Merge Coordinator/Content Management Engineer, focused on acting as the single point coordinator to merge all the disparate branches in the organization. Regularly scheduled meetings attempt to ensure that the organization has "worked out the merge strategy for the week."[8] The teams that aren't chosen to merge often need to re-sync and retest after each of these large merges.

All of that effort in merging and retesting is *pure overhead*. The alternative requires a different paradigm: trunk-based development, rely heavily on testing and CI, keep the build green, and disable incomplete/untested features at runtime. Everyone is responsible to sync to trunk and commit; no "merge strategy" meetings, no large/expensive merges. And, no heated discussions about which version of a library should be used—there can be only one. There must be a single Source of Truth. In the end, there will be a single revision used for a release: narrowing down to a single source of truth is just the "shift left" approach for identifying what is and is not being included.

Release Branches

If the period between releases (or the release lifetime) for a product is longer than a few hours, it may be sensible to create a release branch that represents the exact code that went into the release build for your product. If any critical flaws are discovered between the actual release of that product into the wild and the next release cycle, fixes can be cherry-picked (a minimal, targeted merge) from trunk to your release branch.

By comparison to dev branches, release branches are generally benign: it isn't the technology of branches that is troublesome, it's the usage. The primary difference between a dev branch and a release branch is the expected

end state: a dev branch is expected to merge back to trunk, and could even be further branched by another team. A release branch is expected to be abandoned eventually.

In the highest-functioning technical organizations that Google's DevOps Research and Assessment (DORA) organization has identified, release branches are practically non-existent. Organizations that have achieved Continuous Deployment (CD)—the ability to release from trunk many times a day—likely tend to skip release branches: it's much easier to simply add the fix and redeploy; thus, cherry-picks and branches seem like unnecessary overhead. Obviously, this is more applicable to organizations that deploy digitally (such as web services and apps) than those that push any form of tangible release to customers; it is generally valuable to know exactly what has been pushed to customers.

That same DORA research also suggests a strong positive correlation between "trunk-based development," "no long-lived dev branches," and good technical outcomes. The underlying idea in both of those ideas seems clear: branches are a drag on productivity. In many cases we think complex branch and merge strategies are a perceived safety crutch—an attempt to keep trunk stable. As we see throughout this book, there are other ways to achieve that outcome.

Version Control at Google

At Google, the vast majority of our source is managed in a single repository (monorepo) shared among roughly 50,000 engineers. Almost all projects that are owned by Google live there, except large open source projects like Chromium and Android. This includes public-facing products like Search, Gmail, our advertising products, our Google Cloud Platform offerings, as well as the internal infrastructure necessary to support and develop all of those products.

We rely on an in-house-developed centralized VCS called Piper, built to run as a distributed microservice in our production environment. This has allowed us to use Google-standard storage, communication, and Compute as a Service technology to provide a globally available VCS storing more than 80 TB of content and metadata. The Piper monorepo is then simultaneously edited and committed to by many thousands of engineers every day. Between humans and semi-automated processes that make use of version control (or improve things checked into VCS), we'll regularly handle 60,000 to 70,000

commits to the repository per work day. Binary artifacts are fairly common because the full repository isn't transmitted and thus the normal costs of binary artifacts don't really apply. Because of the focus on Google-scale from the earliest conception, operations in this VCS ecosystem are still cheap at human scale: it takes perhaps 15 seconds total to create a new client at trunk, add a file, and commit an (unreviewed) change to Piper. This low-latency interaction and well-understood/well-designed scaling simplifies a lot of the developer experience.

By virtue of Piper being an in-house product, we have the ability to customize it and enforce whatever source control policies we choose. For instance, we have a notion of granular ownership in the monorepo: at every level of the file hierarchy we can find *OWNERS* files that list the usernames of engineers that are allowed to approve commits within that subtree of the repository (in addition to the OWNERS that are listed at higher levels in the tree). In an environment with many repositories, this might have been achieved by having separate repositories with filesystem permissions enforcement controlling commit access or via a Git "commit hook" (action triggered at commit time) to do a separate permissions check. By controlling the VCS, we can make the concept of ownership and approval more explicit and enforced by the VCS during an attempted commit operation. The model is also flexible: ownership is just a text file, not tied to a physical separation of repositories, so it is trivial to update as the result of a team transfer or organization restructuring.

One Version

The incredible scaling powers of Piper alone wouldn't allow the sort of collaboration that we rely upon. As we said earlier: Version Control is also about policy. In addition to our VCS, one key feature of Google's version control policy is what we've come to refer to as "One Version." This extends the "Single Source ofTruth" concept we looked at earlier—ensuring that a developer knows which branch and repository is their source of truth—to something like "For every dependency in our repository there must be only one version of that dependency to choose."[9] For third-party packages, this means that there can be only a single version of that package checked into our repository, in the steady state.[10] For internal packages, this means no forking without repackaging/renaming: it must be technologically safe to mix both the original and the fork into the same project with no special effort. This is a powerful feature for our ecosystem: there are very few packages with restrictions like "If you include this package (A), you cannot include other package (B)."

This notion of having a single copy on a single branch in a single repository as our Source of Truth is intuitive but also has some subtle depth in application. Let's investigate a scenario in which we have a monorepo (and thus arguably have fulfilled the letter of the law on Single Source of Truth), but have allowed forks of our libraries to propagate on trunk.

Scenario: Multiple Available Versions

Imagine the following scenario: some team discovers a bug in common infrastructure code (in our case, Abseil or Guava or the like). Rather than fix it in place, the team decides to fork that infrastructure and tweak it to work around the bug—without renaming the library or the symbols. It informs other teams near them, "Hey, we have an improved version of Abseil checked in over here: check it out." A few other teams build libraries that themselves rely on this new fork.

As we'll see in Chapter 21, we're now in a dangerous situation. If any project in the codebase comes to depend on both the original and the forked versions of Abseil simultaneously, in the best case the build fails. In the worst case we'll be subjected to difficult-to-understand runtime bugs stemming from linking in two mismatched versions of the same library. The "fork" has effectively added a coloring/partitioning property to the codebase: the transitive dependency set for any given target must include exactly one copy of this library. Any link added from the "original flavor" partition of the codebase to the "new fork" partition will likely break things. This means that in the end that something as simple as "adding a new dependency" becomes an operation that might require running all tests for the entire codebase, to ensure that we haven't violated one of these partitioning requirements. That's expensive, unfortunate, and doesn't scale well.

In some cases, we might be able to hack things together in a way to allow a resulting executable to function correctly. Java, for instance, has a relatively standard practice called *shading*, which tweaks the names of the internal dependencies of a library to hide those dependencies from the rest of the application. When dealing with functions, this is technically sound, even if it is theoretically a bit of a hack. When dealing with types that can be passed from one package to another, shading solutions work neither in theory nor in practice. As far as we know, any technological trickery that allows multiple isolated versions of a library to function in the same binary share this limitation: that approach will work for functions, but there is no good (efficient) solution to shading types—multiple versions for any library that provides a vocabulary type (or any higher-level construct) will fail. Shading

and related approaches are patching over the underlying issue: multiple versions of the same dependency are needed. (we'll discuss how to minimize that in general in

Any policy system that allows for multiple versions in the same codebase is allowing for the possibility of these costly incompatibilities. It's possible that you'll get away with it for a while (we certainly have a number of small violations of this policy), but in general any multiple-version situation has a very real possibility of leading to big problems.

The "One Version" Rule

With that example in mind, on top of the Single Source of Truth model, we can hopefully understand the depth of this seemingly simple rule for source control and branch management:

Developers must never have a choice "What version of this component should I depend upon?"

Colloquially, this becomes something like a "One-Version Rule," In practice, "One Version" is not hard-and-fast,[11] but phrasing this around limiting the versions that can be *chosen* when adding a new dependency conveys a very powerful understanding.

For an individual developer, lack of choice can seem like an arbitrary impediment. Yet we see again and again that for an organization, it's a critical component in efficient scaling. Consistency has a profound importance at all levels in an organization. From one perspective, this is a direct side effect of discussions about consistency and ensuring the ability to leverage consistent "choke points."

(Nearly) No Long-Lived Branches

There are several deeper ideas and policies implicit in our One Version Rule; foremost among them: development branches should be minimal, or at best be very short lived. This follows from a lot of published work over the past 20 years, from Agile processes to DORA research results on trunk-based development and even Phoenix Project[12] lessons on "reducing work-in-progress." When we include the idea of pending work as akin to a dev branch, this further reinforces that work should be done in small increments against trunk, committed regularly.

As a counterexample: in a development community that depends heavily on long-lived development branches, it isn't difficult to imagine opportunity for choice creeping back in.

Imagine this scenario: some infrastructure team is working on a new Widget, better than the old one. Excitement grows. Other newly started projects ask, "Can we depend on your new Widget?" Obviously, this can be handled if you've invested in codebase visibility policies, but the deep problem happens when the new Widget is "allowed" but only exists in a parallel branch. Remember: new development must not have a choice when adding a dependency. That new Widget should be committed to trunk, disabled from the runtime until it's ready, and hidden from other developers by visibility if possible—or the two Widget options should be designed such that they can coexist, linked into the same programs.

Interestingly, there is already evidence of this being important in the industry. In Accelerate and the most recent State of DevOps reports, DORA points out that there is a predictive relationship between trunk-based development and high performing software organizations. Google is not the only organization to have discovered this—nor did we necessarily have expected outcomes in mind when these policies evolved—it just seemed like nothing else worked. DORA's result certainly matches our experience.

Our policies and tools for large-scale changes (LSCs; see) put additional weight on the importance of trunk-based development: broad/shallow changes that are applied across the codebase are already a massive (often tedious) undertaking when modifying everything checked in to the trunk branch. Having an unbounded number of additional dev branches that might need to be refactored at the same time would be an awfully large tax on executing those types of changes, finding an ever-expanding set of hidden branches. In a DVCS model, it might not even be possible to identify all of those branches.

Of course, our experience is not universal. You might find yourself in unusual situations that require longer-lived dev branches in parallel to (and regularly merged with) trunk.

Those scenarios should be rare, and should be understood to be expensive. Across the roughly 1,000 teams that work in the Google monorepo, there are only a couple that have such a dev branch.[13] Usually these exist for a very specific (and very unusual) reason. Most of those reasons boil down to some variation of "We have an unusual requirement for compatibility over time." Oftentimes this is a matter of ensuring compatibility for data at rest across

versions: readers and writers of some file format need to agree on that format over time even if the reader or writer implementations are modified. Other times, long-lived dev branches might come from promising API compatibility over time—when One Version isn't enough and we need to promise that an older version of a microservice client still works with a newer server (or vice versa). That can be a very challenging requirement, something that you should not promise lightly for an actively evolving API, and something you should treat carefully to ensure that period of time doesn't accidentally begin to grow. Dependency across time in any form is far more costly and complicated than code that is time invariant. Internally, Google production services make relatively few promises of that form.[14] We also benefit greatly from a cap on potential version skew imposed by our "build horizon": every job in production needs to be rebuilt and redeployed every six months, maximum. (Usually it is far more frequent than that.)

We're sure there are other situations that might necessitate long-lived dev branches. Just make sure to keep them rare. If you adopt other tools and practices discussed in this book, many will tend to exert pressure against long-lived dev branches. Automation and tooling that works great at trunk and fails (or takes more effort) for a dev branch can help encourage developers to stay current.

What About Release Branches?

Many Google teams use release branches, with limited cherry picks. If you're going to put out a monthly release and continue working toward the next release, it's perfectly reasonable to make a release branch. Similarly, if you're going to ship devices to customers, it's valuable to know exactly what version is out "in the field." Use caution and reason, keep cherry picks to a minimum, and don't plan to remerge with trunk. Our various teams have all sorts of policies about release branches given that relatively few teams have arrived at the sort of rapid release cadence promised by CD (see XREF(Continuous Deployment)) that obviates the need or desire for a release branch. Generally speaking, release branches don't cause any widespread cost in our experience. Or, at least, no noticeable cost above and beyond the additional inherent cost to the VCS.

Monorepos

In 2016, we published a (highly cited, much discussed) paper on Google's monorepo approach (TODO(cite the monorepo paper)). The monorepo

approach has some inherent benefits, and chief among them is that adhering to One Version is trivial: it's usually more difficult to violate One Version than it would be to do the right thing. There's no process of deciding which versions of anything are official, or discovering which repositories are important. Building tools to understand the state of the build () doesn't also require discovering where important repositories exist. Consistency helps scale up the impact of introducing new tools and optimizations. By and large, engineers can see what everyone else is doing and use that to inform their own choices in code and system design. These are all very good things.

Given all of that, and our belief in the merits of the One-Version Rule, it is reasonable to ask whether a monorepo is the One True Way. By comparison, the open source community seems to work just fine with a "manyrepo" approach built on a seemingly infinite number of noncoordinating and nonsynchronized project repositories.

In short: no, we don't think the monorepo approach as we've described it is the perfect answer for everyone. Continuing the parallel between filesystem format and VCS, it's easy to imagine deciding between using 10 drives to provide one very large logical filesystem or 10 smaller filesystems accessed separately. In a filesystem world, there are pros and cons to both. Technical issues when evaluating filesystem choice would range from outage resilience, size constraints, performance characteristics, and so on. Usability issues would likely focus more on the ability to reference files across filesystem boundaries, add symlinks, and synchronize files.

A very similar set of issues governs whether to prefer a monorepo or a collection of finer-grained repositories. The specific decisions of how to store your source code (or store your files, for that matter) are easily debatable, and in some cases the particulars of your organization and your workflow are going to matter more than others. These are decisions you'll need to make yourself.

What is important is not whether we focus on monorepo; it's to adhere to the One Version principle to the greatest extent possible: developers must not have a *choice* when adding a dependency onto some library that is already in use in the organization. Choice violations of the One-Version Rule, lead to merge strategy discussions, diamond dependencies, lost work, and wasted effort.

Software engineering tools including both VCS and build systems are increasingly providing mechanisms to smartly blend between fine-grained repositories and monorepos to provide an experience akin to the monorepo—

an agreed-upon ordering of commits and understanding of the dependency graph. Git submodules, bazel with external dependencies, and cmake subprojects all allow modern developers to synthesize something weakly approximating monorepo behavior without the costs and downsides of a monorepo.[15] For instance, fine-grained repositories are easier to deal with in terms of scale (Git often has performance issues after a few million commits, and tends to be slow to clone when repositories include large binary artifacts) and storage (VCS metadata can add up, especially if you have binary artifacts in your version control system). Fine-grained repositories in a federated/virtual-monorepo (VMR)–style repository can make it easier to isolate experimental or top-secret projects while still holding to One Version and allowing access to common utilities.

To put it another way: if every project in your organization has the same secrecy, legal, privacy, and security requirements,[16] a true monorepo is a fine way to go. Otherwise, *aim* for the functionality of a monorepo, but allow yourself the flexibility of implementing that experience in a different fashion. If you can manage with disjoint repositories and adhere to One Version, or your workload is all disconnected enough to allow truly separate repositories, great. Otherwise, synthesizing something like a VMR in some fashion may represent the best of both worlds.

After all, your choice of filesystem format really doesn't matter as much as what you write to it.

Future of Version Control

Google isn't the only organization to publicly discuss the benefits of a monorepo approach. Microsoft, Facebook, Netflix, and Uber have also publicly mentioned their reliance on the approach. DORA has published about it extensively. It's vaguely possible that all of these successful, long-lived companies are misguided, or at least that their situations are sufficiently different as to be inapplicable to the average smaller organization. Although it's possible, we think it is unlikely.

Most arguments against monorepos focus on the technical limitations of having a single large repository. If cloning a repository from upstream is quick and cheap, developers are more likely to keep changes small and isolated (and to avoid making mistakes with committing to the wrong work-in-progress branch). If cloning a repository (or doing some other common VCS operation) takes hours of wasted developer time, you can easily see why

an organization would shy away from reliance on such a large repository/operation. We luckily avoided this pitfall by focusing on providing a VCS that scales massively.

Looking at the past few years of major improvements to Git, there's clearly a lot of work being done to support larger repositories: shallow clones, sparse branches, better optimization, and more. We expect this to continue, and the importance of "but we need to keep the repository small" to diminish.

The other major argument against monorepos is that it doesn't match how development happens in the Open Source Software (OSS) world. Although true, many of the practices in the OSS world come (rightly) from prioritizing freedom, lack of coordination, and lack of computing resources. Separate projects in the OSS world are effectively separate organizations that happen to be able to see one another's code. Within the boundaries of an organization, we can make more assumptions: we can assume the availability of compute resources, we can assume coordination, and we can assume that there is some amount of centralized authority.

A less-common but perhaps more-legitimate concern with the monorepo approach is that as your organization scales up, it is less and less likely that every piece of code is subject to exactly the same legal, compliance, regulatory, secrecy, and privacy requirements. One native advantage of a manyrepo approach is that separate repositories are obviously capable of having different sets of authorized developers, visibility, permissions, and so on. Stitching that feature into a monorepo can be done, but implies some ongoing carrying costs in terms of customization and maintenance.

At the same time, the industry seems to be inventing lightweight inter-repository linkage over and over again. Sometimes, this is in the VCS (Git submodules) or the build system. So long as a collection of repositories have a consistent understanding of "what is trunk," "which change happened first," and mechanisms to describe dependencies, we can easily imagine stitching together a disparate collection of physical repositories into one larger VMR. Even though Piper has done very well for us, investing in a highly scaling VMR, tools to manage it, and relying on off-the-shelf customization for per-repository policy requirements could have been a better investment.

As soon as someone builds a sufficiently large nugget of compatible and interdependent projects in the OSS community and publishes a VMR view of those packages, we suspect that OSS developer practices will begin to change. We see glimpses of this in the tools that *could* synthesize a virtual monorepo as well as in the work done by (for instance) large Linux

distributions discovering and publishing mutually compatible revisions of thousands of packages. With unit tests, CI, and automatic version bumping for new submissions to one of those revisions, enabling a package owner to update trunk for their package (in nonbreaking fashion, of course), we think that model will catch on in the open source world. It is just a matter of efficiency, after all: a (virtual) monorepo approach with a One-Version Rule cuts down the complexity of software development by a whole (difficult) dimension: time.

We expect version control and dependency management to evolve in this direction in the next 10 to 20 years: VCSs will focus on *allowing* larger repositories with better performance scaling, but also removing the need for larger repositories by providing better mechanisms to stitch them together across project and organizational boundaries. Someone, perhaps the existing package management groups or Linux distributors, will catalyze a de facto standard virtual monorepo. Depending on the utilities in that monorepo will provide easy access to a compatible set of dependencies as one unit. We'll more generally recognize that version numbers are timestamps, and that allowing version skew adds a dimensionality complexity (time) that costs a lot—and that we can learn to avoid. It starts with something logically like a monorepo.

Conclusion

Version control systems are a natural extension of the collaboration challenges and opportunities provided by technology, especially shared compute resources and computer networks. They have historically evolved in lockstep with the norms of software engineering as we understand them at the time.

Early systems provided simplistic file-granularity locking. As typical software engineering projects and teams grew larger, the scaling problems with that approach became apparent, and our understanding of version control changed to match those challenges. Then, as development increasingly moved toward an OSS model, with distributed contributors, VCSs became more decentralized. We expect a shift in VCS technology that assumes constant network availability, focusing more on storage and build in the cloud to avoid transmitting unnecessary files and artifacts. This is increasingly critical for large, long-lived software engineering projects, even if it means a change in approach compared to simple single-dev/single-machine programming projects. This shift to cloud will make concrete what

has emerged with DVCS approaches: even if we allow distributed development, something must still be centrally recognized as the Source of Truth.

The current DVCS decentralization is a sensible reaction of the technology to the needs of the industry (especially the open source community). However, DVCS configuration needs to be tightly controlled and coupled with branch management policies that make sense for your organization. It also can often introduce unexpected scaling problems: perfect fidelity offline operation requires a lot more local data. Failure to rein in the potential complexity of a branching free-for-all can lead to a potentially unbounded amount of overhead between developers and deployment of that code. However, complex technology doesn't need to be used in a complex fashion: as we see in monorepo and trunk-based development models, keeping branch policies simple generally leads to better engineering outcomes.

Choice leads to costs here. We highly endorse the One-Version Rule presented here: developers within an organization must not have a choice where to commit, or which version of an existing component to depend upon. There are few policies we're aware of that can have such an impact on the organization: although it might be annoying for individual developers, in the aggregate, the end result is far better.

TL;DRs

- Use version control for any software development project larger than "toy project with only one developer that will never be updated."
- There's an inherent scaling problem when there are choices in "which version of this should I depend upon?"
- One-Version Rules are surprisingly important for organizational efficiency. Removing choices in where to commit or what to depend upon can result in significant simplification.
- In some languages you might be able to spend some effort to dodge this with technical approaches like shading, separate compilation, linker hiding, and so on. The work to get those approaches working is entirely lost labor—your softwrae engineers aren't producing anything, they're just working around technical debts.
- Previous research (DORA/State of DevOps/Accelerate) has shown that trunk-based development is a predictive factor in high-performing

development organizations. Long-lived dev branches are not a good default plan.

- Use whatever version control system makes sense for you. If your organization wants to prioritize separate repositories for separate projects, it's still probably wise for inter-repository dependencies to be unpinned/"at head"/"trunk based." There are an increasing number of VCS and build system facilities that allow you to have both small, fine-grained repositories as well as a consistent "virtual" head/trunk notion for the whole organization.

1 The "DevOps Research Association," which was acquired by Google between the first draft of this chapter and publication, has published extensively on this in the annual "State of DevOps Report" and the book *Accelerate*. As near as we can tell, it popularized the terminology "trunk-based development."

2 Although the formal idea of what is and is not a repository changes a bit depending on your choice of VCS, and the terminology will vary.

3 Indeed, I've given several public talks that use "adoption of version control" as the canonical example of how the norms of software engineering can *and do* evolve over time. In my experience, in the 1990s version control was pretty well understood as a best practice but not universally followed. In the early 2000s, it was still common to encounter professional groups that didn't use it. Today, the use of tools like Git seems ubiquitous even among college students working on personal projects. Some of this rise in adoption is likely due to better user experience in the tools (nobody wants to go back to RCS), but the role of experience and changing norms is significant.

4 Anecdote: To illustrate this, I looked for information on what Googlers had pending/unsubmitted edits outstanding for a semi-popular file in my most recent project. At the time of this writing, 27 changes are pending, 12 from people on my team, 5 from people on related teams, and 10 from engineers I've never met. This is basically working as expected. Technical systems or policies that require out-of-band coordination certainly don't scale to 24/7 software engineering in distributed locations.

5 Stack Overflow, Developer Survey Results, 2018.

6 Monotonically increasing version numbers, rather than commit hashes, are particularly troublesome. Many systems and scripts have grown up in the Google developer ecosystem that assume that the numeric ordering of

commits is the same as the temporal order—undoing those hidden dependencies is difficult.

7 For that matter, as of the publication of the Monorepo paper, the repository itself had something like 86 TB of data and metadata, ignoring release branches. Fitting that onto a developer workstation directly would be…challenging.

8 Recent informal twitter polling suggests about 25% of software engineers have been subjected to "regularly scheduled" merge strategy meetings.

9 For example, during an upgrade operation there might be two versions checked in, but if a developer is adding a new dependency on an existing package, there should be no *choice* in which version to depend upon.

10 That said, we fail at this in many cases because external packages sometimes have pinned copies of their own dependencies bundled in their source release. You can read more on how all of this goes wrong in .

11 For instance, if there are external/third-party libraries that are periodically updated, it might be infeasible to update that library and update all use of it in a single atomic change. As such, it is often necessary to add a new version of that library, prevent new users from adding dependencies on the old one, and incrementally switch usage from old to new.

12 "The Phoenix Project"

13 It's difficult to get a precise count, but the number of such teams is almost certainly fewer than 10.

14 Cloud interfaces are a different story.

15 We don't think we've seen anything do this particularly smoothly, but the inter-repository dependencies/virtual monorepo idea is clearly in the air.

16 Or you have the willingness and capability to customize your VCS—and maintain that customization for the lifetime of your codebase/organization. Then again, maybe don't plan on that as an option, that is a lot of overhead.

Chapter 17. Code Search

Written by Alexander Neubeck and Ben St. John

Edited by Lisa Carey

Code Search is a tool for browsing and searching code at Google that consists of a frontend UI and various backend elements. Like many of the development tools at Google, it arose directly out of a need to scale to the size of the codebase. Code Search began as a combination of a grep-type tool[1] for internal code with the ranking and UI of external Code Search.[2] Its place as a key tool for Google developers was cemented by the integration of Kythe/Grok,[3] which added cross-references and the ability to jump to symbol definitions.

That integration changed its focus from searching to browsing code, and later development of Code Search was partly guided by a principle of "answering the next question about code in a single click." Now such questions as "Where is this symbol defined?", "Where is it used?", "How do I include it?", "When was it added to the code base?", and even ones like "Fleet-wide, how many CPU cycles does it consume?" are all answerable with one or two clicks.

In contrast to integrated development environments (IDEs) or code editors, Code Search is optimized for the use case of reading, understanding, and exploring code at scale. To do so, it relies heavily on cloud-based backends for searching content and resolving cross-references.

In this chapter, we'll look at Code Search in more detail, including how Googlers use it as part of their developer workflows, why we chose to develop a separate web tool for code searching, and examine how it addresses the challenges of searching and browsing code at Google repository scale.

The Code Search UI

The search box is a central element of the Code Search UI (see Figure 17-1), and like web search, it has "suggestions" that developers can use for quick navigation to files, symbols, or directories. For more complex use cases, a results page with code snippets is returned. The search itself can be thought of as an instant "find in files" (like the Unix `grep` command) with relevance

ranking and some code-specific enhancements like proper syntax highlighting, scope awareness, and awareness of comments and string literals. Search is also available from the command line and can be incorporated into other tools via a Remote Procedure Call (RPC) API. This comes in handy when post-processing is required or if the result set is too large for manual inspection.

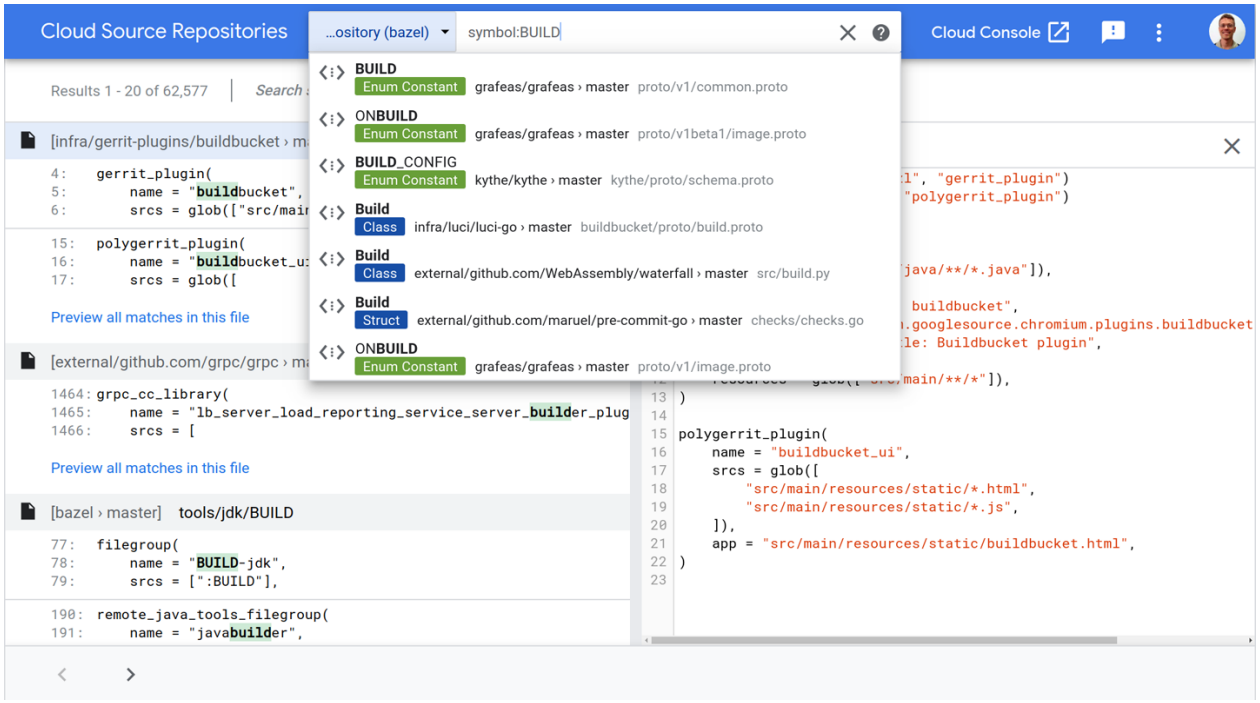

Figure 17-1. The Code Search UI

When viewing a single file, most tokens are clickable to let the user quickly navigate to related information. For example, a function call will link to its function definition, an imported filename to the actual source file, or a bug ID in a comment to the corresponding bug report. This is powered by compiler-based indexing tools like Kythe. Clicking the symbol name opens a panel with all the places the symbol is used. Similarly, hovering over local variables in a function will highlight all occurrences of that variable in the implementation.

Code Search also shows the history of a file, via its integration with Piper (see Chapter 16). This means seeing older versions of the file, which changes have affected it, who wrote them, jumping to them in Critique (see Chapter 19), diffing versions of files, and the classic "blame" view if desired. Even deleted files can be seen from a directory view.

How Do Googlers Use Code Search?

Although similar functionality is available in other tools, Googlers still make heavy use of the Code Search UI, for searching and file viewing, and ultimately for understanding code.[4] The tasks engineers try to complete with Code Search can be thought of answering questions about code, and recurring intents become visible.[5]

Where?

About 16% of Code Searches try to answer the question where a specific piece of information exists in the codebase; for example, a function definition or configuration, all usages of an API, or just where a specific file is in the repository. These questions are very targeted and can be very precisely answered with either search queries or by following semantic links, like "jump to symbol definition." Such questions often arise during larger tasks like refactorings/cleanups or when collaborating with other engineers on a project. Therefore, it is essential that these small knowledge gaps are addressed efficiently.

Code Search provides two ways of helping: ranking the results, and a rich query language. Ranking addresses the common cases, and searches can be made very specific (e.g., restricting code paths, excluding languages, only considering functions) to deal with rarer cases.

The UI makes it easy to share a Code Search result with colleagues. So, for code reviews, you can simply include the link, for example, "Have you considered using this specialized hash map: cool_hash.h?". This is also very useful for documentation, in bug reports, and in post-mortems, and is the canonical way of referring to code within Google. Even older versions of the code can be referenced, so links can stay valid as the codebase evolves.

What?

Roughly one quarter of Code Searches are classic file browsing, to answer the question of what a specific part of the codebase is doing. These kinds of tasks are usually more exploratory, rather than locating a specific result. This is using Code Search to read the source, to better understand code before making a change, or to be able to understand someone else's change.

To ease these kinds of tasks, Code Search introduced browsing via call hierarchies and quick navigation between related files (e.g., between header, implementation, test, and build files). This is about understanding code by easily answering each of the many questions a developer has when looking at it

How?

The most frequent use case, about one third of Code Searches, are about seeing examples of how others have done something. Typically, a developer has already found a specific API (e.g., how to read a file from remote storage) and wants to see how the API should be applied to a particular problem (e.g., how to set up the remote connection robustly and handle certain types of errors). Code Search is also used to find the proper library for specific problems in the first place (e.g., how to compute a fingerprint for integer values efficiently) and then pick the most appropriate implementation. For these kinds of tasks, a combination of searches and cross-reference browsing are typical.

Why?

Related to *what* code is doing, there are more targeted queries around *why* code is behaving differently than expected. About 16% of Code Searches try to answer the question of why a certain piece of code was added, or why it behaves in a certain way. Such questions often arise during debugging; for example, why does an error occur under these particular circumstances.

An important capability here is being able to search and explore the exact state of the code base at a particular point in time. When debugging a production issue, this can mean working with a state of the codebase that is weeks or months old, while debugging test failures for new code usually means working with changes that are only minutes old. Both are possible with Code Search.

Who and When?

About 8% of Code Searches try to answer questions around who or when someone introduced a certain piece of code, interacting with the version control system. For example, it's possible to see when a particular line was introduced (like Git's "blame"), and jump to the relevant code review. This history panel can also be very useful in finding the best person to ask about the code, or to review a change to it.[6]

Why a Separate Web Tool?

Outside Google, most of the aforementioned investigations are done within a local IDE. So, why yet another tool?

Scale

The first answer is that the Google codebase is so large that a local copy of the full codebase—a prerequisite for most IDEs—simply doesn't fit on a single machine. Even before this fundamental barrier is hit, there is a cost to building local search and cross-reference indices for each developer, a cost often paid at IDE startup, slowing developer velocity. Or, without an index, one-off searches (e.g., with `grep`) can become painfully slow. A centralized search index means doing this work once, upfront, and means investments in the process benefit everyone. For example, the Code Search index is incrementally updated with every submitted change, enabling index construction with linear cost.[7]

In normal web search, fast-changing current events are mixed with more slowly changing items, such as stable Wikipedia pages. The same technique can be extended to searching code, making indexing incremental, which reduces its cost and allows changes to the codebase to be visible to everyone instantly. When a code change is submitted, only the actual files touched need to be reindexed, which allows parallel and independent updates to the global index.

Unfortunately, the cross-reference index cannot be instantly updated in the same way. Incrementality isn't possible for it, as any code change can potentially influence the entire code base, and in practice often does affect thousands of files. Many (nearly all of Google's) full binaries need to be built[8] (or at least analyzed) to determine the full semantic structure. It uses a ton of compute resources to produce the index daily (the current frequency). The discrepancy between the instant search index and the daily cross-reference index is a source of rare but recurring issues for users.

Zero Setup Global Code View

Being able to instantly and effectively browse the entire codebase means that it's very easy to find relevant libraries to reuse, and good examples to copy. For IDEs that construct indices at startup, there is a pressure to have a small project or visible scope to reduce this time and avoid flooding tools like autocomplete with noise. With the Code Search web UI, there is no setup required (e.g., project descriptions, build environment), so it's also very easy and fast to learn about code, wherever it occurs, which improves developer

efficiency. There's also no danger of missing code dependencies; for example, when updating an API, reducing merge and library versioning issues.

Specialization

Perhaps surprisingly, one advantage of Code Search is that it is *not* an IDE. This means that the user experience (UX) can be optimized for browsing and understanding code, rather than editing it, which is usually the bulk of an IDE (e.g., keyboard shortcuts, menus, mouse clicks, and even screen space). For example, because there isn't an editor's text cursor, every mouse click on a symbol can be made meaningful (e.g., show all usages or jump to definition), rather than as a way to move the cursor. This advantage is so large that it's extremely common for developers to have multiple Code Search tabs open at the same time as their editor.

Integration with Other Developer Tools

Because it is the primary way to view source code, Code Search is the logical platform for exposing information about source code. It frees up tool creators from needing to create a UI for their results, and ensures the entire developer audience will know of their work without needing to advertise it. Many analyses run regularly over the entire Google codebase, and their results are usually surfaced in Code Search. For example, for many languages we can detect "dead" (uncalled) code, and mark it as such when the file is browsed.

In the other direction, the Code Search link to a source file is considered its canonical "location." This is useful for many developer tools (see Figure 17-2). For example, log file lines typically contain the filename and line number of the logging statement. The production log viewer uses a Code Search link to connect the log statement back to the producing code. Depending on the available information, this can be a direct link to a file at a specific revision, or a basic filename search with the corresponding line number. If there is only one matching file, it is opened at the corresponding line number. Otherwise, snippets of the desired line in each of the matching files are rendered.

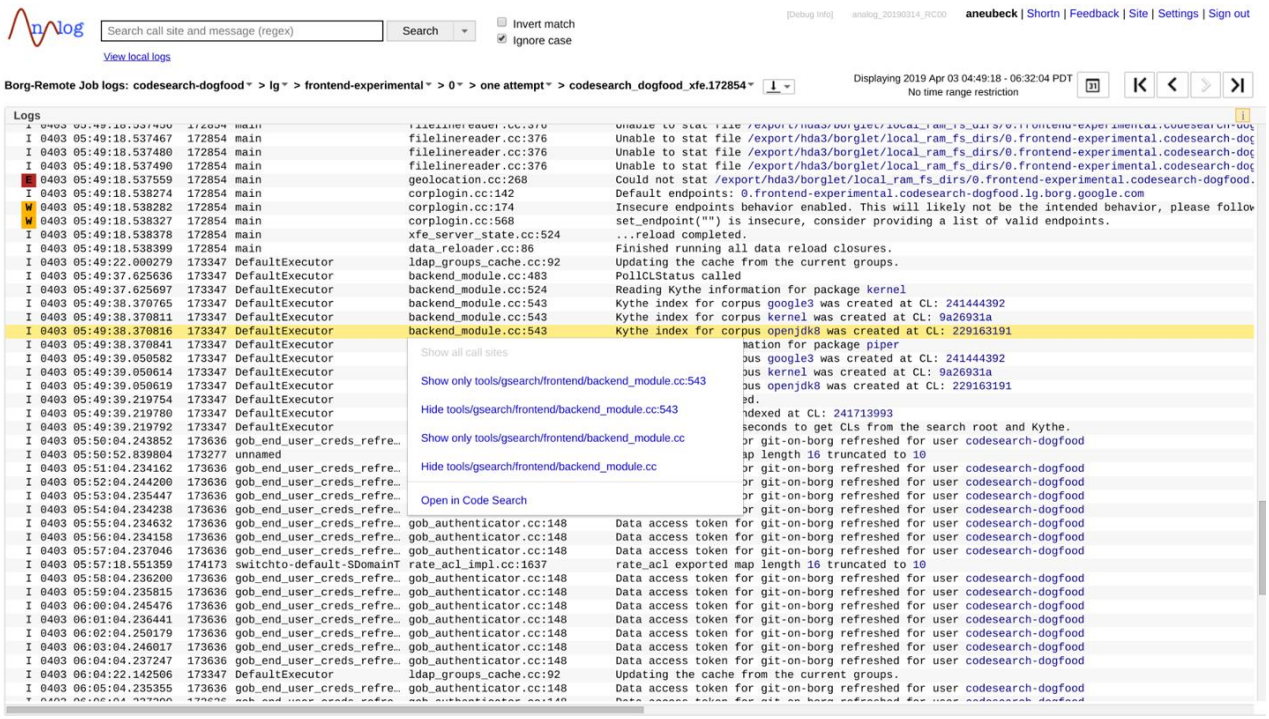

Figure 17-2. Code Search integration in a log viewer

Similarly, stack frames are linked back to source code whether they are shown within a crash reporting tool or in log output, as shown in Figure 17-3. Depending on the programming language, the link will utilize a filename or symbol search. Because the snapshot of the repository at which the crashing binary was built is known, the search can actually be restricted to exactly this version. That way, links remain valid for a long time period, even if the corresponding code is later refactored or deleted.

Figure 17-3. Code Search integration in stack frames

Compilation errors and tests also typically refer back to a code location (e.g., test X in *file* at *line*). These can be linkified even for unsubmitted code given that most development happens in specific cloud-visible workspaces which are accessible and searchable by Code Search.

Finally, codelabs and other documentation refer to APIs, examples, and implementations. Such links can be search queries referencing a specific class or function, which remain valid when the file structure changes. For code

snippets, the most recent implementation at head can easily be embedded into a documentation page, as demonstrated in Figure 17-4, without the need to pollute the source file with additional documentation markers.

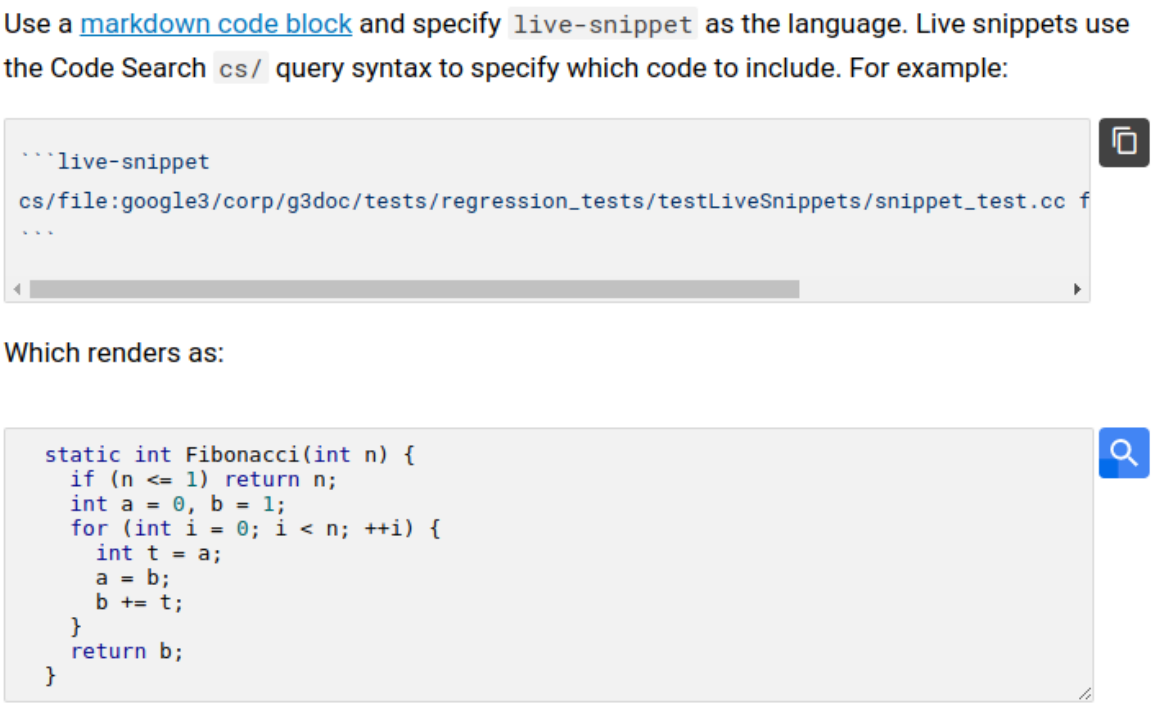

Figure 17-4. Code Search integration in documentation

API Exposure

Code Search exposes its search, cross-reference, and syntax highlighting APIs to tools, so tool developers can bring those capabilities into their tools without needing to reimplement them. Further, plug-ins have been written to provide search and cross-references to editors and IDEs such as vim, emacs, and IntelliJ. These plug-ins restore some of the power lost due to being unable to locally index the codebase, and give back some developer productivity.

Impact of Scale on Design

In the previous section, we looked at various aspects of the Code Search UI and why it's worthwhile having a separate tool for browsing code. In the following sections, we look a bit behind the scenes of the implementation. We first discuss the primary challenge, scaling, and then some of the ways the large scale complicates making a good product for searching and browsing code. After that, we detail how we addressed some of those challenges, and what trade-offs were made when building Code Search.

The biggest[9] scaling challenge for searching code is the corpus size. For a small repository of a couple megabytes, a brute-force search with `grep` search will do. When hundreds of megabytes need to be searched, a simple local index can speed up search by an order of magnitude or more. When gigabytes or terabytes of source code need to be searched, a cloud-hosted solution with multiple machines can keep search times reasonable. The utility of a central solution increases with the number of developers using it, and the size of the code space.

Search Query Latency

Although we take as a given that a fast and responsive UI is better for the user, low latency doesn't come for free. To justify the effort, one can weigh it against the saved engineering time across all users. Within Google, we process much more than one million search queries from developers within Code Search *per day*. For one million queries, an increase of just one second per search request corresponds to about 35 idle full-time engineers every day. In contrast, the search backend can be built and maintained with roughly a tenth of these engineers. This means that with about 100,000 queries per day (corresponding to less than 5,000 developers), just the one-second latency argument is something of a break-even point.

In reality, the productivity loss doesn't simply increase linearly with latency. A UI is considered responsive if latencies are below 200 ms. But after just one second, the developer's attention often begins to drift. If another 10 seconds pass, the developer is likely to switch context completely, which is generally recognized to have high productivity costs. The best way to keep a developer in the productive "flow" state is by targeting sub–200 ms end-to-end latency for all frequent operations and investing in the corresponding backends.

A large number of Code Search queries are performed in order to navigate the codebase. Ideally, the "next" file is only a click away (e.g., for included files, or symbol definitions), but for general navigation, instead of using the classical file tree, it can be much faster to simply search for the desired file or symbol, ideally without needing to fully specify it, and suggestions are provided for partial text. This becomes increasingly true as the code base (and file tree) grows.

Normal navigation to a specific file in another folder or project requires several user interactions. With search, just a couple of keystrokes can be sufficient to get to the relevant file. To make search this effective, additional

information about the search context (e.g., the currently viewed file) can be provided to the search backend. The context can restrict the search to files of a specific project, or influence ranking by preferring files that are in proximity to other files or directories. In the Code Search UI,[10] the user can predefine multiple contexts and quickly switch between them as needed. In editors, the open or edited files are implicitly used as context to prioritize search results in their proximity.

One could consider the power of the search query language (e.g., specifying files, using regular expressions) as another criteria; we discuss this in the trade-offs section a little later in the chapter.

Index Latency

Most of the time, developers won't notice when indices are out of date. They only care about a small subset of code, and even for that they generally won't *know* whether there is more recent code. However, for the cases in which they wrote or reviewed the corresponding change, being out of sync can cause a lot of confusion. It tends not to matter whether the change was a small fix, a refactoring, or a completely new piece of code, developers simply expect a consistent view, such as they experience in their IDE for a small project.

When writing code, instant indexing of modified code is expected. When new files, functions, or classes are added, not being able to find them is frustrating, and breaks the normal workflow for developers used to perfect cross-referencing. Another example are search-and-replace–based refactorings. It is not only more convenient when the removed code immediately disappears from the search results, but it is also essential that subsequent refactorings take the new state into account. When working with a centralized VCS, a developer might need instant indexing for submitted code if the previous change is no longer part of the locally modified file set.

Conversely, sometimes it's useful to be able to go back in time to a previous snapshot of the code; in other words, a release. During an incident, a discrepancy between the index and the running code can be especially problematic because it can hide real causes or introduce irrelevant distractions. This is a problem for cross-references because the current technology for building an index at Google's scale simply takes hours, and the complexity means that only one "version" of the index is kept. Although some patching can be done to align new code with an old index, this is still an issue to be solved.

Google's Implementation

Google's particular implementation of Code Search is tailored to the unique characteristics of its codebase, and the previous section outlined our design constraints for creating a robust and responsive index. The following section outlines how the Code Search team implemented and released its tool to Google developers.

Search Index

Google's codebase is a special challenge for Code Search due to its sheer size. In the early days, a trigram-based approach was taken. Russ Cox subsequently open sourced a simplified version. Currently, Code Search indexes about 1.5 TB of content and processes about 200 queries per second with a median server-side search latency of less than 50 ms and a median indexing latency (time between code commit and visibility in the index) of less than 10 seconds.

Let's roughly estimate the resource requirements to achieve this performance with a `grep`-based brute-force solution. The RE2 library we use for regular expression matching processes about 100 MB/sec for data in RAM. Given a time window of 50 ms, 300,000 cores would be needed to crunch through the 1.5 TB of data. Because in most cases simple substring searches are sufficient, one could replace the regular expression matching with a special substring search that can process about 1 GB/sec[11], [12] under certain conditions, reducing the number of cores by 10 time. So far, we have looked at just the resource requirements for processing a single query within 50 ms. If we're getting 200 requests per second, 10 of those will be simultaneously active in that 50 ms window, bringing us back to 300,000 cores just for substring search.

Although this estimate ignores that the search can stop once a certain number of results are found or that file restrictions can be evaluated much more effectively than content searches, it doesn't take communication overhead, ranking, or the fan out to tens of thousands of machines into account either. But it shows quite well the scale involved and why Google's Code Search team continuously invests into improving indexing. Over the years, our index changed from the original trigram-based solution, through a custom suffix array-based solution, to the current sparse n-gram solution. This latest solution is more than 500 times more efficient than the brute-force solution while being capable of also answering regular expression searches at blazing speed.

One reason we moved from a suffix array–based solution to a token-based n-gram solution was to take advantage of Google's primary indexing and search stack. With a suffix array–based solution, building and distributing the custom indices becomes a challenge in and of itself. By utilizing "standard" technology, we benefit from all the advances in reverse index construction, encoding, and serving made by the core search team. Instant indexing is another feature that exists in standard search stacks, and by itself is a big challenge when solving it at scale.

Relying on standard technology is a trade-off between implementation simplicity and performance. Even though Google's Code Search implementation is based on standard reverse indices, the actual retrieval, matching, and scoring are highly customized and optimized. Some of the more advanced Code Search features wouldn't be possible otherwise. To index the history of file revisions, we came up with a custom compression scheme in which indexing the full history increased the resource consumption by a factor of just 2.5.

In the early days, Code Search served all data from memory. With the growing index size, we moved the inverted index to flash. Although flash storage is at least an order of magnitude cheaper than memory, its access latency is at least two orders of magnitude higher. So, indices that work well in memory might not be suitable when served from flash. For instance, the original trigram index requires fetching not only a large number of reverse indices from flash, but also quite large ones. With n-gram schemes, both the number of inverse indices and their size can be reduced at the expense of a larger index.

To support local workspaces (which have a small delta from the global repository), we have multiple machines doing simple brute-force searches. The workspace data is loaded on the first request and then kept in sync by listening for file changes. When we run out of memory, we remove the least recent workspace from the machines. The unchanged documents are searched with our history index. Thereby the search is implicitly restricted to the repository state to which the workspace is synced.

Ranking

For a very small codebase, ranking doesn't provide much benefit, because there aren't many results anyway. But the larger the codebase becomes, the more results will be found and the more important ranking becomes. In Google's codebase, any short substring will occur thousands, if not millions,

of times. Without ranking, the user either must check all of those results in order to find the correct one, or must refine the query[13] further until the result set is reduced to just a handful of files. Both options waste the developer's time.

Ranking typically starts with a scoring function, which maps a set of features of each file ("signals") to some number: the higher the score, the better the result. The goal of the search is then to find the top *N* results as efficiently as possible. Typically, one distinguishes between two types of signals: those that depend only on the document ("query *independent*") and those that depend on the search query and how it matches the document ("query *dependent*"). The filename length or the programming language of a file would be examples of query independent signals, whereas whether a match is a function definition or a string literal is a query dependent signal.

QUERY INDEPENDENT SIGNALS

Some of the most important query independent signals are the number of file views and the amount of references to a file. File views are important because they indicate which files developers consider important and are therefore more likely to want to find. For instance, utility functions in base libraries have a high view count. It doesn't matter whether the library is already stable and isn't changed anymore or whether the library is being actively developed. The biggest downside of this signal is the feedback loop it creates. By scoring frequently viewed documents higher, the chance increases that developers will look at them and decreases the chance of other documents to make it into the top *N*. This problem is known as *exploitation versus exploration* to which various solutions exists (e.g., advanced A/B search experiments or curation of training data). In practice, it doesn't seem harmful to somewhat over-show high-scoring item: they are simply ignored when irrelevant, and taken if a generic example is needed. However, it is a problem for new files, which don't yet have enough information for a good signal.[14]

We also use the number of references to a file, which parallels the original page rank algorithm, by replacing web links as references with the various kinds of "include/import" statements present in most languages. We can extend the concept up to build dependencies (library/module level references) and down to functions and classes. This global relevance is often referred to as the document's "priority."

When using references for ranking, one must be aware of two challenges. First you must be able to extract reference information reliably. In the early days, Google's Code Search extracted include/import statements with simple

regular expressions and then applied heuristics to convert them into full file paths. With the growing complexity of a codebase, such heuristics became error prone and challenging to maintain. Internally, we replaced this part with correct information from the Kythe graph.

Large-scale refactorings, such as open sourcing core libraries, present a second challenge. Such changes don't happen atomically in a single code update; rather, they need to be rolled out in multiple stages. Typically, indirections are introduced, hiding, for example, the move of files from usages. These kinds of indirections reduce the page rank of moved files and make it more difficult for developers to discover the new location. Additionally, file views usually become lost when files are moved, making the situation even worse. Because such global restructurings of the codebase are comparatively rare (most interfaces move rarely), the simplest solution is to manually boost files during such transition periods. (Or wait until the migration completes and for the natural processes to up-rank the file in its new location.)

QUERY DEPENDENT SIGNALS

Query independent signals can be computed offline, so computational cost isn't a major concern, although it can be high. For example, for the "page" rank, the signal depends on the whole corpus and requires a MapReduce-like batch processing to calculate. Query *dependent* signals, which must be calculated for each query, should be cheap to compute. This means that they are restricted to the query and information quickly accessible from the index.

Unlike web search, we don't just match on tokens. However, if there are clean token matches (that is, the search term matches with content with some form of breaks such as whitespace, around it), a further boost is applied and case sensitivity is considered. This means, for example, a search for "Point" will score higher against "*Point* *p" than against "ap*point*ed to the council."

For convenience, a default search matches filename and qualified symbols[15] in addition to the actual file content. A user *can* specify the particular kind of match, but they don't need to. The scoring boosts symbol and filename matches over normal content matches, to reflect the inferred intent of the developer. Just as with web searches, developers can add more terms to the search to make queries more specific. It's very common for a query to be "qualified" with hints about the filename (e.g., "base" or "myproject"). Scoring leverages this by boosting results where much of the query occurs in the full path of the potential result, putting such results ahead of those that contain only the words in random places in their content.

RETRIEVAL

Before a document can be scored, candidates that are likely to match the search query are found. This phase is called retrieval. Because it is not practical to retrieve all documents, but only retrieved documents can be scored, retrieval and scoring must work well together to find the most relevant documents. A typical example is to search for a class name. Depending on the popularity of the class, it can have thousands of usages, but potentially only one definition. If the search was not explicitly restricted to class definitions, retrieval of a fixed number of results might stop before the file with the single definition was reached. Obviously, the problem becomes more challenging as the codebase grows.

The main challenge for the retrieval phase is to find the few highly relevant files among the bulk of less interesting ones. One solution which works quite well is called *supplemental retrieval*. The idea is to rewrite the original query into more specialized ones. In our example, this would mean that a supplemental query would restrict the search to only definitions and filenames and add the newly retrieved documents to the output of the retrieval phase. In a naive implementation of supplemental retrieval, more documents need to be scored, but the additional partial scoring information gained can be used to fully evaluate only the most promising documents from the retrieval phase.

RESULT DIVERSITY

Another aspect of search is diversity of results, meaning trying to give the best results in multiple categories. A simple example would be to provide both the Java and Python matches for a simple function name, rather than filling the first page of results with one or the other.

This is especially important when the intent of the user is not clear. One of the challenges with diversity is that there are many different categories—like functions, classes, filenames, local results, usages, tests, examples, and so on—into which results can be grouped, but that there isn't a lot of space in the UI to show results for all of them or even all combinations, nor would it always be desirable. Google's Code Search doesn't do this as well as web search does, but in the drop-down list of suggested results (like the autocompletions of web search) is tweaked to provide a diverse set of top filenames, definitions, and matches in the user's current workspace.

Selected Trade-Offs

Implementing Code Search within a codebase the size of Google's, and keeping it responsive, involved making a variety of trade-offs. These are noted in the following section.

Completeness: Repository at Head

We've seen that a larger codebase has negative consequences for search; for example, slower and more expensive indexing, slower queries, and noisier results. Can these costs be reduced by sacrificing completeness; in other words, leaving some content out of the index? The answer is yes, but with caution. Nontext files (binaries, images, videos, sound, etc.) are usually not meant to be read by humans and are dropped apart from their filename. Because they are huge, this saves a lot of resources. A more borderline case involves generated JavaScript files. Due to obfuscation and the loss of structure, they are pretty much unreadable for humans, so excluding them from the index is usually a good trade-off, reducing indexing resources and noise at the cost of completeness. Empirically, multimegabyte files rarely contain information relevant for developers, so excluding extreme cases is probably the correct choice.

However, dropping files from the index has one big drawback. For developers to rely on Code Search, they need to be able to trust it. Unfortunately, it is in general impossible to give feedback about incomplete search results for a specific search, if the dropped files weren't indexed in the first place. The resulting confusion and productivity loss for developers is a high price to pay for the saved resources. Even if developers are fully aware of the limitations, if they still need to perform their search, they will do so in an ad hoc and error-prone way. Given these rare, but potentially high costs, we choose to err on the side of indexing too much, with quite high limits that are mostly picked to prevent abuse and guarantee system stability rather than to save resources.

In the other direction, generated files aren't in the codebase but would often be useful to index. Currently they are not, because indexing them would require integrating the tools and configuration to create them, which would be a massive source of complexity, confusion, and latency.

Completeness: All versus Most-Relevant Results

Normal search sacrifices completeness for speed, essentially gambling that ranking will ensure that the top results will contain all of the desired results. And indeed, for Code Search, ranked search is the more common case in which the user is looking for one particular thing, such as a function definition, potentially among millions of matches. However, sometimes developers want *all* results; for example, finding all occurrences of a particular symbol for refactoring. Needing all results is common for analysis, tooling, or refactoring, such as a global search and replace. The need to deliver all results is a fundamental difference to web search in which many shortcuts can be taken, such as only consider highly ranked items.

Being able to deliver *all* results for very large result sets has high cost, but we felt it was required for tooling, and for developers to trust the results. However, because for most queries only a few results are relevant (either there are only a few matches[16] or only a few are interesting) we didn't want to sacrifice average speed for potential completeness.

To achieve both goals with one architecture, we split the codebase into shards with files ordered by their priority. Then, we usually need to consider only the matches to high priority files from each chunk. This is similar to how web search works. However, if requested, Code Search can fetch *all* results from each chunk, to guarantee finding all results.[17] This lets us address both use cases, without typical searches being slowed down by the less frequently used capability of returning large, complete results sets. Results can also then be delivered in alphabetical order, rather than ranked, which is useful for some tools.

So, here the trade-off was a more complex implementation and API versus greater capabilities, rather than the more obvious latency versus completeness.

Completeness: Head versus Branches versus All History versus Workspaces

Related to the dimension of corpus size is the question of which code versions should be indexed: specifically, whether anything more than the current snapshot of code ("head") should be indexed. System complexity, resource consumption, and overall cost increase drastically if more than a single file revision is indexed. To our knowledge, no IDE indexes anything but the current version of code. When looking at distributed version control

systems like Git or Mercurial, a lot of their efficiency comes from the compression of their historical data. But the compactness of these representations becomes lost when constructing reverse indices. Another issue is that it is difficult to efficiently index graph structures, which are the basis for Distributed Version Control Systems.

Although it is difficult to index multiple versions of a repository, doing so allows the exploration of how code has changed and finding deleted code. Within Google, Code Search indexes the (linear) Piper history. This means that the codebase can be searched at an arbitrary snapshot of the code, for deleted code, or even for code authored by certain people.

One big benefit is that obsolete code can now simply be deleted from the codebase. Before, code was often moved into directories marked as obsolete so that it could still be found later. The full history index also laid the foundation for searching effectively in people's workspaces (unsubmitted changes) which are synced to a specific snapshot of the codebase. For the future, a historical index opens up the possibility of interesting signals to use when ranking, such as authorship, code activity, and so on.

Workspaces are very different from the global repository:

- Each developer can have their own workspaces.
- There are usually a small number of changed files within a workspace.
- The files being worked on are changing frequently.
- A workspace exists only for a relatively short time period.

To provide value, a workspace index must reflect exactly the current state of the workspace.

Expressiveness: Token versus Substring versus Regex

The effect of scale is greatly influenced by the supported search feature set. Code Search supports regular expression (regex) search, which adds power to the query language, allowing whole groups of terms to be specified, or excluded, and they can be used on any text, which is especially helpful for documents and languages for which deeper semantic tools don't exist.

Developers are also used to using regular expressions in other tools (e.g., `grep`) and contexts, so they provide powerful search without adding to a developer's cognitive load. This power comes at a cost given that creating an index to query them efficiently is challenging. What simpler options exist?

A token-based index (i.e., words) scales well because it stores only a fraction of the actual source code and is well supported by standard search engines. The downside is that many use cases are tricky or even impossible to realize efficiently with a token-based index when dealing with source code, which attaches meaning to many characters typically ignored when tokenizing. For example, searching for "function()" versus "function(x)", "(x ^ y)", or "=== myClass" is difficult or impossible in most token-based search.

Another problem of tokenization is that tokenization of code identifiers is ill defined. Identifiers can be written in many ways, such as CamelCase, snake_case, or even justmashedtogether without any word separator. Finding an identifier when remembering only some of the words is a challenge for a token-based index.

Tokenization also typically doesn't care about the case of letters ("r" versus "R"), and will often blur words; for example, reducing "searching" and "searched" to the same stem token search. This lack of precision is a significant problem when searching code. Finally, tokenization makes it impossible to search on whitespace or other word delimiters (commas, parentheses), which can be very important in code.

A next step up[18] in searching power is full substring search in which any sequence of characters can be searched for. One fairly efficient way to provide this is via a trigram-based index.[19] In its simplest form, the resulting index size is still much smaller than the original source code size. However, the small size comes at the cost of relatively low recall accuracy compared to other substring indices. This means slower queries because the nonmatches need to be filtered out of the result set. This is where a good compromise between index size, search latency, and resource consumption must be found that depends heavily on codebase size, resource availability, and searches per second.

If a substring index is available, it's easy to extend it to allow regular expression searches. The basic idea is to convert the regular expression automaton into a set of substring searches. This conversion is straightforward for a trigram index and can be generalized to other substring indices. Because there is no perfect regular expression index, it will always be possible to construct queries which result in a brute-force search. However, given that only a small fraction of user queries are complex regular expressions, in practice the approximation via substring indices works very well.

Conclusion

Code Search grew from an organic replacement for `grep` into a central tool boosting developer productivity, leveraging Google's web search technology along the way. What does this mean for you, though? If you are on a small project that easily fits in your IDE, probably not much. If you are responsible for the productivity of engineers on a larger codebase, there are probably some insights to be gained.

The most important one is perhaps obvious: understanding code is key to developing and maintaining it, and this means that investing in understanding code will yield dividends that might be difficult to measure, but are real. Every feature we added to Code Search was and is used by devs to help them in their daily work (admittedly some more than others). Two of the most important features, Kythe integration (i.e., adding semantic code understanding) and finding working examples, are also the most clearly tied to understanding code (versus, for example, finding it, or seeing how it's changed). In terms of tool impact, no one uses a tool that they don't know exists, so it is also important to make developers aware of the available tooling—at Google it is part of "Noogler" training, the onboarding training for newly hired software engineers.

For you, this might mean setting up a standard indexing profile for IDEs, sharing knowledge about egrep, running ctags, or setting up some custom indexing tooling, like Code Search. Whatever you do, it will almost certainly be used, and used more, and in different ways than you expected—and your developers will benefit.

TL;DRs

- Helping your developers understand code can be a big boost to engineering productivity. At Google, the key tool for this is Code Search.
- Code Search has additional value as a basis for other tools and as a central, standard place that all documentation and developer tools link to.
- The huge size of the Google codebase made a custom tool—as opposed to, for example, `grep` or an IDE's indexing—necessary.
- As an interactive tool, Code Search must be fast, allowing a "question and answer" workflow. It is expected to have low latency in every respect: search, browsing, and indexing.

- It will be widely used only if it is trusted, and will be trusted only if it indexes all code, gives all results, and gives the desired results first. However, earlier, less powerful, versions were both useful and used, as long as their limits were understood.

1 gsearch originally ran on Jeff Dean's personal computer, which once caused company-wide distress when he went on vacation and it was shut down!

2 Shut down in 2013, see *https://en.wikipedia.org/wiki/Google_Code_Search*

3 Now known as Kythe, a service that provides cross-references (among other things): the uses of a particular code symbol—for example, a function—using the full build information to disambiguate it from other ones with the same name.

4 There is an interesting virtuous cycle that a ubiquitous code browser encourages: writing code that is easy to browse. This can mean things like not nesting hierarchies too deep, which requires many clicks to move from call sites to actual implementation, and using named types rather than generic things like strings or integers, because it's then easy to find all usages.

5 "How Developers Use Code Search"

6 That said, given the rate of commits for machine-generated changes, naive "blame" tracking has less value than it does in more change-averse ecosystems.

7 For comparison, the model of "every developer has their own IDE on their own workspace do the indexing calculation" scales roughly quadratically: developers produce a roughly constant amount of code per unit time, so the codebase scales linearly (even with a fixed number of developers). A linear number of IDEs do linearly more work each time—this is not a recipe for good scaling.

8 Kythe instruments the build workflow to extract semantic nodes and edges from source code. This extraction process collects partial cross-reference graphs for each individual build rule. In a subsequent phase, these partial graphs are merged into one global graph and its representation is optimized for the most common queries (go-to-definition, find all usages, fetch all decorations for a file). Each phase—extraction and post processing—is roughly as expensive as a full build; for example, in case of Chromium the construction of the Kythe index is done in about six hours in a distributed

setup and therefore too costly to be constructed by every developer on their own workstation. This computational cost is the why the Kythe index is computed only once per day.

9 Because queries are independent, more users can be addressed by having more servers.

10 The Code Search UI does also have a classical file tree, so navigating this way is also possible.

11 *https://blog.scalyr.com/2014/05/searching-20-gbsec-systems-engineering-before-algorithms/*

12 *http://volnitsky.com/project/str_search/*

13 In contrast to web search, adding more characters to a Code Search query always reduces the result set (apart from a few rare exceptions via regular expression terms).

14 This could likely be somewhat corrected by using recency in some form as a signal, perhaps doing something similar to web search dealing with new pages, but we don't yet do so.

15 In programming languages, a symbol such as a function "Alert" often is defined in a particular scope, such as a class ("Monitor") or namespace ("absl"). The qualified name might then be absl::Monitor::Alert, and this is findable, even if it doesn't occur in the actual text.

16 An analysis of queries showed that about one-third of user searches have less than 20 results.

17 In practice, even more happens behind the scenes so that responses don't become painfully huge and developers don't bring down the whole system by making searches that match nearly everything (imagine searching for the letter 'i" or a single space).

18 There are other intermediate varieties, such as building a prefix/suffix index, but generally they provide less expressiveness in search queries while still having high complexity and indexing costs.

19 Cox, Russ. 2012. "Regular Expression Matching with a Trigram Index or How Google Code Search Worked."

Chapter 18. Build Systems and Build Philosophy

Written by Erik Kueffler

Edited by Lisa Carey

If you ask Google engineers what they like most about working at Google (besides the free food and cool products), you might hear something surprising: engineers love the build system.[1] Google has spent a tremendous amount of engineering effort over its lifetime in creating its own build system from the ground up, with the goal of ensuring that our engineers are able to quickly and reliably build code. The effort has been so successful that Blaze, the main component of the build system, has been reimplemented several different times by ex-Googlers who have left the company.[2] In 2015, Google finally open sourced an implementation of Blaze named Bazel.

Purpose of a Build System

Fundamentally, all build systems have a straightforward purpose: they transform the source code written by engineers into executable binaries that can be read by machines. A good build system will generally try to optimize for two important properties:

Fast

> A developer should be able to type a single command to run the build and get back the resulting binary, often in as little as a few seconds.

Correct

> Every time any developer runs a build on any machine, they should get the same result (assuming that the source files and other inputs are the same).

Many older build systems attempt to make trade-offs between speed and correctness by taking shortcuts that can lead to inconsistent builds. Bazel's main objective is to avoid having to choose between speed and correctness, providing a build system structured to ensure that it's always possible to build code efficiently and consistently.

Build systems aren't just for humans; they also allow machines to create builds automatically, whether for testing or for releases to production. In fact, the large majority of builds at Google are triggered automatically rather than directly by engineers. Nearly all of our development tools tie into the build system in some way, giving huge amounts of value to everyone working on our codebase. Here's a small sample of workflows that take advantage of our automated build system:

- Code is automatically built, tested, and pushed to production without any human intervention. Different teams do this at different rates: some teams push weekly, others daily, and others as fast as the system can create and validate new builds. (See XREF(Continuous Deployment)).
- Developer changes are automatically tested when they're sent for code review (see Chapter 19) so that both the author and reviewer can immediately see any build or test issues caused by the change.
- Changes are tested again immediately before merging them into the trunk, making it much more difficult to submit breaking changes.
- Authors of low-level libraries are able to test their changes across the entire codebase, ensuring that their changes are safe across millions of tests and binaries.
- Engineers are able to create large-scale changes (LSCs) that touch tens of thousands of source files at a time (e.g., renaming a common symbol) while still being able to safely submit and test those changes. We discuss LSCs in greater detail in Chapter 22.

All of this is possible only because of Google's investment in its build system. Though Google might be unique in its scale, any organization of any size can realize similar benefits by making proper use of a modern build system. This chapter describes what Google considers to be a "modern build system" and how to use such systems.

What Happens Without a Build System?

Build systems allow your development to scale. As we'll illustrate in the next section, we run into problems of scaling without a proper build environment.

But All I Need Is a Compiler!

The need for a build system might not be immediately obvious. After all, most of us probably didn't use a build system when we were first learning to code—we probably started by invoking tools like `gcc` or `javac` directly from

the command line, or the equivalent in an integrated development environment (IDE). As long as all of our source code is in the same directory, a command like this works fine:

```
javac *.java
```

This instructs the Java compiler to take every Java source file in the current directory and turn it into a binary class file. In the simplest case, this is all that we need.

However, things become more complicated quickly as soon as our code expands. `javac` is smart enough to look in subdirectories of our current directory to find code that we import. But it has no way of finding code stored in other parts of the filesystem (perhaps a library shared by several of our projects). It also obviously only knows how to build Java code. Large systems often involve different pieces written in a variety of programming languages with webs of dependencies among those pieces, meaning no compiler for a single language can possibly build the entire system.

As soon as we end up having to deal with code from multiple languages or multiple compilation units, building code is no longer a one-step process. We now need to think about what our code depends on and build those pieces in the proper order, possibly using a different set of tools for each piece. If we change any of the dependencies, we need to repeat this process to avoid depending on stale binaries. For a codebase of even moderate size, this process quickly becomes tedious and error-prone.

The compiler also doesn't know anything about how to handle external dependencies, such as third-party JAR files in Java. Often the best we can do without a build system is to download the dependency from the internet, stick it in a `lib` folder on the hard drive, and configure the compiler to read libraries from that directory. Over time, it's easy to forget what libraries we put in there, where they came from, and whether they're still in use. And good luck keeping them up-to-date as the library maintainers release new versions.

Shell Scripts to the Rescue?

Suppose that your hobby project starts out simple enough that you can build it using just a compiler, but you begin running into some of the problems described previously. Maybe you still don't think you need a real build system and can automate away the tedious parts using some simple shell

scripts that take care of building things in the correct order. This helps out for a while, but pretty soon you start running into even more problems:

- It becomes tedious. As your system grows more complex, you begin spending almost as much time working on your build scripts as on real code. Debugging shell scripts is painful, with more and more hacks being layered on top of one another.
- It's slow. To make sure you weren't accidentally relying on stale libraries, you have your build script build every dependency in order every time you run it. You think about adding some logic to detect which parts need to be rebuilt, but that sounds awfully complex and error prone for a script. Or you think about specifying which parts need to be rebuilt each time, but then you're back to square one.
- Good news: it's time for a release! Better go figure out all the arguments you need to pass to the `jar` command to make your final build.[3] And remember how to upload it and push it out to the central repository. And build and push the documentation updates, and send out a notification to users. Hmm, maybe this calls for another script...
- Disaster! Your hard drive crashes, and now you need to recreate your entire system. You were smart enough to keep all of your source files in version control, but what about those libraries you downloaded? Can you find them all again and make sure they were the same version as when you first downloaded them? Your scripts probably depended on particular tools being installed in particular places—can you restore that same environment so that the scripts work again? What about all those environment variables you set a long time ago to get the compiler working just right and then forgot about?
- Despite the problems, your project is successful enough that you're able to begin hiring more engineers. Now you realize that it doesn't take a disaster for the previous problems to arise—you need to go through the same painful bootstrapping process every time a new developer joins your team. And despite your best efforts, there are still small differences in each person's system. Frequently, what works on one person's machine doesn't work on another's, and each time it takes a few hours of debugging tool paths or library versions to figure out where the difference is.
- You decide that you need to automate your build system. In theory, this is as simple as getting a new computer and setting it up to run your build script every night using cron. You still need to go through the painful setup process, but now you don't have the benefit of a human

brain being able to detect and resolve minor problems. Now, every morning when you get in, you see that last night's build failed because yesterday a developer made a change that worked on their system but didn't work on the automated build system. Each time it's a simple fix, but it happens so often that you end up spending a lot of time each day discovering and applying these simple fixes.

- Builds become slower and slower as the project grows. One day, while waiting for a build to complete, you gaze mournfully at the idle desktop of your coworker, who is on vacation, and wish there were a way to take advantage of all that wasted computational power.

You've run into a classic problem of scale. For a single developer working on at most a couple hundred lines of code for at most a week or two (which might have been the entire experience thus far of a junior developer who just graduated university), a compiler is all you need. Scripts can maybe take you a little bit further. But as soon as you need to coordinate across multiple developers and their machines, even a perfect build script isn't enough because it becomes very difficult to account for the minor differences in those machines. At this point, this simple approach breaks down and it's time to invest in a real build system.

Modern Build Systems

Fortunately, all of the problems we started running into have already been solved many times over by existing general-purpose build systems. Fundamentally, they aren't that different from the aforementioned script-based DIY approach we were working on: they run the same compilers under the hood, and you need to understand those underlying tools to be able to know what the build system is really doing. But these existing systems have gone through many years of development, making them far more robust and flexible than the scripts you might try hacking together yourself.

It's All About Dependencies

In looking through the previously described problems, one theme repeats over and over: managing your own code is fairly straightforward, but managing its dependencies is much more difficult (and Chapter 21 is devoted to covering this problem in detail). There are all sorts of dependencies: sometimes there's a dependency on a task (e.g., "push the documentation before I mark a release as complete"), and sometimes there's a dependency on an artifact (e.g., "I need to have the latest version of the computer vision

library to build my code"). Sometimes, you have internal dependencies on another part of your codebase, and sometimes you have external dependencies on code or data owned by another team (either in your organization or a third party). But in any case, the idea of "I need that before I can have this" is something that recurs repeatedly in the design of build systems, and managing dependencies is perhaps the most fundamental job of a build system.

Task-Based Build Systems

The shell scripts we started developing in the previous section were an example of a primitive *task-based build system*. In a task-based build system, the fundamental unit of work is the task. Each task is a script of some sort that can execute any sort of logic, and tasks specify other tasks as dependencies that must run before them. Most major build systems in use today, such as Ant, Maven, Gradle, Grunt, and Rake, are task based.

Instead of shell scripts, most modern build systems require engineers to create *buildfiles* that describe how to perform the build. Take this example from the Ant manual:

```
<project name="MyProject" default="dist" basedir=".">

  <description>

    simple example build file

  </description>

  <!-- set global properties for this build -->

  <property name="src" location="src"/>

  <property name="build" location="build"/>

  <property name="dist" location="dist"/>



  <target name="init">

    <!-- Create the time stamp -->

    <tstamp/>
```

```
    <!-- Create the build directory structure used by compile
-->

    <mkdir dir="${build}"/>

  </target>



  <target name="compile" depends="init"

        description="compile the source">

    <!-- Compile the Java code from ${src} into ${build} -->

    <javac srcdir="${src}" destdir="${build}"/>

  </target>



  <target name="dist" depends="compile"

        description="generate the distribution">

    <!-- Create the distribution directory -->

    <mkdir dir="${dist}/lib"/>



    <!-- Put everything in ${build} into the MyProject-
${DSTAMP}.jar file -->

    <jar jarfile="${dist}/lib/MyProject-${DSTAMP}.jar"
basedir="${build}"/>

  </target>



  <target name="clean"

        description="clean up">
```

```
    <!-- Delete the ${build} and ${dist} directory trees -->
    <delete dir="${build}"/>
    <delete dir="${dist}"/>
  </target>
</project>
```

The buildfile is written in XML, and defines some simple metadata about the build along with a list of tasks (the `<target>` tags in the XML[4]). Each task executes a list of possible commands defined by Ant, which here include creating and deleting directories, running `javac`, and creating a JAR file. This set of commands can be extended by user-provided plug-ins to cover any sort of logic. Each task can also define the tasks it depends on via the `depends` attribute. These dependencies form an acyclic graph (see Figure 18-1):

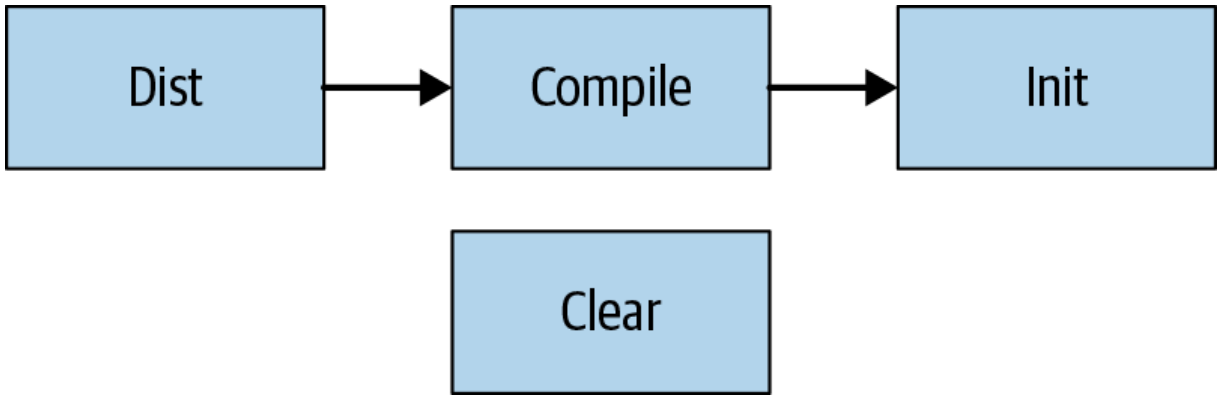

Figure 18-1. An acyclic graph showing dependencies

Users perform builds by providing tasks to Ant's command-line tool. For example, when a user types `ant dist`, Ant takes the following steps:

1. Loads a file named *build.xml* in the current directory and parses it to create the graph structure shown in Figure 18-1.
2. Looks for the task named `dist` that was provided on the command line and discovers that it has a dependency on the task named `compile`.
3. Looks for the task named `compile` and discovers that it has a dependency on the task named `init`.
4. Looks for the task named `init` and discovers that it has no dependencies.
5. Executes the commands defined in the `init` task.

6. Executes the commands defined in the `compile` task given that all of that task's dependencies have been run.
7. Executes the commands defined in the `dist` task given that all of that task's dependencies have been run.

In the end, the code executed by Ant when running the `dist` task is equivalent to the following shell script:

```
./createTimestamp.sh

mkdir build/

javac src/* -d build/

mkdir -p dist/lib/

jar cf dist/lib/MyProject-$(date --iso-8601).jar build/*
```

When the syntax is stripped away, the buildfile and the build script actually aren't too different. But we've already gained a lot by doing this. We can create new buildfiles in other directories and link them together. We can easily add new tasks that depend on existing tasks in arbitrary and complex ways. We need only pass the name of a single task to the `ant` command-line tool, and it will take care of determining everything that needs to be run.

Ant is a very old piece of software, originally released in 2000—not what many people would consider a "modern" build system today! Other tools like Maven and Gradle have improved on Ant in the intervening years and essentially replaced it by adding features like automatic management of external dependencies and a cleaner syntax without any XML. But the nature of these newer systems remains the same: they allow engineers to write build scripts in a principled and modular way as tasks, and provide tools for executing those tasks and managing dependencies among them.

THE DARK SIDE OF TASK-BASED BUILD SYSTEMS

Because these tools essentially let engineers define any script as a task, they are extremely powerful, allowing you to do pretty much anything you can imagine with them. But that power comes with drawbacks, and task-based build systems can become difficult to work with as their build scripts grow more complex. The problem with such systems is that they actually end up giving *too much power to engineers and not enough power to the system*. Because the system has no idea what the scripts are doing, performance

suffers as it must be very conservative in how it schedules and executes build steps. And there's no way for the system to confirm that each script is doing what it should, so scripts tend to grow in complexity and end up being another thing that needs debugging.

Difficulty of parallelizing build steps

Modern development workstations are typically quite powerful, with multiple cores that should theoretically be capable of executing several build steps in parallel. But task-based systems are often unable to parallelize task execution even when it seems like they should be able to. Suppose that task A depends on tasks B and C. Because tasks B and C have no dependency on each other, is it safe to run them at the same time so that the system can more quickly get to task A? Maybe, if they don't touch any of the same resources. But maybe not—perhaps both use the same file to track their statuses and running them at the same time will cause a conflict. There's no way in general for the system to know, so either it has to risk these conflicts (leading to rare but very difficult-to-debug build problems), or it has to restrict the entire build to running on a single thread in a single process. This can be a huge waste of a powerful developer machine, and it completely rules out the possibility of distributing the build across multiple machines.

Difficulty performing incremental builds

A good build system will allow engineers to perform reliable incremental builds such that a small change doesn't require the entire codebase to be rebuilt from scratch. This is especially important if the build system is slow and unable to parallelize build steps for the aforementioned reasons. But unfortunately, task-based build systems struggle here, too. Because tasks can do anything, there's no way in general to check whether they've already been done. Many tasks simply take a set of source files and run a compiler to create a set of binaries; thus, they don't need to be rerun if the underlying source files haven't changed. But without additional information, the system can't say this for sure—maybe the task downloads a file that could have changed, or maybe it writes a timestamp that could be different on each run. To guarantee correctness, the system typically must rerun every task during each build.

Some build systems try to enable incremental builds by letting engineers specify the conditions under which a task needs to be rerun. Sometimes this is feasible, but often it's a much trickier problem than it appears. For example, in languages like C++ that allow files to be included directly by other files, it's impossible to determine the entire set of files that must be

watched for changes without parsing the input sources. Engineers will often end up taking shortcuts, and these shortcuts can lead to rare and frustrating problems where a task result is reused even when it shouldn't be. When this happens frequently, engineers get into the habit of running `clean` before every build to get a fresh state, completely defeating the purpose of having an incremental build in the first place. Figuring out when a task needs to be rerun is surprisingly subtle, and is a job better handled by machines than humans.

Difficulty maintaining and debugging scripts

Finally, the build scripts imposed by task-based build systems are often just difficult to work with. Though they often receive less scrutiny, build scripts are code just like the system being built, and are easy places for bugs to hide. Here are some examples of bugs that are very common when working with a task-based build system:

- Task A depends on task B to produce a particular file as output. The owner of task B doesn't realize that other tasks rely on it, so they change it to produce output in a different location. This can't be detected until someone tries to run task A and finds that it fails.
- Task A depends on task B, which depends on task C, which is producing a particular file as output that's needed by task A. The owner of task B decides that it doesn't need to depend on task C any more, which causes task A to fail even though task B doesn't care about task C at all!
- The developer of a new task accidentally makes an assumption about the machine running the task, such as the location of a tool or the value of particular environment variables. The task works on their machine, but fails whenever another developer tries it.
- A task contains a nondeterministic component, such as downloading a file from the Internet or adding a timestamp to a build. Now, people will get potentially different results each time they run the build, meaning that engineers won't always be able to reproduce and fix each other's failures or failures that occur on an automated build system.
- Tasks with multiple dependencies can create race conditions. If task A depends on both task B and task C, and task B and C both modify the same file, task A will get a different result depending on which one of tasks B and C finishes first.

There's no general-purpose way to solve these performance, correctness, or maintainability problems within the task-based framework laid out here. So long as engineers can write arbitrary code that runs during the build, the system can't have enough information to always be able to run builds quickly and correctly. To solve the problem, we need to take some power out of the hands of engineers and put it back in the hands of the system, and reconceptualize the role of the system not as running tasks, but as producing artifacts. This is the approach that Google takes with Blaze and Bazel, and will be described in the next section.

Artifact-Based Build Systems

To design a better build system, we need to take a step back. The problem with the earlier systems is that they gave too much power to individual engineers by letting them define their own tasks. Maybe instead of letting engineers define tasks, we can have a small number of tasks defined by the system that engineers can configure in a limited way. We could probably deduce the name of the most important task from the name of this chapter: a build system's primary task should be to *build* code. Engineers would still need to tell the system *what* to build, but the *how* of doing the build would be left to the system.

This is exactly the approach taken by Blaze and the other *artifact-based* build systems descended from it (which include Bazel, Pants, and Buck). Like with task-based build systems, we still have buildfiles, but the contents of those buildfiles are very different. Rather than being an imperative set of commands in a Turing-complete scripting language describing how to produce an output, buildfiles in Blaze are a *declarative manifest* describing a set of artifacts to build, their dependencies, and a limited set of options that affect how they're built. When engineers run `blaze` on the command line, they specify a set of targets to build (the "what"), and Blaze is responsible for configuring, running, and scheduling the compilation steps (the "how"). Because the build system now has full control over what tools are being run when, it can make much stronger guarantees that allow it to be far more efficient while still guaranteeing correctness.

A FUNCTIONAL PERSPECTIVE

It's easy to make an analogy between artifact-based build systems and functional programming. Traditional imperative programming languages (e.g., Java, C, and Python) specify lists of statements to be executed one after another, in the same way that task-based build systems let programmers

define a series of steps to execute. Functional programming languages (e.g., Haskell and ML), in contrast, are structured more like a series of mathematical equations. In functional languages, the programmer describes a computation to perform, but leaves the details of when and exactly how that computation is executed to the compiler. This maps to the idea of declaring a manifest in an artifact-based build system and letting the system figure out how to execute the build.

Many problems cannot be easily expressed using functional programming, but the ones that do benefit greatly from it: the language is often able to trivially parallelize such programs and make strong guarantees about their correctness that would be impossible in an imperative language. The easiest problems to express using functional programming are the ones that simply involve transforming one piece of data into another using a series of rules or functions. And that's exactly what a build system is: the whole system is effectively a mathematical function that takes source files (and tools like the compiler) as inputs and produces binaries as outputs. So, it's not surprising that it works well to base a build system around the tenants of functional programming.

Getting concrete with Bazel

Bazel is the open source version of Google's internal build tool, Blaze, and is a good example of an artifact-based build system. Here's what a buildfile (normally named BUILD) looks like in Bazel:

```
java_binary(
  name = "MyBinary",
  srcs = ["MyBinary.java"],
  deps = [
    ":mylib",
  ],
)


java_library(
```

```
    name = "mylib",

    srcs = ["MyLibrary.java", "MyHelper.java"],

    visibility =
["//java/com/example/myproduct:__subpackages__"],

    deps = [

        "//java/com/example/common",

        "//java/com/example/myproduct/otherlib",

        "@com_google_common_guava_guava//jar",

    ],

)
```

In Bazel, *BUILD* files define *targets*—the two types of targets here are `java_binary` and `java_library`. Every target corresponds to an artifact that can be created by the system: `binary` targets produce binaries that can be executed directly, and `library` targets produce libraries that can be used by binaries or other libraries. Every target has a *name* (which defines how it is referenced on the command line and by other targets, *srcs* (which defines the source files that must be compiled to create the artifact for the target), and *deps* (which define other targets that must be built before this target and linked into it). Dependencies can either be within the same package (e.g., `MyBinary`'s dependency on `":mylib"`), on a different package in the same source hierarchy (e.g., `mylib`'s dependency on `"//java/com/example/common"`), or on a third-party artifact outside of the source hierarchy (e.g., `mylib`'s dependency on `"@com_google_common_guava_guava//jar"`). Each source hierarchy is called a *workspace*, and is identified by the presence of a special *WORKSPACE* file at the root.

Like with Ant, users perform builds using Bazel's command-line tool. To build the `MyBinary` target, a user would run `bazel build :MyBinary`. Upon entering that command for the first time in a clean repository, Bazel would do the following:

1. Parse every *BUILD* file in the workspace to create a graph of dependencies among artifacts.

2. Use the graph to determine the *transitive dependencies* of `MyBinary`; that is, every target that `MyBinary` depends on and every target that those targets depend on, recursively.
3. Build (or download for external dependencies) each of those dependencies, in order. Bazel starts by building each target that has no other dependencies and keeps track of which dependencies still need to be built for each target. As soon as all of a target's dependencies are built, Bazel starts building that target. This process continues until every one of `MyBinary`'s transitive dependencies have been built.
4. Build `MyBinary` to produce a final executable binary that links in all of the dependencies that were built in step 3.

Fundamentally, it might not seem like what's happening here is that much different than what happened when using a task-based build system. Indeed, the end result is the same binary, and the process for producing it involved analyzing a bunch of steps to find dependencies among them, and then running those steps in order. But there are critical differences. The first one appears in step 3: because Bazel knows that each target will only produce a Java library, it knows that all it has to do is run the Java compiler rather than an arbitrary user-defined script, so it knows that it's safe to run these steps in parallel. This can produce an order of magnitude performance improvement over building targets one-at-a-time on a multicore machine, and is only possible because the artifact-based approach leaves the build system in charge of its own execution strategy so that it can make stronger guarantees about parallelism.

The benefits extend beyond parallelism, though. The next thing that this approach gives us becomes apparent when the developer types `bazel build :MyBinary` a second time without making any changes: Bazel will exit in less than a second with a message saying that the target is up to date. This is possible due to the functional programming paradigm we talked about earlier—Bazel knows that each target is the result only of running a Java compiler, and it knows that the output from the Java compiler depends only on its inputs, so as long as the inputs haven't changed the output can be reused. And this analysis works at every level; if `MyBinary.java` changes, Bazel knows to rebuild `MyBinary` but reuse `mylib`. If a source file for `//java/com/example/common` changes, Bazel knows to rebuild that library, `mylib`, and `MyBinary`, but reuse `//java/com/example/myproduct/otherlib`. Because Bazel knows about the properties of the tools it runs at every step, it's able to rebuild only the minimum set of artifacts each time while guaranteeing that it won't produce stale builds.

Reframing the build process in terms of artifacts rather than tasks is subtle but powerful. By reducing the flexibility exposed to the programmer, the build system can know more about what is being done at every step of the build. It can use this knowledge to make the build far more efficient by parallelizing build processes and reusing their outputs. But this is really just the first step, and these building blocks of parallelism and reuse will form the basis for a distributed and highly scalable build system that will be discussed later.

OTHER NIFTY BAZEL TRICKS

Artifact-based build systems fundamentally solve the problems with parallelism and reuse that are inherent in task-based build systems. But there are still a few problems that came up earlier that we haven't addressed. Bazel has clever ways of solving each of these, and we should discuss them before moving on.

Tools as dependencies

One problem we ran into earlier was that builds depended on the tools installed on our machine, and reproducing builds across systems could be difficult due to different tool versions or locations. The problem becomes even more difficult when your project uses languages that require different tools based on which platform they're being built on or compiled for (e.g., Windows versus Linux), and each of those platforms requires a slightly different set of tools to do the same job.

Bazel solves the first part of this problem by treating tools as dependencies to each target. Every `java_library` in the workspace implicitly depends on a Java compiler, which defaults to a well-known compiler but can be configured globally at the workspace level. Whenever Blaze builds a `java_library`, it checks to make sure that the specified compiler is available at a known location, and downloads it if not. Just like any other dependency, if the Java compiler changes, every artifact that was dependent upon it will need to be rebuilt. Every type of target defined in Bazel uses this same strategy of declaring the tools it needs to run, ensuring that Bazel is able to bootstrap them no matter what exists on the system where it runs.

Bazel solves the second part of the problem, platform independence, by using toolchains. Rather than having targets depend directly on their tools, they actually depend on types of toolchains. A toolchain contains a set of tools and other properties defining how a type of target is built on a particular platform. The workspace can define the particular toolchain to use for a

toolchain type based on the host and target platform. For more details, see the Bazel manual.

Extending the build system

Bazel comes with targets for several popular programming languages out of the box, but engineers will always want to do more—part of the benefit of task-based systems is their flexibility in supporting any kind of build process, and it would be better not to give that up in an artifact-based build system. Fortunately, Bazel allows its supported target types to be extended by adding custom rules.

To define a rule in Bazel, the rule author declares the inputs that the rule requires (in the form of attributes passed in the *BUILD* file) and the fixed set of outputs that the rule produces. The author also defines the *actions* that will be generated by that rule. Each action declares its inputs and outputs, runs a particular executable or writes a particular string to a file, and can be connected to other actions via its inputs and outputs. This means that actions are the lowest-level composable unit in the build system—an action can do whatever it wants so long as it uses only its declared inputs and outputs, and Bazel will take care of scheduling actions and caching their results as appropriate.

The system isn't foolproof given that there's no way to stop an action developer from doing something like introducing a nondeterministic process as part of their action. But this doesn't happen very often in practice, and pushing the possibilities for abuse all the way down to the action level greatly decreases opportunities for errors. Rules supporting many common languages and tools are widely available online, and most projects will never need to define their own rules. Even for those that do, rule definitions only need to be defined in one central place in the repository, meaning most engineers will be able to use those rules without ever having to worry about their implementation.

Isolating the environment

Actions sound like they might run into the same problems as tasks in other systems—isn't it still possible to write actions that both write to the same file and end up conflicting with one another? Actually, Bazel makes these conflicts impossible by using *sandboxing*. On supported systems, every action is isolated from every other action via a filesystem sandbox. Effectively, each action can see only a restricted view of the filesystem that includes the inputs it has declared and any outputs it has produced. This is

enforced by systems such as LXC on Linux, the same technology behind Docker. This means that it's impossible for actions to conflict with one another because they are unable to read any files they don't declare, and any files that they write but don't declare will be thrown away when the action finishes. Bazel also uses sandboxes to restrict actions from communicating via the network.

Making external dependencies deterministic

There's still one problem remaining: build systems often need to download dependencies (whether tools or libraries) from external sources rather than directly building them. This can be seen in the example via the `@com_google_common_guava_guava//jar` dependency, which downloads a JAR file from Maven.

Depending on files outside of the current workspace is risky. Those files could change at any time, potentially requiring the build system to constantly check whether they're fresh. If a remote file changes without a corresponding change in the workspace source code, it can also lead to unreproducible builds—a build might work one day and fail the next for no obvious reason due to an unnoticed dependency change. Finally, an external dependency can introduce a huge security risk when it is owned by a third party:[5] if an attacker is able to infiltrate that third-party server, they can replace the dependency file with something of their own design, potentially giving them full control over your build environment and its output.

The fundamental problem is that we want the build system to be aware of these files without having to check them into source control. Updating a dependency should be a conscious choice, but that choice should be made once in a central place rather than managed by individual engineers or automatically by the system. This is because even with a "live at head" model, we still want builds to be deterministic, which implies that if you check out a commit from last week, you should see your dependencies as they were then rather than as they are now.

Bazel and some other build systems address this problem by requiring a workspace-wide manifest file that lists a *cryptographic hash* for every external dependency in the workspace.[6] The hash is a concise way to uniquely represent the file without checking the entire file into source control. Whenever a new external dependency is referenced from a workspace, that dependency's hash is added to the manifest, either manually or automatically. When Bazel runs a build, it checks the actual hash of its

cached dependency against the expected hash defined in the manifest and redownloads the file only if the hash differs.

If the artifact we download has a different hash than the one declared in the manifest, the build will fail unless the hash in the manifest is updated. This can be done automatically, but that change must be approved and checked into source control before the build will accept the new dependency. This means that there's always a record of when a dependency was updated, and an external dependency can't change without a corresponding change in the workspace source. It also means that, when checking out an older version of the source code, the build is guaranteed to use the same dependencies that it was using at the point when that version was checked in (or else it will fail if those dependencies are no longer available).

Of course, it can still be a problem if a remote server becomes unavailable or starts serving corrupt data—this can cause all of your builds to begin failing if you don't have another copy of that dependency available. To avoid this problem, we recommend that, for any nontrivial project, you mirror all of its dependencies onto servers or services that you trust and control. Otherwise you will always be at the mercy of a third party for your build system's availability, even if the checked-in hashes guarantee its security.

Distributed Builds

Google's codebase is enormous—with more than two billion lines of code, chains of dependencies can become very deep. Even simple binaries at Google often depend on tens of thousands of build targets. At this scale, it's simply impossible to complete a build in a reasonable amount of time on a single machine: no build system can get around the fundamental laws of physics imposed on a machine's hardware. The only way to make this work is with a build system that supports *distributed builds wherein* the units of work being done by the system are spread across an arbitrary and scalable number of machines. Assuming we've broken the system's work into small enough units (more on this later), this would allow us to complete any build of any size as quickly as we're willing to pay for. This scalability is the holy grail we've been working toward by defining an artifact-based build system.

REMOTE CACHING

The simplest type of distributed build is one that only leverages *remote caching, which* is shown in Figure 18-2.

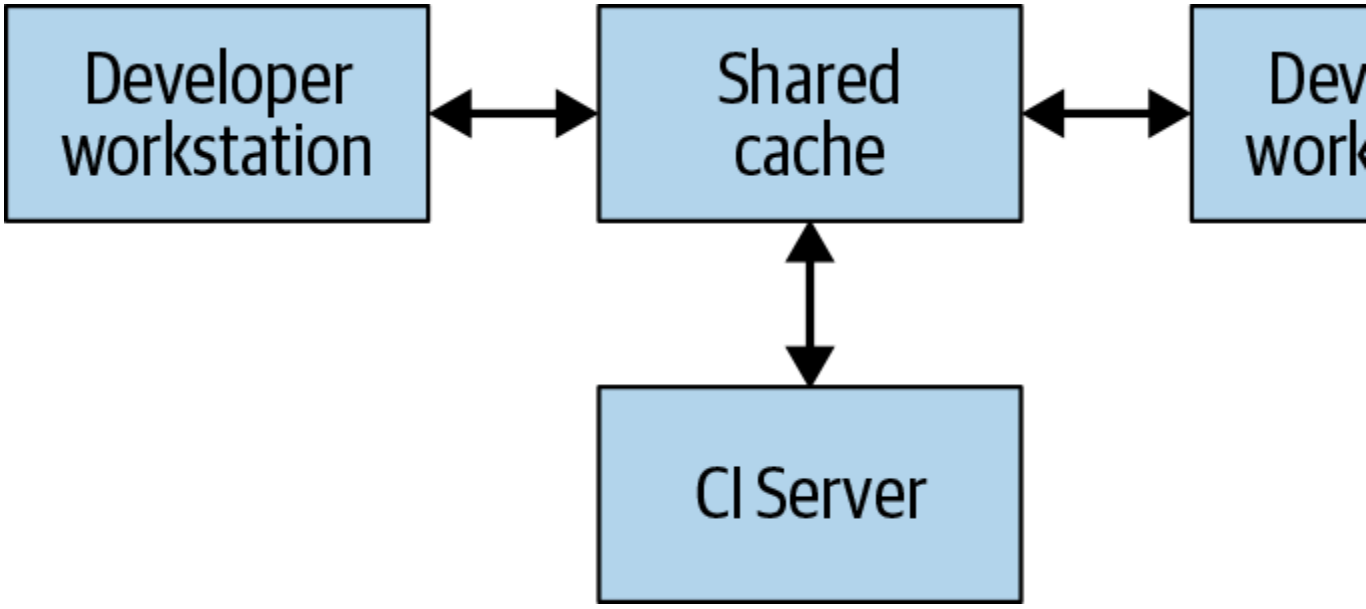

Figure 18-2. A distributed build showing remote caching

Every system that performs builds, including both developer workstations and continuous integration systems, shares a reference to a common remote cache service. This service might be a fast and local short-term storage system like Redis or a cloud service like Google Cloud Storage. Whenever a user needs to build an artifact, whether directly or as a dependency, the system first checks with the remote cache to see if that artifact already exists there. If so, it can download the artifact instead of building it. If not, the system builds the artifact itself and uploads the result back to the cache. This means that low-level dependencies that don't change very often can be built once and shared across users rather than having to be rebuilt by each user. At Google, many artifacts are served from a cache rather than built from scratch, vastly reducing the cost of running our build system.

For a remote caching system to work, the build system must guarantee that builds are completely reproducible. That is, for any build target, it must be possible to determine the set of inputs to that target such that the same set of inputs will produce exactly the same output on any machine. This is the only way to ensure that the results of downloading an artifact are the same as the results of building it oneself. Fortunately, Bazel provides this guarantee and so supports remote caching. Note that this requires that each artifact in the cache be keyed on both its target and a hash of its inputs—that way, different engineers could make different modifications to the same target at the same time, and the remote cache would store all of the resulting artifacts and serve them appropriately without conflict.

Of course, for there to be any benefit from a remote cache, downloading an artifact needs to be faster than building it. This is not always the case,

especially if the cache server is far from the machine doing the build. Google's network and build system is carefully tuned to be able to quickly share build results. When configuring remote caching in your organization, take care to consider network latencies and perform experiments to ensure that the cache is actually improving performance.

REMOTE EXECUTION

Remote caching isn't a true distributed build. If the cache is lost or if you make a low-level change that requires everything to be rebuilt, you still need to perform the entire build locally on your machine. The true goal is to support *remote execution*, in which the actual work of doing the build can be spread across any number of workers. Figure 18-3 depicts a remote execution system.

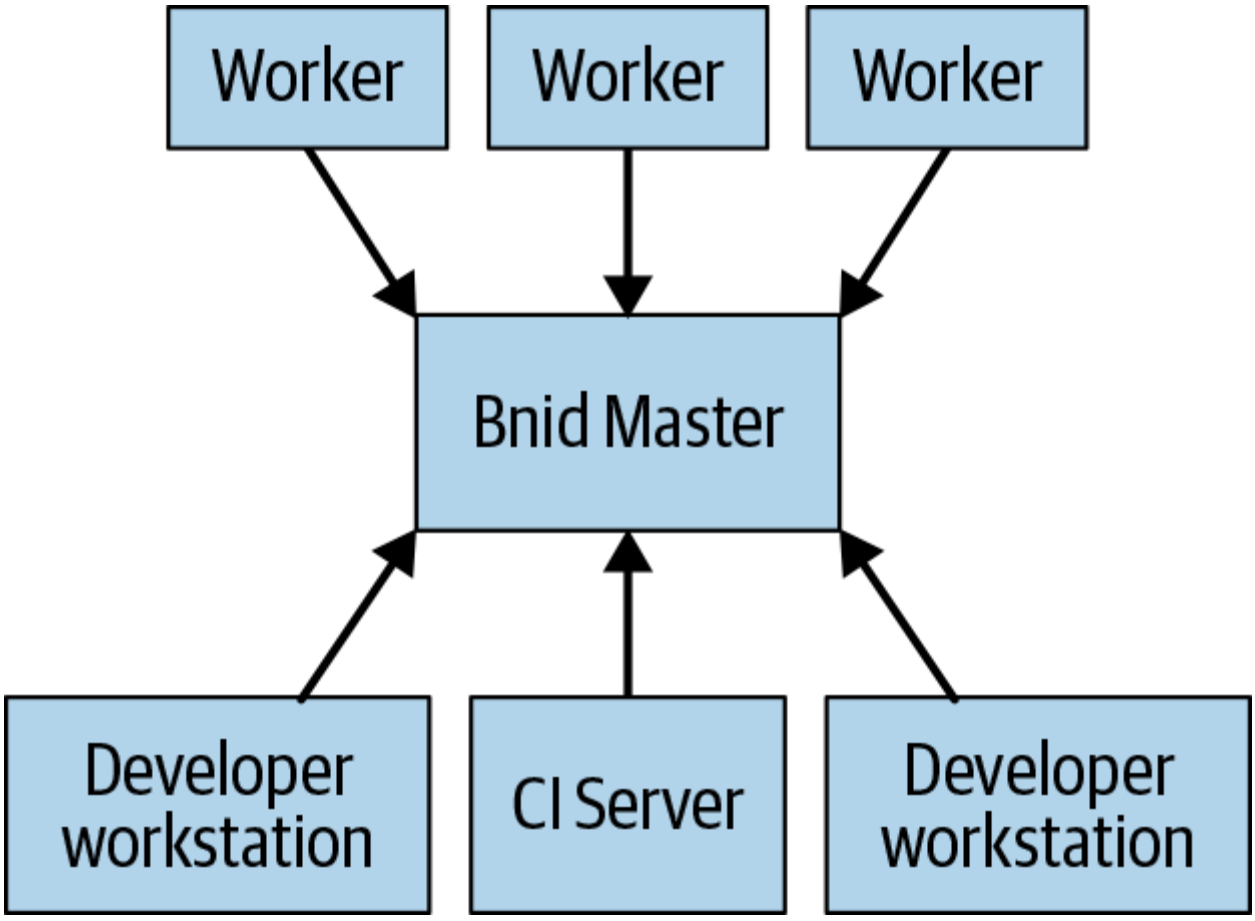

Figure 18-3. A remote execution system

The build tool running on each user's machine (where users are either human engineers or automated build systems) sends requests to a central build master. The build master breaks the requests into their component actions, and schedules the execution of those actions over a scalable pool of workers. Each worker performs the actions asked of it with the inputs specified by the

user and writes out the resulting artifacts. These artifacts are shared across the other machines executing actions that require them until the final output can be produced and sent to the user.

The trickiest part of implementing such a system is managing the communication between the workers, the master, and the user's local machine. Workers might depend on intermediate artifacts produced by other workers, and the final output needs to be sent back to the user's local machine. To do this, we can build on top of the distributed cache described previously by having each worker write its results to and read its dependencies from the cache. The master blocks workers from proceeding until everything they depend on has finished, in which case they'll be able to read their inputs from the cache. The final product is also cached, allowing the local machine to download it. Note that we also need a separate means of exporting the local changes in the user's source tree so that workers can apply those changes before building.

For this to work, all of the parts of the artifact-based build systems described earlier need to come together. Build environments must be completely self-describing so that we can spin up workers without human intervention. Build processes themselves must be completely self-contained because each step might be executed on a different machine. Outputs must be completely deterministic so that each worker can trust the results it receives from other workers. Such guarantees are extremely difficult for a task-based system to provide, which makes it nigh-impossible to build a reliable remote execution system on top of one.

Distributed builds at Google

Since 2008, Google has been using a distributed build system that employs both remote caching and remote execution, which is illustrated in Figure 18-4.

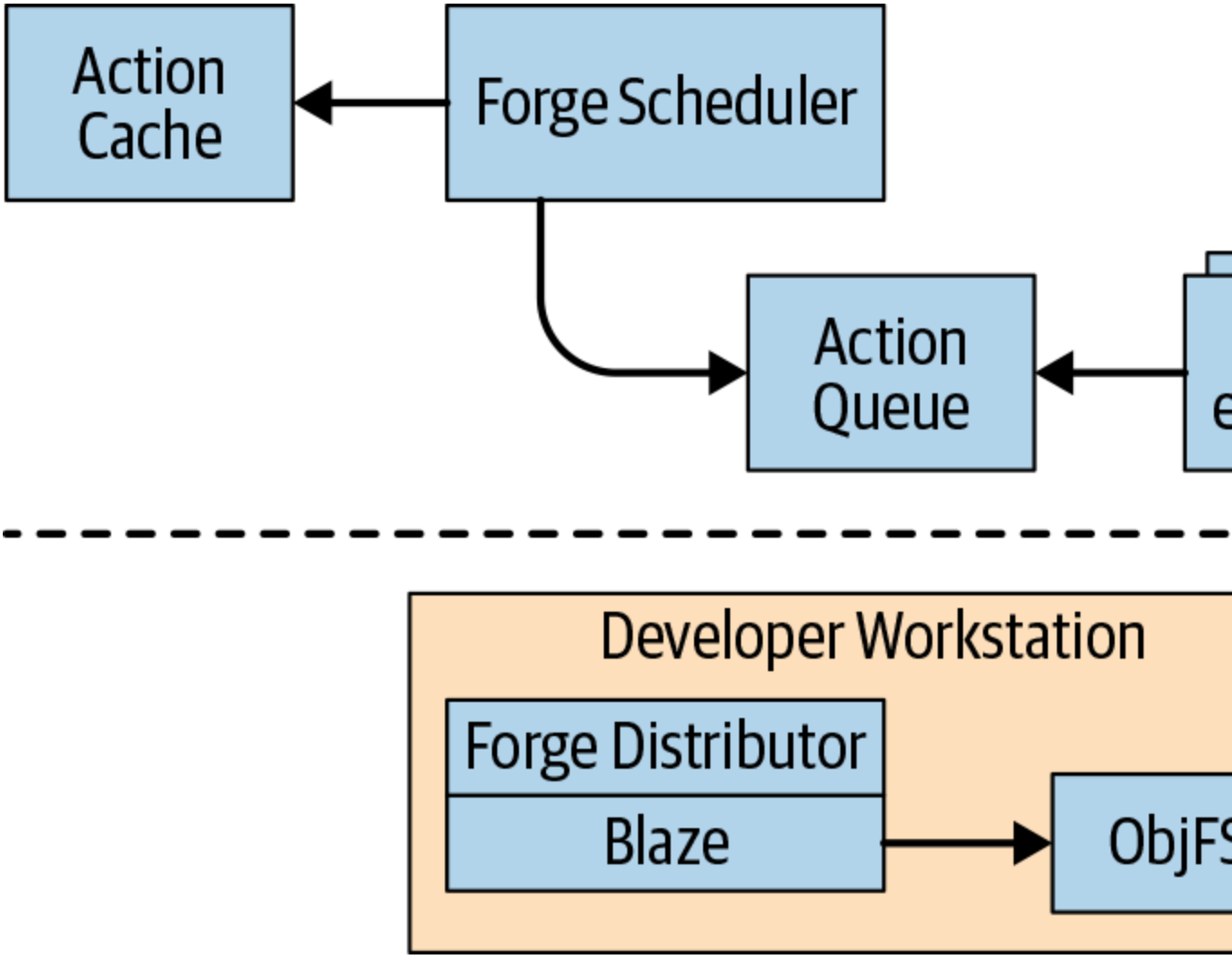

Figure 18-4. Google's distributed build system

Google's remote cache is called ObjFS. It consists of a backend that stores build outputs in Bigtables distributed throughout our fleet of production machines, and a frontend FUSE daemon named objfsd that runs on each developer's machine. The FUSE daemon allows engineers to browse build outputs as if they were normal files stored on the workstation, but with the file content downloaded on-demand only for the few files that are directly requested by the user. Serving file contents on-demand greatly reduces both network and disk usage, and the system is able to build twice as fast compared to when we stored all build output on the developer's local disk.

Google's remote execution system is called Forge. A Forge client in Blaze called the Distributor sends requests for each action to a job running in our datacenters called the Scheduler. The Scheduler maintains a cache of action results, allowing it to return a response immediately if the action has already been created by any other user of the system. If not, it places the action into a queue. A large pool of Executor jobs continually read actions from this

queue, execute them, and store the results directly in the ObjFS Bigtables. These results are available to the executors for future actions, or to be downloaded by the end user via objfsd.

The end result is a system that scales to efficiently support all builds performed at Google. And the scale of Google's builds is truly massive: Google runs millions of builds executing millions of test cases and producing petabytes of build outputs from billions of lines of source code every *day*. Not only does such a system let our engineers build complex codebases quickly, it also allows us to implement a huge number of automated tools and systems that rely on our build. We put many years of effort into developing this system, but nowadays open source tools are readily available such that any organization can implement a similar system. Though it can take time and energy to deploy such a build system, the end result can be truly magical for engineers and is often well worth the effort.

Time, Scale, Trade-Offs

Build systems are all about making code easier to work with at scale and over time. And like everything in software engineering, there are trade-offs in choosing which sort of build system to use. The DIY approach using shell scripts or direct invocations of tools works only for the smallest projects that don't need to deal with code changing over a long period of time, or for languages like Go that have a built-in build system.

Choosing a task-based build system instead of relying on DIY scripts greatly improves your project's ability to scale, allowing you to automate complex builds and more easily reproduce those builds across machines. The trade-off is that you need to actually start putting some thought into how your build is structured and deal with the overhead of writing build files (though automated tools can often help with this). This trade-off tends to be worth it for most projects, but for particularly trivial projects (e.g., those contained in a single source file), the overhead might not buy you much.

Task-based build systems begin to run into some fundamental problems as the project scales further, and these issues can be remedied by using an artifact-based build system, instead. Such build systems unlock a whole new level of scale because as huge builds can now be distributed across many machines, and thousands of engineers can be more certain that their builds are consistent and reproducible. As with so many other topics in this book, the tradeoff here is a lack of flexibility: artifact-based systems don't let you write generic tasks in a real programming language, but require you to work

within the constraints of the system. This is usually not a problem for projects that are designed to work with artifact-based systems from the start, but migration from an existing task-based system can be difficult and is not always worth it if the build isn't already showing problems in terms of speed or correctness.

Changes to a project's build system can be expensive, and that cost increases as the project becomes larger. This is why Google believes that almost every new project benefits from incorporating an artifact-based build system like Bazel right from the start. Within Google, essentially all code from tiny experimental projects up to Google Search is built using Blaze.

Dealing with Modules and Dependencies

Projects that used artifact-based build systems like Bazel are broken into a set of modules, with modules expressing dependencies on one another via *BUILD* files. Proper organization of these modules and dependencies can have a huge effect on both the performance of the build system and how much work it takes to maintain.

Using Fine-Grained Modules and the 1:1:1 Rule

The first question that comes up when structuring an artifact-based build is deciding how much functionality an individual module should encompass. In Bazel, a "module" is represented by a target specifying a buildable unit like a `java_library` or a `go_binary`. At one extreme, the entire project could be contained in a single module by putting one BUILD file at the root and recursively globbing together all of that project's source files. At the other extreme, nearly every source file could be made into its own module, effectively requiring each file to list in a *BUILD* file every other file it depends on.

Most projects fall somewhere between these extremes, and the choice involves a trade-off between performance and maintainability. Using a single module for the entire project might mean that you never need to touch the *BUILD* file except when adding an external dependency, but it means that the build system will always need to build the entire project all at once. This means that it won't be able to parallelize or distribute parts of the build, nor will it be able to cache parts that it's already built. One-module-per-file is the opposite: the build system has the maximum flexibility in caching and scheduling steps of the build, but engineers need to expend more effort

maintaining lists of dependencies whenever they change which files reference which.

Though the exact granularity varies by language (and often even within language), Google tends to favor significantly smaller modules than one might typically write in a task-based build system. A typical production binary at Google will likely depend on tens of thousands of targets, and even a moderate-sized team can own several hundred targets within its codebase. For languages like Java that have a strong built-in notion of packaging, each directory usually contains a single package, target, and *BUILD* file (Pants, another build system based on Blaze, calls this the 1:1:1 rule). Languages with weaker packaging conventions will frequently define multiple targets per *BUILD* file.

The benefits of smaller build targets really begin to show at scale because they lead to faster distributed builds and a less frequent need to rebuild targets. The advantages become even more compelling after testing enters the picture, as finer-grained targets mean that the build system can be much smarter about running only a limited subset of tests that could be affected by any given change. Because Google believes in the systemic benefits of using smaller targets, we've made some strides in mitigating the downside by investing in tooling to automatically manage *BUILD* files to avoid burdening developers. Many of these tools are now open source.

Minimizing Module Visibility

Bazel and other build systems allow each target to specify a visibility: a property that specifies which other targets may depend on it. Targets can be `public`, in which case they can be referenced by any other target in the workspace; `private`, in which case they can be referenced only from within the same *BUILD* file; or visible to only an explicitly defined list of other targets. A visibility is essentially the opposite of a dependency: if target A wants to depend on target B, target B must make itself visible to target A.

Just like in most programming languages, it is usually best to minimize visibility as much as possible. Generally, teams at Google will make targets public only if those targets represent widely used libraries available to any team at Google. Teams that require others to coordinate with them before using their code will maintain a whitelist of customer targets as their target's visibility. Each team's internal implementation targets will be restricted to only directories owned by the team, and most *BUILD* files will have only one target that isn't private.

Managing Dependencies

Modules need to be able to refer to one another. The downside of breaking a codebase into fine-grained module is that you need to manage the dependencies among those modules (though tools can help automate this). Expressing these dependencies usually ends up being the bulk of the content in a *BUILD* file.

INTERNAL DEPENDENCIES

In a large project broken into fine-grained modules, most dependencies are likely to be internal; that is, on another target defined and built in the same source repository. Internal dependencies differ from external dependencies in that they are built from source rather than downloaded as a prebuilt artifact while running the build. This also means that there's no notion of "version" for internal dependencies—a target and all of its internal dependencies are always built at the same commit/revision in the repository.

One issue that should be handled carefully with regard to internal dependencies is how to treat *transitive dependencies* (Figure 18-5). Suppose target A depends on target B, which depends on a common library target C. Should target A be able to use classes defined in target C?

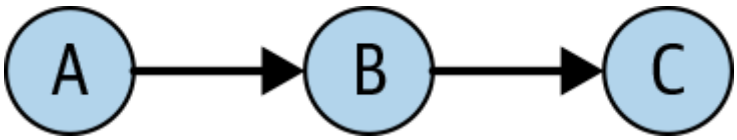

Figure 18-5. Transitive dependencies

As far as the underlying tools are concerned, there's no problem with this; both B and C will be linked into target A when it is built, so any symbols defined in C are known to A. Blaze allowed this for many years, but as Google grew, we began to see problems. Suppose that B was refactored such that it no longer needed to depend on C. If B's dependency on C was then removed, A and any other target that used C via a dependency on B would break. Effectively, a target's dependencies became part of its public contract and could never be safely changed. This meant that dependencies accumulated over time and builds at Google started to slow down.

Google eventually solved this issue by introducing a "strict transitive dependency mode" in Blaze. In this mode, Blaze detects whether a target tries to reference a symbol without depending on it directly and, if so, fails with an error and a shell command that can be used to automatically insert the dependency. Rolling this change out across Google's entire codebase and

refactoring every one of our millions of build targets to explicitly list their dependencies was a multiyear effort, but it was well worth it. Our builds are now much faster given that targets have fewer unnecessary dependencies,[7] and engineers are empowered to remove dependencies they don't need without worrying about breaking targets which depend on them.

As usual, enforcing strict transitive dependencies involved a trade-off. It made build files more verbose as frequently used libraries now need to be listed explicitly in many places rather than pulled in incidentally, and engineers needed to spend more effort adding dependencies to *BUILD* files. We've since developed tools that reduce this toil by automatically detecting many missing dependencies and adding them to *BUILD* files without any developer intervention. But even without such tools, we've found the trade-off to be well worth it as the codebase scales: explicitly adding a dependency to *BUILD* file is a one-time cost, but dealing with implicit transitive dependencies can cause ongoing problems as long as the build target exists. Bazel enforces strict transitive dependencies on Java code by default.

EXTERNAL DEPENDENCIES

If a dependency isn't internal, it must be external. External dependencies are those on artifacts that are built and stored outside of the build system. The dependency is imported directly from an *artifact repository* (typically accessed over the internet) and used as-is rather than being built from source. One of the biggest differences between external and internal dependencies is that external dependencies have *versions*, and those versions exist independently of the project's source code.

Automatic versus manual dependency management

Build systems can allow the versions of external dependencies to be managed either manually or automatically. When managed manually, the buildfile explicitly lists the version it wants to download from the artifact repository, often using a semantic version string such as "1.1.4". When managed automatically, the source file specifies a range of acceptable versions, and the build system always downloads the latest one. For example, Gradle allows a dependency version to be declared as "1.+" to specify that any minor or patch version of a dependency is acceptable so long as the major version is 1.

Automatically managed dependencies can be convenient for small projects, but they're usually a recipe for disaster on projects of nontrivial size or that are being worked on by more than one engineer. The problem with automatically managed dependencies is that you have no control over when

the version is updated. There's no way to guarantee that external parties won't make breaking updates (even when they claim to use semantic versioning), so a build that worked one day might be broken the next with no easy way to detect what changed or to roll it back to a working state. Even if the build doesn't break, there can be subtle behavior or performance changes that are impossible to track down.

In contrast, because manually managed dependencies require a change in source control, they can be easily discovered and rolled back, and it's possible to check out an older version of the repository to build with older dependencies. Bazel requires that versions of all dependencies be specified manually. At even moderate scales, the overhead of manual version management is well worth it for the stability it provides.

The One-Version Rule

Different versions of a library are usually represented by different artifacts, so in theory there's no reason that different versions of the same external dependency couldn't both be declared in the build system under different names. That way, each target could choose which version of the dependency it wanted to use. Google has found this to cause a lot of problems in practice, so we enforce a strict *One-Version Rule* for all third-party dependencies in our internal codebase.

The biggest problem with allowing multiple versions is the *diamond dependency* issue. Suppose that target A depends on target B and on v1 of an external library. If target B is later refactored to add a dependency on v2 of the same external library, target A will break because it now depends implicitly on two different versions of the same library. Effectively, it's never safe to add a new dependency from a target to any third-party library with multiple versions, because any of that target's users could already be depending on a different version. Following the one-version rule makes this conflict impossible—if a target adds a dependency on a third-party library, any existing dependencies will already be on that same version, so they can happily coexist.

We'll examine this further in the context of a large monorepo in Chapter 21.

Transitive external dependencies

Dealing with the transitive dependencies of an external dependency can be particularly difficult. Many artifact repositories such as Maven Central allow artifacts to specify dependencies on particular versions of other artifacts in

the repository. Build tools like Maven or Gradle will often recursively download each transitive dependency by default, meaning that adding a single dependency in your project could potentially cause dozens of artifacts to be downloaded in total.

This is very convenient: when adding a dependency on a new library, it would be a big pain to have to track down each of that library's transitive dependencies and add them all manually. But there's also a huge downside: because different libraries can depend on different versions of the same third-party library, this strategy necessarily violates the One-Version Rule and leads to the diamond dependency problem. If your target depends on two external libraries that use different versions of the same dependency, there's no telling which one you'll get. This also means that updating an external dependency could cause seemingly unrelated failures throughout the codebase if the new version begins pulling in conflicting versions of some of its dependencies.

For this reason, Bazel does not automatically download transitive dependencies. And, unfortunately, there's no silver bullet—Bazel's alternative is to require a global file that lists every single one of the repository's external dependencies and an explicit version used for that dependency throughout the repository. Fortunately, Bazel provides tools that are able to automatically generate such a file containing the transitive dependencies of a set of Maven artifacts. This tool can be run once to generate the initial *WORKSPACE* file for a project, and that file can then be manually updated to adjust the versions of each dependency.

Yet again, the choice here is one between convenience and scalability. Small projects might prefer not having to worry about managing transitive dependencies themselves, and might be able to get away with using automatic transitive dependencies. This strategy becomes less and less appealing as the organization and codebase grows, and conflicts and unexpected results become more and more frequent. At larger scales, the cost of manually managing dependencies is much less than the cost of dealing with issues caused by automatic dependency management.

Caching build results using external dependencies

External dependencies are most often provided by third parties that release stable versions of libraries, perhaps without providing source code. Some organizations might also choose to make some of their own code available as artifacts, allowing other pieces of code to depend on them as third-party

rather than internal dependencies. This can theoretically speed up builds if artifacts are slow to build but quick to download.

However, this also introduces a lot of overhead and complexity: someone needs to be responsible for building each of those artifacts and uploading them to the artifact repository, and clients need to ensure that they stay up to date with the latest version. Debugging also becomes much more difficult because different parts of the system will have been built from different points in the repository, and there is no longer a consistent view of the source tree.

A better way to solve the problem of artifacts taking a long time to build is to use a build system that supports remote caching, as described earlier. Such a build system will save the resulting artifacts from every build to a location that is shared across engineers, so if a developer depends on an artifact that was recently built by someone else, the build system will automatically download it instead of building it. This provides all of the performance benefits of depending directly on artifacts while still ensuring that builds are as consistent as if they were always built from the same source. This is the strategy used internally by Google, and Bazel can be configured to use a remote cache.

Security and reliability of external dependencies

Depending on artifacts from third-party sources is inherently risky. There's an availability risk if the third-party source (e.g., an artifact repository) goes down, because your entire build might grind to a halt if it's unable to download an external dependency. There's also a security risk: if the third-party system is compromised by an attacker, the attacker could replace the referenced artifact with one of their own design, allowing them to inject arbitrary code into your build.

Both problems can be mitigated by mirroring any artifacts you depend on onto servers you control and blocking your build system from accessing third-party artifact repositories like Maven Central. The trade-off is that these mirrors take effort and resources to maintain, so the choice of whether to use them often depends on the scale of the project. The security issue can also be completely prevented with little overhead by requiring the hash of each third-party artifact to be specified in the source repository, causing the build to fail if the artifact is tampered with.

Another alternative that completely sidesteps the issue is to *vendor* your project's dependencies. When a project vendors its dependencies, it checks

them into source control alongside the project's source code, either as source or as binaries. This effectively means that all of the project's external dependencies are converted to internal dependencies. Google uses this approach internally, checking every third-party library referenced throughout Google into a *third_party* directory at the root of Google's source tree. However, this works at Google only because Google's source control system is custom built to handle an extremely large monorepo, so vendoring might not be an option for other organizations.

Conclusion

A build system is one of the most important parts of an engineering organization. Each developer will interact with it potentially dozens or hundreds of times per day, and in many situations it can be the rate-limiting step in determining their productivity. This means that it's worth investing time and thought into getting things right.

As discussed in this chapter, one of the more surprising lessons that Google has learned is that *limiting engineers' power and flexibility can improve their productivity*. We were able to develop a build system that meets our needs not by giving engineers free reign in defining how builds are performed, but by developing a highly structured framework that limits individual choice and leaves most interesting decisions in the hands of automated tools. And despite what you might think, engineers don't resent this: Googlers love that this system mostly works on its own and lets them focus on the interesting parts of writing their applications instead of grappling with build logic. Being able to trust the build is powerful—incremental builds just work, and there is almost never a need to clear build caches or run a "clean" step.

We took this insight and used it to create a whole new type of *artifact-based* build system, contrasting with traditional *task-based* build systems. This reframing of the build as centering around artifacts instead of tasks is what allows our builds to scale to an organization the size of Google. At the extreme end, it allows for a *distributed build system* that is able to leverage the resources of an entire compute cluster to accelerate engineers' productivity. Though your organization might not be large enough to benefit from such an investment, we believe that artifact-based build systems scale down as well as they scale up: even for small projects, build systems like Bazel can bring significant benefits in terms of speed and correctness.

The remainder of this chapter explored how to manage dependencies in an artifact-based world. We came to the conclusion that *fine-grained modules scale better than coarse-grained modules*. We also discussed the difficulties of managing dependency versions, describing the O*ne-Version Rule* and the observation that all dependencies should be *versioned manually and explicitly*. Such practices avoid common pitfalls like the diamond dependency issue, and allow a codebase to achieve Google's scale of billions of lines of code in a single repository with a unified build system.

TL;DRs

- A fully featured build system is necessary to keep developers productive as an organization scales
- Power and flexibility come at a cost. Restricting the build system appropriately makes it easier on developers.
- Build systems organized around artifacts tend to scale better and be more reliable than build systems organized around tasks
- When defining artifacts and dependencies, it's better to aim for fine-grained modules. Fine-grained modules are better able to take advantage of parallelism and incremental builds.
- External dependencies should be versioned explicitly under source control. Relying on "latest" versions is a recipe for disaster and unreproducible builds.

1 In an internal survey, 83% of Googlers reported being satisfied with the build system, making it the fourth most satisfying tool of the 19 surveyed. The average tool had a satisfaction rating of 69%.

2 *https://buck.build/* and *https://www.pantsbuild.org/index.html*

3 *https://xkcd.com/1168/*

4 Ant uses the word "target" to represent what we call a "task" in this chapter, and it uses the word "task" to refer to what we call "commands."

5 Such "software supply chain" attacks are becoming more common.

6 Go recently added preliminary support for modules using the exact same system.

7 Of course, actually *removing* these dependencies was a whole separate process. But requiring each target to explicitly declare what it used was a critical first step. See Chapter 22 for more information about how Google makes large-scale changes like this.

Chapter 19. Critique: Google's Code Review Tool

Written by Caitlin Sadowski

Edited by Lisa Carey

As you saw in Chapter 9, code review is a vital part of software development, particularly when working at scale. The main goal of code review is to improve the readability and maintainability of the code base, and this is supported fundamentally by the review process. However, having a well-defined code review process in only one part of the code-review story. Tooling that supports that process also plays an important part in its success.

In this chapter, we'll look at what makes successful code review tooling via Google's well-loved in-house system, *Critique*. Critique has explicit support for the primary motivations of code review, providing reviewers and authors with a view of the review and ability to comment on the change. Critique also has support for gatekeeping what code is checked into the codebase, discussed in the section on "scoring" changes below. Code review information from Critique also can be useful when doing code archaeology, following some technical decisions that are explained in code review interactions (e.g., when inline comments are lacking). Although Critique is not the only code review tool used at Google, it is the most popular one by a large margin.

Code Review Tooling Principles

We mentioned above that Critique provides functionality to support the goals of code review (we look at this functionality in more detail later in this chapter), but why is it so successful? Critique has been shaped by Google's development culture, which includes code review as a core part of the workflow. This cultural influence translates into a set of guiding principles that Critique was designed to emphasize:

Simplicity

> Critique's user interface (UI) is based around making it easy to do code review without a lot of unnecessary choices, and with a smooth interface. The UI loads fast, navigation is easy and hotkey supported,

and there are clear visual markers for the overall state of whether a change has been reviewed.

Foundation of trust

Code review is not for slowing others down, instead it is to empower others. Trusting colleagues as much as possible makes it work. This might mean, for example, trusting authors to make changes and not requiring an additional review phase to double check that minor comments are actually addressed. Trust also plays out by making changes openly accessible (for viewing and reviewing) across Google.

Generic communication

Communication problems are rarely solved through tooling. Critique prioritizes generic ways for users to comment on the code changes, instead of complicated protocols. Critique encourages users to spell out what they want in their comments or even suggest some edits, instead of making the data model and process more complex. Communication can go wrong, even with the best code review tool because the users are humans.

Workflow integration

Critique has a number of integration points with other core software development tools. Developers can easily navigate to view the code under review in our code search and browsing tool, edit code in our web-based code editing tool, or view test results associated with a code change.

Across these guiding principles, simplicity has probably had the most impact on the tool. There were many interesting features we considered to add, but decided not to make the model more complicated to support a small set of users.

Simplicity also has an interesting tension with workflow integration. We considered but ultimately decided against creating a "Code Central" tool with code editing, reviewing, and searching in one tool. Although Critique has many touchpoints with other tools, we consciously decided to keep code review as the primary focus. Features are linked from Critique but implemented in different subsystems.

Code Review Flow

Code reviews can be executed at many stages of software development, as illustrated in Figure 19-1. Critique reviews typically take place before a change can be committed to the codebase, also known as *precommit reviews*. Although Chapter 9 contains a brief description of the code review flow, here we expand it to describe key aspects of Critique that help at each stage. We'll look at each stage in more detail in the following sections.

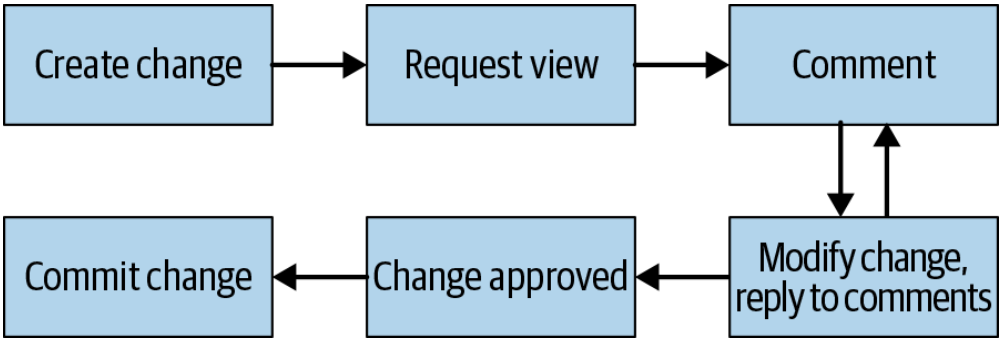

Figure 19-1. The code-review flow

Typical review steps go as follows:

1. **Creating a change.** A user authors a change to the codebase in their workspace. This *author* then uploads a *snapshot* (showing a patch at a particular point in time) to Critique, which triggers the run of automatic code analyzers (Chapter 20).
2. **Request review.** After the author is satisfied with the diff of the change and the result of the analyzers shown in Critique, they mail the change to one or more reviewers.
3. **Comment.** *Reviewers* open the change in Critique and draft comments on the diff. Comments are by default marked as *unresolved*, meaning they are crucial for the author to address. Additionally, reviewers can add *resolved* comments that are optional or informational. Results from automatic code analyzers, if present, are also visible to reviewers. Once a reviewer has drafted a set of comments, they need to *publish* them in order for the author to see them; this has the advantage of allowing a reviewer to provide a complete thought on a change atomically, after having reviewed the entire change. Anyone can comment on changes, providing a "drive-by review" as they see it necessary.
4. **Modify change and reply to comments.** The author modifies the change, uploads new snapshots based on the feedback, and replies back to the reviewers. The author addresses (at least) all unresolved comments, either by changing the code or just replying to the comment and changing the comment type to be *resolved*. The author and reviewers can look at diffs between any pairs of snapshots to see what changed. Steps 3 and 4 might be repeated multiple times.

5. **Change approval.** When the reviewers are happy with the latest state of the change, they approve the change and mark it as "looks good to me" (LGTM). They can optionally include comments to address. After a change is deemed good for submission, it is clearly marked green in the UI to show this state.
6. **Commit a change.** Provided the change is approved (which we'll discuss shortly), the author can trigger the commit process of the change. If automatic analyzers and other precommit hooks (called "presubmits") don't find any problems, the change is committed to the codebase.

Even after the review process is started, the entire system provides significant flexibility to deviate from the regular review flow. For example, reviewers can un-assign themselves from the change, explicitly assign it to someone else, and the author can postpone the review altogether. In emergency cases, the author can forcefully commit their change and have it reviewed after commit.

Notifications

As a change moves through the stages outlined earlier, Critique publishes event notifications that might be used by other supporting tools. This notification model allows Critique to be the code review tool instead of adding arbitrary features yet still integrate into developer workflow. Notifications enable a separation of concerns such that Critique can just emit events and other systems build off of those events.

For example, users can install a Chrome extension that consumes these event notifications. When a change needs the user's attention—for example, because it is their turn to review the change or some presubmit fails—the extension displays a Chrome notification with a button to go directly to the change or silence the notification. We have found that some developers really like immediate notification of changes updates, but others choose not to use this extension because they find it is too disruptive to their flow.

Critique also manages emails related to a change; important Critique events trigger email notifications. Some analyzer findings are configured to also send results out by email (in addition to being displayed in the Critique UI). Critique also processes email replies and translates them to comments, supporting users who prefer an email-based flow. Note that for many users, emails are not a key feature of code review; they use Critique's dashboard view (discussed later) to manage reviews.

Stage 1: Creating a Change

A code review tool should provide support at all stages of the review process and should not be the bottleneck for committing changes. In the prereview step, making it easier for change authors to polish a change before sending it out for review helps reduce the time taken by the reviewers to inspect the change. Critique displays change diffs with knobs to ignore whitespace changes and highlight move-only changes. Critique also surfaces the results from builds, tests, and static analyzers including style checks (as discussed in Chapter 9).

Showing an author the diff of a change gives them the opportunity to wear a different hat: that of a code reviewer. Critique lets a change author see the diff of their changes as their reviewer will, and also see the automatic analysis results. Critique also supports making lightweight modifications to the change from within the review tool and suggests appropriate reviewers. When sending out the request, the author can also include preliminary comments on the change, providing the opportunity to ask reviewers directly about any open questions. Giving authors the chance to see a change just as their reviewers do prevents misunderstanding.

To provide further context for the reviewers, the author can also link the change to a specific bug. Critique uses an autocomplete service to show relevant bugs, prioritizing bugs that are assigned to the author.

Diffing

The core of the code review process is understanding the code change itself. Larger changes are typically more difficult to understand than smaller ones. Optimizing the diff of a change is thus a core requirement for a good code review tool.

In Critique this principle translates onto multiple layers (see Figure 19-2). The diffing component, starting from an optimized longest common subsequence algorithm, is enhanced with the following:

- Syntax highlighting
- Cross-references (powered by Kythe; see Chapter 17)
- Intraline diffing that shows the difference on character-level factoring in the word boundaries (Figure 19-2)
- An option to ignore whitespace differences to a varying degree

- Move detection, in which chunks of code that are moved from one place to another are marked as being moved (as opposed to being marked as removed here and added there, as a naive diff algorithm would).

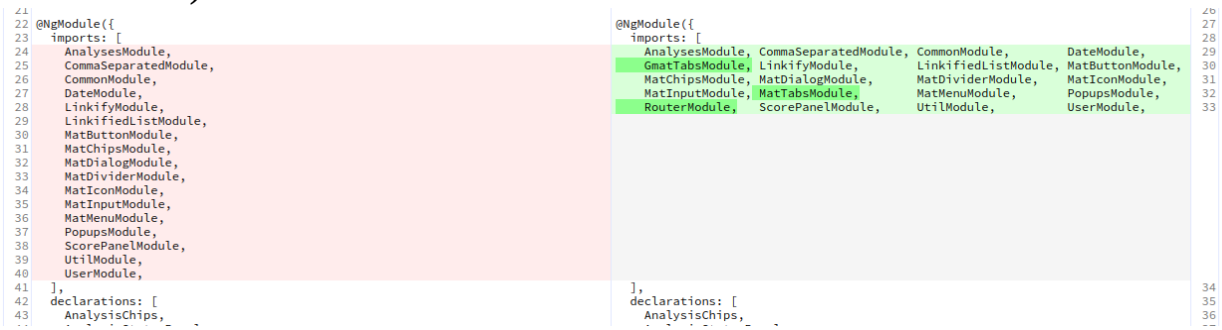

Figure 19-2. Intraline diffing showing character-level differences

Users can also view the diff in various different modes such as overlay and side by side. When developing Critique, we decided that it was important to have side-by-side diffs to make the review process easier. Side-by-side diffs take a lot of space: to make them a reality, we had to simplify the diff view structure, so there is no border, no padding, just the diff and line numbers. We also had to play around with a variety of fonts and sizes until we had a diff view that accommodates even for Java's 100-character line limit for the typical screen-width resolution when Critique launched (1,440 pixels).

Critique further supports a variety of custom tools that provide diffs of artifacts produced by a change, such as a screenshot diff of the UI modified by a change or configuration files generated by a change.

To make the process of navigating diffs smooth, we were careful not to waste space, and spent significant effort ensuring that diffs load quickly, even for images and large files and/or changes. We also provide keyboard shortcuts to quickly navigate through files while visiting only modified sections.

When users drill down to the file level, Critique provides a UI widget with a compact display of the chain of snapshot versions of a file; users can drag and drop to select which versions to compare. This widget automatically collapses similar snapshots, drawing focus to important snapshots. It helps the user understand the evolution of a file within a change; for example, which snapshots have test coverage, have already been reviewed, or have comments. To address concerns of scale, Critique prefetches everything so loading different snapshots is very quick.

Analysis Results

Uploading a snapshot of the change triggers code analyzers (see Chapter 20). Critique displays the analysis results on the change page, summarized by analyzer status chips shown below the change description, as depicted in Figure 19-3, and detailed in the Analysis tab, as illustrated in Figure 19-4.

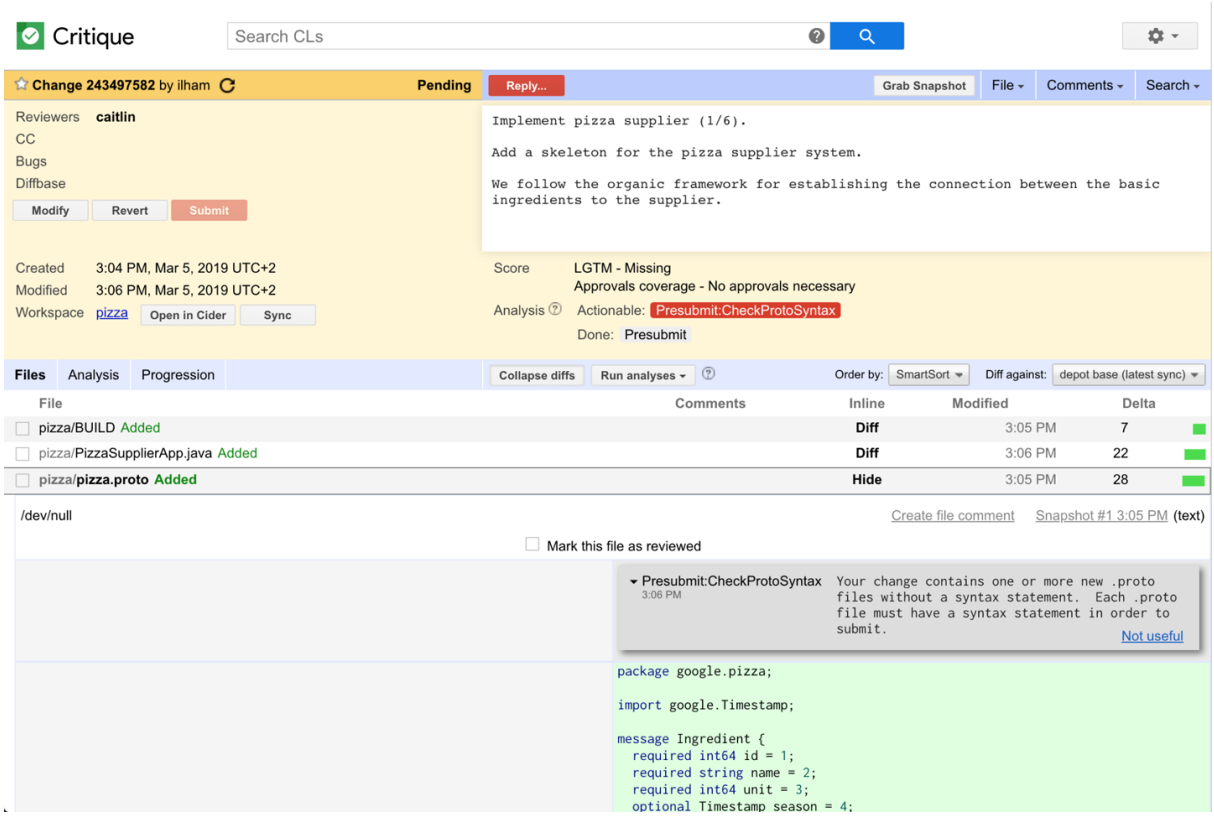

Figure 19-3. Change summary and diff view

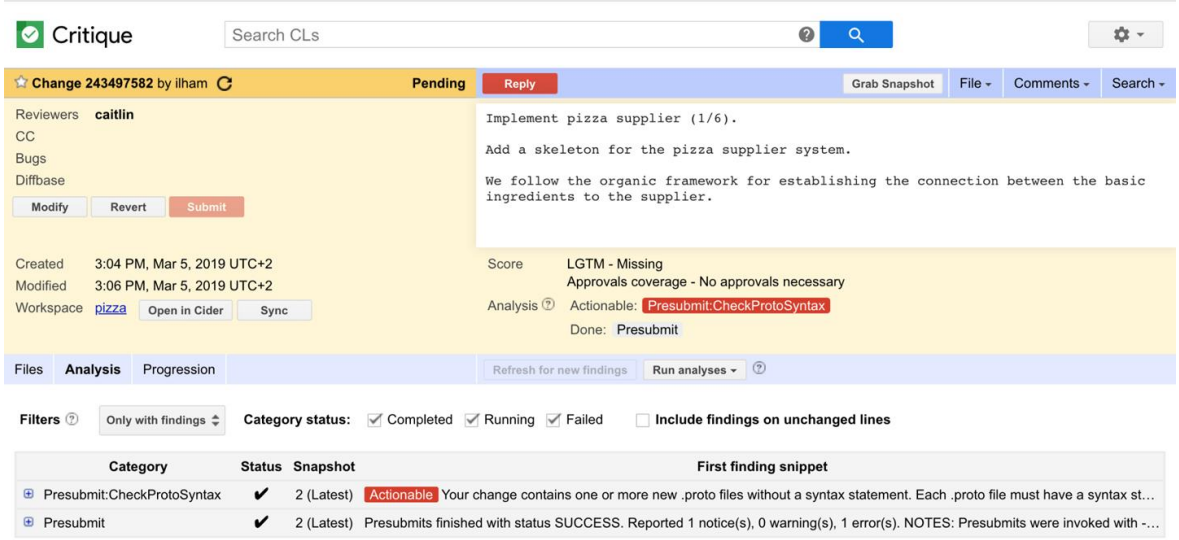

Figure 19-4. Analysis results

Analyzers can mark specific findings to highlight in red for increased visibility. Analyzers that are still in progress are represented by yellow chips,

and gray chips are displayed otherwise. For the sake of simplicity, Critique offers no other options to mark or highlight findings—actionability is a binary option. If an analyzer produces some results ("findings"), clicking the chip opens up the findings. Like comments, findings can be displayed inside the diff but styled differently to make them easily distinguishable. Sometimes, the findings also include fix suggestions, which the author can preview and choose to apply from Critique.

For example, suppose that a linter finds a style violation of extra spaces at the end of the line. The change page will display a chip for that linter. From the chip, the author can quickly go to the diff showing the offending code to understand the style violation with two clicks. Most linter violations also include fix suggestions. With a click, the author can preview the fix suggestion (for example, remove the extra spaces), and with another click apply the fix on the change.

Tight Tool Integration

Google has tools built on top of Piper, its monolithic source-code repository (see Chapter 16), such as the following:

- Cider, an online IDE for editing source code stored in the cloud
- Code Search, a tool for searching code in the codebase
- Tricorder, a tool for displaying static analysis results (mentioned earlier)
- Rapid, a release tool that packages and deploys binaries containing a series of changes
- Zapfhahn, a test coverage calculation tool

Additionally, there are services that provide context on change metadata (for example, about users involved in a change or linked bugs). Critique is a natural melting pot for a quick one-click/hover access or even embedded UI support to these systems, although we need to be careful not to sacrifice simplicity. For example, from a change page in Critique, the author needs to click only once to start editing the change further in Cider. There is support to navigate between cross-references using Kythe or view the mainline state of the code in Code Search Chapter 17. Critique links out to the release tool so that users can see whether a submitted change is in a specific release. For these tools, Critique favors links rather than embedding so as not to distract from the core review experience. One exception here is test coverage: the information whether a line of code is covered by a test is shown by different

background colors on the line gutter in the file's diff view (not all projects use this coverage tool).

Note that tight integration between Critique and a developer's workspace is possible because of the fact that workspaces are stored in a FUSE-based filesystem, accessible beyond a particular developer's computer. The Source of Truth is hosted in the cloud and accessible to all of these tools.

Stage 2: Request Review

After the author is happy with the state of the change, they can send it for review, as depicted in Figure 19-5. This requires the author to pick the reviewers. Within a small team, finding a reviewer might seem simple, but even there it is useful to distribute reviews evenly across team members and consider situations like who is on vacation. To address this, teams can provide an email alias for incoming code reviews. The alias is used by a tool called *GwsQ* (named after the initial team that used this technique: Google Web Server) that assigns specific reviewers based on the configuration linked to the alias. For example, a change author can assign a review to some-team-list-alias, and GwsQ will pick a specific member of some-team-list-alias to perform the review.

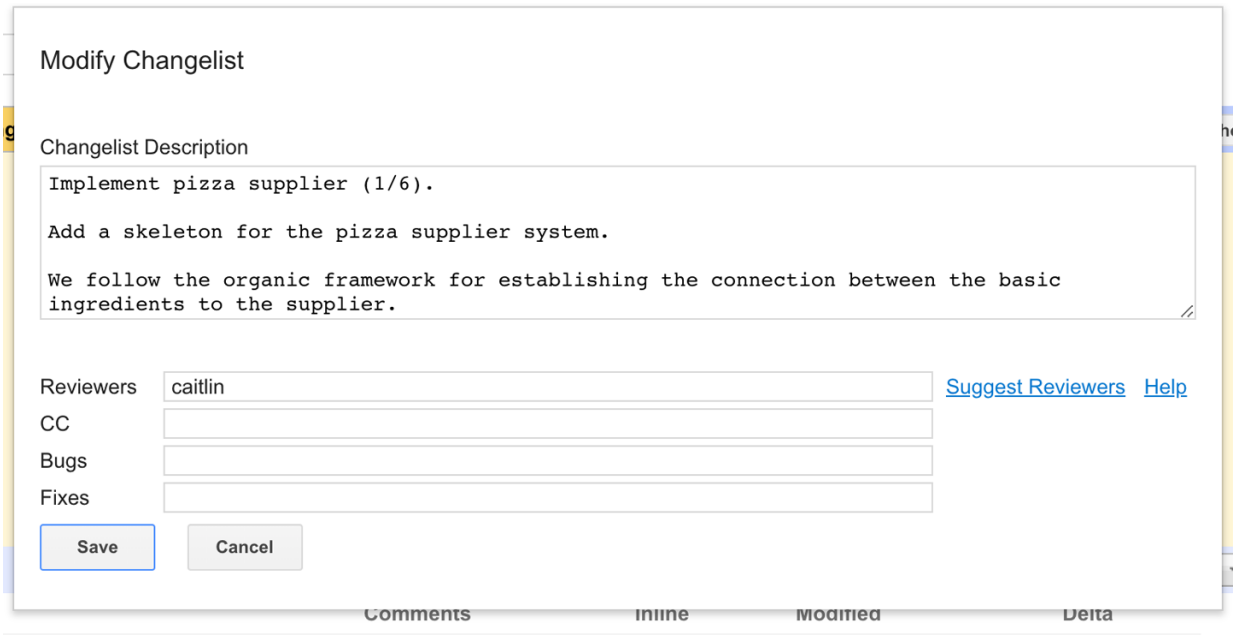

Figure 19-5. Requesting reviewers

Given the size of Google's codebase and the number of people modifying it, it can be difficult to find out who is best qualified to review a change outside your own project. Finding reviewers is a problem to consider when reaching

a certain scale. Critique must deal with scale. Critique offers the functionality to propose sets of reviewers that are sufficient to approve the change. The reviewer selection utility takes into account the following factors:

- Who owns the code that is being changed (see the next section)
- Who is most familiar with the code (i.e., who recently changed it)
- Who is available for review (i.e., not out of office and preferably in the same time zone)
- The GwsQ team alias setup

Assigning a reviewer to a change triggers a review request. This request runs "presubmits" or precommit hooks applicable to the change; teams can configure the presubmits related to their projects in many ways. The most common hooks include the following:

- Automatically adding email lists to changes to raise awareness and transparency
- Running automated test suites for the project
- Enforcing project-specific invariants on both code (to enforce local code style restrictions) and change descriptions (to allow generation of release notes or other forms of tracking)

As running tests is resource-intensive, at Google they are part of presubmits (run when requesting review and when committing changes) rather than for every snapshot like Tricorder checks. Critique surfaces the result of running the hooks in a similar way to how analyzer results are displayed, with an extra distinction to highlight the fact that a failed result blocks the change from being sent for review or committed. Critique notifies the author via email if presubmits fail.

Stage 3 and 4: Understanding and Commenting on a Change

After the review process starts, the author and the reviewers work in tandem to reach the goal of committing changes of high quality.

Commenting

Making comments is the second most common action that users make in Critique after viewing changes (Figure 19-6). Commenting in Critique is free for all. Anyone—not only the change author and the assigned reviewers—can comment on a change.

Critique also offers the ability to track review progress via per-person state. Reviewers have checkboxes to mark individual files at the latest snapshot as reviewed, helping the reviewer keep track of what they have already looked at. When the author modifies a file, the "reviewed" checkbox for that file is cleared for all reviewers because the latest snapshot has been updated.

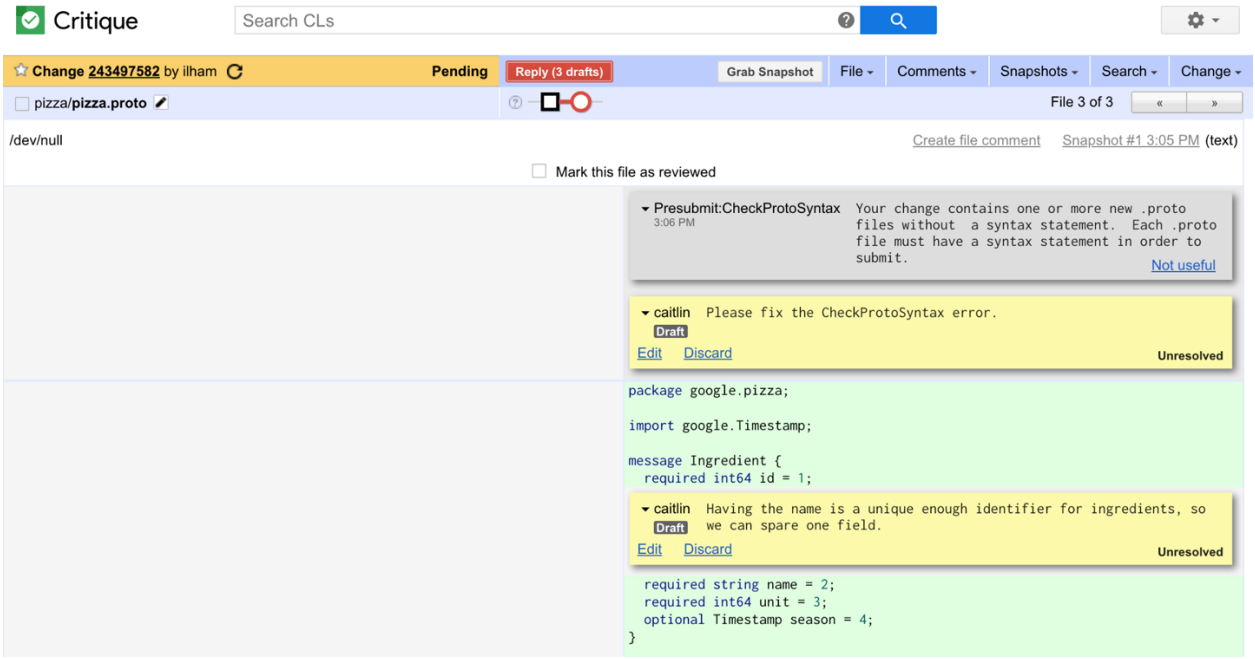

Figure 19-6. Commenting on the diff view

When a reviewer sees a relevant analyzer finding, they can click a "Please fix" button to create an unresolved comment asking the author to address the finding. Reviewers can also suggest a fix to a change by inline editing the latest version of the file. Critique transforms this suggestion into a comment with a fix attached that can be applied by the author.

Critique does not dictate what comments users should create, but for some common comments, Critique provides quick shortcuts. The change author can click the "Done" button on the comment panel to indicate when a reviewer's comment has been addressed, or the "Ack" button to acknowledge that the comment has been read, typically used for informational or optional comments. Both have the effect of resolving the comment thread if it is

unresolved. These shortcuts simplify the workflow and reduce the time needed to respond to review comments.

As mentioned earlier, comments are drafted as-you-go, but then "published" atomically, as shown in Figure 19-7. This allows authors and reviewers to ensure that they are happy with their comments before sending them out.

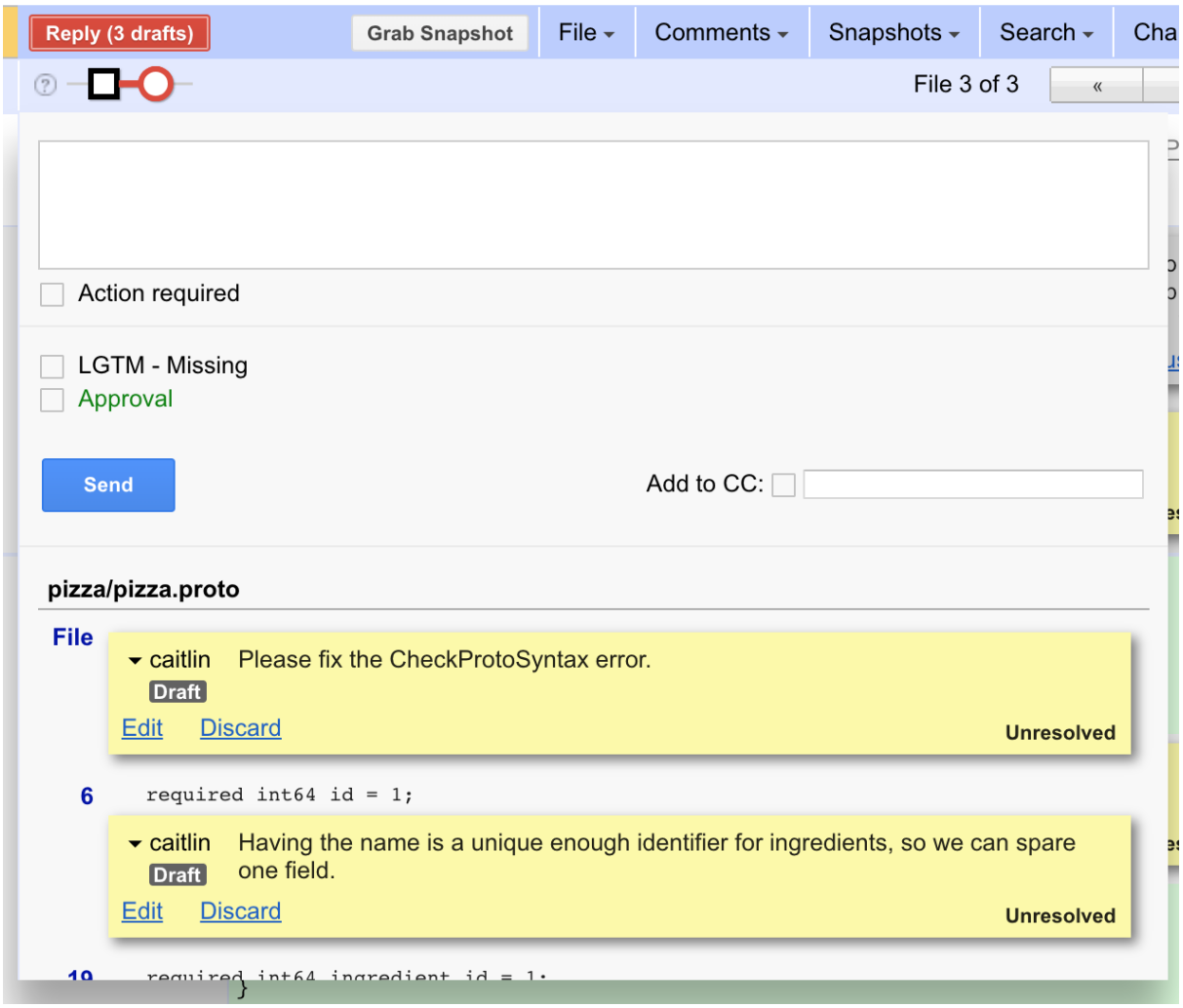

Figure 19-7. Preparing comments to the author

Understanding the State of a Change

Critique provides a number of mechanisms to make it clear where in the comment-and-iterate phase a change is currently located. These include a feature for determining who next needs to take action, and a dashboard view of review/author status for all of the changes with which a particular developer is involved.

"WHOSE TURN" FEATURE

One important factor in accelerating the review process is understanding when it's your turn to act, especially when there are multiple reviewers assigned to a change. This might be the case if the author wants to have their change reviewed by a software engineer and the user-experience person responsible for the feature, or the SRE carrying the pager for the service. Critique helps define who is expected to look at the change next by managing an *attention set* for each change.

The attention set comprises the set of people on which a change is currently blocked. When a reviewer or author is in the attention set, they are expected to respond in a timely manner. Critique tries to be smart about updating the attention set when a user publishes their comments, but users can also manage the attention set themselves. Its usefulness increases even more when there are more reviewers in the change. The attention set is surfaced in Critique by rendering the relevant usernames in bold.

After we implemented this feature, our users had a difficult time imagining the previous state. The prevailing opinion is how did we get along without this? The alternative before we implemented this feature was chatting between reviewers and authors to understand who was dealing with a change. This feature also emphasizes the turn-based nature of code review; it is always at least one person's turn to take action.

DASHBOARD AND SEARCH SYSTEM

Critique's landing page is the user's dashboard page, as depicted in Figure 19-8. The dashboard page is divided into user-customizable sections, each of them containing a list of change summaries.

Critique

Search CLs

Dashboard | Weekly snippets | Starred CLs | Refresh

Needs attention 4 Changes

	Change	Author	Status	Last Action	Re
⊞ ☆	**42972248**	ilham	**Pending**	Apr 11 by gwsq	**ca**
⊞ ☆	**42974683**	ilham	**Pending**	Apr 11 by tap	**ca**
⊞ ☆	**37099895**	ilham	**Pending**	Apr 11 by ilham	**ca**
⊞ ☆	**27761071**	**caitlin**	**Pending**	Jan 8 by ilham	ilh

Incoming reviews 6 Changes

	Change	Author	Status	Last Action	Re
⊞ ☆	**42972248**	ilham	**Pending**	Apr 11 by gwsq	**ca**
⊞ ☆	**42974683**	ilham	**Pending**	Apr 11 by tap	**ca**
⊞ ☆	**37099895**	ilham	**Pending**	Apr 11 by ilham	**ca**
⊞ ☆	42161351	**ilham**	LGTM	Apr 9 by caitlin	ca
⊞ ☆	40374250	**ilham**	Unresolved	Apr 4 by caitlin	ca
⊞ ☆	36387832	**ilham**	Unresolved	Mar 5 by caitlin	ca

Outgoing reviews 3 Changes

	Change	Author	Status	Last Action	Re
⊞ ☆	**27761071**	**caitlin**	**Pending**	Jan 8 by caitlin	ilh
⊞ ☆	15068925	caitlin	Pending	Jan 6 by caitlin	**ilh**
⊞ ☆	15416497	caitlin	Pending	Jan 2 by caitlin	**ilh**

Figure 19-8. Dashboard view

The dashboard page is powered by a search system called *Changelist Search*. Changelist Search indexes the latest state of all available changes (both pre and post submit) across all users at Google and allows its users to look up relevant changes by regular expression–based queries. Each dashboard section is defined by a query to Changelist Search. We have spent time ensuring Changelist Search is fast enough for interactive use; everything is indexed quickly so that authors and reviewers are not slowed down, despite the fact that we have an extremely large number of concurrent changes happening simultaneously at Google.

To optimize the user experience (UX), Critique's default dashboard setting is to have the first section display the changes that need a user's attention, although this is customizable. There is also a search bar for making custom queries over all changes and browsing the results. As a reviewer, you mostly just need the attention set. As an author, you mostly just need to take a look at what is still waiting for review to see if you need to ping any changes. Although we have shied away from customizability in some other parts of the Critique UI, we found that users like to set up their dashboard differently without detracting from the fundamental experience, similar to the way everyone organizes their emails differently.[1]

Stage 5: Change Approvals (Scoring a Change)

Showing whether a reviewer thinks a change is good boils down to providing concerns and suggestions via comments. There also needs to be some mechanism for providing a high-level "OK" on a change. At Google, the scoring for a change is divided into three parts:

- LGTM ("looks good to me")
- Approval
- The number of unresolved comments

An LGTM stamp from a reviewer means that "I have reviewed this change, believe that it meets our standards and I think it is okay to commit it after addressing unresolved comments." An Approval stamp from a reviewer means that "as a gatekeeper, I allow this change to be committed to the codebase". A reviewer can mark comments as unresolved, meaning that the author will need to act upon them. When the change has at least one LGTM, sufficient approvals and no unresolved comments, the author can then

commit the change. Note that every change requires an LGTM regardless of approval status, ensuring that at least two pairs of eyes viewed the change. This simple scoring rule allows Critique to inform the author when a change is ready to commit (shown prominently as green page header).

We made a conscious decision in the process of building Critique to simplify this rating scheme. Initially, Critique had a "Needs More Work" rating and also a "LGTM++". The model we have moved to is to make LGTM/Approval always positive. If a change definitely needs a second review, `primary reviewers can` add comments but without LGTM/Approval. After a change transitions into a mostly-good state, reviewers will typically trust authors to take care of small edits—the tooling does not require repeated LGTMs regardless of change size.

This rating scheme has also had a positive influence on code review culture. Reviewers cannot just thumbs-down a change with no useful feedback; all negative feedback from reviewers must be tied to something specific to be fixed (for example, an unresolved comment). The phrasing "unresolved comment" was also chosen to sound relatively nice.

Critique includes a scoring panel, next to the analysis chips, with the following information:

- Who has LGTM'ed the change
- What approvals are still required and why
- How many unresolved comments are still open

Presenting the scoring information this way helps the author quickly understand what they still need to do to get the change committed.

LGTM and Approval are *hard* requirements and can be granted only by reviewers. Reviewers can also revoke their LGTM and Approval at any time before the change is committed. Unresolved comments are *soft* requirements; the author can mark a comment "resolved" as they reply. This distinction promotes and relies on trust and communication between the author and the reviewers. For example, a reviewer can LGTM the change accompanied with unresolved comments without later on checking precisely whether the comments are truly addressed, highlighting the trust the reviewer places on the author. This trust is particularly important for saving time when there is a significant difference in time zones between the author and the reviewer. Exhibiting trust is also a good way to build trust and strengthen teams.

Stage 6: Commiting a Change

Last but not least, Critique has a button for committing the change after the review to avoid context-switching to a command-line interface.

After Commit: Tracking History

In addition to the core use of Critique as a tool for reviewing source code changes before they are committed to the repository, Critique is also used as a tool for change archaeology. For most files, developers can view a list of the past history of changes that modified a particular file in the Code Search system (see Chapter 17), or navigate directly to a change. Anyone at Google can browse the history of a change to generally viewable files, including the comments on and evolution of the change. This enables future auditing and is used to understand more details about why changes were made or how bugs were introduced. Developers can also use this feature to learn how changes were engineered, and code review data in aggregate is used to produce trainings.

Critique also supports the ability to comment after a change is committed, for example when a problem is discovered later or additional context might be useful for someone investigating the change at another time. Critique also supports the ability to rollback changes and see whether a particular change has already been rolled back.

Gerrit

Although Critique is the most commonly used review tool at Google, it is not the only one. Critique is not externally available due to its tight interdependencies with our large monolithic repository and other internal tools. Because of this, teams at Google that work on open source projects (including Chrome and Android) or internal projects that can't or don't want to be hosted in the monolithic repository use a different code review tool: Gerrit.

Gerrit is a standalone, open source code review tool that is tightly integrated with the Git version control system. As such, it offers a web UI to many Git features including code browsing, merging branches, cherry-picking commits, and, of course code, review. In addition, Gerrit has a fine-grained permission model that we can use to restrict access to repositories and branches.

Both Critique and Gerrit have the same model for code reviews in that each commit is reviewed separately. Gerrit supports stacking commits and uploading them for individual review. It also allows the chain to be committed atomically after it's reviewed.

Being open source, Gerrit accommodates more variants and a wider range of use cases; Gerrit's rich plug-in system enables a tight integration into custom environments. To support these use cases, Gerrit also supports a more sophisticated scoring system. A reviewer can veto a change by placing a –2 score, and the scoring system is highly configurable.

NOTE

You can learn more about Gerrit and see it in action at *https://www.gerritcodereview.com*

Conclusion

There are a number of implicit trade-offs when using a code review tool. Critique builds in a number of features and integrates with other tools to make the review process more seamless for its users. Time spent in code reviews is time not spent coding, so any optimization of the review process can be a productivity gain for the company. Having only two people in most cases (author and reviewer) agree on the change before it can be committed keeps velocity high. Google greatly values the educational aspects of code review, even though they are more difficult to quantify.

To minimize the time it takes for a change to be reviewed, the code review process should flow seamlessly, informing users succinctly of the changes that need their attention and identifying potential issues before human reviewers come in (issues are caught by analyzers and Continuous Integration). When possible, quick analysis results are presented before the longer-running analyses can finish.

There are several ways in which Critique needs to support questions of scale. The Critique tool must scale to the large quantity of review requests produced without suffering a degradation in performance. Because Critique is on the critical path to getting changes committed, it must load efficiently and be usable for special situations such as unusually large changes.[2] The interface must support managing user activities (such as finding relevant changes) over the large codebase and help reviewers and authors navigate the codebase. For example, Critique helps with finding appropriate reviewers for a change

without having to figure out the ownership/maintainer landscape (a feature that is particularly important for large-scale changes such as API migrations that can affect many files).

Critique favors an opinionated process and a simple interface to improve the general review workflow. However, Critique does allow some customizability: custom analyzers and presubmits provide specific context on changes, and some team-specific policies (such as requiring LGTM from multiple reviewers) can be enforced.

Trust and communication are core to the code review process. A tool can enhance the experience, but can't replace them. Tight integration with other tools has also been a key factor in Critique's success.

TL;DRs

- Trust and communication are core to the code review process. A tool can enhance the experience, but it can't replace them.
- Tight integration with other tools is key to great code review experience.
- Small workflow optimizations, like the addition of an explicit "attention set," can increase clarity and reduce friction substantially.

1 Centralized "global" reviewers for large-scale changes (LSCs) are particularly prone to customizing this dashboard to avoid flooding it during an LSC. Chapter 22.

2 Although most changes are small (less than 100 lines), Critique is sometimes used to review large refactoring changes that can touch hundreds or thousands of files, especially for LSCs that must be executed atomically (see Chapter 22).

Chapter 20. Static Analysis

Written by Cailtin Sadowski

Edited by Lisa Carey

Static analysis refers to programs analyzing source code to find potential issues such as bugs, antipatterns, and other potential issues that can be diagnosed *without executing the program*. The "static" part specifically refers to analyzing the source code instead of a running program (referred to as "dynamic" analysis). Static analysis can find bugs in programs early, before they are checked in as production code. For example, static analysis can identify constant expressions that overflow, tests that are never run, or invalid format strings in logging statements that would crash when executed.[1] However, static analysis is useful for more than just finding bugs. Through static analysis at Google, we codify best practices, help keep code current to modern API versions, and prevent or reduce technical debt. Examples of these analyses include verifying that naming conventions are upheld, flagging the use of deprecated APIs, or pointing out simpler but equivalent expressions that make code easier to read. Static analysis is also an integral tool in the API deprecation process, where it can prevent backsliding during migration of the codebase to a new API (see Chapter 22). We have also found evidence that static-analysis checks can educate developers and actually prevent antipatterns from entering the codebase.[2]

In this chapter, we'll look at what makes effective static analysis, some of the lessons we at Google have learned about making static analysis work, and how we implemented these best practices in our static analysis tooling and processes.[3]

Characteristics of Effective Static Analysis

Though there have been decades of static analysis research focused on developing new analysis techniques and specific analyses, a focus on approaches for improving *scalability* and *usability* of static analysis tools has been a relatively recent development.

Scalability

Because modern software has become larger, analysis tools must explicitly address scaling in order to produce results in a timely manner, without slowing down the software development process. Static analysis tools at Google must scale to the size of Google's multibillion-line codebase. To do this, analysis tools are shardable and incremental. Instead of analyzing entire large projects, we focus analyses on files affected by a pending code change, and typically show analysis results only for edited files or lines. Scaling also has benefits: because our codebase is so large, there is a lot of low-hanging fruit in terms of bugs to find. In addition to making sure analysis tools can run on a large codebase, we also must scale up the number and variety of analyses available. Analysis contributions are solicited from throughout the company. Another component to static analysis scalability is ensuring the *process* is scalable. To do this, Google static analysis infrastructure avoids bottlenecking analysis results by showing them directly to relevant engineers.

Usability

When thinking about analysis usability, it is important to consider the cost-benefit trade-off for static analysis tool users. This "cost" could either be in terms of developer time or code quality. Fixing a static analysis warning could introduce a bug. For code that is not being frequently modified, why "fix" code that is running fine in production? For example, fixing a dead code warning by adding a call to the previously dead code could result in untested (possibly buggy) code suddenly running. There is unclear benefit and potentially high cost. For this reason, we generally focus on newly introduced warnings; existing issues in otherwise working code are typically only worth highlighting (and fixing) if they are particularly important (security issues, significant bug fixes, etc.). Focusing on newly introduced warnings (or warnings on modified lines) also means that the developers viewing the warnings have the most relevant context on them.

Also, developer time is valuable! Time spent triaging analysis reports or fixing highlighted issues is weighed against the benefit provided by a particular analysis. If the analysis author can save time (e.g., by providing a fix that can be automatically applied to the code in question), the cost in the trade-off goes down. Anything that can be fixed automatically, should be fixed automatically. We also try to show developers reports about issues that actually have a negative impact on code quality so that they do not waste time slogging through irrelevant results.

To further reduce the cost of reviewing static analysis results, we focus on smooth developer workflow integration. A further strength of homogenizing everything in one workflow is that a dedicated tools team can update tools along with workflow and code, allowing analysis tools to evolve with the source code in tandem.

We believe these choices and trade-offs that we have made in making static analyses scalable and usable arise organically from our focus on three core principles, which we formulate as lessons in the next section.

Key Lessons in Making Static Analysis Work

There are three key lessons that we have learned at Google about what makes static analysis tools work well. Let's take a look at them in the following subsections.

Focus on Developer Happiness

We mentioned some of the ways in which we try to save developer time and reduce the cost of interacting with the aforementioned static analysis tools; we also keep track of how well analysis tools are performing. If you don't measure this, you can't fix problems. We only deploy analysis tools with low false-positive rates (more on that in a minute). We also *actively solicit and act on feedback* from developers consuming static analysis results, in real time. Nurturing this feedback loop between static analysis tool users and tool developers creates a virtuous cycle that has built up user trust and improved our tools. User trust is extremely important for the success of static analysis tools.

For static analysis, a "false negative" is when a piece of code contains an issue that the analysis tool was designed to find, but the tool misses it. A "false positive" occurs when a tool incorrectly flags code as having the issue. Research about static analysis tools traditionally focused on reducing false negatives; in practice, low false-positive rates are often critical for developers to actually want to use a tool—who wants to wade through hundreds of false reports in search of a few true ones?[4]

Furthermore, *perception* is a key aspect of the false positive rate. If a static analysis tool is producing warnings that are technically correct but misinterpreted by users as false positives (e.g., due to confusing messages), users will react the same as if those warnings were in fact false positives. Similarly, warnings that are technically correct but unimportant in the grand

scheme of things provoke the same reaction. We call the user-perceived false-positive rate the "effective false positive" rate. An issue is an "effective false positive" if developers did not take some positive action after seeing the issue. This means that if an analysis incorrectly reports an issue yet the developer happily makes the fix anyway to improve code readability or maintainability, that is not an effective false positive. For example, we have a Java analysis that flags cases in which a developer calls the `contains` method on a hash table (which is equivalent to `containsValue`) when they actually meant to call `containsKey`—even if the developer correctly meant to check for the value, calling `containsValue` instead is clearer. Similarly, if an analysis reports an actual fault, yet the developer did not understand the fault and therefore took no action, that is an effective false positive.

Make Static Analysis a Part of the Core Developer Workflow

At Google, we integrate static analysis into the core workflow via integration with code review tooling. Essentially all code committed at Google is reviewed before being committed; because developers are already in a change mindset when they send code for review, improvements suggested by static analysis tools can be made without too much disruption. There are other benefits to code review integration. Developers typically context switch after sending code for review, and are blocked on reviewers—there is time for analyses to run, even if they take several minutes to do so. There is also peer pressure from reviewers to address static analysis warnings. Furthermore, static analysis can save reviewer time by highlighting common issues automatically; static analysis tools help the code review process (and the reviewers) scale. Code review is a sweet spot for analysis results.[5]

Empower Users to Contribute

There are many domain experts at Google whose knowledge could improve code produced. Static analysis is an opportunity to leverage expertise and apply it at scale, by having domain experts write new analysis tools or individual checks within a tool. For example, experts that know the context for a particular kind of configuration file can write an analyzer that checks properties of those files. In addition to domain experts, analyses are contributed by developers who discover a bug and would like to prevent the same kind of bug from reappearing anywhere else in the codebase. We focus on building a static analysis ecosystem that is easy to plug into instead of integrating a small set of existing tools. We have focused on developing simple APIs that can be used by engineers throughout Google—not just analysis or language experts—to create analyses; for example,

Refaster[6] enables writing an analyzer by specifying pre and post code snippets demonstrating what transformations are expected by that analyzer.

Tricorder: Google's Static Analysis Platform

Tricorder, our static analysis platform, is a core part of static analysis at Google[7]. Tricorder came out of several failed attempts to integrate static analysis with the developer workflow at Google;[8] the key difference between Tricorder and previous attempts was our relentless focus on having Tricorder deliver only valuable results to its users. Tricorder is integrated with the main code review tool at Google, Critique. Tricorder warnings show up on Critique's diff viewer as gray comment boxes, as demonstrated in Figure 20-1.

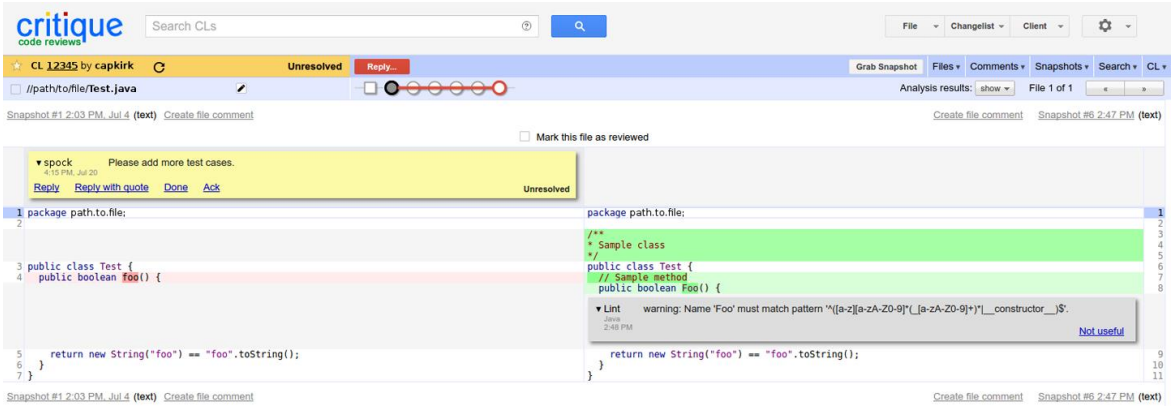

Figure 20-1. Critique's diff viewing, showing a static analysis warning from Tricorder in gray

To scale, Tricorder uses a microservices architecture. The Tricorder system sends analyze requests to analysis servers along with metadata about a code change. These servers can use that metadata to read the versions of the source code files in the change via a FUSE-based filesystem, and can access cached build inputs and outputs. The analysis server then starts running each individual analyzer and writes the output to a storage layer; the most recent results for each category are then displayed in Critique. Because analyses sometimes take a few minutes to run, analysis servers also post status updates to let change authors and reviewers know that analyzers are running, and post a completed status when they have finished. Tricorder analyzes more than 50,000 code review changes per day and is often running several analyses per second.

Developers throughout Google write Tricorder analyses (called "analyzers") or contribute individual "checks" to existing analyses. There are four criteria for new Tricorder checks:

Be understandable

Be easy for any engineer to understand the output.

Be actionable and easy to fix

The fix might require more time, thought, or effort than a compiler check, and the result should include guidance as to how the issue might indeed be fixed.

Produce less than 10% effective false positives

Developers should feel the check is pointing out an actual issue at least 90% of the time.

Have the potential for significant impact on code quality

The issues might not affect correctness, but developers should take them seriously and deliberately choose to fix them.

Tricorder analyzers report results for more than 30 languages and support a variety of analysis types. Tricorder includes more than 100 analyzers, with most being contributed from outside the Tricorder team. Seven of these analyzers are themselves plug-in systems that have hundreds of additional checks, again contributed from developers across Google. The overall effective false-positive rate is just below 5%.

Integrated Tools

There are many different types of static analysis tools integrated with Tricorder.

Error Prone and Clang Tidy extend the compiler to identify AST antipatterns for Java and C++, respectively. These antipatterns could represent real bugs. For example, consider the following code snippet hashing a field `f` of type `long`:

```
result = 31 * result + (int) (f ^ (f >>> 32));
```

Now consider the case in which the type of `f` is `int`. The code will still compile, but the right shift by 32 is a no-op so that `f` is XORed with itself and no longer affects the value produced. We fixed 31 occurrences of this bug in Google's codebase while enabling the check as a compiler error in Error Prone. There are many more such examples. AST antipatterns can also result in code readability improvements, such as removing a redundant call to `.get()` on a smart pointer.

Other analyzers showcase relationships between disparate files in a corpus. The Deleted Artifact Analyzer warns if a source file is deleted that is referenced by other non-code places in the codebase (such as inside checked-in documentation). IfThisThenThat allows developers to specify that portions of two different files must be changed in tandem (and warns if they are not). Chrome's Finch analyzer runs on configuration files for A/B experiments in Chrome, highlighting common problems including not having the right approvals to launch an experiment or crosstalk with other currently running experiments that affect the same population. The Finch analyzer makes Remote Procedure Calls (RPCs) to other services in order to provide this information.

In addition to the source code itself, some analyzers run on other artifacts produced by that source code; many projects have enabled a binary size checker that warns when changes significantly affect a binary size.

Almost all analyzers are intraprocedural, meaning that the analysis results are based on code within a procedure (function). Compositional or incremental interprocedural analysis techniques are technically feasible but would require additional infrastructure investment (e.g., analyzing and storing method summaries as analyzers run).

Integrated Feedback Channels

As mentioned earlier, establishing a feedback loop between analysis consumers and analysis writers is critical to track and maintain developer happiness. With Tricorder, we display the option to click a "Not useful" button on an analysis result; this click provides the option to file a bug *directly against the analyzer writer* about why the result is not useful with information about analysis result prepopulated. Code reviewers can also ask change authors to address analysis results by clicking a "Please fix" button. The Tricorder team tracks analyzers with high "Not useful" click rates, particularly relative to how often reviewers ask to fix analysis results, and will disable analyzers if they don't work to address problems and improve the "not useful" rate. Establishing and tuning this feedback loop took a lot of work, but has paid dividends many times over in improved analysis results and a better user experience (UX)—before we established clear feedback channels, many developers would just ignore analysis results they did not understand.

And sometimes the fix is pretty simple—such as updating the text of the message an analyzer outputs! For example, we once rolled out an Error Prone

check that flagged when too many arguments were being passed to a `printf`-like function in Guava that accepted only `%s` (and no other `printf` specifiers). The Error Prone team received weekly "Not useful" bug reports claiming the analysis was incorrect because the number of format specifiers matched the number of arguments—all due to users trying to pass specifiers other than `%s`. After the team changed the diagnostic text to state directly that the function accepts only the `%s` placeholder, the influx of bug reports stopped. Improving the message produced by an analysis provides an explanation of what is wrong, why, and how to fix it exactly at the point where that is most relevant and can make the difference for developers learning something when they read the message.

Suggested Fixes

Tricorder checks also, when possible, *provide fixes*, as shown in Figure 20-2.

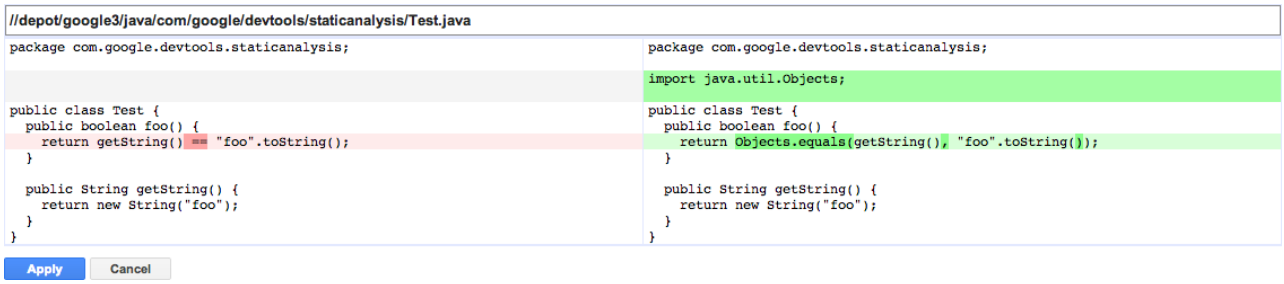

Figure 20-2. View of an example static analysis fix in Critique

Automated fixes serve as an additional documentation source when the message is unclear and, as mentioned earlier, reduce the cost to addressing static analysis issues. Fixes can be applied directly from within Critique, or over an entire code change via a command-line tool. Although not all analyzers provide fixes, many do. We take the approach that *style* issues in particular should be fixed automatically; for example, by formatters that automatically reformat source code files. Google has style guides for each language that specify formatting issues; pointing out formatting errors is not a good use of a human reviewer's time. Reviewers click "Please Fix" thousands of times per day, and authors applied the automated fixes approximately 3,000 times per day. And Tricorder analyzers received "Not useful" clicks 250 times per day.

Per-Project Customization

After we had built up a foundation of user trust by showing only high-confidence analysis results, we added the ability to run additional "optional" analyzers to specific projects in addition to the on-by-default ones. The *Proto*

Best Practices analyzer is an example of an optional analyzer. This analyzer highlights potentially breaking data format changes to protocol buffers—Google's language-independent data serialization format. These changes are only breaking when serialized data is stored somewhere (e.g., in server logs); protocol buffers for projects that do not have stored serialized data do not need to enable the check. We have also added the ability to customize existing analyzers, although typically this customization is limited and many checks are applied by default uniformly across the codebase.

Some analyzers have even started as optional, improved based on user feedback, built up a large userbase, and then graduated into on-by-default status as soon as we could capitalize on the user trust we had built up. For example, we have an analyzer that suggests Java code readability improvements that typically do not actually change code behavior. Tricorder users initially worried about this analysis being too "noisy," but eventually wanted more analysis results available.

The key insight to making this customization successful was to focus on *project-level customization, not user-level customization*. Project-level customization ensures that all team members have a consistent view of analysis results for their project, and prevents situations in which one developer is trying to fix an issue while another developer introduces it.

Early on in the development of Tricorder, a set of relatively straightforward style checkers ("linters") displayed results in Critique, and Critique provided user settings to choose the confidence level of results to display and suppress results from specific analyses. We removed all of this user customizability from Critique and immediately started getting complaints from users about annoying analysis results. Instead of reenabling customizability, we asked users why they were annoyed and found all kinds of bugs and false positives with the linters. For example, the C++ linter also ran on Objective-C files but produced incorrect, useless results. We fixed the linting infrastructure so that this would no longer happen. The HTML linter had an extremely high false-positive rate with very little useful signal and was typically suppressed from view by developers writing HTML. Because the linter was so rarely helpful, we just disabled this linter. In short, user customization resulted in hidden bugs and suppressing feedback.

Presubmits

In addition to code review, there are also other workflow integration points for static analysis at Google. Because developers can choose to ignore static

analysis warnings displayed in code review, Google additionally has the ability to add an analysis that blocks committing a pending code change, which we call a *presubmit check*. Presubmit checks include very simple customizable built-in checks on the contents or metadata of a change, such as ensuring that the commit message does not say "DO NOT SUBMIT" or that test files are always included with corresponding code files. Teams can also specify a suite of tests that must pass or verify that there are no Tricorder issues for a particular category. Presubmits also check that code is well formatted. Presubmit checks are typically run when a developer mails out a change for review and again during the commit process, but they can be triggered on an ad hoc basis in between those points. See Chapter 23 for more details on presubmits at Google.

Some teams have written their own custom presubmits. These are additional checks on top of the base presubmit set that add the ability to enforce higher best-practice standards than the company as a whole and add project-specific analysis. This enables new projects to have stricter best-practice guidelines than projects with large amounts of legacy code (for example). Team-specific presubmits can make the large-scale change (LSC) process (see Chapter 22) more difficult, so some are skipped for changes with "CLEANUP=" in the change description.

Compiler Integration

Although blocking commits with static analysis is great, it is even better to notify developers of problems even earlier in the workflow. When possible, we try to push static analysis into the compiler. Breaking the build is a warning that is not possible to ignore, but is infeasible in many cases. However, some analyses are highly mechanical and have no effective false positives. An example is Error Prone "ERROR" checks. These checks are all enabled in Google's Java compiler, preventing instances of the error from ever being introduced again into our codebase. Compiler checks need to be fast so that they don't slow down the build. In addition, we enforce these three criteria (similar criteria exist for the C++ compiler):

- Actionable and easy to fix (whenever possible, the error should include a suggested fix that can be applied mechanically)
- Produce no effective false positives (the analysis should never stop the build for correct code)
- Report issues affecting only correctness rather than style or best practices

To enable a new check, we first need to clean up all instances of that problem in the codebase so that we don't break the build for existing projects just because the compiler has evolved. This also implies that the value in deploying a new compiler-based check must be high enough to warrant fixing all existing instances of it. Google has infrastructure in place for running various compilers (such as clang and javac) over the entire codebase in parallel via a cluster—as a MapReduce operation. When compilers are run in this MapReduce fashion, the static analysis checks run must produce fixes in order to automate the cleanup. After a pending code change is prepared and tested that applies the fixes across the entire codebase, we commit that change and remove all existing instances of the problem. We then turn the check on in the compiler so that no new instances of the problem can be committed without breaking the build. Build breakages are caught after commit by our Continuous Integration (CI) system, or before commit by presubmit checks (see the earlier discussion).

We also aim to never issue compiler warnings. We have found repeatedly that developers ignore compiler warnings. We either enable a compiler check as an error (and break the build) or don't show it in compiler output. Because the same compiler flags are used throughout the codebase, this decision is made globally. Checks that can't be made to break the build are either suppressed or shown in code review (e.g., through Tricorder). Although not every language at Google has this policy, the most frequently used ones do. Both of the Java and C++ compilers have been configured to avoid displaying compiler warnings. The Go compiler takes this to extreme; some things that other languages would consider warnings (such as unused variables or package imports) are errors in Go.

Analysis While Editing and Browsing Code

Another potential integration point for static analysis is in an integrated development environment (IDE). However, IDE analyses require quick analysis times (typically less than 1 second and ideally less than 100 ms), and so some tools are not suitable to integrate here. In addition, there is the problem of making sure the same analysis runs identically in multiple IDEs. We also note that IDEs can rise and fall in popularity (we don't mandate a single IDE); hence IDE integration tends to be messier than plugging into the review process. Code review also has specific benefits for displaying analysis results. Analyses can take into account the entire context of the change; some analyses can be inaccurate on partial code (such as a dead code analysis when a function is implemented before adding callsites). Showing analysis results in code review also means that code authors have to convince reviewers, as

well, if they want to ignore analysis results. That said, IDE integration for suitable analyses is another great place to display static analysis results.

Although we mostly focus on showing newly introduced static analysis warnings, or warnings on edited code, for some analyses developers actually do want the ability to view analysis results over the entire codebase during code browsing. An example of this are some security analyses. Specific security teams at Google want to see a holistic view of all instances of a problem. Developers also like viewing analysis results over the codebase when planning a cleanup. In other words, there are times when showing results when code browsing is the right choice.

Conclusion

Static analysis can be a great tool to improve a codebase, find bugs early, and allow more expensive processes (such as human review and testing) to focus on issues that are not mechanically verifiable. By improving the scalability and usability of our static analysis infrastructure, we have made static analysis an effective component of software development at Google.

TL;DRs

- *Focus on developer happiness*. We have invested considerable effort in building feedback channels between analysis users and analysis writers in our tools, and aggressively tune analyses to reduce the number of false positives.
- *Make static analysis part of the core developer workflow*. The main integration point for static analysis at Google is through code review, where analysis tools provide fixes and involve reviewers. However, we also integrate analyses at additional points (via compiler checks, gating code commits, in IDEs, and when browsing code).
- *Empower users to contribute*. We can scale the work we do building and maintaining analysis tools and platforms by leveraging the expertise of domain experts. Developers are continuously adding new analyses and checks that make their lives easier and our codebase better.

1 *http://errorprone.info/bugpatterns*

2 Sadowski, Caitlin, Jeffrey van Gogh, Ciera Jaspan, Emma Soederberg, Collin Winter. (2015). "Tricorder: Building a Program Analysis Ecosystem." International Conference on Software Engineering (ICSE).

3 A good academic reference for static analysis theory is Principles of Program Analysis by Flemming Nielson, Hanne R. Nielson, Chris Hankin, 2004.

4 Note that there are some specific analyses for which reviewers might be willing to tolerate a much higher false-positive rate: one example is security analyses that identify critical problems.

5 See later in this chapter for more information on additional integration points when editing and browsing code.

6 Wasserman, Louis. (2013). "Scalable, Example-Based Refactorings with Refaster." Workshop on Refactoring Tools.

7 Sadowski, Caitlin, Jeffrey van Gogh, Ciera Jaspan, Emma Soederberg, Collin Winter. (2015). Tricorder: Building a Program Analysis Ecosystem. International Conference on Software Engineering (ICSE).

8 Sadowski, Caitlin, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, Ciera Jaspan. (2018). Lessons from Building Static Analysis Tools at Google. Communications of the ACM (CACM), vol. 61 Issue 4; pp. 58–66.

Chapter 21. Dependency Management

Written by Titus Winters

Edited by Lisa Carey

Dependency management—the management of networks of libraries, packages, and dependencies that we don't control—is one of the least understood and most challenging problems in software engineering. Dependency management focuses on questions like: how do we update between versions of external dependencies? How do we describe versions, for that matter? What types of changes are allowed or expected in our dependencies? How do we decide when it is wise to depend on code produced by other organizations?

For comparison, the most closely related topic here is source control. Both areas describe how we work with source code. Source control covers the easier part: where do we check things in? How do we get things into the build? After we accept the value of trunk-based development, most of the day-to-day source-control questions for an organization are fairly mundane: "I've got a new thing, what directory do I add it to?"

Dependency management adds additional complexity in both time and scale. In a trunk-based source-control problem, it's fairly clear when you make a change that you need to run the tests and not break existing code. That's predicated on the idea that you're working in a shared codebase, have visibility into how things are being used, can trigger the build and run the tests. Dependency management focuses on the problems that arise when changes are being made outside of your organization, without full access or visibility. Because your upstream dependencies can't coordinate with your private code, they are more likely to break your build and cause your tests to fail. How do we manage that? Should we not take external dependencies? Should we ask for greater consistency between releases of external dependencies? When do we update to a new version?

Scale makes all of these questions more complex, with the realization that we aren't really talking about single dependency imports, and in the general case that we're depending on an entire network of external dependencies. When we begin dealing with a network, it is easy to construct scenarios in which your organization's use of two dependencies becomes unsatisfiable at some point in time. Generally, this happens because one dependency stops working

without some requirement,[1] whereas the other is incompatible with the same requirement. Simple solutions about how to manage a single outside dependency usually fail to account for the realities of managing a large network. We'll spend much of this chapter discussing various forms of these conflicting requirement problems.

Source control and dependency management are related issues separated by the question: "Does our organization control the development/update/management of this subproject?" For example, if every team in your company has separate repositories, goals, and development practices, the interaction and management of code produced by those teams is going to have more to do with dependency management than source control. On the other hand, a large organization with a (virtual?) single repository (monorepo) can scale up significantly farther with source-control policies—this is Google's approach. Separate open source projects certainly count as separate organizations: interdependencies between unknown and not-necessarily-collaborating projects are a dependency management problem. Perhaps our strongest single piece of advice on this topic is this: *All else being equal, prefer source-control problems over dependency-management problems*. If you have the option to redefine "organization" more broadly (your entire company rather than just one team), that's very often a good trade-off. Source-control problems are a lot easier to think about and a lot cheaper to deal with than dependency-management ones.

As the Open Source Software (OSS) model continues to grow and expand into new domains, and the dependency graph for many popular projects continues to expand over time, dependency management is perhaps becoming the most important problem in software engineering policy. We are no longer disconnected islands built on one or two layers outside an API. Modern software is built on towering pillars of dependencies; but just because we can build those pillars doesn't mean we've yet figured out how to keep them standing and stable over time.

In this chapter, we'll look at the particular challenges of dependency management, explore solutions (common and novel) and their limitations, and look at the realities of working with dependencies, including how we've handled things in Google. It is important to preface all of this with an admission: we've invested a lot of *thought* into this problem and have extensive experience with refactoring and maintenance issues that show the practical shortcomings with existing approaches. We don't have firsthand evidence of solutions that work well across organizations at scale. To some extent, this chapter is a summary of what we know does not work (or at least

might not work at larger scales) and where we think there is the potential for better outcomes. We definitely cannot claim to have all the answers here; if we could, we wouldn't be calling this one of the most important problems in software engineering.

Why Is Dependency Management So Difficult?

Even defining the dependency=management problem presents some unusual challenges. Many half-baked solutions in this space focus on a too-narrow problem formulation: "How do we import a package that our locally developed code can depend upon?" This is a necessary-but-not-sufficient formulation. The trick isn't just finding a way to manage one dependency, the trick is how to manage a *network* of dependencies and their changes over time. Some subset of this network is directly necessary for your first-party code, some of it is only pulled in by transitive dependencies. Over a long enough period, all of the nodes in that dependency network will have new versions, and some of those updates will be important.[2] How do we manage the resulting cascade of upgrades for the rest of the dependency network? Or, specifically, how do we make it easy to find mutually compatible versions of all of our dependencies given that we do not control those dependencies? How do we analyze our dependency network? How do we manage that network, especially in the face of an ever-growing graph of dependencies?

Conflicting Requirements and Diamond Dependencies

The central problem in dependency management highlights the importance of thinking in terms of dependency networks, not individual dependencies. Much of the difficulty stems from one problem: what happens when two nodes in the dependency network have conflicting requirements, and your organization depends on them both? This can arise for many reasons, ranging from platform considerations (operating system [OS], language version, compiler version, etc.) to the much more mundane issue of version incompatibility. The canonical example of version incompatibility as an unsatisfiable version requirement is the *diamond dependency* problem. Although we don't generally include things like "what version of the compiler" are you using in a dependency graph, most of these conflicting requirements problems are isomorphic to "add a (hidden) node to the dependency graph representing this requirement." As such, we'll primarily discuss conflicting requirements in terms of diamond dependencies, but keep in mind that `libbase` might actually be absolutely any piece of software

involved in the construction of two or more nodes in your dependency network.

The diamond dependency problem, and other forms of conflicting requirements, require at least three layers of dependency, as demonstrated in Figure 21-1.

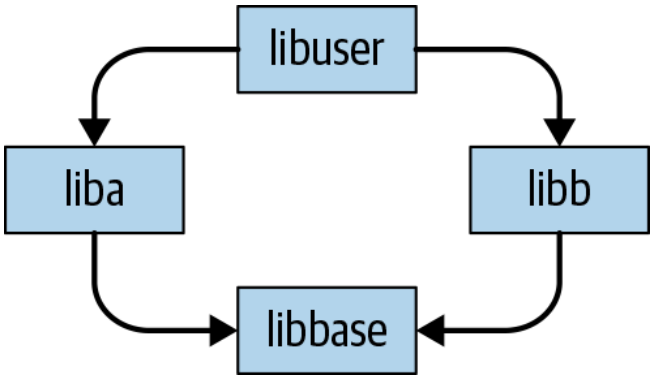

Figure 21-1. The diamond dependency problem

In this simplified model, `libbase` is used by both `liba` and `libb`, and `liba` and `libb` are both used by a higher-level component libuser. If `libbase` ever introduces an incompatible change, there is a chance that `liba` and `libb`, as products of separate organizations, don't update simultaneously. If `liba` depends on the new `libbase` version and `libb` depends on the old version, there's no general way for `libuser` (aka your code) to put everything together. This diamond can form at any scale: in the entire network of your dependencies, if there is ever a low-level node that is required to be in two incompatible versions at the same time (by virtue of there being two paths from some higher level node to those two versions), there will be a problem.

Different programming languages tolerate the diamond dependency problem to different degrees. For some languages it is possible to embed multiple (isolated) versions of a dependency within a build: a call into `libbase` from `liba` might call a different version of the same API as a call into `libbase` from `libb`. For example, Java provides fairly well-established mechanisms to rename the symbols provided by such a dependency.[3] Meanwhile, C++ has nearly zero tolerance for diamond dependencies in a normal build, and they are very likely to trigger arbitrary bugs and undefined behavior (UB) as a result of a clear violation of C++'s One Definition Rule. You can at best use a similar idea as Java's shading to hide some symbols in a dynamic-link library (DLL) or .so in cases in which you're building and linking separately. However, in all programming languages that we're aware of, these workarounds are partial

solutions at best: embedding multiple versions can be made to work by tweaking the names of *functions*, but if there are *types* that are passed around between dependencies, all bets are off. For example, there is simply no way for a `map` defined in `libbase` v1 to be passed through some libraries to an API provided by `libbase` v2 in a semantically consistent fashion. Language-specific hacks to hide or rename entities in separately compiled libraries can provide some cushion for diamond dependency problems, but are not a solution in the general case.

If you encounter a conflicting requirement problem the only easy answer is to skip forward or backward in versions for those dependencies to find something compatible. When that isn't possible, we must resort to locally patching the dependencies in question, which is particularly challenging because the cause of the incompatibility in both provider and consumer is probably not known to the engineer that first discovers the incompatibility. This is inherent: `liba` developers are still working in a compatible fashion with `libbase` v1, and `libb` devs have already upgraded to v2. Only a dev who is pulling in both of those projects has the chance to discover the issue, and it's certainly not guaranteed that they are familiar enough with `libbase` and `liba` to work through the upgrade. The easier answer is to downgrade `libbase` and `libb`, although that is not an option if the upgrade was originally forced because of security issues.

Systems of policy and technology for Dependency Management largely boil down to the question, "How do we avoid conflicting requirements while still allowing change among noncoordinating groups?" If you have a solution for the general form of the diamond dependency problem that allows for the reality of continuously changing requirements (both dependencies and platform requirements) at all levels of the network, you've described the interesting part of a dependency-management solution.

Importing Dependencies

In programming terms, it's clearly better to reuse some existing infrastructure rather than build it yourself. This is obvious, and part of the fundamental march of technology: if every novice had to reimplement their own JSON parser and regular expression engine, we'd never get anywhere. Reuse is healthy, especially compared to the cost of redeveloping quality software from scratch. So long as you aren't downloading trojaned software, if your external dependency satisfies the requirements for your programming task, you should use it.

Compatibility Promises

When we start considering time, the situation gains some complicated trade-offs. Just because you get to avoid a *development* cost doesn't mean importing a dependency is the correct choice. In a software engineering organization that is aware of time and change, we need to also be mindful of its ongoing maintenance costs. Even if we import a dependency with no intent of upgrading it, discovered security vulnerabilities, changing platforms, and evolving dependency networks can conspire to force that upgrade, regardless of our intent. When that day comes, how expensive is it going to be? Some dependencies are more explicit than others about the expected maintenance cost for merely using that dependency: how much compatibility is assumed? How much evolution is assumed? How are changes handled? For how long are releases supported?

We suggest that a dependency provider should be clearer about the answers to these questions. Consider the example set by large infrastructure projects with millions of users and their compatibility promises.

C++

For the C++ standard library, the model is one of nearly indefinite backward compatibility. Binaries built against an older version of the standard library are expected to build and link with the newer standard: the standard provides not only API compatibility, but ongoing backward compatibility for the binary artifacts, known as *ABI compatibility*. The extent to which this has been upheld varies from platform to platform. For users of gcc on Linux, it's likely that most code works fine over a range of roughly a decade. The standard doesn't explicitly call out its commitment to ABI compatibility—there are no public-facing policy documents on that point. However, the standard does publish Standing Document 8 (SD-8), which calls out a small set of types of change that the standard library can make between versions, defining implicitly what type of changes to be prepared for. Java is similar: source is compatible between language versions, and JAR files from older releases will readily work with newer versions.

GO

Not all languages prioritize the same amount of compatibility. The Go programming language explicitly promises source compatibility between most releases, but no binary compatibility. You cannot build a library in Go with one version of the language and link that library into a Go program built with a different version of the language.

ABSEIL

Google's Abseil project is much like Go, with an important caveat about time. We are unwilling to commit to compatibility *indefinitely*: Abseil lies at the foundation of most of our most computationally heavy services internally, which we believe are likely to be in use for many years to come. This means we're careful to reserve the right to make changes, especially in implementation details and ABI, in order to allow better performance. We have experienced far too many instances of an API turning out to be confusing and error prone after the fact; publishing such known faults to tens of thousands of developers for the indefinite future feels wrong. Internally, we already have roughly 250 million lines of C++ code that depends on this library—we aren't going to make API changes lightly, but it must be possible. To that end, Abseil explicitly does not promise ABI compatibility, but does promise a slightly limited form of API compatibility: we won't make a breaking API change without also providing an automated refactoring tool that will transform code from the old API to the new transparently. We feel that shifts the risk of unexpected costs significantly in favor of users: no matter what version a dependency was written against, a user of that dependency and Abseil should be able to use the most current version. The highest cost should be "run this tool," and presumably send the resulting patch for review in the mid-level dependency (`liba` or `libb`, continuing our example from earlier). In practice, the project is new enough that we haven't had to make any significant API breaking changes. We can't say how well this will work for the ecosystem as a whole, but in theory it seems like a good balance for stability versus ease of upgrade.

BOOST

By comparison, the Boost C++ library makes no promises of compatibility between versions.[4] Most code doesn't change, of course, but "many of the Boost libraries are actively maintained and improved, so backward compatibility with prior version isn't always possible." Users are advised to upgrade only at a period in their project life cycle in which some change will not cause problems. The goal for Boost is fundamentally different than the standard library or Abseil: Boost is an experimental proving ground. A particular release from the Boost stream is probably perfectly stable and appropriate for use in many projects, but Boost's project goals do not prioritize compatibility between versions—other long-lived projects might experience some friction keeping up to date. The Boost developers are every bit as expert as the developers for the standard library[5]—none of this is about

technical expertise: this is purely a matter of what a project does or does not promise and prioritize.

Looking at the libraries in this discussion, it's important to recognize that these compatibility issues are *software engineering* issues, not *programming* issues. You can download something like Boost with no compatibility promise and embed it deeply in the most critical, long-lived systems in your organization; it will *work* just fine. All of the concerns here are about how those dependencies will change over time, keeping up with updates, and the difficulty of getting developers to worry about maintenance instead of just getting features working. Within Google there is a constant stream of guidance directed to our engineers to help them consider this difference between "I got it to work" and "this is working in a supported fashion." That's unsurprising: it's basic application of Hyrum's Law, after all.

Put more broadly: it is important to realize that dependency management has a wholly different nature in a programming task versus a software engineering task. If you're in a problem space for which maintenance over time is relevant, dependency management is difficult. If you're purely developing a solution for today with no need to ever update anything, it is perfectly reasonable to grab as many readily available dependencies as you like with no thought of how to use them responsibly or plan for upgrades. Getting your program to work today by violating everything in SD-8 and also relying on binary compatibility from Boost and Abseil works fine…so long as you never upgrade the standard library, Boost, or Abseil, and neither does anything that depends on you.

Considerations When Importing

Importing a dependency for use in a programming project is nearly free: assuming that you've taken the time to ensure that it does what you need and isn't secretly a security hole, it is almost always cheaper to reuse than to reimplement functionality. Even if that dependency has taken the step of clarifying what compatibility promise it will make, so long as we aren't ever upgrading, anything you build on top of that snapshot of your dependency is fine, no matter how many rules you violate in consuming that API. But when we move from programming to software engineering, those dependencies become subtly more expensive, and there are a host of hidden costs and questions that need to be answered. Hopefully, you consider these costs before importing, and, hopefully, you know when you're working on a programming project versus working on a software engineering project.

When engineers at Google try to import dependencies at Google, we encourage them to ask this (incomplete) list of questions first:

- Does the project have tests that you can run?
- Do those tests pass?
- Who is providing that dependency? Even among "No warranty implied" Open Source Software (OSS) projects there is a significant range of experience and skill set—it's a very different thing to depend on compatibility from the C++ standard library or Java's Guava library than it is to select a random project from GitHub or npm. Reputation isn't everything, but it is worth investigating.
- What sort of compatibility is the project aspiring to?
- Does the project detail what sort of usage is expected to be supported?
- How popular is the project?
- How long will we be depending on this project?
- How often does the project make breaking changes?

Add to this a short selection of internally focused questions:

- How complicated would it be to implement that functionality within Google?
- What incentives will we have to keep this dependency up-to-date?
- Who will perform an upgrade?
- How difficult do we expect it to be to perform an upgrade?

Our own Russ Cox has written about this more extensively. We can't give a perfect formula for deciding when it's cheaper in the long term to import versus reimplement; we fail at this ourselves, more often than not.

How Google Handles Importing Dependendencies

In short: we could do better.

The overwhelming majority of dependencies in any given Google project are internally developed. This means that the vast majority of our internal dependency-management story isn't really dependency management, it's just source control—by design. As we have mentioned, it is a far easier thing to manage and control the complexities and risks involved in adding dependencies when the providers and consumers are part of the same

organization and have proper visibility and Continuous Integration (CI; see Chapter 23) available. Most problems in dependency management stop being problems when you can see exactly how your code is being used and know exactly the impact of any given change. Source control (when you control the projects in question) is far easier than dependency management (when you don't).

That ease-of-use begins failing when it comes to our handling of external projects. For projects that we are importing from the OSS ecosystem or commercial partners, those dependencies are added into a separate directory of our monorepo, labeled *third_party*. Let's examine how a new OSS project is added to *third_party*.

Alice, a software engineer at Google, is working on a project and realizes that there is an open source solution available. She would really like to have this project completed and demo'ed soon, to get it out of the way before going on vacation. The choice then is whether to reimplement that functionality from scratch or download the OSS package and get it added to *third_party*. It's very likely that Alice decides that the faster development solution makes sense: she downloads the package and follows a few steps in our *third_party* policies. These are a fairly simple checklist: make sure it builds with our build system, make sure there isn't an existing version of that package, and make sure that at least two engineers are signed up as OWNERS to maintain the package in the event that any maintenance is necessary. Alice gets her teammate Bob to say, "Yes, I'll help." Neither of them need to have any experience maintaining a *third_party* package, and they have conveniently avoided the need to understand anything about the *implementation* of this package. At most, they have gained a little experience with its interface as part of using it to solve the prevacation demo problem.

From this point on, the package is usually available to other Google teams to use in their own projects. The act of adding additional dependencies is completely transparent to Alice and Bob: they might be completely unaware that the package they downloaded and promised to maintain has become popular. Subtly, even if they are monitoring for new direct usage of their package, they might not necessarily notice growth in the *transitive* usage of their package. If they use it for a demo, while Charlie adds a dependency from within the guts of our Search infrastructure, the package will have suddenly moved from fairly innocuous to being in the critical infrastructure for important Google systems. However, we don't have any particular signals sent to Charlie when he is considering whether to add this dependency.

Now, it's possible that this scenario is perfectly fine. Perhaps that dependency is well written, has no security bugs, and isn't depended upon by other OSS projects. It might be *possible* for it to go quite a few years without being updated. It's not necessarily *wise* for that to happen: changes externally might have optimized it or added important new functionality, or cleaned up security holes before CVEs[6] were discovered. The longer that the package exists, the more dependencies (direct and indirect) are likely to accrue. The more that the package remains stable, the more that we are likely to accrete Hyrum's Law reliance on the particulars of the version that is checked into *third_party*.

One day, Alice and Bob are informed that an upgrade is critical. It could be the disclosure of a security vulnerability in the package itself or in an OSS project that depends upon it that forces an upgrade. Bob has transitioned to management and hasn't touched the codebase in a while. Alice has moved to another team since the demo and hasn't used this package again. Nobody changed the OWNERS file. Thousands of projects depend on this indirectly—we can't just delete it without breaking the build for Search and a dozen other big teams. Nobody has any experience with the implementation details of this package. Alice isn't necessarily on a team that has a lot of experience undoing Hyrum's Law subtleties that have accrued over time.

All of which is to say: Alice and the other users of this package are in for a costly and difficult upgrade, with the security team exerting pressure to get this resolved immediately. Nobody in this scenario has practice in performing the upgrade, and the upgrade is extra difficult because it is covering many smaller releases covering the entire period between initial introduction of the package into *third_party* and the security disclosure.

Our *third_party* policies don't work for these unfortunately common scenarios. We roughly understand that we need a higher bar for ownership, we need to make it easier (and more rewarding) to update regularly, and more difficult for *third_party* packages to be orphaned and important at the same time. The difficulty is that it is difficult for codebase maintainers and *third_party* leads to say, "No, you can't use this thing that solves your development problem perfectly because we don't have resources to update everyone with new versions constantly." Projects that are popular and have no compatibility promise (like Boost) are particularly risky: our developers might be very familiar with using that dependency to solve programming problems outside of Google, but allowing it to become ingrained into the fabric of our codebase is a big risk. Our codebase has an expected lifespan of

decades at this point: upstream projects that are not explicitly prioritizing stability are a risk.

Dependency Management, In Theory

Having looked at the ways that dependency management is difficult and how it can go wrong, let's discuss more specifically the problems we're trying to solve and how we might go about solving them. Throughout this chapter, we call back to the formulation, "How do we manage code that comes from outside our organization (or that we don't perfectly control): how do we update it, how do we manage the things it depends upon over time?" We need to be clear that any good solution here avoids conflicting requirements of any form, including diamond dependency version conflicts, even in a dynamic ecosystem in which new dependencies or other requirements might be added (at any point in the network). We also need to be aware of the impact of time: all software has bugs, some of those will be security critical, and some fraction of our dependencies will therefore be *critical* to update over a long enough period of time.

A stable dependency-management scheme must therefore be flexible with time and scale: we can't assume indefinite stability of any particular node in the dependency graph, nor can we assume that no new dependencies are added (either in code we control or in code we depend upon). If a solution to dependency management prevents conflicting requirement problems among your dependencies, it's a good solution. If it does so without assuming stability in dependency version or dependency fan-out, coordination or visibility between organizations, or significant compute resources, it's a great solution.

When proposing solutions to dependency management, there are four common options that we know of that exhibit at least some of the appropriate properties: nothing ever changes, semantic versioning, bundle everything that you need (coordinating not per project, but per distribution), or live-at-head.

Nothing Changes (aka The Static Dependency Model)

The simplest way to ensure stable dependencies is to never change them: no API changes, no behavioral changes, nothing. Bugfixes are allowed only if no user code could be broken. This prioritizes compatibility and stability over all else. Clearly, such a scheme is not-ideal, due to the assumption of indefinite stability. If, somehow, we get to a world in which security issues

and bug fixes are a nonissue and dependencies aren't changing, the Nothing Changes model is very appealing: if we start with satisfiable constraints, we'll be able to maintain that property indefinitely.

Although not sustainable in the long term, practically speaking, this is where every organization starts: up until you've demonstrated that the expected lifespan of your project is long enough that change becomes necessary, it's really easy to live in a world where we assume that nothing changes. It's also important to note: this is probably the right model for most new organizations. It is comparatively rare to know that you're starting a project that is going to live for decades and have a *need* to be able to update dependencies smoothly. It's much more reasonable to hope that stability is a real option, and pretend that dependencies are perfectly stable for the first few years of a project.

The downside to this model is that over a long enough time period it *is* false, and there isn't a clear indication of exactly how long you can pretend that it is legitimate. We don't have long-term early warning systems for security bugs or other critical issues that might force you to upgrade a dependency—and because of chains of dependencies, a single upgrade can in theory become a forced update to your entire dependency network.

In this model, version selection is simple: there are no decisions to be made, because there are no versions.

Semantic Versioning

The de facto standard for "how do we manage a network of dependencies today" is semantic versioning (SemVer).[7] SemVer is the nearly ubiquitous practice of representing a version number for some dependency (especially libraries) using three decimal-separated integers, such as 2.4.72 or 1.1.4. In the most common convention, the three component numbers represent major, minor, and patch versions, with the implication that a changed major number indicates a change to an existing API that can break existing usage, a changed minor number indicates purely added functionality that should not break existing usage, and a changed patch version is reserved for non-API-impacting implementation details and bug fixes that are viewed as particularly low risk.

With the SemVer separation of major/minor/patch versions, the assumption is that a version requirement can generally be expressed as "anything newer than," barring API-incompatible changes (major version changes). Commonly, we'll see "Requires `libbase` $\geq$ 1.5," that requirement would be

compatible with any `libbase` in 1.5, including 1.5.1, and anything in 1.6 onward, but not `libbase` 1.4.9 (missing the API introduced in 1.5) or 2.x (some APIs in `libbase` were changed incompatibly). Major version changes are a significant incompatibility: because an existing piece of functionality has changed (or been removed), there are potential incompatibilities for all dependents. Version requirements exist (explicitly or implicitly) whenever one dependency uses another: we might see "liba requires libbase ≥ 1.5" and "libb requires libbase ≥ 1.4.7."

If we formalize these requirements, we can conceptualize a dependency network as a collection of software components (nodes) and the requirements between them (edges). Edge labels in this network change as a function of the version of the source node, either as dependencies are added (or removed) or as the SemVer requirement is updated because of a change in the source node (requiring a newly added feature in a dependency, for instance). Because this whole network is changing asynchronously over time, the process of finding a mutually compatible set of dependencies that satisfy all the transitive requirements of your application can be challenging.[8] Version-satisfiability solvers for SemVer are very much akin to SAT-solvers in logic and algorithms research: given a set of constraints (version requirements on dependency edges), can we find a set of versions for the nodes in question that satisfies all constraints? Most package management ecosystems are built on top of these sorts of graphs, governed by their SemVer SAT-solvers.

SemVer and its SAT-solvers aren't in any way promising that there *exists* a solution to a given set of dependency constraints. Situations in which dependency constraints cannot be satisfied are created constantly, as we've already seen: if a lower-level component (`libbase`) makes a major-number bump, and some (but not all) of the libraries that depend on it (`libb` but not `liba`) have upgraded, we will encounter the diamond dependency issue.

SemVer solutions to dependency management are usually SAT-solver based. Version selection is a matter of running some algorithm to find an assignment of versions for dependencies in the network that satisfies all of the version-requirement constraints. When no such satisfying assignment of versions exists, we colloquially call it "dependency hell."

We'll look at some of the limitations of SemVer in more detail later in this chapter.

Bundled Distribution Models

As an industry, we've seen the application of a powerful model of managing dependencies for decades now: an organization gathers up a collection of dependencies, finds a mutually compatible set of those, and releases the collection as a single unit. This is what happens, for instance, with Linux distributions—there's no guarantee that the various pieces that are included in a distro are cut from the same point in time. In fact, it's somewhat more likely that the lower-level dependencies are somewhat older than the higher-level ones, just to account for the time it takes to integrate them.

This "draw a bigger box around it all and release that collection" model introduces entirely new actors: the distributors. Although the maintainers of all of the individual dependencies may have little or no knowledge of the other dependencies, these higher-level *distributors* are involved in the process of finding, patching, and testing a mutually compatible set of versions to include. Distributors are the engineers responsible for proposing a set of versions to bundle together, testing those to find bugs in that dependency tree, and resolving any issues.

For an outside user, this works great, so long as you can properly rely on only one of these bundled distributions. This is effectively the same as changing a dependency network into a single aggregated dependency and giving that a version number. Rather than saying, "I depend on these 72 libraries at these versions," this is, "I depend on RedHat version N," or, "I depend on the pieces in the NPM graph at time T".

In the bundled distribution approach, version selection is handled by dedicated distributors.

Live at Head

The model that some of us at Google[9] have been pushing for is theoretically sound, but places new and costly burdens on participants in a dependency network. It's wholly unlike the models that exist in OSS ecosystems today, and it is not clear how to get from here to there as an industry. Within the boundaries of an organization like Google, it is costly but effective, and we feel that it places most of the costs and incentives into the correct places. We call this model "Live at Head." It is viewable as the dependency-management extension of trunk-based development: where trunk-based development talks about source control policies, we're extending that model to apply to upstream dependencies, as well.

Live at Head presupposes that we can unpin dependencies, drop SemVer, and rely on dependency providers to test changes against the entire ecosystem before committing. Live at Head is an explicit attempt to take time and choice out of the issue of dependency management: always depend on the current version of everything, and never change anything in a way in which it would be difficult for your dependents to adapt. A change that (unintentionally) alters API or behavior will in general be caught by CI on downstream dependencies, and thus should not be committed. For cases in which such a change *must* happen (i.e., for security reasons), such a break should be made only after either the downstream dependencies are updated or an automated tool is provided to perform the update in place. (This tooling is essential for closed-source downstream consumers: the goal is to allow any user the ability to update use of a changing API without expert knowledge of the use or the API. That property significantly mitigates the "mostly bystanders" costs of breaking changes.) This philosophical shift in responsibility in the open source ecosystem is difficult to motivate initially: putting the burden on an API provider to test against and change all of its downstream customers is a significant revision to the responsibilities of an API provider.

Changes in a Live at Head model are not reduced to a SemVer "I think this is safe or not." Instead, tests and CI systems are used to test against visible dependents to determine experimentally how safe a change is. So, for a change that alters only efficiency or implementation details, all of the visible affected tests might likely pass, which demonstrates that there are no obvious ways for that change to impact users—it's safe to commit. A change that modifies more obviously observable parts of an API (syntactically or semantically) will often yield hundreds or even thousands of test failures. It's then up to the author of that proposed change to determine whether the work involved to resolve those failures is worth the resulting value of committing the change. Done well, that author will work with all of their dependents to resolve the test failures ahead of time (i.e., unwinding brittle assumptions in the tests) and might potentially create a tool to perform as much of the necessary refactoring as possible.

The incentive structures and technological assumptions here are materially different than other scenarios: we assume that there exist unit tests and CI, we assume that API providers will be bound by whether downstream dependencies will be broken, and we assume that API consumers are keeping their tests passing and relying on their dependency in supported ways. This works significantly better in an open source ecosystem (in which fixes can be distributed ahead of time) than it does in the face of hidden/closed-source

dependencies. API providers are incentivized when making changes to do so in a way that can be smoothly migrated to. API consumers are incentivized to keep their tests working so as not to be labeled as a low-signal test and potentially skipped, reducing the protection provided by that test.

In the Live at Head approach, version selection is handled by asking "What is the most recent stable version of everything?" If providers have made changes responsibly, it will all work together smoothly.

The Limitations of SemVer

The live-at-head approach may build on recognized practices for version control (trunk-based development) but is largely unproven at scale. SemVer is the de facto standard for dependency management today, but as we've suggested, it is not without its limitations. Because it is such a popular approach, it is worth looking at it in more detail and highlighting what we believe to be its potential pitfalls.

There's a lot to unpack in the SemVer definition of what a dotted-triple version number really means. Is this a promise? Or is the version number chosen for a release an estimate? That is, when the maintainers of `libbase` cut a new release and choose whether this is a major, minor, or patch release, what are they saying? Is it provable that an upgrade from 1.1.4 to 1.2.0 is safe and easy, because there were only API additions and bugfixes? Of course not. There's a host of things that ill-behaved users of `libbase` could have done that could cause build breaks or behavioral changes in the face of a "simple" API addition.[10] Fundamentally, you can't *prove* anything about compatibility when only considering the source API, you have to know *with which* things you are asking about compatibility.

However, this idea of "estimating" compatibility begins to weaken when we talk about networks of dependencies and SAT-solvers applied to those networks. The fundamental problem in this formulation is the difference between node values in traditional SAT and version values in a SemVer dependency graph. A node in a three-SAT graph *is* either True or False. A version value (1.1.14) in a dependency graph is provided by the maintainer as an *estimate* of how compatible the new version is, given code that used the previous version. We're building all of our version-satisfaction logic on top of a shaky foundation, treating estimates and self-attestation as absolute. As we'll see, even if that works OK in limited cases, in the aggregate it doesn't necessarily have enough fidelity to underpin a healthy ecosystem.

If we acknowledge that SemVer is a lossy estimate and represents only a subset of the possible scope of changes, we can begin to see it as a blunt instrument. In theory, it works fine as a shorthand. In practice, especially when we build SAT-solvers on top of it, SemVer can (and does) fail us by both overconstraining and underprotecting us.

SemVer Might Overconstrain

Consider what happens when `libbase` is recognized to be more than a single monolith: there are almost always independent interfaces within a library. Even if there are only two functions, we can see situations in which SemVer overconstrains us. Imagine that `libbase` is indeed composed of only two functions, Foo and Bar. Our mid-level dependencies `liba` and `libb` use only Foo. If the maintainer of `libbase` makes a breaking change to Bar, it is incumbent on them to bump the major version of `libbase` in a SemVer world. `liba` and `libb` are known to depend on `libbase` 1.x—SemVer dependency solvers won't accept a 2.x version of that dependency. However, in reality these libraries would work together perfectly: only Bar changed, and that was unused. The compression inherent in "I made a breaking change; I must bump the major version number" is lossy when it doesn't apply at the granularity of an individual atomic API unit. Although some dependencies might be fine grained enough for that to be accurate,[11] that is not the norm for a SemVer ecosystem.

If SemVer overconstrains, either because of an unnecessarily severe version bump or insufficiently fine-grained application of SemVer numbers, automated package managers and SAT-solvers will report that your dependencies cannot be updated or installed, even if everything would work together flawlessly by ignoring the SemVer checks. Anyone who has ever been exposed to dependency hell during an upgrade might find this particularly infuriating: some large fraction of that effort was a complete waste of time.

SemVer Might Overpromise

On the flip side, the application of SemVer makes the explicit assumption that an API provider's estimate of compatibility can be fully predictive and that changes fall into three buckets: breaking (by modification or removal), strictly additive, or non-API-impacting. If SemVer is a perfectly faithful representation of the risk of a change by classifying syntactic and semantic changes, how do we characterize a change that adds a one-millisecond delay to a time-sensitive API? Or, more plausibly: how do we characterize a change

that alters the format of our logging output? Or that alters the order that we import external dependencies? Or that alters the order that results are returned in an "unordered" stream? Is it reasonable to assume that those changes are "safe" merely because those aren't part of the syntax or contract of the API in question? What if the documentation said "This may change in the future?" Or the API was named "ForInternalUseByLibBaseOnlyDoNotTouchThisIReallyMeanIt?"[12]

The idea that SemVer patch versions, which in theory are only changing implementation details, are "safe" changes absolutely runs afoul of Google's experience with Hyrum's Law—"With a sufficient number of users, every observable behavior of your system will be depended upon by someone." Changing the order that dependencies are imported, or changing the output order for an "unordered" producer will, at scale, invariably break assumptions that some consumer was (perhaps incorrectly) relying upon. The very term "breaking change" is misleading: there are changes that are theoretically breaking but safe in practice (removing an unused API). There are also changes that are theoretically safe but break client code in practice (any of our earlier Hyrum's Law examples). We can see this in any SemVer/dependency-management system for which the version-number requirement system allows for restrictions on the patch number: if you can say `liba requires libbase >1.1.14` rather than `liba requires libbase 1.1`, that's clearly an admission that there are observable differences in patch versions.

A change in isolation isn't breaking or nonbreaking—that statement can be evaluated only in the context of how it is being used. There is no absolute truth in the notion of "This is a breaking change"; a change can been seen to be breaking for only a (known or unknown) set of existing users and use cases. The reality of how we evaluate a change inherently relies upon information that isn't present in the SemVer formulation of dependency management: how are downstream users consuming this dependency?

Because of this, a SemVer constraint solver might report that your dependencies work together when they don't, either because a bump was applied incorrectly or because something in your dependency network had a Hyrum's Law dependence on something that wasn't considered part of the observable API surface. In these cases, you might have either build errors or runtime bugs, with no theoretical upper bound on their severity.

Motivations

There is a further argument that SemVer doesn't always incentivise the creation of stable code. For a maintainer of an arbitrary dependency, there is variable systemic incentive to *not* make breaking changes and bump major versions. Some projects care deeply about compatibility and will go to great lengths to avoid a major-version bump. Others are more aggressive, even intentionally bumping major versions on a fixed schedule. The trouble is that most users of any given dependency are indirect users—they wouldn't have any significant reasons to be aware of an upcoming change. Even most direct users don't subscribe to mailing lists or other release notifications.

All of which combines to suggest that no matter how many users will be inconvenienced by adoption of an incompatible change to a popular API, the maintainers bear a tiny fraction of the cost of the resulting version bump. For maintainers who are also users, there can also be an incentive *toward* breaking: it's always easier to design a better interface in the absence of legacy constraints. This is part of why we think projects should publish clear statements of intent with respect to compatibility, usage, and breaking changes. Even if those are best-effort, nonbinding, or ignored by many users, it still gives us a starting point to reason about whether a breaking change/major version bump is "worth it," without bringing in these conflicting incentive structures.

Go and Clojure both handle this nicely: in their standard package management ecosystems, the equivalent of a major-version bump is expected to be a fully new package. This has a certain sense of justice to it: if you're willing to break backward compatibility for your package, why do we pretend this is the same set of APIs? Repackaging and renaming everything seems like a reasonable amount of work to expect from a provider, in exchange for them taking the nuclear option and throwing away backward compatibility.

Finally, there's the human fallibility of the process. In general, SemVer version bumps should be applied to *semantic* changes just as much as syntactic ones; changing the behavior of an API matters just as much as changing its structure. Although it's plausible that tooling could be developed to evaluate whether any particular release involves syntactic changes to a set of public APIs, discerning whether there are meaningful and intentional semantic changes is computationally infeasible.[13] Practically speaking, even the potential tools for identifying syntactic changes are limited. In almost all cases, it is up to the human judgement of the API provider whether to bump

major, minor, or patch versions for any given change. If you're relying on only a handful of professionally-maintained dependencies, your expected exposure to this form of SemVer clerical error is probably low.[14] If you have a network of thousands of dependencies underneath your product, you should be prepared for some amount of chaos simply from human error.

Minimum Version Selection

In 2018, as part of an essay series on building a package management system for the Go programming language, Google's own Russ Cox described an interesting variation on SemVer dependency management: Minimum Version Selection (MVS). When updating the version for some node in the dependency network, it is possible that its dependencies need to be updated to newer versions to satisfy an updated SemVer requirement—this can then trigger further changes transitively. In most constraint-satisfaction/version-selection formulations, the newest possible versions of those downstream dependencies are chosen: after all, you'll need to update to those new versions eventually, right?

MVS makes the opposite choice: when `liba`'s specification requires `libbase` ≥1.7, we'll try `libbase` 1.7 directly, even if a 1.8 is available. This "produces high-fidelity builds in which the dependencies a user builds are as close as possible to the ones the author developed against." There is a critically important truth revealed in this point: when `liba` says it requires `libbase` ≥1.7, that almost certainly means that the developer of `liba` had `libbase` 1.7 installed. Assuming that the maintainer performed even basic testing before publishing,[15] we have at least anecdotal evidence of interoperability testing for that version of `liba` and version 1.7 of `libbase`. It's not CI or proof that everything has been unit tested together, but it's something.

Absent accurate input constraints derived from 100% accurate prediction of the future, it's best to make the smallest jump forward possible, Just as it's usually safer to commit an hour of work to your project instead of dumping a year of work all at once, smaller steps forward in your dependency updates are safer. MVS just walks forward each affected dependency only as far as is required and says, "OK, I've walked forward far enough to get what you asked for (and not farther). Why don't you run some tests and see if things are good?"

Inherent in the idea of MVS is the admission that a newer version might introduce an incompatibility in practice, even if the version numbers *in*

theory say otherwise. This is recognizing the core concern with SemVer, using MVS or not: there is some loss of fidelity in this compression of software changes into version numbers. MVS gives some additional practical fidelity, trying to produce selected versions closest to those that have presumably been tested together. This might be enough of a boost to make a larger set of dependency networks function properly. Unfortunately, we haven't found a good way to empirically verify that idea. The jury is still out on whether MVS makes SemVer "good enough" without fixing the basic theoretical and incentive problems with the approach, but we still believe it represents a manifest improvement in the application of SemVer constraints as they are used today.

So, Does SemVer Work?

SemVer works well enough in limited scales. It's deeply important, however, to recognize what it is actually saying and what it cannot. SemVer will work fine provided that:

- Your dependency providers are accurate and responsible (to avoid human error in SemVer bumps)
- Your dependencies are fine grained (to avoid falsely overconstraining when unused/unrelated APIs in your dependencies are updated, and the associated risk of unsatisfiable SemVer requirements)
- All usage of all APIs is within the expected usage (to avoid being broken in surprising fashion by an assumed-compatible change, either directly or in code you depend upon transitively)

When you have only a few carefully chosen and well-maintained dependencies in your dependency graph, SemVer can be a perfectly suitable solution.

However, our experience at Google suggests that it is unlikely that you can have *any* of those three properties at scale and keep them working constantly over time. Scale tends to be the thing that shows the weaknesses in SemVer. As your dependency network scales up, both in the size of each dependency and the number of dependencies (as well as any monorepo effects from having multiple projects depending on the same network of external dependencies), the compounded fidelity loss in SemVer will begin to dominate. These failures manifest as both false positives (practically incompatible versions that theoretically should have worked) and false negatives (compatible versions disallowed by SAT-solvers and resulting dependency hell).

Dependency Management with Infinite Resources

Here's a useful thought experiment when considering dependency-management solutions: what would dependency management look like if we all had access to infinite compute resources? That is, what's the best we could hope for, if we aren't resource constrained but are limited only by visibility and weak coordination among organizations? As we see it currently, the industry relies on SemVer for three reasons:

- It requires only local information (an API provider doesn't *need* to know the particulars of downstream users).
- It doesn't assume the availability of tests (not ubiquitous in the industry yet, but definitely moving that way in the next decade), compute resources to run the tests, or CI systems to monitor the test results.
- It's the existing practice.

The "requirement" of local information isn't really necessary, specifically because dependency networks tend to form in only two environments:

- Within a single organization
- Within the OSS ecosystem, where source is visible even if the projects are not necessarily collaborating

In either of those cases, significant information about downstream usage is *available*, even if it isn't being readily exposed or acted upon today. That is, part of SemVer's effective dominance is that we're choosing to ignore information that is theoretically available to us. If we had access to more compute resources and that dependency information was surfaced readily, the community would probably find a use for it.

Although an OSS package can have innumerable closed-source dependents, the common case is that popular OSS packages are popular both publicly and privately. Dependency networks don't (can't) aggressively mix public and private dependencies: generally, there is a public subset and a separate private subgraph.[16]

Next, we must remember the *intent* of SemVer: "In my estimation, this change will be easy (or not) to adopt." Is there a better way of conveying that information? Yes, in the form of practical experience demonstrating that the change is easy to adopt. How do we get such experience? If most (or at least a representative sample) of our dependencies are publicly visible, we run the

tests for those dependencies with every proposed change. With a sufficiently large number of such tests, we have at least a statistical argument that the change is safe in the practical Hyrum's-Law sense. The tests still pass, the change is good—it doesn't matter whether this is API impacting, bug fixing, or anything in between, there's no need to classify or estimate.

Imagine, then, that the OSS ecosystem moved to a world in which changes were accompanied with *evidence* of whether they are safe. If we pull compute costs out of the equation, the *truth*[17] of "how safe is this" comes from running affected tests in downstream dependencies.

Even without formal CI applied to the entire OSS ecosystem, we can of course use such a dependency graph and other secondary signals to do a more targeted presubmit analysis. Prioritize tests in dependencies that are heavily used. Prioritize tests in dependencies that are well maintained. Prioritize tests in dependencies that have a history of providing good signal and high-quality test results. Beyond just prioritizing tests based on the projects that are likely to give us the most information about experimental change quality, we might be able to use information from the change authors to help estimate risk and select an appropriate testing strategy. Running "all affected" tests is theoretically necessary, if the goal is "nothing that anyone relies upon is change in a breaking fashion." If we consider the goal to be more in line with "risk mitigation," a statistical argument becomes a more appealing (and cost-effective) approach.

In Chapter 12, we identified four varieties of change, ranging from pure refactorings to modification of existing functionality. Given a CI-based model for dependency updating, we can begin to map those varieties of change onto a SemVer-like model for which the author of a change estimates the risk and applies an appropriate level of testing. For example, a pure refactoring change that modifies only internal APIs might be assumed to be low risk and justify running tests only in our own project and perhaps a sampling of important direct dependents. On the other hand, a change that removes a deprecated interface or changes observable behaviors might require as much testing as we can afford.

What changes would we need to the OSS ecosystem to apply such a model? Unfortunately, quite a few:

- All dependencies must provide unit tests. Although we are moving inexorably toward a world in which unit testing is both well accepted and ubiquitous, we are not there yet.

- The dependency network for the majority of the OSS ecosystem is understood. It is unclear that any mechanism is currently available to perform graph algorithms on that network—the information is *public* and *available,* but not actually generally indexed or usable. Many package-management systems/dependency-management ecosystems allow you to see the dependencies of a project, but not the reverse edges, the dependents.
- The availability of compute resources for executing CI is still very limited. Most developers don't have access to build-and-test compute clusters.
- Dependencies are often expressed in a pinned fashion. As a maintainer of `libbase`, we can't experimentally run a change through the tests for `liba` and `libb` if those dependencies are explicitly depending on a specific pinned version of `libbase`.
- We might want to explicitly include history and reputation in CI calculations. A proposed change that breaks a project that has a longstanding history of tests continuing to pass gives us a different form of evidence than a breakage in a project that was only added recently and has a history of breaking for unrelated reasons.

Inherent in this is a scale question: against which versions of each dependency in the network do you test presubmit changes? If we test against the full combination of all historical versions, we're going to burn a truly staggering amount of compute resource, even by Google standards. The most obvious simplification to this version-selection strategy would seem to be "test the current stable version" (trunk-based development is the goal, after all). And thus, the model of dependency management given infinite resources is effectively that of the Live at Head model. The outstanding question is whether that model can apply effectively with a more practical resource availability and whether API providers are willing to take greater responsibility for testing the practical safety of their changes. Recognizing where our existing low-cost facilities are an oversimplification of the difficult-to-compute truth that we are looking for is still a useful exercise.

Exporting Dependencies

So far, we've only talked about taking on dependencies; that is, depending on software that other people have written. It's also worth thinking about how we build software that can be *used* as a dependency. This goes beyond just the mechanics of packaging software and uploading it to a repository: we

need to think about the benefits, costs, and risks of providing software, for both us and our potential dependents.

There are two major ways that an innocuous and hopefully charitable act like "open sourcing a library" can become a possible loss for an organization. First, it can eventually become a drag on the reputation of your organization if implemented poorly or not maintained properly. As the Apache community saying goes, we ought to prioritize "community over code." If you provide great code but are a poor community member, that can still be harmful to your organization and the broader community. Second, a well-intentioned release can become a tax on engineering efficiency if you can't keep things in-sync. Given time, all forks will become expensive.

EXAMPLE: OPEN SOURCING GFLAGS

For reputation loss, consider the case of something like Google's experience circa 2006 open sourcing our C++ command-line flag libraries. Surely giving back to the open source community is a purely good act that won't come back to haunt us, right? Sadly, no. A host of reasons conspired to make this good act into something that certainly hurt our reputation and possibly damaged the OSS community, as well:

- At the time, we didn't have the ability to execute large-scale refactorings, so everything that used that library internally had to remain exactly the same—we couldn't move the code to a new location in the codebase.
- We segregated our repository into "code developed in-house" (which can be copied freely if it needs to be forked, so long as it is renamed properly) and "code that may have legal/licensing concerns" (which can have more nuanced usage requirements).
- If an OSS project accepts code from outside developers, that's generally a legal issue—the project originator doesn't *own* that contribution, they only have rights to it.

As a result, the gflags project was doomed to be either a "throw over the wall" release or a disconnected fork. Patches contributed to the project couldn't be reincorporated into the original source inside of Google, and we couldn't move the project within our monorepo because we hadn't yet mastered that form of refactoring, nor could we make everything internally depend on the OSS version.

Further, like most organizations, our priorities have shifted and changed over time. Around the time of the original release of that flags library we were interested in products outside of our traditional space (web applications, search), including things like Google Earth, which had a much more traditional distribution mechanism: precompiled binaries for a variety of platforms. In the late 2000s, it was unusual but not unheard of for a library in our monorepo, especially something low-level like flags, to be used on a variety of platforms. As time went on and Google grew, our focus narrowed to the point that it was extremely rare for any libraries to be built with anything other than our in-house configured toolchain, then deployed to our production fleet. The "portability" concerns for properly supporting an OSS project like flags were nearly impossible to maintain: our internal tools simply didn't have support for those platforms, and our average developer didn't have to interact with external tools. It was a constant battle to try to maintain portability.

As the original authors and OSS supporters moved on to new companies or new teams, it eventually became clear that nobody internally was really supporting our OSS flags project—nobody could tie that support back to the priorities for any particular team. Given that it was no specific team's job, and nobody could say why it was important, it isn't surprising that we basically let that project rot externally.[18] The internal and external versions diverged slowly over time, and eventually some external developers took the external version and forked it, giving it some proper attention.

Other than the initial "Oh look, Google contributed something to the open source world," no part of that made us look good, and yet every little piece of it made sense given the priorities of our engineering organization. Those of us who have been close to it have learned, "Don't release things without a plan (and a mandate) to support it for the long term." Whether the whole of Google engineering has learned that or not remains to be seen. It's a big organization.

Above and beyond the nebulous "We look bad," there are also parts of this story that illustrate how we can be subject to technical problems stemming from poorly released/poorly maintained external dependencies. Although the flags library was shared but ignored, there were still some Google backed open source projects, or projects that needed to be shareable outside of our monorepo ecosystem. Unsurprisingly, the authors of those other projects were able to identify[19] the common API subset between the internal and external forks of that library. Because that common subset stayed fairly stable between the two versions for a long period, it silently became "the way to do

this" for the rare teams that had unusual portability requirements between roughly 2008 and 2017. Their code could build in both internal and external ecosystems, switching out forked versions of the flags library depending on environment.

Then, for unrelated reasons, C++ library teams began tweaking observable-but-not-documented pieces of the internal flag implementation. At that point, everyone who was depending on the stability and equivalence of an unsupported external fork started screaming that their builds and releases were suddenly broken. An optimization opportunity worth some thousands of aggregate CPUs across Google's fleet was significantly delayed, not because it was difficult to update the API that 250 million lines of code depended upon, but because a tiny handful of projects were relying on unpromised and unexpected things. Once again, Hyrum's Law affects software changes, in this case even for forked APIs maintained by separate organizations.

EXAMPLE: APPENGINE

A more serious example of exposing ourselves to greater risk of unexpected technical dependency comes from publishing Google's AppEngine service. This service allows users to write their applications on top of an existing framework in one of several popular programming languages. So long as the application is written with a proper storage/state management model, the AppEngine service allows those applications to scale up to huge usage levels: backing storage and frontend management are managed and cloned on demand by Google's production infrastructure.

Originally, AppEngine's support for Python was a 32-bit build running with an older version of the Python interpreter. The AppEngine system itself was (of course) implemented in our monorepo and built with the rest of our common tools, in Python and in C++ for backend support. In 2014 we started the process of doing a major update to the Python runtime alongside our C++ compiler and standard library installations, with the result being that we effectively tied "code that builds with the current C++ compiler" to "code that uses the updated Python version"—a project that upgraded one of those dependencies inherently upgraded the other at the same time. For most projects this was a non-issue. For a few projects, because of edge cases and Hyrum's Law, our language platform experts wound up doing some investigation and debugging to unblock the transition. In a terrifying instance of Hyrum's Law running into business practicalities, AppEngine discovered that many of its users, our paying customers, couldn't (or wouldn't) update: either they didn't want to take the change to the newer Python version, or

they couldn't afford the resource consumption changes involved in moving from 32-bit to 64-bit Python. Because there were some customers that were paying a significant amount of money for AppEngine services, AppEngine was able to make a strong business case that a forced switch to the new language and compiler versions must be delayed. This inherently meant that every piece of C++ code in the transitive closure of dependencies from AppEngine had to be compatible with the older compiler and standard library versions: any bug fixes or performance optimizations that could be made to that infrastructure had to be compatible across versions. That situation persisted for almost three years.

With enough users, any observable of your system will come to be depended upon by somebody. At Google, we constrain all of our internal users within the boundaries of our technical stack, and ensure visibility into their usage with the monorepo and code indexing systems, so it is far easier to ensure that useful change remains possible. When we shift from source control to dependency management, and we lose visibility into how code is used, or are subject to competing priorities from outside groups (especially ones that are paying you), it becomes much more difficult to make pure engineering trade-offs. Releasing APIs of any sort exposes you to the possibility of competing priorities and unforeseen constraints by outsiders. This isn't to say that you shouldn't release APIs; it serves only to provide the reminder: external users of an API cost a lot more to maintain than internal ones.

Sharing code with the outside world, either as an open source release or as a closed-source library release, is not a simple matter of charity (in the OSS case) or business opportunity (in the closed-source case). Dependent users that you cannot monitor, in different organizations, with different priorities, will eventually exert some form of Hyrum's Law inertia on that code. Especially if you are working with long timescales, it is impossible to accurately predict the set of necessary or useful changes that could become valuable. When evaluating whether to release something, be aware of the long-term risks: externally shared dependencies are often much more expensive to modify over time.

Conclusion

Dependency management is inherently challenging—we're looking for solutions to management of complex API surfaces and webs of dependencies, where the maintainers of those dependencies generally have little or no assumption of coordination. The de facto standard for managing a network of

dependencies is semantic versioning, or SemVer, which provides a lossy summary of the perceived risk in adopting any particular change. SemVer presupposes that we can *a priori* predict the severity of a change, in the absence of knowledge of how the API in question is being consumed: Hyrum's Law informs us otherwise. However, SemVer works well enough at small scale, and even better when we include the MVS approach. As the size of the dependency network grows, Hyrum's Law issues and fidelity loss in SemVer make managing the selection of new versions increasingly difficult.

It is possible, however, that we move toward a world in which maintainer provided estimates of compatibility (SemVer version numbers) are dropped in favor of experience-driven evidence: running the tests of affected downstream packages. If API providers take greater responsibility for testing against their users and clearly advertise what types of changes are expected, we have the possibility of higher-fidelity dependency networks at even larger scale.

TL;DRs

- Prefer source control problems to dependency management problems: if you can get more code from your organization to have better transparency and coordination, those are important simplifications.
- Adding a dependency isn't free for a software engineering project, and the complexity in establishing an "ongoing" trust relationship is challenging. Importing dependencies into your organization needs to be done carefully, with an understanding of the ongoing support costs.
- A dependency is a contract: there is a give and take, both providers and consumers have some rights and responsibilities in that contract. Providers should be clear about what they are trying to promise over time.
- SemVer is a lossy-compression shorthand estimate for "How risky does a human think this change is." SemVer with a SAT-solver in a package manager takes those estimates and escalates them to function as absolutes. This can result in either overconstraint (dependency hell) or underconstraint (versions that should work together that don't).
- By comparison, testing and CI provide actual evidence of whether a new set of versions work together.
- Minimum-version update strategies in SemVer/package management are higher fidelity. This still relies on humans being able to assess incremental version risk accurately, but distinctly improves the chance

that the link between API provider and consumer has been tested by an expert.

- Unit testing, CI, and (cheap) compute resources have the potential to change our understanding and approach to dependency management. That phase-change requires a fundamental change in how the industry considers the problem of dependency management, and the responsibilities of providers and consumers both.
- Providing a dependency isn't free: "throw it over the wall and forget" can cost you reputation and become a challenge for compatibility. Supporting it with stability can limit your choices and pessimize internal usage. Supporting without stability can cost goodwill or expose you to risk of important external groups depending on something via Hyrum's Law and messing up your "no stability" plan.

1 This could be any of language version, version of a lower-level library, hardware version, operating system, compiler flag, compiler version, and so on.

2 For instance, security bugs, deprecations, being in the dependency set of a higher-level dependency that has a security bug, and so on.

3 This is called *shading* or *versioning*.

4 *https://www.boost.org/users/faq.html*

5 In many cases there is significant overlap in those populations.

6 Common Vulnerabilities and Exposures

7 Strictly speaking, SemVer refers only to the emerging practice of applying semantics to major/minor/patch version numbers, not the application of compatible version requirements among dependencies numbered in that fashion. There are numerous minor variations on those requirements among different ecosystems, but in general the version-number-plus-constraints system described here as SemVer is representative of the practice at large.

8 In fact, it has been proven that SemVer constraints applied to a dependency network are NP-complete.

9 Especially the author and others in the Google C++ community.

10 For example: a poorly implemented polyfill that adds the new libbase API ahead of time, causing a conflicting definition. Or, use of language reflection

APIs to depend upon the precise number of APIs provided by `libbase`, introducing crashes if that number changes. These shouldn't happen and are certainly rare even if they do happen by accident—the point is that the `libbase` providers can't *prove* compatibility.

11 The Node ecosystem has noteworthy examples of dependencies that provide exactly one API.

12 It's worth noting: in our experience, naming like this doesn't fully solve the problem of users reaching in to access private APIs. Prefer languages that have good control over public/private access to APIs of all forms.

13 In a world of ubiquitous unit tests, we could identify changes that required a change in test behavior, but it would still be difficult to algorithmically separate "This is a behavioral change" from "This is a bug fix to a behavior that wasn't intended/promised."

14 So, when it matters in the long term, choose well-maintained dependencies.

15 If that assumption doesn't hold, you should really stop depending on liba.

16 Because the public OSS dependency network can't generally depend on a bunch of private nodes, graphics firmware notwithstanding.

17 Or something very close to it.

18 That isn't to say it's *right or wise*, just that as an organization we let some things slip through the cracks.

19 Often through trial and error.

Chapter 22. Large-Scale Changes

Written by Hyrum Wright

Edited by Lisa Carey

Think for a moment about your own codebase. How many files can you reliably update in a single simultaneous commit? What are the factors which constrain that number? Have you ever tried committing a change that large? Would you be able to do it in a reasonable amount of time in an emergency? How does your largest commit size compare to the actual size of your codebase? How would you test such a change? How many people would need to review the change before it is committed? Would you be able to roll back that change if it did get committed? The answers to these questions might surprise you (both what you *think* the answers are and what they actually turn out to be for your organization.)

At Google, we've long ago abandoned the idea of making sweeping changes across our codebase in these types of large atomic changes. Our observation has been that as a codebase and the number of engineers working in it grows, the largest atomic change possible counterintuitively *decreases*—running all affected presubmit checks and tests becomes difficult, to say nothing of even ensuring that every file in the change is up to date before submission. As it has become more difficult to make sweeping changes to our codebase, given our general desire to be able to continually improve underlying infrastructure, we've had to develop new ways of reasoning about large-scale changes and how to implement them.

In this chapter, we'll talk about the techniques, both social and technical, which enable us to keep the large Google codebase flexible and responsive to changes in underlying infrastructure. We'll also provide some real-life examples of how and where we've used these approaches. Although your codebase might not look like Google's, understanding these principles and adapting them locally will help your development organization scale while still being able to make broad changes across your codebase.

What Is a Large-Scale Change?

Before going much further, we should dig into what qualifies as a large-scale change (LSC). In our experience, an LSC is any set of changes that are

logically related but cannot practically be submitted as a single atomic unit. This might be because it touches so many files that the underlying tooling can't commit them all at once, or it might be because the change is so large that it would always have merge conflicts. In many cases, an LSC is dictated by your repository topology: if your organization uses a collection of distributed or federated repositories,[1] making atomic changes across them might not even be technically possible.[2] We'll look at potential barriers to atomic changes in more detail later in this chapter.

LSCs at Google are almost always generated using automated tooling. Reasons for making an LSC vary, but the changes themselves generally fall into a few basic categories:

- Cleaning up common antipatterns using codebase-wide analysis tooling
- Replacing uses of a deprecated library features
- Enabling low-level infrastructure improvements, such as compiler upgrades
- Moving users from an old system to a newer one[3]

The number of engineers working on these specific tasks in a given organization might be low, but it is useful for their customers to have insight into the LSC tools and process. By their very nature, LSCs will affect a large number of customers, and the LSC tools easily scale down to teams making only a few dozen related changes.

There can be broader motivating causes behind specific LSCs. For example, a new language standard might introduce a more efficient idiom for accomplishing a given task, an internal library interface might change, or a new compiler release might require fixing existing problems that would be flagged as errors by the new release. The majority of LSCs across Google actually have near-zero functional impact: they tend to be widespread textual updates for clarity, optimization or future compatibility. But LSCs are not theoretically limited to this behavior-preserving/refactoring class of change.

In all of these cases, on a codebase the size of Google's, infrastructure teams might routinely need to change hundreds of thousands of individual references to the old pattern or symbol. In the largest cases so far, we've touched millions of references, and we expect the process to continue to scale well. Generally, we've found it advantageous to invest early and often in tooling to enable LSCs for the many teams doing infrastructure work. We've also found that efficient tooling also helps engineers performing smaller

changes. The same tools that make changing thousands of files efficient, also scale down to tens-of-files reasonably well.

Who Deals with LSCs?

As just indicated, the infrastructure teams that build and manage our systems are responsible for much of the work of performing LSCs, but the tools and resources are available across the company. If you skipped Chapter 1, you might wonder why infrastructure teams are the ones responsible for this work. Why can't we just introduce a new class, function, or system and dictate that everybody who uses the old one move to the updated analogue? Although this might seem easier in practice, it turns out not to scale very well for several reasons.

First, the infrastructure teams that build and manage the underlying systems are also the ones with the domain knowledge required to fix the hundreds of thousands of references to them. Teams that consume the infrastructure are unlikely to have the context for handling many of these migrations, and it is globally inefficient to expect them to each relearn expertise that infrastructure teams already have. Centralization also allows for faster recovery when faced with errors because errors generally fall into a small set of categories, and the team running the migration can have a playbook—formal or informal—for addressing them.

Consider the amount of time it takes to do the first of a series of semi-mechanical changes that you don't understand. You probably spend some time reading about the motivation and nature of the change, find an easy example, try to follow the provided suggestions, and then try to apply that to your local code. Repeating this for every team in an organization greatly increases the overall cost of execution. By making only a few centralized teams responsible for LSCs, Google both internalizes those costs and drives them down by making it possible for the change to happen more efficiently.

Second, nobody likes unfunded mandates.[4] Even though a new system might be categorically better than the one it replaces, those benefits are often diffused across an organization and thus unlikely to matter enough for individual teams to want to update on their own initiative. If the new system is important enough to migrate to, the costs of migration will be borne somewhere in the organization. Centralizing the migration and accounting for its costs is almost always faster and cheaper than depending on individual teams to organically migrate.

Additionally, having teams that own the systems requiring LSCs helps align incentives to ensure the change gets done. In our experience, organic migrations are unlikely to fully succeed, in part because engineers tend to use existing code as examples when writing new code. Having a team that has a vested interest in removing the old system responsible for the migration effort helps ensure that it actually gets done. Although funding and staffing a team to run these kinds of migrations can seem like an additional cost, it is actually just internalizing the externalities that an unfunded mandate creates, with the additional benefits of economies of scale.

Filling Potholes

Although the LSC systems at Google are used for high-priority migrations, we've also discovered that just having them available opens up opportunities for various small fixes across our codebase, which just wouldn't have been possible without them. Much like transportation infrastructure tasks consist of building new roads as well as repairing old ones, infrastructure groups at Google spend a lot of time fixing existing code, in addition to developing new systems and moving users to them.

For example, early in our history, a template library emerged to supplement the C++ Standard Template Library. Aptly named the Google Template Library, this library consisted of several header files' worth of implementation. For reasons lost in the mists of time, one of these header files was named *stl_util.h* and another was named *map-util.h* (note the different separators in the file names). In addition to driving the consistency purists nuts, this difference also led to reduced productivity, and engineers had to remember which file used which separator, and only discovered when they got it wrong after a potentially lengthy compile cycle.

Although fixing this single-character change might seem pointless, particularly across a codebase the size of Google's, the maturity of our LSC tooling and process enabled us to do it with just a couple weeks' worth of background-task effort. Library authors could find and apply this change *en masse* without having to bother end users of these files, and we were able to quantitatively reduce the number of build failures caused by this specific issue. The resulting increases in productivity (and happiness) more than paid for the time to make the change.

As the ability to make changes across our entire codebase has improved, the diversity of changes has also expanded, and we can make some engineering

decisions knowing that they aren't immutable in the future. Sometimes, it's worth the effort to fill a few potholes.

Barriers to Atomic Changes

Before we discuss the process that Google uses to actually effect LSCs, we should talk about why many kinds of changes can't be committed atomically. In an ideal world, all logical changes could be packaged into a single atomic commit that could be tested, reviewed, and committed independent of other changes. Unfortunately, as a repository—and the number of engineers working in it—grows, that ideal becomes less feasible. It can be completely infeasible even at small scale when using a set of distributed or federated repositories.

Technical Limitations

To begin with, most Version Control Systems (VCSs) have operations that scale linearly with the size of a change. Your system might be able to handle small commits (e.g., on the order of tens of files) just fine, but might not have sufficient memory or processing power to atomically commit thousands of files at once. In centralized VCSs, commits can block other writers (and in older systems, readers) from using the system as they process, meaning that large commits stall other users of the system.

In short, it might not be just "difficult" or "unwise" to make a large change atomically: it might simply be impossible with a given set of infrastructures. Splitting the large change into smaller independent chunks gets around these limitations, although it makes the execution of the change more complex.[5]

Merge Conflicts

As the size of a change grows, the potential for merge conflicts also increases. Every version control system we know of requires updating and merging, potentially with manual resolution, if a newer version of a file exists in the central repository. As the number of files in a change increases, the probability of encountering a merge conflict also grows, and is compounded by the number of engineers working in the repository.

If your company is small, you might be able to sneak in a change that touches every file in the repository on a weekend when nobody is doing development. Or you might have an informal system of grabbing the global repository lock by passing a virtual (or even physical!) token around your development team.

At a large, global company like Google these approaches are just not feasible: somebody is always making changes to the repository.

With few files in a change, the probability of merge conflicts shrinks, so they are more likely to be committed without problems. This property also holds for the following areas, as well.

No Haunted Graveyards

The SREs who run Google's production services have a mantra: "No Haunted Graveyards." A haunted graveyard in this sense is a system that is so ancient, obtuse, or complex that no one dares enter it. Haunted graveyards are often business-critical systems that are frozen in time because any attempt to change them could cause the system to fail in incomprehensible ways, costing the business real money. They pose a real existential risk and can consume an inordinate amount of resources.

Haunted graveyards don't just exist in production systems, however; they can be found in codebases. Many organizations have bits of software that are old and unmaintained, written by someone long off the team, and on the critical path of some important revenue-generating functionality. These systems are also frozen in time, with layers of bureaucracy built up to prevent changes that might cause instability. Nobody wants to be the network support engineer II who flipped the wrong bit!

These parts of a codebase are anathema to the LSC process because they prevent the completion of large migrations, the decommissioning of other systems upon which they rely, or the upgrade of compilers or libraries that they use. From an LSC perspective, haunted graveyards prevent all kinds of meaningful progress.

At Google, we've found the counter to this to be good, ol'-fashioned testing. When software is thoroughly tested, we can make arbitrary changes to it and know with confidence whether those changes are breaking, no matter the age or complexity of the system. Writing those tests takes a lot of effort, but it allows a codebase like Google's to evolve over long periods of time, consigning the notion of haunted software graveyards to a graveyard of its own.

Heterogeneity

LSCs really work only when the bulk of the effort for them can be done by computers, not humans. As good as humans can be with ambiguity,

computers rely upon consistent environments to apply the proper code transformations to the correct places. If your organization has many different VCSs, Continuous Integration (CI) systems, project-specific tooling, or formatting guidelines, it is difficult to make sweeping changes across your entire codebase. Simplifying the environment to add more consistency will help both the humans who need to move around in it, and the robots making automated transformations.

For example, many projects at Google have presubmit tests configured to run before changes are made to their codebase. Those checks can be very complex, ranging from checking new dependencies against a whitelist, to running tests, to ensuring that the change has an associated bug. Many of these checks are relevant for teams writing new features, but for LSCs, they just add additional irrelevant complexity.

We've decided to embrace some of this complexity, such as running presubmit tests, by making it standard across our codebase. For other inconsistencies we advise teams to omit their special checks when parts of LSCs touch their project code. Most teams are happy to help given the benefit these kinds of changes are to their projects.

NOTE

Much of the benefits of consistency for humans mentioned in Chapter 8 also applies to automated tooling.

Testing

Every change should be tested (a process we'll talk about more in just a moment), but the larger the change, the more difficult it is to actually test it appropriately. Google's CI system will run not only the tests immediately impacted by a change, but also any tests that transitively depend on the changed files.6 This means a change gets broad coverage, but we've also observed that the farther away in the dependency graph a test is from the impacted files, the more unlikely a failure is to have been caused by the change itself.

Small, independent changes are easier to validate, because each of them affects a smaller set of tests, but also because test failures are easier to diagnose and fix. Finding the root cause of a test failure in a change of 25 files is pretty straightforward; finding 1 in a 10,000-file change is like the proverbial needle in a haystack.

The trade-off in this decision is that smaller changes will cause the same tests to be run multiple times, particularly tests that depend on large parts of the codebase. Because engineer time spent tracking down test failures is much more expensive than the compute time required to run these extra tests, we've made the conscious decision that this is a trade-off we're willing to make. That same trade-off might not hold for all organizations, but it is worth examining what the proper balance is for yours.

Testing LSCs

By Adam Bender

Today it is common for a double-digit percentage (10% to 20%) of the changes in a project to be the result of LSCs, meaning a substantial amount of code is changed in projects by people whose full-time job is unrelated to those projects. Without good tests, such work would be impossible and Google's codebase would quickly atrophy under its own weight. LSCs enable us to systematically migrate our entire codebase to newer APIs, deprecate older APIs, change language versions, and remove popular but dangerous practices.

Even a simple one-line signature change becomes complicated when made in 1,000 different places across hundreds of different products and services.[7] After the change is written, you need to coordinate code reviews across dozens of teams. Lastly, after reviews are approved, you need to run as many tests as you can to be sure the change is safe.[8] We say "as many as you can," because a good-sized LSC could trigger a rerun of every single test at Google, and that can take a while. In fact, many LSCs have to plan time to catch downstream clients whose code backslides while the LSC makes its way through the process.

Testing an LSC can be a slow and frustrating process. When a change is sufficiently large, your local environment is almost guaranteed to be permanently out of sync with head as the codebase shifts like sand around your work. In such circumstances, it is easy to find yourself running and rerunning tests just to ensure your changes continue to be valid. When a project has flaky tests or is missing unit test coverage, it can require a lot of manual intervention and slow down the entire process. To help speed things up, we use a strategy called the TAP (Test Automation Platform) train.

Riding the TAP Train

The core insight to LSCs is that they rarely interact with one another, and most affected tests are going to pass for most LSCs. As a result, we can test more than one change at a time and reduce the total number of tests executed. The train model has proven to be very effective for testing LSCs.

The TAP train takes advantage of two facts:

- LSCs tend to be pure refactorings and therefore very narrow in scope, preserving local semantics.
- Individual changes are often simpler and highly scrutinized, so they are correct more often than not.

The train model also has the advantage that it works for multiple changes at the same time and doesn't require that each individual change ride in isolation[9].

The train has five steps, and is started fresh every three hours:

1. For each change on the train, run a sample of 1,000 randomly-selected tests.
2. Gather up all the changes that passed their 1,000 tests and create one uber-change from all of them: "the train."
3. Run the union of all tests directly affected by the group of changes. Given a large enough (or low-level enough) LSC, this can mean running every single test in Google's repository. This process can take more than six hours to complete.
4. For each nonflaky test that fails, rerun it individually against each change that made it into the train to determine which changes caused it to fail.
5. TAP generates a report for each change that boarded the train. The report describes all passing and failing targets and can be used as evidence that an LSC is safe to submit.

Code Review

Finally, as we mentioned in Chapter 9, all changes need to be reviewed before submission, and this policy applies even for LSCs. Reviewing large commits can be tedious, onerous, and even error prone, particularly if the changes are generated by hand (a process you want to avoid, as we'll discuss

shortly). In just a moment, we'll look at how tooling can often help in this space, but for some classes of changes, we still want humans to explicitly verify they are correct. Breaking an LSC into separate shards makes this much easier.

scoped_ptr to std::unique_ptr

Since its earliest days, Google's C++ codebase has had a self-destructing smart pointer for wrapping heap-allocated C++ objects and ensuring that they are destroyed when the smart pointer goes out of scope. This type was called `scoped_ptr` and was used extensively throughout Google's codebase to ensure that object lifetimes were appropriately managed. It wasn't perfect, but given the limitations of the then-current C++ standard (C++ 98) when the type was first introduced, it made for safer programs.

In C++11, the language introduced a new type: `std::unique_ptr`. It fulfilled the same function as `scoped_ptr`, but also prevented other classes of bugs that the language now could detect. `std::unique_ptr` was strictly better than `scoped_ptr`, yet Google's codebase had more than 500,000 references to `scoped_ptr` scattered among millions of source files. Moving to the more modern type required the largest LSC attempted to that point within Google.

Over the course of several months, several engineers attacked the problem in parallel. Using Google's large-scale migration infrastructure, we were able to change references to `scoped_ptr` into references to `std::unique_ptr` as well as slowly adapt `scoped_ptr` to behave more closely to `std::unique_ptr`. At the height of the migration process, we were consistently generating, testing and committing more than 700 independent changes, touching more than 15,000 files *per day*. Today, we sometimes manage 10 times that throughput, having refined our practices and improved our tooling.

Like almost all LSCs, this one had a very long tail of tracking down various nuanced behavior dependencies (another manifestation of Hyrum's Law), fighting race conditions with other engineers, and uses in generated code that weren't detectable by our automated tooling. We continued to work on these manually as they were discovered by the testing infrastructure.

`scoped_ptr` was also used as a parameter type in some widely used APIs, which made small independent changes difficult. We contemplated writing a call-graph analysis system which could change an API and its callers, transitively, in one commit, but were concerned that the resulting changes would themselves be too large to commit atomically.

In the end, we were able to finally remove `scoped_ptr` by first making it a type alias of `std::unique_ptr` and then performing the textual substitution between the old alias and the new, before eventually just removing the old `scoped_ptr` alias. Today, Google's codebase benefits from using the same standard type as the rest of the C++ ecosystem, which was possible only because of our technology and tooling for LSCs.

LSC Infrastructure

Google has invested in a significant amount of infrastructure to make LSCs possible. This infrastructure includes tooling for change creation, change management, change review, and testing. However, perhaps the most important support for LSCs has been the evolution of cultural norms around large-scale changes and the oversight given to them. Although the sets of technical and social tools might differ for your organization, the general principles should be the same.

Policies and Culture

As we've described in Chapter 16, Google stores the bulk of its source code in a single monolithic repository (monorepo), and every engineer has visibility into almost all of this code. This high degree of openness means that any engineer can edit any file and send those edits for review to those who can approve them. However, each of those edits has costs, both to generate as well as review.[10]

Historically, these costs have been somewhat symmetric, which limited the scope of changes a single engineer or team could generate. As Google's LSC tooling improved, it became easier to generate a large number of changes very cheaply, and it became equally easy for a single engineer to impose a burden on a large number of reviewers across the company. Even though we want to encourage widespread improvements to our codebase, we want to make sure there is some oversight and thoughtfulness behind them, rather than indiscriminate tweaking.[11]

The end result is a lightweight approval process for teams and individuals seeking to make LSCs across Google. This process is overseen by a group of experienced engineers who are familiar with the nuances of various languages, as well as invited domain experts for the particular change in question. The goal of this process is not to prohibit LSCs, but to help change authors produce the best possible changes, which make the most use of

Google's technical and human capital. Occasionally, this group might suggest that a cleanup just isn't worth it: for example, cleaning up a common typo without any way of preventing recurrence.

Related to these policies was a shift in cultural norms surrounding LSCs. Although it is important for code owners to have a sense of responsibility for their software, they also needed to learn that LSCs were an important part of Google's effort to scale our software engineering practices. Just as product teams are the most familiar with their own software, library infrastructure teams know the nuances of the infrastructure, and getting product teams to trust that domain expertise is an important step toward social acceptance of LSCs. As a result of this culture shift, local product teams have grown to trust LSC authors to make changes relevant to those authors' domains.

Occasionally local owners question the purpose of a specific commit being made as part of a broader LSC, and change authors respond to these comments just as they would other review comments. Socially, it's important that code owners understand the changes happening to their software, but they also have come to realize that they don't hold a veto over the broader LSC. Over time, we've found that a good FAQ and a solid historic track record of improvements have generated widespread endorsement of LSCs throughout Google.

Codebase Insight

To do LSCs, we've found it invaluable to be able to do large-scale analysis of our codebase, both on a textual level using traditional tools, as well as on a semantic level. For example, Google's use of the semantic indexing tool Kythe provides a complete map of the links between parts of our codebase, allowing us to ask questions such as "Where are the callers of this function?" or "Which classes derive from this one?" Kythe and similar tools also provide programmatic access to their data so that they can be incorporated into refactoring tools. (For further examples, see Chapter 17 and Chapter 20.)

We also use compiler-based indices to run abstract syntax tree-based analysis and transformations over our codebase. Tools such as ClangMR, JavacFlume, or Refaster, which can perform transformations in a highly parallelizable way, depend on these insights as part of their function. For smaller changes, authors can use specialized, custom tools, `perl` or `sed`, regular expression matching, or even a simple shell script.

Whatever tool your organization uses for change creation, it's important that its human effort scale sublinearly with the codebase; in other words, it should take roughly the same amount of human time to generate the collection of all required changes, no matter the size of the repository. The change creation tooling should also be comprehensive across the codebase, so that an author can be assured that their change covers all of the cases they're trying to fix.

As with other areas in this book, an early investment in tooling usually pays off in the short to medium term. As a rule of thumb, we've long held that if a change requires more than 500 edits, it's usually more efficient for an engineer to learn and execute our change-generation tools rather than manually execute that edit. For experienced "code janitors," that number is often much smaller.

Change Management

Arguably the most important piece of large-scale change infrastructure is the set of tooling that shards a master change into smaller pieces and manages the process of testing, mailing, reviewing, and committing them independently. At Google, this tool is called Rosie, and we discuss its use more completely in a few moments when we examine our LSC process. In many respects, Rosie is not just a tool, but an entire platform for making LSCs at Google scale. It provides the ability to split the large sets of comprehensive changes produced by tooling into smaller shards, which can be tested, reviewed, and submitted independently.

Testing

Testing is another important piece of large-scale-change-enabling infrastructure. As discussed in Chapter 11, tests are one of the important ways that we validate our software will behave as expected. This is particularly important when applying changes that are not authored by humans. A robust testing culture and infrastructure means that other tooling can be confident that these changes don't have unintended effects.

Google's testing strategy for LSCs differs slightly from that of normal changes while still using the same underlying CI infrastructure. Testing LSCs means not just ensuring the large master change doesn't cause failures, but that each shard can be submitted safely and independently. Because each shard can contain arbitrary files, we don't use the standard project-based presubmit tests. Instead, we run each shard over the transitive closure of every test it might affect, which we discussed earlier.

Language Support

LSCs at Google are typically done on a per-language basis, and some languages support them much more easily than others. We've found that language features such as type aliasing and forwarding functions are invaluable for allowing existing users to continue to function while we introduce new systems and migrate users to them non-atomically. For languages that lack these features, it is often difficult to migrate systems incrementally.[12]

We've also found that statically typed languages are much easier to perform large automated changes in than dynamically typed languages. Compiler-based tools, along with strong static analysis provide a significant amount of information that we can use to build tools to effect LSCs, and reject invalid transformations before they even get to the testing phase. The unfortunate result of this is that languages like Python, Ruby, and JavaScript that are dynamically typed are extra difficult for maintainers. Language choice is, in many respects, intimately tied to the question of code lifespan: languages that tend to be viewed as more focused on developer productivity tend to be more difficult to maintain. Although this isn't an intrinsic design requirement, it is where the current state of the art happens to be.

Finally, it's worth pointing out that automatic language formatters are a crucial part of the LSC infrastructure. Because we work toward optimizing our code for readability, we want to make sure that any changes produced by automated tooling are intelligible to both immediate reviewers and future readers of the code. All of the LSC-generation tools run the automated formatter appropriate to the language being changed as a separate pass so that the change-specific tooling does not need to concern itself with formatting specifics. Applying automated formatting, such as google-java-format or clang-format to our codebase means that automatically produced changes will "fit in" with code written by a human, reducing future development friction. Without automated formatting, large-scale automated changes would never have become the accepted status quo at Google.

Operation RoseHub

LSCs have become a large part of Google's internal culture, but they are starting to have implications in the broader world. Perhaps the best known case so far was "Operation RoseHub."

In early 2017, a vulnerability in the Apache Commons library allowed any Java application with a vulnerable version of the library in its transitive classpath to become susceptible to remote execution. This bug became known as the Mad Gadget. Among other things, it allowed an avaricious hacker to encrypt the San Francisco Municipal Transportation Agency's systems and shut down its operations. Because the only requirement for the vulnerability was having the wrong library somewhere in its classpath, anything that depended on even one of many open source projects on GitHub was vulnerable.

To solve this problem, some enterprising Googlers launched their own version of the LSC process. By using tools such as BigQuery, volunteers identified affected projects and sent more than 2,600 patches to upgrade their versions of the Commons library to one that addressed Mad Gadget. Instead of automated tools managing the process, more than 50 humans made this LSC work.

The LSC Process

With these pieces of infrastructure in place, we can now talk about the process for actually making an LSC. This roughly breaks down into four phases (with very nebulous boundaries between them):

1. Authorization
2. Change creation
3. Shard management
4. Cleanup

Typically, these steps happen after a new system, class, or function has been written, but it's important to keep them in mind during the design of the new system. At Google, we aim to design successor systems with a migration path from older systems in mind, so that system maintainers can move their users to the new system automatically.

Authorization

We ask potential authors to fill out a brief document explaining the reason for a proposed change, its estimated impact across the codebase (i.e., how many smaller shards the large change would generate) and answers to any questions potential reviewers might have. This process also forces authors to

think about how they will describe the change to an engineer unfamiliar with it in the form of an FAQ and proposed change description. Authors also get "domain review" from the owners of the API being refactored.

This proposal is then forwarded to an email list with about a dozen people who have oversight over the entire process. After discussion, the committee gives feedback on how to move forward. For example, one of the most common changes made by the committee is to direct all of the code reviews for an LSC to go to a single "global approver." Many first time LSC authors tend to assume that local project owners should review everything, but for most mechanical LSCs, it's cheaper to have a single expert understand the nature of the change and build automation around reviewing it properly.

After the change is approved, the author can move forward in getting their change submitted. Historically, the committee has been very liberal with their approval,[13] and often gives approval not just for a specific change but also for a broad set of related changes. Committee members can, at their discretion, fast track obvious changes without the need for full deliberation.

The intent of this process is to provide oversight and an escalation path, without being too onerous for the LSC authors. The committee is also empowered as the escalation body for concerns or conflicts about an LSC: local owners who disagree with the change can appeal to this group who can then arbitrate any conflicts. In practice, this has rarely been needed.

Change Creation

After getting the required approval, an LSC author will begin to produce the actual code edits. Sometimes, these can be generated comprehensively into a single large global change that will be subsequently sharded into many smaller independent pieces. Usually, the size of the change is too large to fit in a single global change, due to technical limitations of the underlying version control system.

The change generation process should be as automated as possible, so that the parent change can be updated as users backslide into old uses[14] or textual merge conflicts occur in the changed code. Occasionally, for the rare case in which technical tools aren't able to generate the global change, we have sharded change generation across humans (see "Operation RoseHub"). Although much more labor intensive than automatically generating changes, this allows global changes to happen much more quickly for time-sensitive applications.

Keep in mind that we optimize for human readability of our codebase, so whatever tool generates changes, we want the resulting changes to look as much like human-generated changes as possible. This requirement leads to the necessity of style guides and automatic formatting tools Chapter 8.[15]

Sharding and Submitting

After a global change has been generated, the author then starts running Rosie. Rosie takes a large change, shards it based upon project boundaries and ownership rules into changes that *can* be submitted atomically. It then puts each individually sharded change through an independent test-mail-submit pipeline. Rosie can be a heavy user of other pieces of Google's developer infrastructure, so it caps the number of outstanding shards for any given LSC, runs at lower priority, and communicates with the rest of the infrastructure about how much load it is acceptable to generate on our shared testing infrastructure.

We talk more about the specific test-mail-submit process for each shard below.

CATTLE VERSUS PETS

We often use the "cattle and pets" analogy when referring to individual machines in a distributed computing environment, but the same principles can apply to changes within a codebase.

At Google, as at most organizations, typical changes to the codebase are handcrafted by individual engineers working on specific features or bug fixes. Engineers might spend days or weeks working through the creation, testing, and review of a single change. They come to know the change intimately, and are proud when it is finally committed to the main repository. The creation of such a change is akin to owning and raising a favorite pet.

In contrast, effective handling of LSCs requires a high degree of automation and produces an enormous number of individual changes. In this environment, we've found it useful to treat specific changes as cattle: nameless and faceless commits that might be rolled back or otherwise rejected at any given time with little cost unless the entire herd is affected. Often this happens because of an unforeseen problem not caught by tests, or even something as simple as a merge conflict.

With a "pet" commit, it can be difficult to not take rejection personally, but when working with many changes as part of a large-scale change, it's just the nature of the job. Having automation means that tooling can be updated and new changes generated at very low cost, so losing a few cattle now and then isn't a problem.

TESTING

Each independent shard is tested by running it through TAP, Google's CI framework. We run every test that depends on the files in a given change transitively, which often creates high load on our CI system.

This might sound computationally expensive, but in practice, the vast majority of shards affect fewer than one thousand tests, out of the millions across our codebase. For those that affect more, we can group them together: first running the union of all affected tests for all shards, and then for each individual shard running just the intersection of its affected tests with those that failed the first run. Most of these unions cause almost every test in the codebase to be run, so adding additional changes to that batch of shards is nearly free.

One of the drawbacks of running such a large number of tests is that independent low-probability events are almost certainties at large enough scale. Flaky and brittle tests, such as those discussed in Chapter 11, which often don't harm the teams that write and maintain them, are particularly difficult for LSC authors. Although fairly low impact for individual teams, flaky tests can seriously affect the throughput of an LSC system. Automatic flake detection and elimination systems help with this issue, but it can be a constant effort to ensure that teams that write flaky tests are the ones that bear their costs.

In our experience with LSCs as semantic-preserving, machine-generated changes, we are now much more confident in the correctness of a single change than a test with any recent history of flakiness—so much so that recently flaky tests are now ignored when submitting via our automated tooling. In theory, this means that a single shard can cause a regression that is detected only by a flaky test going from flaky to failing. In practice, we see this so rarely that it's easier to deal with it via human communication rather than automation.

For any LSC process, individual shards should be committable independently. This means that they don't have any interdependence or that the sharding mechanism can group dependent changes (such as to a header file and its implementation) together. Just like any other change, large-scale change shards must also pass project-specific checks before being reviewed and committed.

MAILING REVIEWERS

After Rosie has validated that a change is safe through testing, it mails the change to an appropriate reviewer. In a company as large as Google, with thousands of engineers, reviewer discovery itself is a challenging problem. Recall from Chapter 9 that code in the repository is organized with *OWNERS* files, which list users with approval privileges for a specific subtree in the repository. Rosie uses an owners detection service that understands these *OWNERS* files and weights each owner based upon their expected ability to review the specific shard in question. If a particular owner proves to be unresponsive, Rosie adds additional reviewers automatically in an effort to get a change reviewed in a timely manner.

As part of the mailing process, Rosie also runs the per-project precommit tools, which might perform additional checks. For LSCs, we selectively disable certain checks such as those for nonstandard change description formatting. Although useful for individual changes on specific projects, such checks are a source of heterogeneity across the codebase and can add significant friction to the LSC process. This heterogeneity is a barrier to scaling our processes and systems, and LSC tools and authors can't be expected to understand special policies for each team.

We also aggressively ignore presubmit check failures that preexist the change in question. When working on an individual project, it's easy for an engineer to fix those and continue with their original work, but that technique doesn't scale when making LSCs across Google's codebase. Local code owners are responsible for having no preexisting failures in their codebase as part of the social contract between them and infrastructure teams.

REVIEWING

As with other changes, changes generated by Rosie are expected to go through the standard code review process. In practice, we've found that local owners don't often treat LSCs with the same rigor as regular changes—they trust the engineers generating LSCs too much. Ideally these changes would be reviewed as any other, but in practice, local project owners have come to trust infrastructure teams to the point where these changes are often given only cursory review. We've come to only send changes to local owners for which their review is required for context, not just approval permissions. All other changes can go to a "global approver": someone who has ownership rights to approve *any* change throughout the repository.

When using a global approver, all of the individual shards are assigned to that person, rather than to individual owners of different projects. Global approvers generally have specific knowledge of the language and/or libraries they are reviewing, and work with the large-scale change author to know what kinds of changes to expect. They know what the details of the change are, and what potential failure modes for it might exist and can customize their workflow accordingly.

Instead of reviewing each change individually, global reviewers use a separate set of pattern-based tooling to review each of the changes and automatically approve ones that meet their expectations. Thus, they need to manually examine only a small subset that are anomalous because of merge conflicts or tooling malfunctions, which allows the process to scale very well.

SUBMITTING

Finally, individual changes are committed. As with the mailing step, we ensure that the change passes the various project precommit checks before actually finally being committed to the repository.

With Rosie, we are able to effectively create, test, review, and submit thousands of changes per day across all of Google's codebase, and have given teams the ability to effectively migrate their users. Technical decisions that used to be final, such as the name of a widely used symbol or the location of a popular class within a codebase, no longer need to be final.

Cleanup

Different LSCs have different definitions of "done," which can vary from completely removing an old system, to migrating only high-value references and leaving old ones to organically disappear.[16] In almost all cases, it's important to have a system that prevents additional introductions of the symbol or system that the large-scale change worked hard to remove. At Google, we use the Tricorder framework mentioned in Chapter 20 and Chapter 19 to flag at review time when an engineer introduces a new use of a deprecated object, and this has proven an effective method to prevent backsliding. We talk more about the entire deprecation process in Chapter 15.

Conclusion

LSCs form an important part of Google's software engineering ecosystem. At design time, they open up more possibilities, knowing that some design decisions don't need to be as fixed as they once were. The LSC process also allows maintainers of core infrastructure the ability to migrate large swaths of Google's codebase from old systems, language versions, and library idioms to new ones, keeping the codebase consistent, spatially and temporally. And all of this happens with only a few dozen engineers supporting tens of thousands of others.

No matter the size of your organization, it's reasonable to think about how you would make these kinds of sweeping changes across your collection of source code. Whether by choice or by necessity, having this ability will allow greater flexibility as your organization scales while keeping your source code malleable over time.

TL;DRs

- An LSC process makes it possible to rethink the immutability of certain technical decisions.
- Traditional models of refactoring break at large scales.
- Making LSCs means making a habit of making LSCs.

1 For some ideas about why, see Chapter 16.

2 It's possible in this federated world to say "we'll just commit to each repo as fast as possible to keep the duration of the build break small!" But that approach really doesn't scale as the number of federated repositories grows.

3 For a further discussion about this practice, see Chapter 15.

4 By "unfunded mandate," we mean "additional requirements imposed by an external entity without balancing compensation." Sort of like when the CEO says that everybody must wear an evening gown for "formal Fridays," but doesn't give you a corresponding raise to pay for your formal wear.

5 See *https://ieeexplore.ieee.org/abstract/document/8443579*.

6 This probably sounds like overkill, and it likely is. We're doing active research on the best way to determine the "right" set of tests for a given change, balancing the cost of compute time to run the tests, and the human cost of making the wrong choice.

7 The largest series of LSCs ever executed removed more than one billion lines of code from the repository over the course of three days. This was largely to remove an obsolete part of the repository that had been migrated to a new home; but still, how confident do you have to be to delete one billion lines of code?

8 LSCs are usually supported by tools that make finding, making, and reviewing changes relatively straight forward.

9 It is possible to ask TAP for single change "isolated" run, but these are very expensive and are performed only during off-peak hours.

10 There are obvious technical costs here in terms of compute and storage, but the human costs in time to review a change far outweigh the technical ones.

11 For example, we do not want the resulting tools to be used as a mechanism to fight over the proper spelling of "gray" or "grey" in comments.

12 In fact, Go recently introduced these kinds of language features specifically to support large-scale refactorings (see *https://talks.golang.org/2016/refactor.article*).

13 The only kinds of changes that the committee has outright rejected have been those that are deemed dangerous, such as converting all `NULL` instances to `nullptr`, or extremely low-value, such as changing spelling from British English to American English, or vice versa. As our experience with such changes has increased and the cost of LSCs has dropped, the threshold for approval has, as well.

14 This happens for many reasons: copy-and-paste from existing examples, committing changes that have been in development for some time, or simply reliance on old habits.

15 In actuality, this is the reasoning behind the original work on clang-format for C++.

16 Sadly, the systems we most want to organically decompose are those that are the most resilient to doing so. They are the plastic six-pack rings of the code ecosystem.

Chapter 23. Continuous Integration

Written by Rachel Tannenbaum

Edited by Lisa Carey

Continuous Integration, or CI, is generally defined as "a software development practice where members of a team integrate their work frequently [...] Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible."[1] Simply put, the fundamental goal of CI is to automatically catch problematic changes as early as possible.

In practice, what does "integrating work frequently" mean for the modern, distributed application? Today's systems have many moving pieces beyond just the latest versioned code in the repository. In fact, with the recent trend toward microservices, the changes that break an application are less likely to live inside the project's immediate codebase, and more likely to be in loosely coupled microservices on the other side of a network call. Whereas a traditional continuous build tests changes in your binary, an extension of this might test changes to upstream microservices. The dependency is just shifted from your function call stack to an HTTP request or Remote Procedure Calls (RPC).

Even further from code dependencies, an application might periodically ingest data or update machine learning models. It might execute on evolving operating systems, runtimes, cloud hosting services, and devices. It might be a feature that sits on top of a growing platform or be the platform that must accommodate a growing feature base. All of these things should be considered dependencies, and we should aim to "continuously integrate" their changes, too. Further complicating things, these changing components are often owned by developers outside our team, organization, or company and deployed on their own schedules.

So, perhaps a better definition for CI in today's world, particularly when developing at scale, is the following:

Continuous Integration (2): the continuous assembling and testing of our entire complex and rapidly evolving ecosystem.

It is natural to conceptualize CI in terms of testing because the two are tightly coupled, and we'll do so throughout this chapter. In previous chapters, we've

discussed a comprehensive range of testing, from unit to integration, to larger-scoped system.

From a testing perspective, CI is a paradigm to inform the following:

- *Which* tests to run *when* in the development/release workflow, as code (and other) changes are continuously integrated into it
- *How* to compose the system under test (SUT) at each point, balancing concerns like fidelity and setup cost

For example, which tests do we run on presubmit, which do we save for post-submit, and which do we save even later until our staging deploy? Accordingly, how do we represent our SUT at each of these points? As you might imagine, requirements for a presubmit SUT can differ significantly from those of a staging environment under test. For example, it can be dangerous for an application built from code pending review on presubmit to talk to real production backends (think security and quota vulnerabilities), whereas this is often acceptable for a staging environment.

And *why* should we try to optimize this often-delicate balance of testing "the right things" at "the right times" with CI? Plenty of prior work has already established the benefits of CI to the engineering organization and the overall business alike.[2] These outcomes are driven by a powerful guarantee: verifiable, and timely, proof that the application is good to progress to the next stage. We don't need to just hope that all contributors are very careful, responsible, and thorough; we can instead guarantee the working state of our application at various points from build throughout release, thereby improving confidence and quality in our products, and productivity of our teams.

In the rest of this chapter, we'll introduce some key CI concepts, best practices, and challenges, before looking at how we manage CI at Google with an introduction to our continuous build tool, TAP, and an in-depth study of one application's CI transformation.

CI Concepts

First, let's begin by looking at some core concepts of CI.

Fast Feedback Loops

As discussed in Chapter 11, the cost of a bug grows almost exponentially the later it is caught. Figure 23-1 shows all the places a problematic code change might be caught in its lifetime.

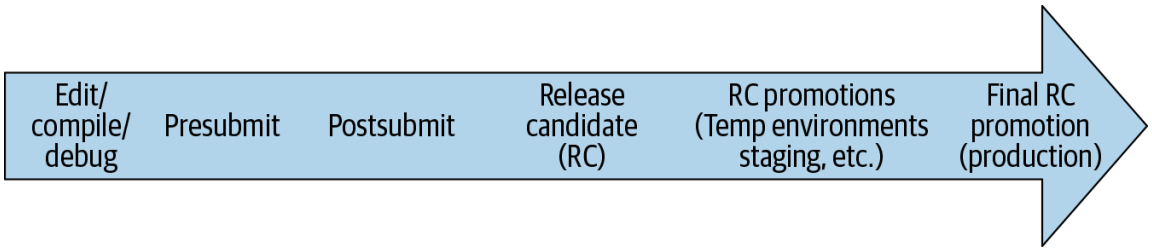

Figure 23-1. Life of a code change

In general, as issues progress to the "right" in our diagram, they become costlier for the following reasons:

- They must be triaged by an engineer who is likely unfamiliar with the problematic code change.
- They require more work for the code change author to recollect and investigate the change.
- They negatively affect others, whether engineers in their work or ultimately the end user.

To minimize the cost of bugs, CI encourages us to use *fast feedback loops*.[3] Each time we integrate a code (or other) change into a testing scenario and observe the results, we get a new *feedback loop*. Feedback can take many forms; following are some common ones (in order of fastest to slowest):

- The edit-compile-debug loop of local development
- Automated test results to a code change author on presubmit
- An integration error between changes to two projects, detected after both are submitted and tested together (i.e., on postsubmit)
- An incompatibility between our project and an upstream microservice dependency, detected by a QA tester in our staging environment, when the upstream service deploys its latest changes
- Bug reports by internal users who are opted in to a feature before external users
- Bug or outage reports by external users or the press

Canarying—or deploying to a small percentage of production first—can help minimize issues that do make it to production, with a subset-of-production initial feedback loop preceding all-of-production. However, canarying can cause problems, too, particularly around compatibility between deployments when multiple versions are deployed at once. This is sometimes known as *version skew*, a state of a distributed system in which it contains multiple incompatible versions of code, data, and/or configuration. Like many issues we look at in this book, version skew is another example of a challenging problem that can arise when trying to develop and manage software over time.

Experiments and *feature flags* are extremely powerful feedback loops. They reduce deployment risk by isolating changes within modular components that can be dynamically toggled in production. Relying heavily on feature-flag-guarding is a common paradigm for Continuous Delivery, which we explore further in Chapter 24.

ACCESSIBLE AND ACTIONABLE FEEDBACK

It's also important that feedback from CI be widely accessible. In addition to our open culture around code visibility, we feel similarly about our test reporting. We have a unified test reporting system in which anyone can easily look up a build or test run, including all logs (excluding user Personally Identifiable Information [PII]), whether for an individual engineer's local run or on an automated development or staging build.

Along with logs, our test reporting system provides a detailed history of when build or test targets began to fail, including audits of where the build was cut at each run, where it was run, and by whom. We also have a system for flake classification, which uses statistics to classify flakes at a Google-wide level, so engineers don't need to figure this out for themselves to determine whether their change broke another project's test (if the test is flaky: probably not).

Visibility into test history empowers engineers to share and collaborate on feedback, an essential requirement for disparate teams to diagnose and learn from integration failures between their systems. Similarly, bugs (e.g., tickets or issues) at Google are open with full comment history for all to see and learn from (with the exception, again, of customer PII).

Finally, any feedback from CI tests should not just be accessible but actionable—easy to use to find and fix problems. We'll look at an example of improving user-unfriendly feedback in our case study later in this chapter. By

improving test output readability, you automate the understanding of feedback.

Automation

It's well known that automating development-related tasks saves engineering resources in the long run. Intuitively, because we automate processes by defining them as code, peer review when changes are checked in will reduce the probability of error. Of course, automated processes, like any other software, will have bugs; but when implemented effectively, they are still faster, easier, and more reliable than if they were attempted manually by engineers.

CI, specifically, automates the *build* and *release* processes, with a Continuous Build and Continuous Delivery. Continuous testing is applied throughout, which we'll look at in the next section.

CONTINUOUS BUILD

The *Continuous Build* (CB) integrates the latest code changes at head,[4] and runs an automated build and test. Because the CB runs tests as well as building code, "breaking the build" or "failing the build" includes breaking tests as well as breaking compilation.

After a change is submitted, the CB should run all relevant tests. If a change passes all tests, the CB marks it passing or "green," as it is often displayed in user interfaces (UIs). This process effectively introduces two different versions of head in the repository: *true head*, or the latest change that was committed, and *green head*, or the latest change the CB has verified. Engineers are able to sync to either version in their local development. It's common to sync against green head to work with a stable environment, verified by the CB, while coding a change but have a process that requires changes to be synced to true head before submission.

CONTINUOUS DELIVERY

The first step in Continuous Delivery (CD; discussed more fully in "Continuous Delivery") is *release automation*, which continuously assembles the latest code and configuration from head into release candidates. At Google, most teams cut these at green, as opposed to true, head.

Release candidate (RC): A cohesive, deployable unit created by an automated process,[5] assembled of code, configuration, and other dependencies that have passed the continuous build.

Note that we include configuration in release candidates—this is extremely important, even though it can slightly vary between environments as the candidate is promoted. We're not necessarily advocating you compile configuration into your binaries—actually, we would recommend dynamic configuration, such as experiments or feature flags, for many scenarios.[6]

Rather, we are saying that any static configuration you *do* have should be promoted as part of the release candidate so that it can undergo testing along with its corresponding code. Remember, a large percent of production bugs are caused by "silly" configuration problems, so it's just as important to test your configuration as it is your code (and to test it along *with* the same code that will use it). Version skew is often caught in this release-candidate-promotion process. This assumes, of course, that your static configuration is in version control—at Google static configuration is in version control along with the code, and hence, goes through the same code review process.

We then define CD as follows:

Continuous Delivery (CD): a continuous assembling of release candidates, followed by the promotion and testing of those candidates throughout a series of environments—sometimes reaching production and sometimes not.

The promotion and deployment process often depends on the team. We'll show how our case study navigated this process.

For teams at Google that want continuous feedback from new changes in production (e.g., Continuous Deployment), it's usually infeasible to continuously push entire binaries, which are often quite large, on green. For that reason, doing a *selective* Continuous Deployment, through experiments or feature flags, is a common strategy.[7]

As an RC progresses through environments, its artifacts (e.g., binaries, containers) ideally should not be recompiled or rebuilt. Using containers such as Docker helps enforce consistency of an RC between environments, from local development onward. Similarly, using orchestration tools like Kubernetes (or in our case, usually Borg), helps enforce consistency between deployments. By enforcing consistency of our release and deployment between environments, we achieve higher-fidelity earlier testing and fewer surprises in production.

Continuous Testing

Let's look at how CB and CD fit in as we apply Continuous Testing (CT) to a code change throughout its lifetime, as shown Figure 23-2.

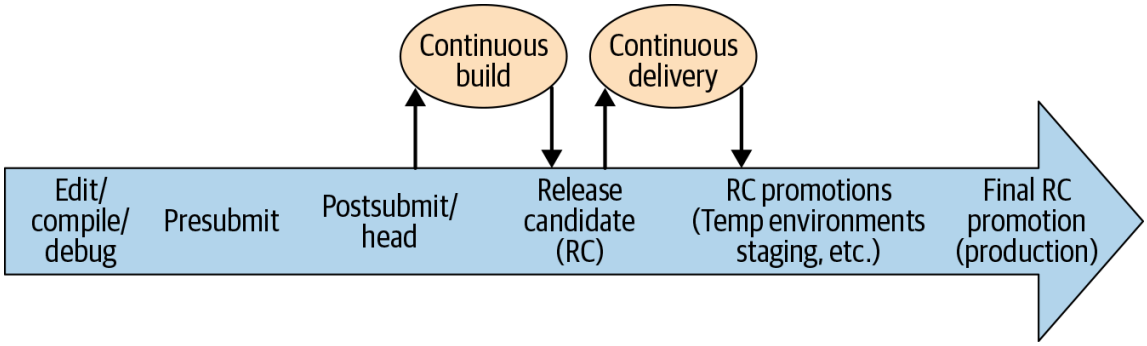

Figure 23-2. Life of a code change with CB and CD

The rightward arrow shows the progression of a single code change from local development to production. Again, one of our key objectives in CI is determining *what* to test *when* in this progression. Later in this chapter, we'll introduce the different testing phases and provide some considerations for what to test in presubmit versus post-submit, and in the RC and beyond. We'll show that as we shift to the right, the code change is subjected to progressively larger-scoped automated tests.

WHY PRESUBMIT ISN'T ENOUGH

With the objective to catch problematic changes as soon as possible and the ability to run automated tests on presubmit, you might be wondering: why not just run all tests on presubmit?

The main reason is that it's too expensive. Engineer productivity is extremely valuable and waiting a long time to run every test during code submission can be severely disruptive. Further, by removing the constraint for presubmits to be exhaustive, a lot of efficiency gains can be made if tests pass far more frequently than they fail. For example, the tests that are run can be restricted to certain scopes, or selected based on a model that predicts their likelihood of detecting a failure.

Similarly, it's expensive for engineers to be blocked on presubmit by failures arising from instability or flakiness that has nothing to do with their code change.

Another reason is that during the time we run presubmit tests to confirm that a change is safe, the underlying repository might have changed in a manner

that is incompatible with the changes being tested. That is, it is possible for two changes that touch completely different files to cause a test to fail. We call this a mid-air collision,[7] and though generally rare, it happens most days at our scale. CI systems for smaller repositories or projects can avoid this problem by serializing submits so that there is no difference between what is about to enter and what just did.

PRESUBMIT VERSUS POSTSUBMIT

So, which tests *should* be run on presubmit? Our general rule of thumb is: only fast, reliable ones. You can accept some loss of coverage on presubmit, but that means you need to catch any issues that slip by on post-submit, and accept some number of rollbacks. On post-submit you can accept longer times and some instability, as long as you have proper mechanisms to deal with it

NOTE

We'll show how TAP and our case study handle failure management in "CI at Google".

We don't want to waste valuable engineer productivity by waiting too long for slow tests or for too many tests—we typically limit presubmit tests to just those for the project where the change is happening. We also run tests concurrently, so there is a resource decision to consider, as well. Finally, we don't want to run unreliable tests on presubmit, because the cost of having many engineers affected by them, debugging the same problem that is not related to their code change, is too high.

Most teams at Google run their small tests (like unit tests) on presubmit[8]—these are the obvious ones to run as they tend to be the fastest and most reliable. Whether and how to run larger-scoped tests on presubmit is the more interesting question, and this varies by team. For teams that do want to run them, hermetic testing is a proven approach to reduce their inherent instability. Another option is to allow large-scoped tests to be unreliable on presubmit but disable them aggressively when they start failing.

RELEASE CANDIDATE TESTING

After a code change has passed the CB (this might take multiple cycles if there were failures), it will soon encounter CD, and be included in a pending release candidate.

As CD builds RCs, it will run larger tests against the entire candidate. We test a release candidate by promoting it through a series of test environments and testing it at each deployment. This can include a combination of sandboxed, temporary environments and shared test environments, like dev or staging. It's common to include some manual QA testing of the RC in shared environments, too.

There are several reasons why it's important to run a comprehensive, automated test suite against an RC, even if it is the same suite that CB just ran against the code on postsubmit (assuming the CD cuts at green):

As a sanity check

> We double check that nothing strange happened when the code was cut and recompiled in the RC.

For auditability

> If an engineer wants to check an RC's test results, they are readily available and associated with the RC, so they don't need to dig through CB logs to find them.

To allow for cherry picks

> If you apply a cherry-pick fix to an RC, your source code has now diverged from the latest cut tested by the CB.

For emergency pushes

> In that case, CD can cut from true head, and run the minimal set of tests necessary to feel confident about an emergency push, without waiting for the full CB to pass.

PRODUCTION TESTING

Our continuous, automated testing process goes all the way to the final deployed environment: production. We should run the same suite of tests against production (sometimes called *probers*), that we did against the release candidate earlier on to verify 1) the working state of production, according to our tests, and 2) the relevance of our tests, according to production.

Continuous testing at each step of the application's progression, each with its own trade-offs, serves as a reminder of the value in a "defense in depth" approach to catching bugs—it isn't just one bit of technology or policy that we rely upon for quality and stability, it's many testing approaches combined.

CI Is Alerting

By Titus Winters

As with responsibly running production systems, sustainably maintaining software systems also requires continual automated monitoring. Just as we use a monitoring and alerting system to understand how production systems respond to change, CI reveals how our software is responding to changes in its environment. Whereas production monitoring relies on passive alerts and active probers of running systems, CI uses unit and integration tests to detect changes to the software before it is deployed. Drawing comparisons between these two domains lets us apply knowledge from one to the other.

Both CI and alerting serve the same overall purpose in the developer workflow: identify problems as quickly as reasonably possible. CI emphasizes the early side of the developer workflow, and catches problems by surfacing test failures. Alerting focuses on the late end of the same workflow and catches problems by monitoring metrics and reporting when they exceed some threshold. Both are forms of "identify problems automatically, as soon as possible."

A well-managed alerting system helps to ensure that your Service-Level Objectives (SLOs) are being met. A good CI system helps to ensure that your build is in good shape—the code compiles, tests pass, and you could deploy a new release if you needed to. Best practice policies in both spaces focus a lot on ideas of fidelity and actionable alerting: tests should fail only when the important underlying invariant is violated, rather than because the test is brittle or flaky. A flaky test that fails every few CI runs is just as much of a problem as a spurious alert going off every few minutes and generating a page for the on-call. If it isn't actionable, it shouldn't be alerting. If it isn't actually violating the invariants of the SUT, it shouldn't be a test failure.

CI and alerting share an underlying conceptual framework. For instance, there's a similar relationship between localized signals (unit tests, monitoring of isolated statistics/cause-based alerting) and cross-dependency signals (integration and release tests, black-box probing). The highest fidelity indicators of whether an aggregate system is working are the end-to-end signals, but we pay for that fidelity in flakiness, increasing resource costs, and difficulty in debugging root causes.

Similarly, we see an underlying connection in the failure modes for both domains. Brittle cause-based alerts fire based on crossing an arbitrary threshold (say, retries in the past hour), without there necessarily being a

fundamental connection between that threshold and system health as seen by an end user. Brittle tests fail when an arbitrary test requirement or invariant is violated, without there necessarily being a fundamental connection between that invariant and the correctness of the software being tested. In most cases these are easy to write, and potentially helpful in debugging a larger issue. In both cases they are rough proxies for overall health/correctness, failing to capture the holistic behavior. If you don't have an easy end-to-end probe, but you do make it easy to collect some aggregate statistics, teams will write threshold alerts based on arbitrary statistics. If you don't have a high-level way to say, "Fail the test if the decoded image isn't roughly the same as this decoded image," teams will instead build tests that assert that the byte streams are identical.

Cause-based alerts and brittle tests can still have value; they just aren't the ideal way to identify potential problems in an alerting scenario. In the event of an actual failure, having more debug detail available can be useful. When SREs are debugging an outage, it can be useful to have information of the form, "An hour ago users started experiencing more failed requests. Around the same time the number of retries started ticking up. Let's start investigating there." Similarly, brittle tests can still provide extra debugging information, "The image rendering pipeline started spitting out garbage. One of the unit tests suggests that we're getting different bytes back from the JPEG compressor. Let's start investigating there."

Although monitoring and alerting are considered a part of the SRE/production management domain, where the insight of "Error Budgets" is well understood,[9] CI comes from a perspective that still tends to be focused on absolutes. Framing CI as the "left shift" of alerting starts to suggest ways to reason about those policies and propose better best practices:

- Having a 100% green rate on CI, just like having 100% uptime for a production service, is awfully expensive. If that is *actually* your goal, one of the biggest problems is going to be a race condition between testing and submission.
- Treating every alert as an equal cause for alarm is not generally the correct approach. If an alert fires in production, but the service isn't actually impacted, silencing the alert is the correct choice. The same is true for test failures: until our CI systems learn how to say, "This test is known to be failing for irrelevant reasons," we should probably be more liberal in accepting changes that disable a failed test. Not all test failures are indicative of upcoming production issues.

- Policies that say, "Nobody can commit if our latest CI results aren't green" are probably misguided. If CI reports an issue, such failures should definitely be *investigated* before letting people commit or compound the issue. But if the root cause is well understood and clearly would not affect production, blocking commits is unreasonable.

This "CI is alerting" insight is new, and we're still figuring out how to fully draw parallels. Given the higher stakes involved, it's unsurprising that SRE has put a lot of thought into best practices surrounding monitoring and alerting, whereas CI has been viewed as more of a luxury feature.[10] For the next few years, the task in software engineering will be to see where existing SRE practice can be reconceptualized in a CI context to help reformulate the testing and CI landscape—and perhaps where best practices in testing can help clarify goals and policies on monitoring and alerting.

CI Challenges

We've discussed some of the established best practices in CI and have introduced some of the challenges involved, such as the potential disruption to engineer productivity of unstable, slow, conflicting, or simply too many tests at presubmit. Some common additional challenges when implementing CI include the following:

- *Presubmit optimization*, including *which* tests to run at presubmit time given the potential issues we've already described, and *how* to run them.
- *Culprit finding* and *failure isolation*: Which code or other change caused the problem, and which system did it happen in? "Integrating upstream microservices" is one approach to failure isolation in a distributed architecture, when you want to figure out whether a problem originated in your own servers or a backend. In this approach, you stage combinations of your stable servers along with upstream microservices' new servers. (Thus, you are integrating the microservices' latest changes into your testing). This approach can be particularly challenging due to version skew: not only are these environments often incompatible, but you're also likely to encounter false positives—problems that occur in a particular staged combination that wouldn't actually be spotted in production.
- *Resource constraints*: Tests need resources to run, and large tests can be very expensive. In addition, the cost for the infrastructure for inserting automated testing throughout the process can be considerable.

There's also the challenge of *failure management*—what to do when tests fail. Although smaller problems can usually be fixed quickly, many of our teams find that it's extremely difficult to have a consistently green test suite when large end-to-end tests are involved. They inherently become broken or flaky and are difficult to debug; there needs to be a mechanism to temporarily disable and keep track of them so that the release can go on. A common technique at Google is to use bug "hotlists" filed by an on-call or release engineer, and triaged to the appropriate team. Even better is when these bugs can be automatically generated and filed—some of our larger products, like Google Web Server (GWS) and Google Assistant, do this. These hotlists should be curated to make sure any release-blocking bugs are fixed immediately. Nonrelease blockers should be fixed, too; they are less urgent, but should also be prioritized so the test suite remains useful and is not simply a growing pile of disabled, old tests. Often, the problems caught by end-to-end test failures are actually with tests rather than code.

Flaky tests pose another problem to this process. They erode confidence similar to a broken test, but finding a change to rollback is often more difficult because the failure won't happen all the time. Some teams rely on a tool to remove such flaky tests from presubmit temporarily while the flakiness is investigated and fixed. This keeps confidence high while allowing for more time to fix the problem.

Test instability is another significant challenge that we've already looked at in the context of presubmits. One tactic for dealing with this is to allow multiple attempts of the test to run. This is a common test configuration setting that teams use. Also, within test code, retries can be introduced at various points of specificity.

Another approach that helps with test instability (and other CI challenges) is hermetic testing, which we'll look at in the next section.

Hermetic Testing

Because talking to a live backend is unreliable, we often use hermetic backends for larger-scoped tests. This is particularly useful when we want to run these tests on presubmit, when stability is of utmost importance.
In Chapter 11, we introduced the concept of hermetic tests:

Hermetic tests: tests run against a test environment (i.e., application servers and resources) that is entirely self-contained (i.e., no external dependencies like production backends).

Hermetic tests have two important properties: greater determinism (i.e., stability), and isolation. Hermetic servers are still prone to some sources of nondeterminism, like system time, random number generation, and race conditions. But, what goes into the test doesn't change based on outside dependencies, so when you run a test twice with the same application and test code, you should get the same results. If a hermetic test fails, you know that it's due to a change in your application code or tests (with a minor caveat: they can also fail due to a restructuring of your hermetic test environment, but this should not change very often). For this reason, when CI systems rerun tests hours or days later to provide additional signals, hermeticity makes test failures easier to narrow down.

The other important property, isolation, means that problems in production should not affect these tests. We generally run these tests all on the same machine, as well, so we don't have to worry about network connectivity issues. The reverse also holds: problems caused by running hermetic tests should not affect production.

Hermetic test success should not depend on the user running the test. This allows people to reproduce tests run by the CI system, and allows people (e.g., library developers) to run tests owned by other teams.

One type of hermetic backend is a fake. As discussed in Chapter 13, these can be cheaper than running a real backend, but they take work to maintain and have limited fidelity.

The cleanest option to achieve a presubmit-worthy integration test is with a fully hermetic setup; that is, starting up the entire stack sandboxed,[11] and Google provides out-of-the-box sandbox configurations for popular components, like databases, to make it easier. This is more feasible for smaller applications with fewer components, but there are exceptions at Google, even one (by DisplayAds) that starts about four hundred servers from scratch on every presubmit as well as continuously on post-submit. Since the time that system was created, though, record/replay has emerged as a more popular paradigm for larger systems and tends to be cheaper than starting up a large sandboxed stack.

Record/replay[12] systems record live backend responses, cache them, and replay them in a hermetic test environment. Record/replay is a powerful tool for reducing test instability, but one downside is that it leads to brittle tests: it's difficult to strike a balance between the following:

False positives

The test passes when it probably shouldn't have because we are hitting the cache too much and missing problems that would surface when capturing a new response.

False negatives

The test fails when it probably shouldn't have because we are hitting the cache too little. This requires responses to be updated, which can take a long time and lead to test failures that must be fixed, many of which might not be actual problems. This process is often submit-blocking, which is not ideal.

Ideally, a record/replay system should detect only problematic changes and cache-miss only when a request has changed in a meaningful way. In the event that that change causes a problem, the code change author would rerun the test with an updated response, see that the test is still failing, and thereby be alerted to the problem. In practice, knowing when a request has changed in a meaningful way can be incredibly difficult in a large and ever-changing system.

THE HERMETIC GOOGLE ASSISTANT

Google Assistant provides a framework for engineers to run end-to-end tests, including a test fixture with functionality for setting up queries, specifying whether to simulate on a phone or a smart home device, and validating responses throughout an exchange with Google Assistant.

One of its greatest success stories was making its test suite fully hermetic on presubmit. When the team previously used to run nonhermetic tests on presubmit, the tests would routinely fail. In some days, the team would see more than 50 code changes bypass and ignore the test results. In moving presubmit to hermetic, the team cut the runtime by a factor of 14, with virtually no flakiness. It still see failures, but those failures tend to be fairly easy to find and rollback.

Now that nonhermetic tests have been pushed to post-submit, it results in failures accumulating there, instead. Debugging failing end-to-end tests is still difficult, and some teams don't have time to even try, so they just disable them. That's better than having it stop all development for everyone, but it can result in production failures.

One of the team's current challenges is to continue to fine-tuning its caching mechanisms so that presubmit can catch more types of issues that have been discovered only post-submit in the past, without introducing too much brittleness.

Another is how to do presubmit testing for the decentralized Assistant given that components are shifting into their own microservices. Because the Assistant has a

large and complex stack, the cost of running a hermetic stack on presubmit, in terms of engineering work, coordination and resources, would be very high.

Finally, the team is taking advantage of this decentralization in a clever new post-submit failure-isolation strategy. For each of the N microservices within the Assistant, the team will run a post-submit environment containing the microservice built at head, along with production (or close to it) versions of the other $N - 1$ services, to isolate problems to the newly built server. This setup would normally be $O(N^2)$ cost to facilitate, but the team leverages a cool feature called *hotswapping* to cut this cost to $O(N)$. Essentially, hotswapping allows a request to instruct a server to "swap" in the address of a backend to call instead of the usual one. So only N servers need to be run, one for each of the microservices cut at head—and they can reuse the same set of prod backends swapped in to each of these N "environments."

As we've seen in this section, hermetic testing can both reduce instability in larger-scoped tests and help isolate failures—addressing two of the significant CI challenges we identified in the previous section. However, hermetic backends can also be more expensive because they use more resources and are slower to set up. Many teams use combinations of hermetic and live backends in their test environments.

CI at Google

Now let's look in more detail at how CI is implemented at Google. First, we'll look at our global continuous build, TAP, used by the vast majority of teams at Google, and how it enables some of the practices and addresses some of the challenges that we looked at in the previous section. We'll also look at one application, Google Takeout, and how a CI transformation helped it scale both as a platform and as a service.

TAP: Google's Global Continuous Build

We run a massive continuous build, called the Test Automation Platform (TAP), of our entire codebase. It is responsible for running the majority of our automated tests. As a direct consequence of our use of a monorepo), TAP is the gateway for almost all changes at Google. Every day it is responsible for handling more than 50,000 unique changes *and* running more than four billion individual test cases.

TAP is the beating heart of Google's development infrastructure. Conceptually, the process is very simple. When an engineer attempts to submit code, TAP runs the associated tests and reports success or failure. If the tests pass, the change is allowed into the codebase.

PRESUBMIT OPTIMIZATION

To catch issues quickly and consistently, it is important to ensure that tests are run against every change. Without a CB, running tests is usually left to individual engineer discretion and that often leads to a few motivated engineers trying to run all tests and keep up with the failures.

As discussed earlier, waiting a long time to run every test on presubmit can be severely disruptive, in some cases taking hours. To minimize the time spent waiting, Google's CB approach allows potentially breaking changes to land in the repository (remember that they become immediately visible to the rest of the company!). All we ask is for each team to create a fast subset of tests, often a project's unit tests, that can be run before a change is submitted (usually before it is sent for code review)—the presubmit. Empirically, a change that passes the presubmit has a very high likelihood (95%+) of passing the rest of the tests, and we optimistically allow it to be integrated so that other engineers can then begin to use it.

After a change has been submitted, we use TAP to asynchronously run all potentially affected tests, including larger and slower tests.

When a change causes a test to fail in TAP, it is imperative that the change be fixed quickly to prevent blocking other engineers. We have established a cultural norm that strongly discourages committing any new work on top of known failing tests, though flaky tests make this difficult. Thus, when a change is committed that breaks a team's build in TAP, that change may prevent the team from making forward progress or building a new release. As a result, dealing with breakages quickly is imperative.

To deal with such breakages each team has a "Build Cop." The Build Cop's responsibility is keeping all the tests passing in their particular project, regardless of who breaks them. When a Build Cop is notified of a failing test in their project, they drop whatever they are doing and fix the build. This is usually by identifying the offending change and determining whether it needs to be rolled back (the preferred solution) or can be fixed going forward (a riskier proposition).

In practice the trade-off of allowing changes to be committed before verifying all tests has really paid off; the average wait time to submit a change is around 11 minutes, often run in the background. Coupled with the discipline of the Build Cop, we are able to efficiently detect and address breakages detected by longer running tests with a minimal amount of disruption.

CULPRIT FINDING

One of the problems we face with large test suites at Google is finding the specific change that broke a test. Conceptually, this should be really easy: grab a change, run the tests, if any tests fail, mark the change as bad. Unfortunately, due to a prevalence of flakes and the occasional issues with the testing infrastructure itself, having confidence that a failure is real isn't easy. To make matters more complicated, TAP must evaluate so many changes a day (more than one a second) that it can no longer run every test on every change. Instead, it falls back to batching related changes together, which reduces the total number of unique tests to be run. Although this approach can make it faster to run tests, it can obscure which change in the batch caused a test to break.

To speed up failure identification we use two different approaches. First, TAP automatically splits a failing batch up into individual changes and reruns the tests against each change in isolation. This process can sometimes take a while to converge on a failure, so in addition we have created culprit finding tools that an individual developer can use to binary search through a batch of changes and identify which one is the likely culprit.

FAILURE MANAGEMENT

After a breaking change has been isolated it is important to fix it as quickly as possible. The presence of failing tests can quickly begin to erode confidence in the test suite. As mentioned previously, fixing a broken build is the responsibility of the Build Cop. The most effective tool the Build Cop has is the *rollback*.

Rolling a change back is often the fastest and safest route to fix a build because it quickly restores the system to a known good state.[13] In fact, TAP has recently been upgraded to automatically roll back changes when it has high confidence that they are the culprit.

Fast rollbacks work hand in hand with a test suite to ensure continued productivity. Tests give us confidence to change, rollbacks give us confidence to undo. Without tests, rollbacks can't be done safely. Without rollbacks, broken tests can't be fixed quickly, thereby reducing confidence in the system.

RESOURCE CONSTRAINTS

Although engineers can run tests locally, most test executions happen in a distributed build-and-test system called *Forge*. Forge allows engineers to run their builds and tests in our datacenters, which maximizes parallelism. At our scale, the resources required to run all tests executed on-demand by engineers, and all tests being run as part of the CB process, are enormous. Even given the amount of compute resource we have, systems like Forge and TAP are resource constrained. To work around these constraints, engineers working on TAP have come up with some clever ways to determine which tests should be run at which times to ensure that the minimal amount of resources are spent to validate a given change.

The primary mechanism for determining which tests need to be run is an analysis of the downstream dependency graph for every change. Google's distributed build tools, Forge and Blaze, maintain a near-real-time version of the global dependency graph and make it available to TAP. As a result, TAP can quickly determine which tests are downstream from any change and run the minimal set to be sure the change is safe.

Another factor influencing the use of TAP is the speed of tests being run. TAP is often able to run changes with fewer tests sooner than those with more tests. This bias encourages engineers to write small, focused changes. The difference in waiting time between a change that triggers 100 tests and one that triggers 1,000 can be tens of minutes on a busy day. Engineers who want to spend less time waiting end up making smaller, targeted changes, which is a win for everyone.

CI Case Study: Google Takeout

Google Takeout started out as a data backup and download product in 2011. Its founders pioneered the idea of "data liberation"—that users should be able to easily take their data with them, in a usable format, wherever they go. They began by integrating Takeout with a handful of Google products themselves, producing archives of users' photos, contact lists, and so on for download at their request. However, Takeout didn't stay small for long, growing as both a platform and a service for a wide variety of Google products. As we'll see, effective CI is central to keeping any large project healthy, but is especially critical when applications rapidly grow.

SCENARIO #1: CONTINUOUSLY BROKEN DEV DEPLOYS

Problem: As Takeout gained a reputation as a powerful Google-wide data fetching, archiving, and download tool, other teams at the company began to turn to it, requesting APIs so that their own applications could provide backup and download functionality, too, including Google Drive (folder downloads are served by Takeout) and Gmail (for zip file previews). All in all, Takeout grew from being the backend for just the original Google Takeout product, to providing APIs for at least 10 other Google products, offering a wide range of functionality.

The team decided to deploy each of the new APIs as a customized instance, using the same original Takeout binaries but configuring them to work a little differently. For example, the environment for Drive bulk downloads has the largest fleet, the most quota reserved for fetching files from the Drive API, and some custom authentication logic to allow nonsigned-in users to download public folders.

Before long, Takeout faced "flag issues." Flags added for one of the instances would break the others, and their deployments would break when servers could not start up due to configuration incompatibilities. Beyond feature configuration, there was security and ACL configuration, too. For example, the consumer Drive download service should not have access to keys that encrypt enterprise Gmail exports. Configuration quickly became complicated and led to nearly nightly breakages.

Some efforts were made to detangle and modularize configuration, but the bigger problem this exposed was that when a Takeout engineer wanted to make a code change, it was not practical to manually test that each server started up under each configuration. They didn't find out about configuration failures until the next day's deploy. There were unit tests that ran on presubmit and post-submit (by TAP), but those weren't sufficient to catch these kinds of issues.

What the team did

The team created temporary, sandboxed mini-environments for each of these instances that ran on presubmit and tested that all servers were healthy on startup. Running the temporary environments on presubmit prevented 95% of broken servers from bad configuration, and reduced nightly deployment failures by 50%.

Although these new sandboxed presubmit tests dramatically reduced deployment failures, they didn't remove them entirely. In particular, Takeout's end-to-end tests would still frequently break the deploy, and these tests were difficult to run on presubmit (because they use test accounts, which still behave like real accounts in some respects and are subject to the same security and privacy safeguards). Redesigning them to be presubmit friendly would have been too big an undertaking.

If the team couldn't run end-to-end tests in presubmit, when could it run them? It wanted to get end-to-end test results more quickly than the next day's dev deploy and decided every two hours was a good starting point. But the team didn't want to do a full dev deploy this often—this would incur overhead and disrupt long-running processes that engineers were testing in dev. Making a new shared test environment for these tests also seemed like too much overhead to provision resources for, plus culprit finding (i.e., finding the deployment that led to a failure) could involve some undesirable manual work.

So, the team reused the sandboxed environments from presubmit, easily extending them to a new post-submit environment. Unlike presubmit, post-submit was compliant with security safeguards to use the test accounts (for one, because the code has been approved), so the end-to-end tests could be run there. The post-submit CI runs every two hours, grabbing the latest code and configuration from green head, creates an RC, and runs the same end-to-end test suite against it that is already run in dev.

Lesson learned

Faster feedback loops prevent problems in dev deploys:

- Moving tests for different Takeout products from "after nightly deploy" to presubmit prevented 95% of broken servers from bad configuration, and reduced nightly deployment failures by 50%.
- Though end-to-end tests couldn't be moved all the way to presubmit, they were still moved from "after nightly deploy" to "post-submit within two hours". This effectively cut the "culprit set" by 12 times.

SCENARIO #2: INDECIPHERABLE TEST LOGS

Problem: As Takeout incorporated more Google products, it grew into a mature platform into which product teams inserted directly into Takeout's binary plug-ins with product-specific data-fetching client code. For example, the Google Photos plug-in knows how to fetch photos, album metadata, and

the like. Takeout expanded from its original "handful" of products to now integrate with more than *90*.

Takeout's end-to-end tests dumped its failures to a log, and this approach didn't scale to 90 product plugins. As more products integrated, more failures were introduced. Even though the team was running the tests earlier and more often with the addition of the post-submit CI, multiple failures would still pile up inside and were easy to miss. Going through these logs became a frustrating time sink, and the tests were almost always failing.

What the team did

The team refactored the tests into a dynamic, configuration-based suite (using a parameterized test runner) that reported results in a friendlier UI, clearly showing individual test results as green or red: no more digging through logs. They also made failures much easier to debug, most notably, by displaying failure information, with links to logs, directly to the error message. For example, if Takeout failed to fetch a file from Gmail, the test would dynamically construct a link that searched for that file's ID in the Takeout logs and include it in the test failure message. This automated much of the debugging process for product plug-in engineers and required less of the Takeout team's assistance in sending them logs, as demonstrated in Figure 23-3.

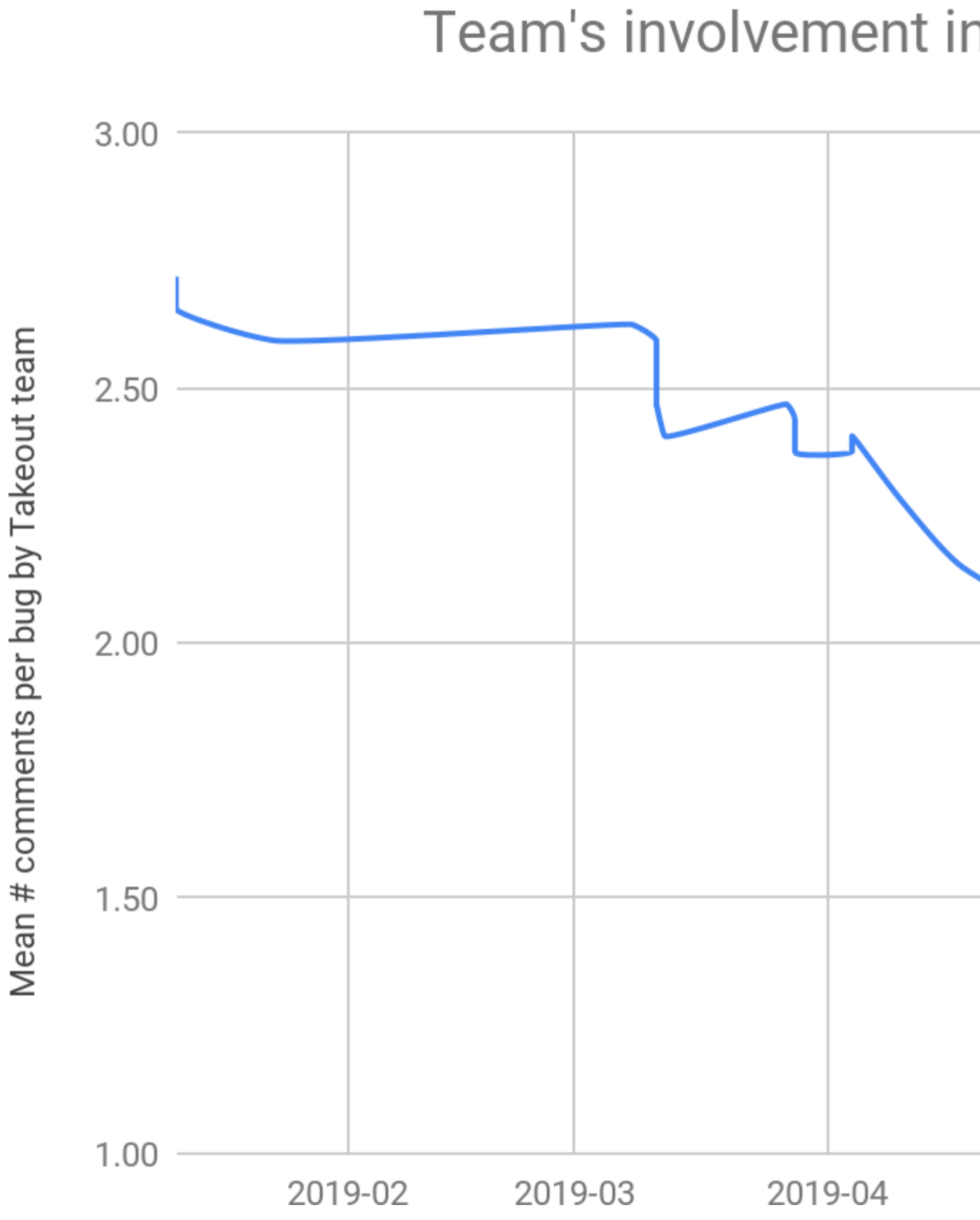
Team's involvement in
Mean # comments per bug by Takeout team
3.00
2.50
2.00
1.50
1.00
2019-02
2019-03
2019-04

Figure 23-3. The team's involvement in debugging client features

Lesson learned

Accessible, actionable feedback from CI reduces test failures and improves productivity. These initiatives reduced the Takeout team's involvement in debugging client (product plugin) test failures by 35%.

SCENARIO #3: DEBUGGING "ALL OF GOOGLE"

Problem: An interesting side effect of the Takeout CI that the team did not anticipate was that because it verified the output of 90-some odd end-user-facing products, in the form of an archive, they were basically testing "all of Google" and catching issues that had nothing to do with Takeout. This was a good thing—Takeout was able to help contribute to the quality of Google's products overall. However, this introduced a problem for their CI processes: they needed better failure isolation so that they could determine which problems were in their build (which were the minority) and which lay in loosely coupled microservices behind the product APIs they called.

What the team did

The team's solution was to run the exact same test suite continuously against production as it already did in its post-submit CI. This was cheap to implement and allowed the team to isolate which failures were new in its build, and which were in production; for instance, the result of a microservice release somewhere else "in Google."

Lesson learned

Running the same test suite against prod and a post-submit CI (with newly built binaries, but the same live backends) is a cheap way to isolate failures.

Remaining challenge

Going forward, the burden of testing "all of Google" (obviously, this is an exaggeration as most product problems are caught by their respective teams) grows as Takeout integrates with more products and as those products become more complex. Manual comparisons between this CI and prod are an expensive use of the BuildCop's time.

Future improvement

This presents an interesting opportunity to try hermetic testing with record/replay in Takeout's post-submit CI. In theory, this would eliminate failures from backend product APIs surfacing in Takeout's CI, which would make the suite more stable, and effective at catching failures in the last two hours of Takeout changes—which is its intended purpose.

SCENARIO #4: KEEPING IT GREEN

Problem: As the platform supported more product plug-ins, which each included end-to-end tests, these tests would fail and the end-to-end test suites were nearly always broken. The failures could not all be immediately fixed. Many were due to bugs in product plug-in binaries, which the Takeout team had no control over. And some failures mattered more than others—low priority bugs and bugs in the test code did not need to block a release, whereas higher priority bugs did. The team could easily disable tests by commenting them out, but that would make the failures too easy to forget about.

One common source of failures: tests would break when product plug-ins were rolling out a feature. For example, a playlist-fetching feature for the YouTube plug-in might be enabled for testing in dev for a few months before being enabled in prod. The Takeout tests only knew about one result to check, so that often resulted in the test needing to be disabled in particular environments and manually curated as the feature rolled out.

What the team did

The team came up with a strategic way to disable failing tests, by tagging them with an associated bug and filing that off to the responsible team (usually a product plug-in team). When a failing test was tagged with a bug, the team's testing framework would suppress its failure. This allowed the test suite to stay green and still provide confidence that everything else, besides the known issues, was passing, as illustrated in Figure 23-4.

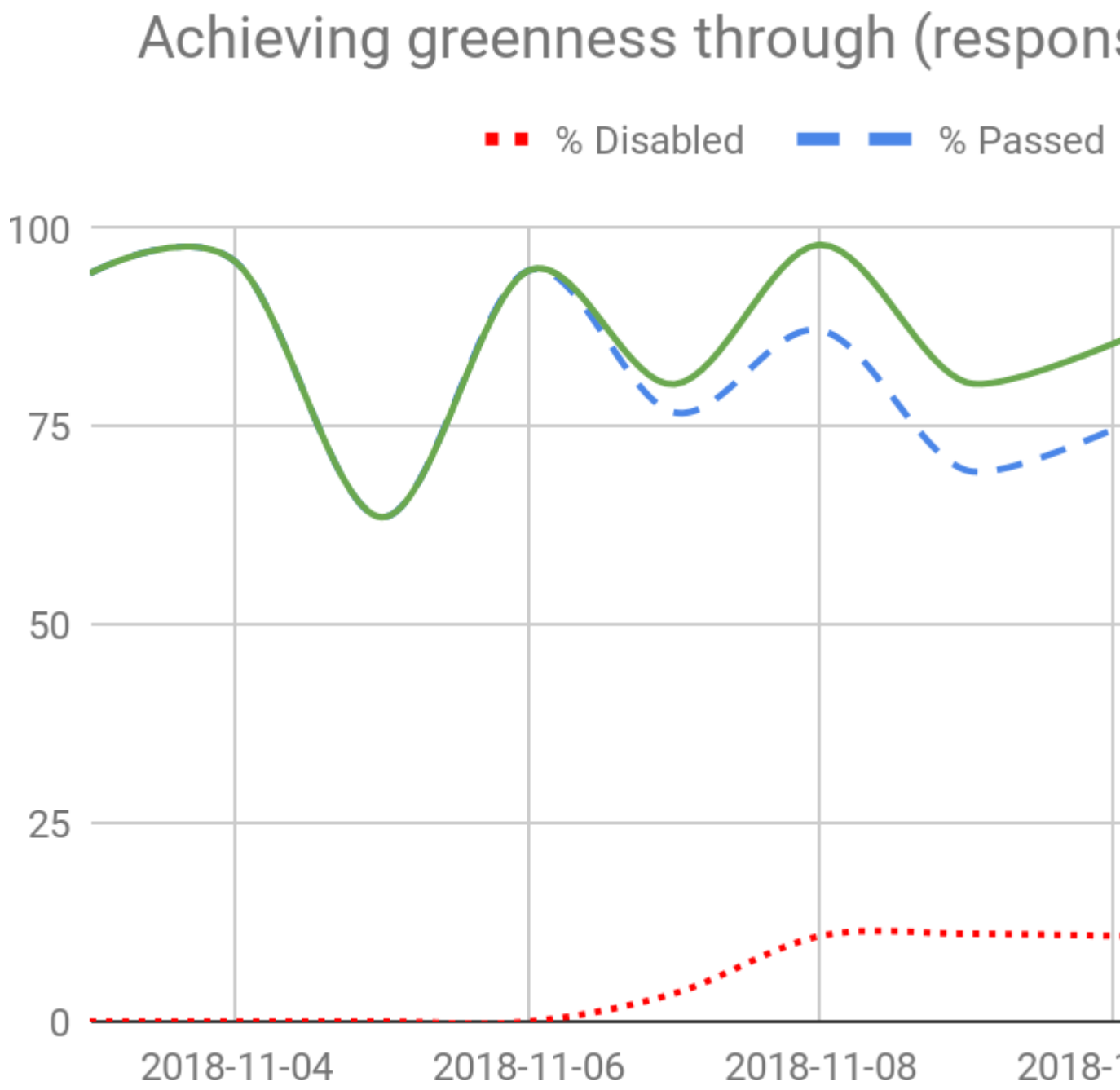

Figure 23-4. Achieving greenness through (responsible) test disablement

For the rollout problem, the team added capability for plug-in engineers to specify the name of a feature flag, or ID of a code change, that enabled a particular feature along with the output to expect both with and without the feature. The tests were equipped to query the test environment to determine whether the given feature was enabled there, and verified the expected output accordingly.

When bug tags from disabled tests began to accumulate and were not updated, the team automated their cleanup. The tests would now check whether a bug was closed by querying our bug system's API. If a tagged-failing test actually passed, and was passing for longer than a configured time

limit, the test would prompt to clean up the tag (and mark the bug fixed, if it wasn't already). There was one exception for this strategy: flaky tests. For these, the team would allow a test to be tagged as flaky, and the system wouldn't prompt a tagged "flaky" failure for cleanup if it passed.

These changes made a mostly self-maintaining test suite, as illustrated in Figure 23-5.

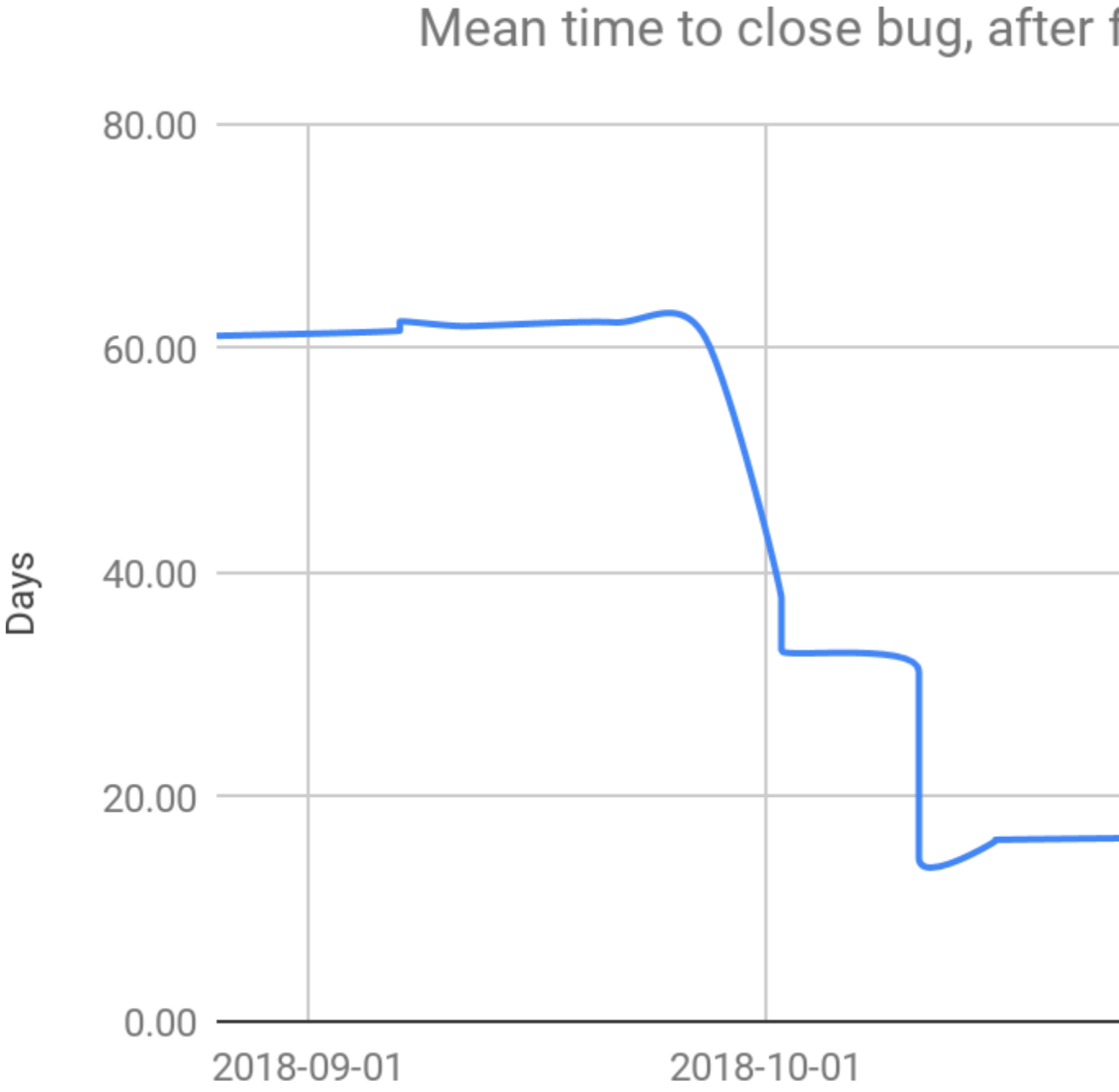

Figure 23-5. Mean time to close bug, after fix submitted

Lessons learned

Disabling failing tests that can't be immediately fixed is a practical approach to keeping your suite green, which gives confidence that you're aware of all

test failures. Also, automating the test suite's maintenance, including rollout management and updating tracking bugs for fixed tests, keeps the suite clean and prevents technical debt. In DevOps parlance, we could call the metric in Figure 23-5 MTTCU: mean time to clean up.

Future improvement

Automating the filing and tagging of bugs would be a helpful next step. This is still a manual and burdensome process. As mentioned earlier, some of our larger teams already do this.

Further challenges

The scenarios we've described are far from the only CI challenges faced by Takeout, and there are still more problems to solve. For example, we mentioned the difficulty of isolating failures from upstream services in "CI Challenges". This is a problem that Takeout still faces with rare breakages originating with upstream services, such as when a security update in the streaming infrastructure used by Takeout's "Drive folder downloads" API broke archive decryption when it deployed to production. The upstream services are staged and tested themselves, but there is no simple way to automatically check with CI if they are compatible with Takeout after launched into production. An initial solution involved creating an "upstream staging" CI environment to test production Takeout binaries against the staged versions of their upstream dependencies. However, this proved difficult to maintain, with additional compatibility issues between staging and production versions.

But I Can't Afford CI

You might be thinking that's all well and good, but you have neither the time nor money to build any of this. We certainly acknowledge that Google might have more resources to implement CI than the typical startup does. Yet many of our products have grown so quickly that they didn't have time to develop a CI system either (at least not an adequate one).

In your own products and organizations, try and think of the cost you are already paying for problems discovered and dealt with in production. These negatively affect the end user or client, of course, but they also affect the team. Frequent production fire-fighting is stressful and demoralizing. Although building out CI systems is expensive, it's not necessarily a new cost as much as a cost shifted left to an earlier—and more preferable—stage, reducing the incidence, and thus the cost, of problems occurring too far to the

right. CI leads to a more stable product and happier developer culture in which engineers feel more confident that "the system" will catch problems, and they can focus more on features and less on fixing.

Conclusion

Even though we've described our CI processes and some of how we've automated them, none of this is to say that we have developed perfect CI systems. After all, a CI system itself is just software and is never complete, and should be adjusted to meet the evolving demands of the application and engineers it is meant to serve. We've tried to illustrate this with the evolution of Takeout's CI and the future areas of improvement we point out.

TL;DRs

- A CI system decides what tests to use, and when.
- CI systems become progressively more necessary as your codebase ages and grows in scale.
- CI should optimize quicker, more reliable tests on presubmit, slower, less deterministic tests on post-submit.
- Accessible, actionable feedback allows a CI system to become more efficient.

1 *https://www.martinfowler.com/articles/continuousIntegration.html*

2 Forsgren, Nicole, et al. (2018). Accelerate: The Science of Lean Software and DevOps: Building and Scaling High Performing Technology Organizations. IT Revolution.

3 This is also sometimes called "shifting left on testing."

4 *Head* is the latest versioned code in our monorepo. In other workflows, this is also referred to as *master*, *mainline*, or *trunk*. Correspondingly, integrating at head is also known as *trunk-based development*.

5 At Google, release automation is managed by a separate system from TAP. We won't focus on *how* release automation assembles RCs, but if you're interested, we do refer you to the XREF(SRE Book) in which our release automation technology (a system called Rapid) is discussed in detail.

6 CD with experiments and feature flags is discussed further in "Fast Feedback Loops" as well as "Continuous Delivery".

7 We call these "mid-air collisions" because the probability of it occurring is extremely low; however, when this does happen, the results can be quite surprising.

8 Each team at Google configures a subset of its project's tests to run on presubmit (versus postsubmit). In reality, our continuous build actually optimizes some presubmit tests to be saved for postsubmit, behind the scenes. We discuss this further in XREF("TAP").

9 Aiming for 100% uptime is the wrong target. Pick something like 99.9% or 99.999% as a business or product trade-off, define and monitor your actual uptime, and use that "budget" as an input to how aggressively you're willing to push risky releases.

10 We believe CI is actually critical to the software engineering ecosystem: a must-have, not a luxury. But that is not universally understood yet.

11 In practice, it's often difficult to make a *completely* sandboxed test environment, but the desired stability can be achieved by minimizing outside dependencies.

12 See Chapter 14

13 Any change to Google's codebase can be rolled back with two clicks!

Chapter 24. Continuous Delivery

Written by Radha Narayan, Bobbi Jones, Sheri Shipe, and David Owens

Edited by Lisa Carey

Given how quickly and unpredictably the technology landscape shifts, the competitive advantage for any product lies in its ability to quickly go to market. An organization's velocity is a critical factor in its ability to compete with other players, maintain product and service quality, or adapt to new regulation. This velocity is bottlenecked by the time to deployment. Deployment doesn't just happen once, at initial launch. There is a saying among educators, that no lesson-plan survives its first contact with the student body. In much the same way, no software is perfect at first launch, and the only guarantee is that you'll have to update it. Quickly.

The long-term life cycle of a software product involves rapid exploration of new ideas, rapid responses to landscape shifts or user issues, and enabling developer velocity at scale. From Eric Raymond's *The Cathedral and the Bazaar* to Eric Reis' *Lean Startup*, the key to any organization's long-term success has always been in its ability to get ideas executed and into users' hands as quickly as possible, and reacting quickly to their feedback. Martin Fowler, in his book *Continuous Delivery*[1] (aka CD), points out that "the biggest risk to any software effort is that you end up building something that isn't useful. The earlier and more frequently you get working software in front of real users, the quicker you get feedback to find out how valuable it really is."

Work that stays in progress for a long time before delivering user value is high risk and high cost, and can even be a drain on morale. At Google, we strive to release early and often, or "launch and iterate," to enable teams to see the impact of their work quickly, and to adapt faster to a shifting market. The value of code is not realized at the time of submission but when features are available to your users. Reducing the time between "code complete" and user feedback minimizes the cost of work that is in progress.

You get extraordinary outcomes by realizing that the launch never lands but that it begins a learning cycle where you then fix the next most important thing, measure how it went, fix the next thing, etc.—and it is never complete.

—David Weekly, Former Google product manager

At Google, the practices we describe in this book allow hundreds (or in some cases thousands) of engineers to quickly troubleshoot problems, to independently work on new features without worrying about the release, and to understand the effectiveness of new features through A/B experimentation. This chapter focuses on the key levers of rapid innovation, including managing risk, enabling developer velocity at scale, and understanding the cost and value trade-off of each feature you launch.

Idioms of Continuous Delivery at Google

A core tenet of Continuous Delivery (CD) as well as of Agile methodology is that over time, smaller batches of changes result in higher quality; in other words, *faster is safer*. This can seem deeply controversial to teams at first glance, especially if the prerequisites for setting up CD—for example, Continuous Integration (CI) and testing—are not yet in place. Because it might take a while for all teams to realize the ideal of CD, we focus on developing various aspects that deliver value independently en route to the end goal. Here are some of these:

Agility

: Release frequently and in small batches

Automation

: Reduce or remove repetitive overhead of frequent releases

Isolation

: Strive for modular architecture to isolate changes and make troubleshooting easier

Reliability

: Measure key health indicators like crashes or latency and keep improving them

Data-driven decision making

: Use A/B testing on health metrics to ensure quality

Phased rollout

: Roll out changes to a few users before shipping to everyone

At first, releasing new versions of software frequently might seem risky. As your userbase grows, you might fear the backlash from angry users if there are any bugs that you didn't catch in testing, and you might quite simply have too much new code in your product to test exhaustively. But this is precisely

where CD can help. Ideally, there are so few changes between one release and the next that troubleshooting issues is trivial. In the limit, with CD, every change goes through the QA pipeline and is automatically deployed into production. This is often not a practical reality for many teams, and so there is often work of culture change toward CD as an intermediate step, during which teams can build their readiness to deploy at any time without actually doing so, building up their confidence to release more frequently in the future.

Velocity Is a Team Sport: How to Break Up a Deployment into Manageable Pieces

When a team is small, changes come into a codebase at a certain rate. We've seen an antipattern emerge as a team grows over time or splits into subteams: a subteam branches off its code to avoid stepping on anyone's feet, but then struggles, later, with integration and culprit-finding. At Google, we prefer that teams continue to develop at head in the shared codebase and set up CI testing, automatic rollbacks and culprit-finding to identify issues quickly. This is discussed at length in the Chapter 23.

One of our codebases, YouTube, is a large, monolithic Python application. The release process is laborious, with Build Cops, release managers and other volunteers. Almost every release has multiple cherry-picked changes and respins. There is also a 50-hour manual regression testing cycle run by a remote QA team on every release. When the operational cost of a release is this high, a cycle begins to develop in which you wait to push out your release until you're able to test it a bit more. Meanwhile someone wants to add just one more feature that's almost ready, and pretty soon you have yourself a release process that's laborious, error prone, and slow. Worst of all, the experts who did the release last time are burned out and have left the team, and now nobody even knows how to troubleshoot those strange crashes that happen when you try to release an update, leaving you panicky at the very thought of pushing that button.

If your releases are costly and sometimes risky, the *instinct* is to slow down your release cadence and increase your stability period. However, this only provides short-term stability gains, and over time it slows velocity and frustrates teams and users. The *answer* is to reduce cost, increase discipline, and make the risks more incremental, but it is critical to resist the obvious operational fixes and invest in long-term architectural changes. The obvious operational fixes to this problem lead to a few traditional approaches:

reverting to a traditional planning model that leaves little room for learning or iteration, adding more governance and oversight to the development process, and implementing risk reviews or rewarding low-risk (and often low-value) features.

The investment with the best return, though, is migrating to a microservice architecture, which can empower a large product team with the ability to remain scrappy and innovative while simultaneously reducing risk. In some cases, at Google, the answer has been to rewrite an application from scratch rather than simply migrating it, establishing the desired modularity into the new architecture. Although either of these options can take months and is likely painful in the short term, the value gained in terms of operational cost and cognitive simplicity will pay off over an application's lifespan of years.

Evaluating Changes in Isolation: Flag-Guarding Features

A key to reliable continuous releases is to make sure engineers *"flag guard" all changes*. As a product grows, there will be multiple features under various stages of development coexisting in a binary. Flag guarding can be used to control the inclusion or expression of feature code in the product on a feature-by-feature basis and can be expressed differently for release and development builds. A feature flag disabled for a build should allow build tools to strip the feature from the build if the language permits it. For instance, a stable feature that has already shipped to customers might be enabled for both development and release builds. A feature under development might be enabled only for development, protecting users from an unfinished feature. New feature code lives in the binary alongside the old codepath—both can run, but the new code is guarded by a flag. If the new code works, you can remove the old codepath and launch the feature fully in a subsequent release. If there's a problem, the flag value can be updated independently from the binary release via a dynamic config update.

In the old world of binary releases, we had to time press releases closely with our binary rollouts. We had to have a successful rollout before a press release about new functionality or a new feature could be issued. This meant that the feature would be out in the wild before it was announced and the risk of it being discovered ahead of time was very real.

This is where the beauty of the flag guard comes to play. If the new code has a flag, the flag can be updated to turn your feature on immediately before the

press release, thus minimizing the risk of leaking a feature. Note that flag-guarded code is not a *perfect* safety net for truly sensitive features. Code can still be scraped and analyzed if it's not well obfuscated, and not all features can be hidden behind flags without adding a lot of complexity. Moreover, even flag configuration changes must be rolled out with care. Turning on a flag for 100% of your users all at once is not a great idea, so a configuration service that manages safe configuration rollouts is a good investment, Nevertheless, the level of control and the ability to decouple the destiny of a particular feature from the overall product release are powerful levers for long-term sustainability of the application.

Striving for Agility: Setting Up a Release Train

Google's Search binary is its first and oldest. Large and complicated, its codebase can be tied back to Google's origin—a search through our codebase can still find code written at least as far back as 2003, often earlier. When smartphones began to take off, feature after mobile feature was shoehorned into a hairball of code written primarily for server deployment. Even though the Search experience was becoming more vibrant and interactive, deploying a viable build became more and more difficult. At one point, we were releasing the Search binary into production only once per week, and even hitting that target was rare and often based on luck.

When one of our contributing authors, Sheri Shipe, took on the project of increasing our release velocity in Search, each release cycle was taking a group of engineers days to complete. They built the binary, integrated data, and then began testing. Each bug had to be manually triaged to make sure it wouldn't impact Search quality, the user experience (UX), and/or revenue. This process was grueling and time consuming and did not scale to the volume or rate of change. As a result, a developer could never know when their feature was going to be released into production. This made timing press releases and public launches challenging.

Releases don't happen in a vacuum, and having reliable releases make the dependent factors easier to synchronize. Over the course of several years, a dedicated group of engineers implemented a continuous release process, which streamlined everything about sending a Search binary into the world. We automated what we could, set deadlines for submitting features, and simplified the integration of plug-ins and data into the binary. We could now consistently release a new Search binary into production every other day.

What were the trade-offs we made to get predictability in our release cycle? They narrow down to two main ideas we baked into the system.

No Binary Is Perfect

The first is that *no binary is perfect*, especially for builds that are incorporating the work of tens or hundreds of developers independently developing dozens of major features. Even though it's impossible to fix every bug, we constantly need to weigh questions such as: If a line has been moved two pixels to the left, will it affect an ad display and potential revenue? What if the shade of a box has been altered slightly? Will it make it difficult for visually impaired users to read the text? The rest of this book is arguably about minimizing the set of unintended outcomes for a release, but in the end we must admit that software is fundamentally complex. There is no perfect binary—decisions and trade-offs have to be made every time a new change is released into production. Key performance indicator metrics with clear thresholds allow features to launch even if they aren't perfect,[2] and can also create clarity in otherwise contentious launch decisions.

One bug involved a rare dialect spoken on only one island in the Philippines. If a user asked a search question in this dialect, instead of an answer to their question, they would get a blank web page. We had to determine whether the cost of fixing this bug was worth delaying the release of a major new feature.

We ran from office to office trying to determine how many people actually spoke this language, if it happened every time a user searched in this language, and whether these folks even used Google on a regular basis. Every quality engineer we spoke with deferred us to a more senior person. Finally, data in hand, we put the question to Search's senior vice president. Should we delay a critical release to fix a bug that affected only a very small Philippine island? It turns out that no matter how small your island, you should get reliable and accurate search results: we delayed the release and fixed the bug.

Meet Your Release Deadline

The second idea is that *if you're late for the release train, it will leave without you*. There's something to be said for the adage, "deadlines are certain, life is not." At some point in the release timeline, you must put a stake in the ground and turn away developers and their new features. Generally speaking, no amount of pleading or begging will get a feature into today's release after the deadline has passed.

There is the *rare* exception. The situation usually goes like this. It's late Friday evening and six software engineers come storming into the release manager's cube in a panic. They have a contract with the NBA and finished the feature moments ago. But it must go live before the big game tomorrow. The release must stop and we must cherry-pick the feature into the binary or we'll be in breach of contract! A bleary-eyed release engineer shakes their head and says it will take four hours to cut and test a new binary. It's their kid's birthday and they still need to pick up the balloons.

A world of regular releases means that if a developer misses the release train, they'll be able to catch the next train in a matter of hours rather than days. This limits developer panic and greatly improves work–life balance for release engineers.

Quality and User-Focus: Ship Only What Gets Used

Bloat is an unfortunate side effect of most software development life cycles, and the more successful a product becomes, the more bloated its code base typically becomes. One downside of a speedy, efficient release train is that this bloat is often magnified and can manifest in challenges to the product team and even to the users. Especially if the software is delivered to the client, as in the case of mobile apps, this can mean the user's device pays the cost in terms of space, download, and data costs, even for features they never use, whereas developers pay the cost of slower builds, complex deployments and rare bugs. In this section, we'll talk about how dynamic deployments allow you to ship only what is used, forcing necessary trade-offs between user value and feature cost. At Google, this often means staffing dedicated teams to improve the efficiency of the product on an ongoing basis.

Whereas some products are web-based and run on the cloud, many are client applications that use shared resources on a user's device—a phone or tablet. This choice in itself showcases a trade-off between native apps that can be more performant and resilient to spotty connectivity, but also more difficult to update and more susceptible to platform-level issues. A common argument against frequent, continuous deployment for native apps is that users dislike frequent updates and must pay for the data cost and the disruption. There might be other limiting factors such as access to a network, or a limit to the reboots required to percolate an update.

Even though there is a trade-off to be made in terms of how frequently to update a product, the goal is to *have these choices be intentional*. With a smooth, well-running CD process, how often a viable release is *created* can be separated from how often a user *receives* it. You might achieve the goal of being able to deploy weekly, daily or hourly, without actually doing so, and you should intentionally choose release processes in the context of your users' specific needs and the larger organizational goals, and determine the staffing and tooling model that will best support the long-term sustainability of your product.

Earlier in the chapter, we talked about keeping your code modular. This allows for dynamic, configurable deployments that allow better utilization of constrained resources, such as the space on a user's device. In the absence of this practice, every user must receive code they will never use to support translations they don't need or architectures that were meant for other kinds of devices. Dynamic deployments allow apps to maintain small sizes while only shipping code to a device that brings its users value, and A/B experiments allow for intentional trade-offs between a feature's cost and its value to users and your business.

There is an upfront cost to setting up these processes, and identifying and removing frictions that keep the frequency of releases lower than is desirable is a painstaking process. But the long-term wins in terms of risk management, developer velocity, and enabling rapid innovation are so high that these initial costs become worthwhile.

Shifting Left: Making Data-Driven Decisions Earlier

If you're building for all users, you might have clients on smart screens, speakers, Android and iOS phones and tablets, and your software may be flexible enough to allow users to customize their experience. Even if you're building for only Android devices, the sheer diversity of the more-than two billion Android devices can make the prospect of qualifying a release overwhelming. And with the pace of innovation, by the time someone reads this chapter, whole new categories of devices might have bloomed.

One of our release managers shared a piece of wisdom that turned the situation around when he said that the diversity of our client market was not a *problem*, but a *fact*. After we accepted that, we could switch our release qualification model in the following ways:

- If *comprehensive* testing is practically infeasible, aim for *representative* testing, instead.
- Staged rollouts to slowly increasing percentages of the userbase allow for fast fixes.
- Automated A/B releases allow for statistically significant results proving a release's quality, without tired humans needing to look at dashboards and make decisions.

When it comes to developing for Android clients, Google apps use specialized testing tracks and staged rollouts to an increasing percentage of user traffic, carefully monitoring for issues in these channels. Because the Play Store offers unlimited testing tracks, we can also set up a QA team in each country in which we plan to launch, allowing for a global overnight turnaround in testing key features.

One issue we noticed when doing deployments to Android was that we could expect a statistically significant change in user metrics *simply from pushing an update*. This meant that even if we made no changes to our product, pushing an update could affect device and user behavior in ways that were difficult to predict. As a result, although canarying the update to a small percentage of user traffic could give us good information about crashes or stability problems, it told us very little about whether the newer version of our app was in fact better than the older one.

Dan Siroker and Pete Koomen have already discussed the value of A/B testing[3] your features, but at Google, some of our larger apps also A/B test their *deployments*. This means sending out two versions of the product, one that is the desired update, with the baseline being a placebo (your old version just gets shipped again). As the two versions roll out simultaneously to a large enough base of similar users, you can compare one release against the other to see whether the latest version of your software is in fact an improvement over the previous one. With a large enough userbase, you should be able to get statistically significant results within days, or even hours. An automated metrics pipeline can enable the fastest possible release by pushing forward a release to more traffic as soon as there is enough data to know that the guardrail metrics will not be affected.

Obviously, this method does not apply to every app and can be a lot of overhead when you don't have a large enough userbase. In these cases, the recommended best practice is to aim for change-neutral releases. All new features are flag guarded so that the only change being tested during a rollout is the stability of the deployment itself.

Changing Team Culture: Building Discipline into Deployment

Although "Always Be Deploying" helps address several issues affecting developer velocity, there are also certain practices that address issues of scale. The initial team launching a product can be fewer than 10 people, each taking turns at deployment and production-monitoring responsibilities. Over time, your team might grow to hundreds of people, with subteams responsible for specific features. As this happens and the organization scales up, the number of changes in each deployment and the amount of risk in each release attempt is increasing superlinearly. Each release contains months of sweat and tears. Making the release successful becomes a high-touch and labor-intensive effort. Developers can often be caught trying to decide which is worse: abandoning a release that contains a quarter's worth of new features and bug fixes, or pushing out a release without confidence in its quality.

At scale, increased complexity usually manifests as increased release latency. Even if you release every day, a release can take a week or longer to fully roll out safely, leaving you a week behind when trying to debug any issues. This is where "Always Be Deploying" can return a development project to effective form. Frequent release trains allow for minimal divergence from a known good position, with the recency of changes aiding in resolving issues. But how can a team ensure that the complexity inherent with a large and quickly expanding code base doesn't weigh down progress?

On Google Maps, we take the perspective that features are very important, but only very seldom is any feature so important that a release should be held for it. If releases are frequent, the pain a feature feels for missing a release is small in comparison to the pain all the new features in a release feel for a delay, and especially the pain users can feel if a not quite ready feature is rushed to be included.

One release responsibility is to protect the product from the developers.

When making trade-offs, the passion and urgency a developer feels about launching a new feature can never trump the user experience with an existing product. This means that new features must be isolated from other components via interfaces with strong contracts, separation of concerns, rigorous testing, communication early and often, and conventions for new feature acceptance.

Conclusion

Over the years, and across all of our software products, we've found that, counterintuitively, faster is safer. The health of your product and the speed of development are not actually in opposition to each other, and products that release more frequently and in small batches have better quality outcomes. They adapt faster to bugs encountered in the wild and to unexpected market shifts. Not only that, faster is *cheaper*, because having a predictable, frequent release train forces you to drive down the cost of each release and makes the cost of any abandoned release very low.

Simply having the structures in place that *enable* continuous deployment generates the majority of the value, *even if you don't actually push those releases out to users*. What do we mean? We don't actually release a wildly different version of Search, Maps, or YouTube every day, but to be able to do so requires a robust, well-documented continuous deployment process, accurate and real-time metrics on user satisfaction and product health, and a coordinated team with clear policies on what makes it in or out and why. In practice, getting this right often also requires binaries that can be configured in production, configuration managed like code (in version control), and a toolchain that allow safety measures like dry-run verification, rollback/rollforward mechanisms, and reliable patching.

TL;DRs

- *Velocity is a team sport*: The optimal workflow for a large team that develops code collaboratively requires modularity of architecture and near-continuous integration.
- *Evaluate changes in isolation*: Flag-guard any features to be able to isolate problems early.
- *Make reality your benchmark*: Use a staged rollout to address device diversity and the breadth of the userbase. Release qualification in a synthetic environment that isn't similar to the production environment can lead to late surprises.
- *Ship only what gets used*: Monitor the cost and value of any feature in the wild to know whether it's still relevant and delivering sufficient user value.
- *Shift left*: Enable faster, more data-driven decision making earlier on all changes through CI and continuous deployment.
- *Faster is safer*: ship early and often and in small batches to reduce the risk of each release and to minimize time to market.

1 *https://martinfowler.com/bliki/ContinuousDelivery.html*

2 Remember the SRE "error-budget" formulation: perfection is rarely the best goal. Understand how much room for error is acceptable, how much of that budget has been spent recently, and use that to adjust the trade-off between velocity and stability.

3 Siroker, Dan, and Pete Koomen. (2013). *A/B Testing: The Most Powerful Way to Turn Clicks Into Customers*. Wiley.

Chapter 25. Compute as a Service

Written by Onufry Wojtaszczyk

Edited by Lisa Carey

I don't try to understand computers. I try to understand the programs

Barbara Liskov

After doing the hard work of writing code, you need some hardware to run it. Thus, you go to buy or rent that hardware. This, in essence, is *Compute as a Service* (CaaS) in which "Compute" is a shorthand for the computing power needed to actually run your programs.

This chapter is about how this simple concept—just give me the hardware to run my stuff[1]—maps into a system that will survive and scale as your organization evolves and grows. It is somewhat long because the topic is complex, and divided into four sections:

- "Taming the Compute Environment" covers how Google arrived at its solution for this problem and explains some of the key concepts of CaaS.
- "Writing Software for Managed Compute" shows how a managed compute solution affects how engineers write software. We believe that the "cattle, not pets"/flexible scheduling model has been fundamental to Google's success in the past 15 years and is an important tool in a software engineer's toolbox.
- "CaaS over time and scale" goes deeper into a few lessons Google learned about how various choices about a compute architecture play out as the organization grows and evolves.
- Finally, "Choosing a Compute Service" is dedicated primarily at those engineers who will make a decision about what compute service to use in their organization.

Taming the Compute Environment

Google's internal Borg system[2] was a precursor for many of today's CaaS architectures (like Kubernetes or Mesos). To better understand how the

particular aspects of such a service answer the needs of a growing and evolving organization, we'll trace the evolution of Borg and the efforts of Google engineers to tame the compute environment.

Automation of Toil

Imagine being a student at the university around the turn of the century. If you wanted to deploy some new, beautiful code, you'd SFTP the code onto one of the machines in the university's computer lab, SSH into the machine, compile and run the code. This is a tempting solution in its simplicity, but it runs into considerable issues over time and at scale. However, because that's roughly what many projects begin with, multiple organizations end up with processes that are somewhat streamlined evolutions of this system, at least for some tasks—the number of machines grows (so you SFTP and SSH into many of them), but the underlying technology remains. For example, in 2002 Jeff Dean, one of Google's most senior engineers, wrote the following about running an automated data-processing task as a part of the release process:

[Running the task] is a logistical, time-consuming nightmare. It currently requires getting a list of 50+ machines, starting up a process on each of these 50+ machines, and monitoring its progress on each of the 50+ machines. There is no support for automatically migrating the computation to another machine if one of the machines dies, and monitoring the progress of the jobs is done in an ad hoc manner [...] Furthermore, since processes can interfere with each other, there is a complicated, human-implemented "sign up" file to throttle the use of machines, which results in less-than-optimal scheduling, and increased contention for the scarce machine resources

This was an early trigger in Google's efforts to tame the compute environment, which explains well how the naive solution becomes unmaintainable at larger scale.

SIMPLE AUTOMATIONS

There are simple things that an organization can do to mitigate some of the pain. The process of deploying a binary onto each of the 50-plus machines and starting it there can easily be automated through a shell script; and then—if this is to be a reusable solution—through a more robust piece of code in an easier-to-maintain language that will perform the deployment in parallel (especially since the "50+" is likely to grow over time).

More interestingly, the monitoring of each machine can also be automated. Initially, the person in charge of the process would like to know (and be able

to intervene) if something went wrong with one of the replicas. This means exporting some monitoring metrics (like "the process is alive" and "number of documents processed") from the process—by having it write to a shared storage, or call out to a monitoring service, where they can see anomalies at a glance. Current open source solutions in that space are, for instance, setting up a dashboard in a monitoring tool like Graphana or Prometheus.

If an anomaly is detected, the usual mitigation strategy is to SSH into the machine, kill the process (if it's still alive) and start it again. This is tedious, possibly error prone (be sure you connect to the right machine, and be sure to kill the right process) and could be automated:

- Instead of manually monitoring for failures, one can use an agent on the machine that detects anomalies (like "the process did not report it's alive for the past five minutes" or "the process did not process any documents over the past 10 minutes"), and kills the process if an anomaly is detected.
- Instead of logging in to the machine to start the process again after death, it might be enough to wrap the whole execution in a "`while true; do run && break; done`" shell script.

The cloud world equivalent is setting an autohealing policy (to kill and re-create a VM or container after it fails a health check).

These relatively simple improvements address a part of Jeff Dean's problem described earlier, but not all of it; human-implemented throttling, and moving to a new machine, require more involved solutions.

AUTOMATED SCHEDULING

The natural next step is to automate machine assignment. This requires the first real "service," that will eventually grow into "Compute as a Service." That is, to automate scheduling, we need a central service that knows the complete list of machines available to it and can—on demand—pick a number of unoccupied machines and automatically deploy your binary to those machines. This eliminates the need for a hand-maintained "sign-up" file, instead delegating the maintenance of the list of machines to computers. This system is strongly reminiscent of earlier time-sharing architectures.

A natural extension of this idea is to combine this scheduling with reaction to machine failure. By scanning machine logs for expressions that signal bad health (e.g., mass disk read errors), we can identify machines that are broken, signal (to humans) the need to repair such machines, and avoid scheduling

any work onto those machines in the meantime. Extending the elimination of toil further, automation can try some fixes first before involving a human, like rebooting the machine, with the hope that whatever was wrong goes away, or running an automated disk scan.

One last complaint from Jeff's quote is the need for a human to migrate the computation to another machine if the machine it's running on breaks. The solution here is simple: because we already have scheduling automation, and the capability to detect that a machine is broken, we can simply have the scheduler allocate a new machine and restart the work on this new machine, abandoning the old one. The signal to do this might come from the machine introspection daemon or from monitoring of the individual process.

All of these improvements systematically deal with the growing scale of the organization. When the fleet was a single machine, SFTP and SSH were perfect solutions, but at the scale of hundreds or thousands of machines, automation needs to take over. The quote we started from came from a 2002 design document for the "Global WorkQueue," an early CaaS internal solution for some workloads at Google.

Containerization and Multitenancy

So far, we implicitly assumed a one-to-one mapping between machines and the programs running on them. This is highly inefficient in terms of computing resource (RAM, CPU) consumption, in many ways:

- It's very likely to have many more different types of jobs (with different resource requirements) than types of machines (with different resource availability), so many jobs will need to use the same machine type (which will need to be provisioned for the largest of them).
- Machines take a long time to deploy, whereas program resource needs grow over time. If obtaining new, larger machines takes your organization months, you need to also make them large enough to accommodate expected growth of resource needs over the time needed to provision new ones, which leads to waste, as new machines are not utilized to their full capacity.[3]
- Even when the new machines arrive, you still have the old ones (and it's likely wasteful to throw them away), and so you must manage a heterogeneous fleet that does not adapt itself to your needs.

The natural solution is to specify, for each program, its resource requirements (in terms of CPU, RAM, disk space), and then ask the scheduler to bin-pack replicas of the program onto the available pool of machines.

MY NEIGHBOR'S DOG BARKS IN MY RAM

The aforementioned solution works perfectly if everybody plays nicely. However, if I specify in my configuration that each replica of my data-processing pipeline will consume one CPU and 200 MB of RAM, and then—due to a bug, or organic growth—it starts consuming more, the machines it gets scheduled onto will run out of resources. In the CPU case, this will cause neighboring serving jobs to experience latency blips; in the RAM case, it will either cause out-of-memory kills by the kernel or horrible latency due to disk swap.[4]

Two programs on the same computer can interact badly in other ways, as well. Many programs will want their dependencies installed on a machine, in some specific version—and these might collide with the version requirements of some other program. A program might expect certain system-wide resources (think about `/tmp`) to be available for its own exclusive use. Security is an issue—a program might be handling sensitive data, and need to be sure that other programs on the same machine cannot access it.

Thus, a multitenant compute service must provide a degree of *isolation*, a guarantee of some sort that a process will be able to safely proceed without being disturbed by the other tenants of the machine.

A classical solution to isolation is the use of virtual machines (VMs). These, however, come with significant overhead[5] in terms of resource usage (they need the resources to run a full operating system inside) and startup time (again, they need to boot up a full operating system). This makes them a less-than-perfect solution for batch job containerization for which small resource footprints and short runtimes are expected. This led Google's engineers designing Borg in 2003 to look to different solutions, ending up with *containers*—a lightweight mechanism based on cgroups (contributed by Google engineers into the Linux kernel in 2007) and chroot jails, bind mounts and/or union/overlay filesystems for filesystem isolation. Open source container implementations include Docker and LMCTFY.

Over time, and the evolution of the organization, more and more potential isolation failures are discovered. To give a specific example, in 2011 engineers working on Borg discovered that the exhaustion of the process ID space (which was set by default to 32,000 PIDs) was becoming an isolation

failure, and limits on the total number of processes/threads a single replica can spawn had to be introduced. We look at this example in more detail later in this chapter.

RIGHTSIZING AND AUTOSCALING

The Borg of 2006 scheduled work based on the parameters provided by the engineer in the configuration, such as the number of replicas and the resource requirements.

Looking at the problem from a distance, the idea of asking humans to determine the resource requirement numbers is somewhat flawed: these are not numbers that humans interact with daily. And so, these configuration parameters become themselves, over time, a source of inefficiency. Engineers need to spend time determining them upon initial service launch, and as your organization accumulates more and more services, the cost to determine them scales up. Moreover, as time passes, the program evolves (likely grows), but the configuration parameters do not keep up. This ends in an outage—where it turns out that over time the new releases had resource requirements that ate into the slack left for unexpected spikes or outages; and when such a spike or outage actually occurs, the slack remaining turns out to be insufficient.

The natural solution is to automate the setting of these parameters. Unfortunately, this proves surprisingly tricky to do well. As an example, Google has only recently reached a point at which more than half of the resource usage over the whole Borg fleet is determined by rightsizing automation. That said, even though it is only half of the usage, it is a larger fraction of configurations, which means that the majority of engineers do not need to concern themselves with the tedious and error-prone burden of sizing their containers. We view this as a successful application of the idea that "easy things should be easy, and complex things should be possible"—just because some fraction of Borg workloads is too complex to be properly managed by rightsizing doesn't mean there isn't great value in handling the easy cases.

Summary

As your organization grows, and your products become more popular, you will grow in all of these axes:

- Number of different applications to be managed
- Number of copies of an application that needs to run

- The size of the largest application

To effectively manage scale, automation is needed that will enable you to address all these growth axes. You should, over time, expect the automation itself to become more involved, both to handle new types of requirements (for instance, scheduling for GPUs and TPUs is a major change in Borg that happened over the past 10 years) and increased scale. Actions that, at a smaller scale, could be manual, will need to be automated to avoid a collapse of the organization under the load.

One example, a transition that Google is still in the process of figuring out, is automating the management of our *datacenters*. Ten years ago, each datacenter was a separate entity. We manually managed them. Turning a datacenter up was an involved manual process, requiring a specialized skill set, that took weeks (from the moment when all the machines are ready), and was inherently risky. However, the growth of the number of datacenters Google manages meant that we move toward a model in which turning up a datacenter is an automated process that does not require human intervention.

Writing Software for Managed Compute

The move from a world of hand-managed lists of machines to the automated scheduling and rightsizing made management of the fleet much easier for Google, but it also took profound changes to the way we write and think about software.

Architecting for Failure

Imagine an engineer is to process a batch of one million documents and validate their correctness. If processing a single document takes one second, the entire job would take one machine roughly 12 days—which is probably too long. So, we shard the work across 200 machines, which reduces the runtime to a much more manageable 100 minutes.

As discussed in "Automated scheduling", in the Borg world the scheduler can unilaterally kill one of the 200 workers and move it to a different machine.[6] The "move it to a different machine" part implies that a new instance of your worker can be stamped out automatically, without the need for a human to SSH into the machine and tune some environment variables or install packages.

The move from "the engineer has to manually monitor each of the 100 tasks, and attend to them if broken" to "if something goes wrong with one of the tasks, the system is architected so that the load is picked up by others, while the automated scheduler kills it and reinstantiates it on a new machine" has been described many years later through the analogy of "pets versus cattle."[7]

If your server is a pet, when it's broken, a human comes to look at it (usually in a panic), understand what went wrong, and hopefully nurse it back to health. It's difficult to replace. If your servers are cattle, you name them replica001 to replica100, and if one fails, automation will remove it and provision a new one in its place. The distinguishing characteristic of "cattle" is that it's easy to stamp out a new instance of the job in question—it doesn't require manual setup and can be done fully automatically. This allows for the self-healing property described earlier—in the case of a failure, automation can take over and replace the unhealthy job with a new, healthy one without human intervention. Note that although the original metaphor spoke of servers (VMs), the same applies to containers: if you can stamp out a new version of the container from an image without human intervention, your automation will be able to autoheal your service when required.

If your servers are pets, your maintenance burden will grow linearly, or even superlinearly, with the size of your fleet, and that's a burden that no organization should accept lightly. On the other hand, if your servers are cattle, your system will be able to return to a stable state after a failure, and you will not need to spend your weekend nursing a pet server or container back to health.

Having your VMs or containers be cattle is not enough to guarantee that your system will behave well in the face of failure, though. With 200 machines, one of the replicas being killed by Borg is quite likely to happen, possibly more than once, and each time it extends the overall duration by 50 minutes (or however much processing time was lost). To deal with this gracefully, the architecture of the processing needs to be different: instead of statically assigning the work, we instead divide the entire set of one million documents into, say, 1,000 chunks of 1,000 documents each. Whenever a worker is finished with a particular chunk, it reports the results, and picks up another. This means that we lose at most one chunk of work on a worker failure, in the case when the worker dies after finishing the chunk, but before reporting it. This, fortunately, fits very well with the data-processing architecture that was Google's standard at that time: work isn't assigned equally to the set of workers at the start of the computation, it's dynamically assigned during the overall processing in order to account for workers that fail.

Similarly, for systems serving user traffic, you would ideally want a container being rescheduled not resulting in errors being served to your users. The Borg scheduler, when it plans to reschedule a container for maintenance reasons, signals its intent to the container to give it notice ahead of time. The container can react to this by refusing new requests while still having the time to finish the requests it has ongoing. This, in turn, requires the load-balancer system to understand the "I cannot accept new requests" response (and redirect traffic to other replicas).

To summarize: treating your containers or servers as cattle means that your service can get back to a healthy state automatically, but additional effort is needed to make sure that it can function smoothly while experiencing a moderate rate of failures.

Batch versus Serving

The Global WorkQueue (which we described in the first section of this chapter) addressed the problem of what Google engineers call "batch jobs"—programs that are expected to complete some specific task (like data processing) and that run to completion. Canonical examples of batch jobs would be logs analysis or machine learning model learning. Batch jobs stood in contrast to "serving jobs"—programs that are expected to run indefinitely and serve incoming requests, the canonical example being the job that served actual user search queries from the prebuilt index.

These two types of jobs have (typically) different characteristics,[8] in particular:

- Batch jobs are primarily interested in throughput of processing. Serving jobs care about latency of serving a single request.
- Batch jobs are short lived (minutes, or at most hours). Serving jobs are typically long lived (by default only restarted with new releases).
- Because they're long lived, serving jobs are more likely to have longer startup times.

So far, most of our examples were about batch jobs. As we have seen, to adapt a batch job to survive failures, we need to make sure that work is spread into small chunks and assigned dynamically to workers. The canonical framework for doing this at Google was MapReduce,[9] later replaced by Flume.[10]

Serving jobs are, in many ways, more naturally suited to failure resistance than batch jobs. Their work is naturally chunked into small pieces (individual user requests) that are assigned dynamically to workers—the strategy of handling a large stream of requests through load balancing across a cluster of servers has been used since the early days of serving internet traffic.

However, there are also multiple serving applications that do not naturally fit that pattern. The canonical example would be any server that you intuitively describe as a "leader" of a particular system. Such a server will typically maintain the state of the system (in memory or on its local filesystem), and if the machine it is running on goes down, a newly created instance will typically be unable to re-create the system's state. Another example is when you have large amounts of data to serve—more than fits on one machine—and so you decide to shard the data among, for instance, 100 servers, each holding 1% of the data, and handling requests for that part of the data. This is similar to statically assigning work to batch job workers; if one of the servers goes down, you (temporarily) lose the ability to serve a part of your data. A final example is if your server is known to other parts of your system by its hostname. In that case, regardless of how your server is structured, if this specific host loses network connectivity, other parts of your system will be unable to contact it.[11]

Managing State

NOTE

Note that, besides distributed state, there are other requirements to setting up an effective "servers as cattle" solution, like a discovery and load-balancing systems (so that your application, which moves around the datacenter, can be accessed effectively). Because this book is less about building a full CaaS infrastructure, and more how such an infrastructure relates to the art of software engineering, we won't go into more detail here.

One common theme in the previous description focused on *state* as a source of issues when trying to treat jobs like cattle. Whenever you replace one of your cattle jobs, you lose all the in-process state (as well as everything that was on local storage, if the job is moved to a different machine). This means that the in-process state should be treated as transient, whereas "real storage" needs to occur elsewhere.

The simplest way of dealing with this is extracting all storage to an external storage system. This means that anything that should survive past the scope of serving a single request (in the serving job case) or processing one chunk

of data (in the batch case) needs to be stored off machine, in durable, persistent storage. If all your local state is immutable, making your application failure resistant should be relatively painless.

Unfortunately, most applications are not that simple. One natural question that might come to mind is, "How are these durable, persistent storage solutions implemented—are *they* cattle?" The answer should be "yes." Persistent state can be managed by cattle through state replication. On a different level, RAID arrays are an analogous concept; we treat disks as transient (accept the fact one of them can be gone) while still maintaining state. In the servers world, this might be realized through multiple replicas holding a single piece of data and synchronizing to make sure every piece of data is replicated a sufficient number of times (usually 3 to 5). Note that setting this up correctly is difficult (some way of consensus handling is needed to deal with writes), and so Google developed a number of specialized storage solutions[12] that were enablers for most applications adopting a model where all state is transient.

Other types of local storage that cattle can use covers "re-creatable" data that is held locally to improve serving latency. Caching is the most obvious example here: a cache is nothing more than transient local storage that holds state in a transient location, but banks on the state not going away all the time, which allows for better performance characteristics on average. A key lesson for Google production infrastructure has been to provision the cache to meet your latency goals, but provision the core application for the total load. this has allowed us to avoid outages when the cache layer was lost because the noncached path was provisioned to handle the total load (although with higher latency). However, there is a clear trade-off here: how much to spend on the redundancy to mitigate the risk of an outage when cache capacity is lost.

In a similar vein to caching, data might be pulled in from external storage to local in the warm-up of an application, in order to improve request serving latency.

One more case of using local storage—this time in case of data that's written more than read—is batching writes. This is a common strategy for monitoring data (think, for instance, about gathering CPU utilization statistics from the fleet for the purposes of guiding the autoscaling system), but it can be used anywhere where it is acceptable for a fraction of data to perish, either because we do not need 100% data coverage (this is the monitoring case), or because the data that perishes can be re-created (this is the case of a batch job

that processes data in chunks, and writes some output for each chunk). Note that in many cases even if a particular calculation has to take a long time, it can be split into smaller time windows by periodic checkpointing of state to persistent storage.

Connecting to a Service

As mentioned earlier, if anything in the system has the name of the host on which your program runs hardcoded (or even provided as a configuration parameter at startup), your program replicas are not cattle. However, to connect to your application, another application does need to get your address from somewhere. Where?

The answer is to have an extra layer of indirection; that is, other applications refer to your application by some identifier which is durable across restarts of the specific "backend" instances. That identifier can be resolved by another system that the scheduler writes to when it places your application on a particular machine. Now, to avoid distributed storage lookups on the critical path of making a request to your application, clients will likely look up the address that your app can be found on, and set up a connection, at startup time, and monitor it in the background. This is generally called *service discovery*, and many compute offerings have built-in or modular solutions. Most such solutions also include some form of load balancing, which reduces coupling to specific backends even more.

A repercussion of this model is that you will likely need to repeat your requests in some cases, because the server you are talking to might be taken down before it manages to answer.[13] Retrying requests is standard practice for network communication (e.g., mobile app to a server) because of network issues, but it might be less intuitive for things like a server communicating with its database. This makes it important to design the API of your servers in a way that handles such failures gracefully. For mutating requests, dealing with repeated requests is tricky. The property you want to guarantee is some variant of *idempotency*—that the result of issuing a request twice is the same as issuing it once. One useful tool to help with idempotency is client-assigned identifiers: if you are creating something (e.g., an order to deliver a pizza to a specific address), the order is assigned some identifier by the client; and if an order with that identifier was already recorded, the server assumes it's a repeated request and reports success (it might also validate that the parameters of the order match).

One more surprising thing that we saw happen is that sometimes the scheduler loses contact with a particular machine due to some network problem. It then decides that all of the work there is lost and reschedules it onto other machines—and then the machine comes back! Now we have two programs on two different machines, both thinking they are "replica072." The way for them to disambiguate is to check which one of them is referred to by the address resolution system (and the other one should terminate itself or be terminated); but it also is one more case for idempotency: two replicas performing the same work and serving the same role are another potential source of request duplication.

One-Off Code

Most of the previous discussion focused on production-quality jobs, either those serving user traffic, or data processing pipelines producing production data. However, the life of a software engineer also involves running one-off analyses, exploratory prototypes, custom data processing pipelines, and more. These need compute resources.

Often, the engineer's workstation is a satisfactory solution to the need for compute resources. If one want to, say, automate the skimming through the 1 GB of logs that a service produced over the last day to check whether a suspicious line A always occurs before the error line B, they can just download the logs, write a short Python script, and let it run for a minute or two.

But if they want to automate the skimming through 1 TB of logs that service produced over the last year (for a similar purpose), waiting for roughly a day for the results to come in is likely not acceptable. A compute service that allows the engineer to just run the analysis on a distributed environment in several minutes (utilizing a few hundred cores) means the difference between having the analysis now and having it tomorrow. For tasks that require iteration—for example, if I will need to refine the query after seeing the results—the difference may be between having it done in a day and not having it done at all.

One concern that arises at times with this approach is that allowing engineers to just run one-off jobs on the distributed environment risks them wasting resources. This is, of course, a trade-off, but one that should be made consciously. It's very unlikely that the cost of processing that the engineer runs is going to be more expensive than the engineer's time spent on writing the processing code. The exact trade-off values differ depending on an

organization's compute environment and how much it pays its engineers, but it's unlikely that a thousand core hours costs anything close to a day of engineering work. Compute resources, in that respect, are similar to markers, which we discussed in the opening of the book; there is a small savings opportunity for the company in instituting a process to acquire more compute resources, but this process is likely to cost much more in lost engineering opportunity and time than it saves.

That said, compute resources differ from markers in that it's easy to take waaay too many by accident. Although it's unlikely someone will carry off a thousand markers, it's totally possible someone will accidentally write a program that occupies a thousand machines without noticing.[14] The natural solution to this is instituting quotas for resource usage by individual engineers. An alternative used by Google is to observe that because we're running low-priority batch workloads effectively for free (see the section on multitenancy later on), we can provide engineers with almost unlimited quota for low-priority batch, which is good enough for most one-off engineering tasks.

CaaS over Time and Scale

We talked above how CaaS evolved at Google, and what are the basic parts needed to make it happen—how the simple mission of "just give me resources to run my stuff" translates to an actual architecture like Borg. Several aspects of how a CaaS architecture affects the life of software across time and scale deserve a closer look.

Containers as an abstraction

Containers, as we described them earlier, were shown primarily as an isolation mechanism, a way to enable multitenancy, while minimizing the interference between different tasks sharing a single machine. That was the initial motivation, at least in Google. But containers turned out to also serve a very important role in abstracting away the compute environment.

A container provides an abstraction boundary between the deployed software and the actual machine it's running on. This means that as—over time—the machine changes, it is only the container software (presumably managed by a single team) that has to be adapted, whereas the application software (managed by each individual team, as the organization grows) can remain unchanged.

Let's discuss two examples of how a containerized abstraction allows an organization to manage change.

A *filesystem abstraction* provides a way to incorporate software that was not written in the company without the need to manage custom machine configurations. This might be open source software an organization runs in its datacenter, or acquisitions that it wants to onboard onto its CaaS. Without a filesystem abstraction, onboarding a binary that expects a different filesystem layout (e.g., expecting a helper binary at */bin/foo/bar*) would require either modifying the base layout of all machines in the fleet, or fragmenting the fleet, or modifying the software (which might be difficult, or even impossible due to licence considerations).

Even though these solutions might be feasible if importing an external piece of software is something that happens once in a lifetime, it is not a sustainable solution if importing software becomes a common (or even only-somewhat-rare) practice.

A filesystem abstraction of some sort also helps with dependency management because it allows the software to predeclare and prepackage the dependencies (e.g., specific versions of libraries) that the software needs to run. Depending on the software installed on the machine presents a leaky abstraction that forces everybody to use the same version of precompiled libraries, and makes upgrading any component very difficult, if not impossible.

A container also provides a simple way to manage *named resources* on the machine. The canonical example is network ports; other named resources include specialized targets; for example, GPUs and other accelerators.

Google initially did not include network ports as a part of the container abstraction, and so binaries had to search for unused ports themselves. As a result, the "PickUnusedPortOrDie" function has more than 20,000 usages in the Google C++ codebase. Docker, which was built after Linux namespaces were introduced, uses namespaces to provide containers with a virtual-private NIC, which means that applications can listen on any port they want. The docker networking stack then maps a port on the machine to the in-container port. Kubernetes, which was originally built on top of Docker, goes one step further and requires the network implementation to treat containers ("pods" in Kubernetes parlance) as "real" IP addresses, available from the host network. Now every app can listen on any port they want without fear of conflicts.

These improvements are particularly important when dealing with software not designed to run on the particular compute stack. Although many popular open source programs have configuration parameters for which port to use, there is no consistency between them for how to configure this.

CONTAINERS AND IMPLICIT DEPENDENCIES

As with any abstraction, Hyrum's Law of implicit dependencies applies to the container abstraction. It probably applies *even more than usual*, both because of the huge number of users (at Google, all production software and much else will run on Borg) and because the users do not feel that they are using an API when using things like the filesystem (and are even less likely to think whether this API is stable, versioned, etc.).

To illustrate, let's return to the example of process ID space exhaustion that Borg experienced in 2011. You might wonder why are process IDs exhaustible. Are they not simply integer IDs that can be assigned from the 32-bit or 64-bit space? In Linux, they are in practice assigned in the range [0,..., PID_MAX - 1], where PID_MAX defaults to 32,000. PID_MAX, however, can be raised through a simple configuration change (to a considerably higher limit). Problem solved?

Well, no. By Hyrum's Law, the fact that the PIDs that processes running on Borg got were limited to the 0...32,000 range became an implicit API guarantee that people started depending on; for instance, log storage processes depended on the fact that the PID can be stored in five digits, and broke for six-digit PIDs, because record names exceeded the maximum allowed length. Dealing with the problem became a lengthy, two-phase project. First, a temporary upper bound on the number of PIDs a single container can use (so that a single thread-leaking job cannot render the whole machine unusable). Second, splitting the PID space for threads and processes (because it turned out very few users depended on the 32,000 guarantee for the PIDs assigned to threads, as opposed to processes. So, we could increase the limit for threads and keep it at 32,000 for processes). Phase three would be to introduce PID namespaces to Borg, giving each container its own complete PID space. Predictably (Hyrum's Law again), a multitude of systems ended up assuming that the triple {hostname, timestamp, pid} uniquely identifies a process, which would break if PID namespaces were introduced. The effort to identify all these places and fix them (and backport any relevant data) is still ongoing eight years later.

The point here is not that you should run your containers in PID namespaces. Although it's a good idea, it's not the interesting lesson here. When Borg's

containers were built, PID namespaces did not exist; and even if they did, it's unreasonable to expect engineers designing Borg in 2003 to recognize the value of introducing them. Even now there are certainly resources on a machine that are not sufficiently isolated, which will probably cause problems one day. This underlines the challenges of designing a container system that will prove maintainable over time and thus the value of using a container system developed and used by a broader community, where these types of issues have already occurred for others, and the lessons learned have been incorporated.

One Service to Rule Them All

As discussed earlier, the original WorkQueue design was targeted at only some batch jobs, which ended up all sharing a pool of machines managed by the WorkQueue, and a different architecture was used for serving jobs, with each particular serving job running in its own, dedicated pool of machines. The open source equivalent would be running a separate Kubernetes cluster for each type of workload (plus one pool for all the batch jobs).

In 2003, the Borg project was started, aiming (and eventually succeeding at) building a compute service that assimilates these disparate pools into one large pool. Borg's pool covered both serving and batch jobs, and became the only pool in any datacenter (the equivalent would be running a single large Kubernetes cluster for all workloads in each geographical location). There are two significant efficiency gains here worth discussing.

The first one is that serving machines became cattle (the way the Borg design doc put it: "*Machines are anonymous:* programs don't care which machine they run on as long as it has the right characteristics."). If every team managing a serving job must manage their own pool of machines (their own cluster), the same organizational overhead of maintaining and administering that pool is applied to every one of these teams. As time passes, the management practices of these pools will diverge over time, making company-wide changes (like moving to a new server architecture, or switching datacenters) more and more complex. A unified management infrastructure—that is, a *common* compute service for all the workloads in the organization—allows Google to avoid this linear scaling factor; there aren't *n* different management practices for the physical machines in the fleet, there's just Borg.[15]

The second one is more subtle and might not be applicable to every organization, but it was very relevant to Google. The distinct needs of batch

and serving jobs turn out to be complementary. Serving jobs usually need to be overprovisioned because they need to have capacity to serve user traffic without significant latency decreases even in the case of a usage spike or partial infrastructure outage. This means that a machine running only serving jobs will be underutilized. It's tempting to try to take advantage of that slack by overcommitting the machine, but that defeats the purpose of the slack in the first place because if the spike/outage does happen, the resources we need will not be available.

However, this reasoning applies only to serving jobs! If we have a number of serving jobs on a machine and these jobs are requesting RAM and CPU that sum up to the total size of the machine, no more serving jobs can be put in there, even if real utilization of resources is only 30% of capacity. But we *can* (and, in Borg, will) put batch jobs in the spare 70%, with the policy that if any of the serving jobs need the memory or CPU, we will reclaim it from the batch jobs (by freezing them in the case of CPU or killing in the case of RAM). Because the batch jobs are interested in throughput (measured in aggregate across hundreds of workers, not for individual tasks) and their individual replicas are cattle anyway, they will be more than happy to soak up this spare capacity of serving jobs.

Depending on the shape of the workloads in a given pool of machines, this means that either all of the batch workload is effectively running on free resources (because we are paying for them in the slack of serving jobs anyway) or all the serving workload is effectively paying for only what they use, not for the slack capacity they need for failure resistance (because the batch jobs are running in that slack). In Google's case, most of the time it turns out we run batch effectively for free.

MULTITENANCY FOR SERVING JOBS

Earlier, we discussed a number of requirements that a compute service must satisfy to be suitable for running serving jobs. As previously discussed, there are multiple advantages to having the serving jobs be managed by a common compute solution, but this also comes with challenges. One particular requirement worth repeating is a discovery service, discussed in "Connecting to a Service". There are a number of other requirements that are new when we want to extend the scope of a managed compute solution to serving tasks, for example:

- Rescheduling of jobs needs to be throttled: although it's probably acceptable to kill and restart 50% of a batch job's replicas (because it will cause only a temporary blip in processing, and what we really care

about is throughput), it's unlikely to be acceptable to kill and restart 50% of a serving job's replicas (because the remaining jobs are likely too few to be able to serve user traffic while waiting for the restarted jobs to come back up again).

- A batch job can usually be killed without warning. What we lose is some of the already performed processing, which can be redone. When a serving job is killed without warning, we likely risk some user-facing traffic returning errors or (at best) having increased latency; it is preferable to give several seconds of warning ahead of time so that the job can finish serving requests it has in flight and not accept new ones.

For the aforementioned efficiency reasons, Borg covers both batch and serving jobs, but multiple compute offerings split the two concepts—typically, a shared pool of machines for batch jobs, and dedicated, stable pools of machines for serving jobs. Regardless of whether the same compute architecture is used for both types of jobs, however, both groups benefit from being treated like cattle.

Submitted Configuration

The Borg scheduler receives the configuration of a replicated service or batch job to run in the cell as the contents of a Remote Procedure Call (RPC) call. It's possible for the operator of the service to manage it by using a command-line interface (CLI) that sends those RPCs, and have the parameters to the CLI stored in shared documentation, or in their head.

Depending on documentation and tribal knowledge over code submitted to a repository is rarely a good idea in general because both documentation and tribal knowledge have a tendency to deteriorate over time (see Chapter 3). However, the next natural step in the evolution—wrapping the execution of the CLI in a locally developed script—is still inferior to using a dedicated configuration language to specify the configuration of your service.

Over time, the runtime presence of a logical service will typically grow beyond a single set of replicated containers in one datacenter across many axes:

- It will spread its presence across multiple datacenters (both for user affinity and failure resistance).
- It will fork into having staging and development environments in addition to the production environment/configuration.

- It will accrue additional replicated containers of different types in the form of attached services, like a memcached accompanying the service.

Management of the service is much simplified if this complex setup can be expressed in a standardized configuration language that allows easy expression of standard operations (like "update my service to the new version of the binary, but taking down no more than 5% of capacity at any given time").

A standardized configuration language provides standard configuration that other teams can easily include in their service definition. As usual, we emphasize the value of such standard configuration over time and scale. If every team writes a different snippet of custom code to stand up their memcached service, it becomes very difficult to perform organization-wide tasks like swapping out to a new memcache implementation (e.g., for performance or licencing reasons) or to push a security update to all the memcache deployments. Also note that such a standardized configuration language is a requirement for automation in deployment (see XREF(Continuous Deployment)).

Choosing a Compute Service

It's unlikely any organization will go down the path that Google went, building its own compute architecture from scratch. These days, modern compute offerings are available both in the open source world (like Kubernetes or Mesos, or, at a different level of abstraction, OpenWhisk or Knative), or as public cloud managed offerings (again, at different levels of complexity, from things like Google Cloud Platform's Managed Instance Groups or Amazon Web Services Elastic Compute Cloud [Amazon EC2] autoscaling, to managed containers similar to Borg, like Microsoft Azure Kubernetes Service (AKS) or Google Kubernetes Engine (GKE), to a serverless offering like AWS Lambda or Google's Cloud Functions).

However, most organizations will *choose* a compute service, just as Google did internally. Note that a compute infrastructure has a high lock-in factor. One reason for that is because code will be written in a way that takes advantage of all the properties of the system (Hyrum's Law); thus, for instance, if you choose a VM-based offering, teams will tweak their particular VM images; and if you choose a specific container-based solution, teams will call out to the APIs of the cluster manager. If your architecture allows code to treat VMs (or containers) as pets, teams will do so, and then a

move to a solution that depends on them being treated like cattle (or even different forms of pets) will be difficult.

To show how even the smallest details of a compute solution can end up locked in, consider how Borg runs the command that the user provided in the configuration. In most cases, the command will be the execution of a binary (possibly followed by a number of arguments). However, for convenience, the authors of Borg also included the possibility of passing in a shell script; for example, `while true; do ./my_binary; done`.16 However, whereas a binary execution can be done through a simple fork-and-exec (which is what Borg does), the shell script needs to be run by a shell like Bash. So, Borg actually executed `/usr/bin/bash -c $USER_COMMAND`, which works in the case of a simple binary execution, as well.

At some point, the Borg team realized that at Google's scale, the resources—mostly memory—consumed by this Bash wrapper are non-negligible, and decided to move over to using a more lightweight shell: ash. So, the team made a change to the process runner code to run `/usr/bin/ash -c $USER_COMMAND`, instead.

You would think that this is not a risky change; after all, we control the environment, we know that both of these binaries exist, and so there should be no way this doesn't work. In reality, the way this didn't work is that the Borg engineers were not the first to notice the extra memory overhead of running Bash. Some teams were creative in their desire to limit memory usage, and replaced (in their custom filesystem overlay) theBash command with a custom-written piece of "execute the second argument" code. These teams, of course, were very aware of their memory usage, and so when the Borg team changed the process runner to use ash (which was not overwritten by the custom code), their memory usage increased (because it started including ash usage instead of the custom code usage), and this caused alerts, rolling back the change, and a certain amount of unhappiness.

Another reason that a compute service choice is difficult to change over time is that any compute service choice will eventually become surrounded by a large ecosystem of helper services—tools for logging, monitoring, debugging, alerting, visualization, on-the-fly analysis, configuration languages and meta-languages, user interfaces, and more. These tools would need to be rewritten as a part of a compute service change, and even understanding and enumerating those tools is likely to be a challenge for a medium or large organization

Thus, the choice of a compute architecture is important. As with most software engineering choices, this one involves trade-offs. Let's discuss a few.

Centralization versus Customization

From the point of view of management overhead of the compute stack (and also from the point of view of resource efficiency), the best an organization can do is adopt a single CaaS solution to manage its entire fleet of machines and use only the tools available there for everybody. This ensures that as the organization grows, the cost of managing the fleet remains manageable. This path is basically what Google has done with Borg.

NEED FOR CUSTOMIZATION

However, a growing organization will have increasingly diverse needs. For instance, when Google launched the Google Compute Engine (the "VM as a Service" public cloud offering) in 2012, the VMs, just as most everything else at Google, were managed by Borg. This means that each VM was running in a separate container controlled by Borg. However, the "cattle" approach to task management did not suit Cloud's workloads, because each particular container was actually a VM that some particular user was running, and Cloud's users did not, typically, treat the VMs as cattle.[17]

Reconciling this difference required considerable work on both sides. The Cloud organization made sure to support live migration of VMs; that is, the ability to take a VM running on one machine, spin up a copy of that VM on another machine, bring the copy to be a perfect image, and finally redirect all traffic to the copy, without causing a noticeable period when service is unavailable.[18] Borg, on the other hand, had to be adapted to avoid at-will killing of containers containing VMs (to provide the time to migrate the VM's contents to the new machine), and also, given that the whole migration process is more expensive, Borg's scheduling algorithms were adapted to optimize for decreasing the risk of rescheduling being needed.[19] Of course, these modifications were rolled out only for the machines running the cloud workloads, leading to a (small, but still noticeable) bifurcation of Google's internal compute offering.

A different example—but one that also leads to a bifurcation—comes from Search. Around 2011, one of the replicated containers serving Google Search web traffic had a giant index built up on local disks, storing the less-often-accessed part of the Google index of the web (the more common queries were served by in-memory caches from other containers). Building up this index

on a particular machine required the capacity of multiple hard drives and took several hours to fill in the data. However, at the time Borg assumed that if any of the disks that a particular container had data on had gone bad, the container will be unable to continue, and needs to be rescheduled to a different machine. This combination (along with the relatively high failure rate of spinning disks, compared to other hardware) caused severe availability problems; containers were taken down all the time and then took forever to start up again. To address this, Borg had to add the capability for a container to deal with disk failure by itself, opting out of Borg's default treatment; while the Search team had to adapt the process to continue operation with partial data loss.

Multiple other bifurcations, covering areas like filesystem shape, filesystem access, memory control, allocation and access, CPU/memory locality, special hardware, special scheduling constraints, and more, caused the API surface of Borg to become large and unwieldy, and the intersection of behaviors became difficult to predict, and even more difficult to test. Nobody really knew whether the expected thing happened if a container requested *both* the special Cloud treatment for eviction *and* the custom Search treatment for disk failure (and in many cases, it was not even obvious what "expected" means).

After 2012, the Borg team devoted significant time to cleaning up the API of Borg. It discovered some of the functionalities Borg offered were no longer used at all.[20] The more concerning group of functionalities were those that were used by multiple containers, but it was unclear whether intentionally—the process of copying the configuration files between projects led to proliferation of usage of features that were originally intended for power users only. Whitelisting was introduced for certain features to limit their spread, and clearly mark them as power-user-only. However, the cleanup is still ongoing, and some changes (like using labels for identifying groups of containers) are still not fully done.[21]

As usual with trade-offs, although there are ways to invest effort and get some of the benefits of customization while not suffering the worst downsides (like the aforementioned whitelisting for power functionality), in the end there are hard choices to be made. These choices usually take the form of multiple small questions: do we accept expanding the explicit (or worse, implicit) API surface to accommodate a particular user of our infrastructure, or do we significantly inconvenience that user, but maintain higher coherence?

Level of Abstraction: Serverless

The description of taming the compute environment by Google can easily be read as a tale of increasing and improving abstraction—the more advanced versions of Borg took care of more management responsibilities, and isolated the container more from the underlying environment. It's easy to get the impression this is a simple story: more abstraction is good; less abstraction is bad.

Of course, it is not that simple. The landscape here is complex, with multiple offerings. In "Taming the Compute Environment", we discussed the progression from dealing with pets running on bare-metal machines (either owned by your organization or rented from a colocation center) to managing containers as cattle. In between, as an alternative path, are VM-based offerings in which VMs can progress from being a more flexible substitute for bare metal (in Infrastructure as a Service offerings like Google Compute Engine [GCE] or Amazon EC2) to heavier substitutes for containers (with autoscaling, rightsizing, and other management tools).

In Google's experience, the choice of managing cattle (and not pets) is the solution to managing at scale. To reiterate, if each of your teams will need just one pet machine in each of your datacenters, your management costs will rise superlinearly with your organization's growth (because both the number of teams *and* the number of datacenters a team occupies are likely to grow). And after the choice to manage cattle is made, containers are a natural choice for management; they are lighter weight (implying smaller resource overheads and startup times) and configurable enough that should you need to provide specialized hardware access to a specific type of workload, you can (if you choose so) allow punching a hole through easily.

The advantage of VMs as cattle lies primarily in the ability to bring our own operating system, which matters if your workloads require a diverse set of operating systems to run. Multiple organizations will also have preexisting experience in managing VMs, and preexisting configurations and workloads based on VMs, and so might choose to use VMs instead of containers to ease migration costs.

WHAT IS SERVERLESS?

An even higher level of abstraction is *serverless* offerings[22]. Assume that an organization is serving web content, and is using (or willing to adopt) a common server framework for handling the HTTP requests and serving responses. The key defining trait of a framework is the inversion of control—

so, the user will only be responsible for writing an "Action" or "Handler" of some sort—a function in the chosen language that takes the request parameters, and returns the response.

In the Borg world, the way you run this code is that you stand up a replicated container, each replica containing a server consisting of framework code and your functions. If traffic increases, you will handle this by scaling up (adding replicas or expanding into new datacenters). If traffic decreases, you will scale down. Note that a minimal presence (Google usually assumes at least three replicas in each datacenter a server is running in) is required.

However, if multiple different teams are using the same framework, a different approach is possible: instead of just making the machines multitenant, we can also make the framework servers themselves multitenant. In this approach, we end up running a larger number of framework servers, dynamically load/unload the action code on different servers as needed, and dynamically direct requests to those servers that have the relevant action code loaded. Individual teams no longer run servers, hence "serverless."

Most discussions of serverless frameworks compare them to the "VMs as pets" model. In this context, the serverless concept is a true revolution, as it brings in all of the benefits of cattle management—autoscaling, lower overhead, lack of explicit provisioning of servers. However, as described earlier, the move to a shared, multitenant, cattle-based model should already be a goal for an organization planning to scale; and so the natural comparison point for serverless architectures should be "persistent containers" architecture like Borg, Kubernetes, or Mesosphere.

PROS AND CONS

First note that a serverless architecture requires your code to be *truly stateless*; it's unlikely we will be able to run your users' VMs or implement Spanner inside the serverless architecture. All the ways of managing local state (except not using it) that we talked about earlier do not apply. In the containerized world, you might spend a few seconds or minutes at startup setting up connections to other services, populating caches from cold storage, and so on and you expect that in the typical case you will be given a grace period before termination. In a serverless model, there is no local state that is really persisted across requests; everything that you want to use, you should set up in request-scope.

In practice, most organizations have needs that cannot be served by truly stateless workloads. This can either lead to depending on specific solutions

(either home grown or third party) for specific problems (like a managed database solution, which is a frequent companion to a public cloud serverless offering) or to having two solutions: a container-based one and a serverless one. It's worth mentioning that many or most serverless frameworks are built on top of other compute layers: AppEngine runs on Borg, Knative runs on Kubernetes, Lambda runs on Amazon EC2.

The managed serverless model is attractive for *adaptable scaling* of the resource cost, especially at the low-traffic end. In, say, Kubernetes, your replicated container cannot scale down to zero containers (because the assumption is that spinning up both a container and a node is too slow to be done at request serving time). This means that there is a minimum cost of just having an application available in the persistent cluster model. On the other hand, a serverless application can easily scale down to zero; and so the cost of just owning it scales with the traffic.

At the very high-traffic end, you will necessarily be limited by the underlying infrastructure, regardless of the compute solution. If your application needs to use 100,000 cores to serve its traffic, there needs to be 100,000 physical cores available in whatever physical equipment is backing the infrastructure you are using. At the somewhat lower end, where your application does have enough traffic to keep multiple servers busy but not enough to present problems to the infrastructure provider, both the persistent container solution and the serverless solution can scale to handle it, although the scaling of the serverless solution will be more reactive and more granular than that of the persistent container one.

Finally, adopting a serverless solution implies a certain loss of control over your environment. On some level, this is a good thing: having control means having to exercise it, and that means management overhead. But, of course, this also means that if you need some extra functionality that's not available in the framework you use, it will become a problem for you.

To take one specific instance of that, the Google Code Jam team (running a programming contest for thousands of participants, with a frontend running on Google AppEngine) had a custom-made script to hit the contest webpage with an artificial traffic spike several minutes before the contest start, in order to warm up enough instances of the app to serve the actual traffic that happened when the contest started. This worked, but it's the sort of hand-tweaking (and also hacking) that one would hope to get away from by choosing a serverless solution.

THE TRADE-OFF

Google's choice in this trade-off was not to invest heavily into serverless solutions. Google's persistent containers solution, Borg, is advanced enough to offer most of the serverless benefits (like autoscaling, various frameworks for different types of applications, deployment tools, unified logging and monitoring tools, and more). The one thing missing is the more aggressive scaling (in particular, the ability to scale down to zero), but the vast majority of Google's resource footprint comes from high-traffic services, and so it's comparably cheap to overprovision the small services. At the same time, Google runs multiple applications that would not work in the "truly stateless" world, from GCE, through home-grown database systems like BigQuery or Spanner, to servers that take a long time to populate the cache, like the aforementioned long-tail search serving jobs. Thus, the benefits of having one common unified architecture for all of these things outweigh the potential gains for having a separate serverless stack for a part of a part of the workloads.

However, Google's choice is not necessarily the correct choice for every organization: other organizations have successfully built out on mixed container/serverless architectures, or on purely serverless architectures utilizing third-party solutions for storage.

The main pull of serverless, however, comes not in the case of a large organization making the choice, but in the case of a smaller organization or team; in that case, the comparison is inherently unfair. The serverless model, though being more restrictive, allows the infrastructure vendor to pick up a much larger share of the overall management overhead and thus *decrease the management overhead* for the users. Running the code of one team on a shared serverless architecture, like AWS Lambda or Google's Cloud Run, is significantly simpler (and cheaper) than setting up a cluster to run the code on a managed container service like GKE or AKS if the cluster is not being shared among many teams. If your team wants to reap the benefits of a managed compute offering, but your larger organization is unwilling or unable to move to a persistent containers-based solution, a serverless offering by one of the public cloud providers is likely to be attractive to you because the cost (in resources and management) of a shared cluster amortizes well only if the cluster is truly shared (between multiple teams in the organization).

Note, however, that as your organization grows and adoption of managed technologies spreads, you are likely to outgrow the constraints of a purely

serverless solution. This makes solutions where a break-out path exists (like from KNative to Kubernetes) attractive given that they provide a natural path to a unified compute architecture like Google's, should your organization decide to go down that path.

Public versus Private

Back when Google was starting, the CaaS offerings were primarily homegrown; if you wanted one, you built it. Your only choice in the public-versus-private space was between owning the machines and renting them, but all the management of your fleet was up to you.

In the age of public cloud, there are cheaper options, but there are also more choices, and an organization will have to make them.

An organization using a public cloud is effectively outsourcing (a part of) the management overhead to a public cloud provider. For many organizations this is an attractive proposition - they can focus on providing value in their specific area of expertise, and do not need to grow significant infrastructure expertise. Although the cloud providers (of course) charge more than the bare cost of the metal to recoup the management expenses, they have the expertise already built up, and they are sharing it across multiple customers.

Additionally, a public cloud is a way to scale the infrastructure more easily. As the level of abstraction grows—rom colocations, through buying VM time, up to managed containers and serverless offerings—the ease of scaling up increases: from having to sign a rental agreement for colocation space, through the need to run a CLI to get a few more VMs, up to autoscaling tools for which your resource footprint changes automatically with the traffic you receive. Especially for young organizations or products, predicting resource requirements is challenging, and so the advantages of not having to provision resources up front are significant.

One significant concern when choosing a cloud provider is the fear of lock-in—the provider might suddenly increase their prices or maybe just fail, leaving an organization in a very difficult position. One of the first serverless offering providers, Zimki, a Platform as a Service environment for running JavaScript, shut down in 2007 with three months' notice.

A partial mitigation for this is to use public cloud solutions that run using an open source architecture (like Kubernetes). This is intended to make sure that a migration path exists, even if the particular infrastructure provider becomes unacceptable for some reason. Although this mitigates a significant part of

the risk, it is not a perfect strategy. Because of Hyrum's Law, it's difficult to guarantee no parts that are specific to a given provider will be used.

Two extensions of that strategy are possible. One is to use a lower-level public cloud solution (like Amazon EC2) and run a higher-level open source solution (like OpenWhisk or KNative) on top of it. This tries to ensure that if you want to migrate out, you can take whatever tweaks you did to the higher-level solution, tooling you built on top of it, and implicit dependencies you have along with you. The other is to run multicloud; that is, to use managed services based on the same open source solutions from two or more different cloud providers (say, GKE and AKS for Kubernetes). This provides an even easier path for migration out of one of them, and also makes it more difficult to depend on specific implementation details available in one one of them.

One more related strategy—less for managing lock-in, and more for managing migration—is to run in a hybrid cloud; that is, have a part of your overall workload on your private infrastructure, and part of it run on a public cloud provider. One of the ways this can be used is to use the public cloud as a way to deal with overflow. An organization can run most of its typical workload on a private cloud, but in case of resource shortage scale some of the workloads out to a public cloud. Again, to make this work effectively the same open source compute infrastructure solution needs to be used in both spaces.

Both multicloud and hybrid cloud strategies require the multiple environments to be connected well, through direct network connectivity between machines in different environments and common APIs that are available in both.

Conclusion

Over the course of building, refining, and running its compute infrastructure Google learned the value of a well-designed, common compute infrastructure. Having a single infrastructure for the entire organization (e.g., one or a small number of shared Kubernetes clusters per region) provides significant efficiency gains in management and resource costs and allows the development of shared tooling on top of that infrastructure. In the building of such an architecture, containers are a key tool, to allow sharing a physical (or virtual) machine between different tasks (leading to resource efficiency) as well as to provide an abstraction layer between the application and the operating system that provides resilience over time.

Utilizing a container-based architecture well requires designing applications to use the "cattle" model: engineering your application to consist of nodes that can be easily and automatically replaced allows scaling to thousands of instances. Writing software to be compatible with that model requires different thought patterns; for example, treating all local storage (including disk) as ephemeral and avoiding hardcoding hostnames.

That said, although Google has, overall, been both satisfied and successful with its choice of architecture, other organizations will choose from a wide range of compute services exists—from the "pets" model of hand-managed VMs or machines, through "cattle" replicated containers, to the abstract "serverless" model, all available in managed and open source flavors; your choice is a complex trade-off of many factors.

TL;DRs

- Scale requires a common infrastructure for running workloads in production.
- A compute solution can provide a standardized, stable abstraction and environment for software.
- Software needs to be adapted to a distributed, managed compute environment.
- The compute solution for an organization should be chosen thoughtfully to provide appropriate level of abstraction.

1 Disclaimer: for some applications, the "hardware to run it" is the hardware of your customers (think, for example, of a shrink-wrapped game you bought a decade ago). This presents very different challenges that we do not cover in this chapter.

2 See Verma, A. L. Pedros, M. Korupolu, D. Oppenheimer, E. Tune, J. Wilkes. (215). "Large-scale cluster management at Google with Borg." Proceedings of the European Conference on Computer Systems (EuroSys).

3 Note that this and the next point apply less if your organization is renting machines from a public cloud provider.

4 Google has chosen, long ago, that the latency degradation due to disk swap is so horrible that an out-of-memory kill and a migration to a different machine is universally preferable—so in Google's case, it's always an out-of-memory kill.

5 Although a considerable amount of research is going into decreasing this overhead, it will never be as low as a process running natively.

6 The scheduler does not do this arbitrarily, but for concrete reasons (like the need to update the kernel, or a disk going bad on the machine, or a reshuffle to make the overall distribution of workloads in the datacenter bin-packed better). However, the point of having a compute service is that as a software author I should neither know nor care why regarding the reasons this might happen.

7 The "cattle versus pets" metaphor is attributed by Randy Bias to Bill Baker (*http://cloudscaling.com/blog/cloud-computing/the-history-of-pets-vs-cattle/*), and it's become extremely popular as a way to describe the "replicated software unit" concept. As an analogy, it can also be used to describe other concepts than servers; for example, see Chapter 22.

8 Like all categorizations, this one isn't perfect; there are types of programs that don't fit neatly into any of the categories, or that possess characteristics typical of both serving and batch jobs. However, like most useful categorizations, it still captures a distinction present in many real-life cases.

9 See Dean, J., S. Ghemawat. (2004). "MapReduce: Simplified Data Processing on Large Clusters." 6th Symposium on Operating System Design and Implementation (OSDI).

10 See Chambers, C., A. Raniwala, F. Perry, S. Adams, R. Henry, R. Bradshaw, N. Weizenbaum. (2010). "FlumeJava: Easy, Efficient Data-Parallel Pipelines." ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).

11 See also Adya, A., et al. (2019). "Auto-sharding for datacenter applications, OSDI, and Adya, A., R. Grandl, D. Myers, H. Qin. (2019). "Fast key-value stores: An idea whose time has come and gone." HotOS XVII.

12 See, for example, Ghemawat, S. H., Gobioff, S.-T. Leung. (2003). "The Google File System." Proceedings of the 19th ACM Symposium on Operating Systems; Change, F., J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, R. Gruber. (2006). "Bigtable: A Distributed Storage System for Structured Data." 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI); or Corbett, J. et al. (2012)." Spanner: Google's Globally-Distributed Database." OSDI.

13 Note that retries need to be implemented correctly—with backoff, graceful degradation and tools to avoid cascading failures like jitter. Thus, this should likely be a part of Remote Procedure Call library, instead of implemented by hand by each developer. See, for example, Chapter 22 (Addressing Cascading Failures) in the SRE book.

14 This has happened multiple times at Google; for instance, because of someone leaving load-testing infrastructure occupying a thousand Google Compute Engine VMs running when they went on vacation, or because a new employee was debugging a master binary on their workstation without realizing it was spawning 8,000 full-machine workers in the background.

15 As in any complex systems, there are exceptions. Not all machines owned by Google are Borg-managed, and not every datacenter is covered by a single Borg cell. But the majority of engineers work in an environment in which they don't touch non-Borg machines, or nonstandard cells.

16 This particular command is actively harmful under Borg because it prevents Borg's mechanisms for dealing with failure from kicking in. However, more complex wrappers that echo parts of the environment to logging, for example, are still in use to help debug startup problems.

17 My mail server is not interchangeable with your graphics rendering job, even if both of those tasks are running in the same form of VM.

18 This is not the only motivation for making user VMs possible to live migrate, it also offers considerable user-facing benefits because it means the host operating system can be patched and the host hardware updated without disrupting the VM. The alternative (used by other major cloud vendors) is to deliver "maintenance event notices," which mean the VM can be, for example, rebooted or stopped and later started up by the cloud provider.

19 This is particularly relevant given that not all customer VMs are opted into live migration; for some workloads even the short period of degraded performance during the migration is unacceptable. These customers will receive maintenance event notices, and Borg will avoid evicting the containers with those VMs unless strictly necessary.

20 A good reminder that monitoring and tracking the usage of your features is valuable over time.

21 This means that Kubernetes, which benefited from the experience of cleaning up Borg, but was not hampered by a broad existing userbase to

begin with, was significantly more modern in quite a few aspects (like its treatment of labels) from the beginning. That said, Kubernetes suffers some of the same issues now that it has broad adoption across a variety of types of applications.

22 FaaS (Function as a Service) and PaaS (Platform as a Service) are related terms to serverless. There are differences between the three terms, but there are more similarities, and the boundaries are somewhat blurred.

Part V. Conclusion

Chapter 26. Conclusion

Written by Asim Husain

Vice President of Engineering, Google

Software engineering at Google has been an extraordinary experiment in how to develop and maintain a large and evolving codebase. I've seen engineering teams break ground on this front during my time here, moving Google forward both as a company that touches billions of users and as a leader in the tech industry. This wouldn't have been possible without the principles outlined in this book, so I'm very excited to see these pages come to life.

If the past 50 years (or the preceding pages here) have proven anything, it's that software engineering is far from stagnant. In an environment in which technology is steadily changing, the software engineering function holds a particularly important role within a given organization. Today, software engineering principles aren't simply about how to effectively run an organization, they're about how to be a more responsible company for users and the world at large.

Solutions to common software engineering problems are not always hidden in plain sight—most require a certain level of resolute agility to identify solutions that will work for current day problems and also withstand inevitable changes to technical systems. This agility is a common quality of the software engineering teams I've had the privilege to work with and learn from since joining Google back in 2008.

The idea of sustainability is also central to software engineering. Over a codebase's expected lifespan, we must be able to react and adapt to changes, be that in product direction, technology platforms, underlying libraries, operating systems, and more. Today, we rely on the principles outlined in this book to achieve crucial flexibility in changing pieces of our software ecosystem.

We certainly can't prove that the ways we've found to attain sustainability will work for every organization, but I think it's important to share these key learnings. Software engineering is a new discipline, so very few organizations have had the chance to achieve both sustainability and scale. By providing this overview of what we've seen, as well as the bumps along the way, our hope is to demonstrate the value and feasibility of long-term

planning for code health. The passage of time and the importance of change cannot be ignored.

This book outlines some of our key guiding principles as they relate to software engineering. At a high level, it also illuminates the influence of technology on society. As software engineers, it's our responsibility to ensure that our code is designed with inclusion, equity, and accessibility for everyone. Building for the sole purpose of innovation is no longer acceptable; technology that helps only a set of users isn't innovative at all.

Our responsibility at Google has always been to provide developers, internally and externally, with a well-lit path. With the rise of new technologies like artificial intelligence, quantum computing, and ambient computing, there's still plenty for us to learn as a company. I'm particularly excited to see where the industry takes software engineering in the coming years, and I'm confident that this book will help shape that path.

About the Authors

Titus Winters is a Senior Staff Software Engineer at Google, where he has worked since 2010. Today, he is the chair of the global subcommittee for the design of the C++ standard library. At Google, he is the library lead for Google's C++ codebase: 250 million lines of code that will be edited by 12K distinct engineers in a month. For the last 7 years, Titus and his teams have been organizing, maintaining, and evolving the foundational components of Google's C++ codebase using modern automation and tooling. Along the way he has started several Google projects that believed to be in the top 10 largest refactorings in human history. As a direct result of helping to build out refactoring tooling and automation, Titus has encountered first-hand a huge swath of the shortcuts that engineers and programmers may take to "just get something working". That unique scale and perspective has informed all of his thinking on the care and feeding of software systems.

Tom Manshreck is a Staff Technical Writer within Software Engineering at Google since 2005, responsible for developing and maintaining many of Google's core programming guides in infrastructure and language. Since 2011, he has been a member of Google's C++ Library Team, developing Google's C++ documentation set, launching (with Titus Winters) Google's C++ training classes, and documenting Abseil, Google's open source C++ code. Tom holds a BS in Political Science and a BS in History from the Massachusetts Institute of Technology. Before Google, Tom worked as a Managing Editor at Pearson/Prentice Hall and various startups.

Hyrum K. Wright is a Staff Software Engineer at Google, where he has worked since 2012, mainly in the areas of large-scale maintenance of Google's C++ codebase. Hyrum has made more individual edits to Google's codebase than any other engineer in the history of the company. He is a member of the Apache Software and an occasional visiting faculty member at Carnegie Mellon University. Hyrum received a PhD in Software Engineering from the University of Texas at Austin, and also holds an MS from the University of Texas and a BS from Brigham Young University. He is an active speaker at conferences and contributor to the academic literature on software maintenance and evolution.

Colophon

The animal on the cover of *Software Engineering at Google* is an American Flamingo (*Phoenicopterus ruber*).

Many of the animals on O'Reilly covers are endangered; all of them are important to the world.

The cover illustration is by Karen Montgomery, based on a black and white engraving from *Cassell's Natural History*. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.