O'REILLY®

Software Engineering at Google

Lessons Learned from Programming Over Time

Early Release
RAW & UNEDITED

Curated by Titus Winters, Tom Manshreck & Hyrum Wright

1. Preface
    a. Programming Over Time
    b. Google’s Perspective
    c. What this Book Isn’t
    d. Parting Remarks
    e. Conventions Used in This Book
    f. Using Code Examples
    g. O’Reilly Online Learning
    h. How to Contact Us
2. 1. What is Software Engineering?
    a. Time & Change
        i. Hyrum’s Law
        ii. Example: Hash Ordering
        iii. Why Not Just Aim for “Nothing Changes”?
    b. Scale & Efficiency
        i. Policies that Don’t Scale
        ii. Policies that Scale Well
        iii. Example: Compiler Upgrade
        iv. Shifting Left
    c. Tradeoffs & Costs
        i. Example: Markers
        ii. Inputs to Decision Making

iii. Example: Distributed Builds
iv. Example: Deciding Between Time and Scale
v. Revisiting Decisions, Making Mistakes
d. Software Engineering vs. Programming
e. Conclusion
f. TL;DRs
3. 2. Build Systems & Build Philosophy
a. Purpose of a Build System
b. What Happens without a Build System?
i. But All I Need is a Compiler!
ii. Shell Scripts to the Rescue?
c. Modern Build Systems
i. It’s All About Dependencies
ii. Task-Based Build Systems
iii. Artifact-Based Build Systems
iv. Distributed Builds
v. Time, Scale, Trade-offs
d. Dealing with Modules and Dependencies

        i. Using Fine-Grained Modules and the 1:1:1 Rule
        ii. Minimizing Module Visibility
        iii. Managing Dependencies
    e. Conclusion
4. 3. Large-Scale Changes
    a. What is a Large-Scale Change?
    b. Who Deals with Large-Scale Changes?
    c. Barriers to Atomic Changes
        i. Technical Limitations
        ii. Merge Conflicts
        iii. Heterogeneity
        iv. Testing
        v. Code Review
    d. Large-Scale Change Infrastructure
        i. Policies and Culture
        ii. Codebase Insight
        iii. Change Management
        iv. Testing
        v. Language Support
    e. The Large-Scale Change Process
        i. Authorization
        ii. Change Creation

iii. Sharding and Submitting

iv. Cleanup

f. Conclusion

i. TL;DRs

Software Engineering at Google

Lessons Learned from Programming Over Time

Titus Winters, Tom Manshreck, and Hyrum Wright

Software Engineering at Google

by Titus Winters, Tom Manshreck, and Hyrum Wright

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.

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

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

978-1-492-08279-8

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, etc 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 we must rely on more rigorous engineering methods as software becomes more integrated into our lives. 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 their 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 hope that Google’s collective experiences over the last 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
- Tradeoffs and Costs, or how an organization makes decisions, based on the lessons of Time and Scale

Throughout the chapters, we have tried to tie back to these themes and point out ways such principles affect engineering practices and allow them to be sustainable. (See "What is Software Engineering" 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 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 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, UI 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 which helps move our industry to adopt more forward-thinking and sustainable practices. Most importantly, 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

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)

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

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 tradeoffs 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 and 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 may 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 lifespan[1] of your code?" Reasonable answers to this question vary by roughly a factor of 100,000x. 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, 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 lifespans, 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 lifespan of your software, you are capable of reacting to whatever valuable change comes along, for either technical or business reasons. Importantly, we are only looking for capability - you may 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 may 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 multi-person development of multi-version 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 get 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*. 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 have to be made and their stakes. In software engineering, we are regularly forced to evaluate the tradeoffs 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 tradeoffs and make rational decisions. We may sometimes defer maintenance changes, or even embrace policies that don't scale well, with the knowledge that we'll have 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 range of 100,000x for reasonable answers on "How long will this software live", a range of perhaps 10,000x 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'll 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 us first elaborate on these primary themes of time, scale, and tradeoffs.

Time & Change

When a novice is learning to program, the lifespan 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. They may have 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 lifespan[5], and for better or worse, full rewrites are relatively common. Engineers at an early-stage startup may rightly choose to focus on immediate goals over long-term investments: the company may 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 lifespan - we can't reasonably predict an endpoint for Google Search, the Linux kernel, or the Apache HTTP Server project. For most Google projects, we have to assume they will live indefinitely: we cannot predict when we won't need to upgrade our dependencies, language versions, etc. As their lifetimes grow, these long-lived projects *eventually* have a different feel to them than programming assignments or startup development.

Consider two software projects on opposite ends of this "expected lifetime" spectrum. For a programmer working on a task with an expected lifespan of hours, what types of maintenance are reasonable to expect? That is, if a new version of your operating system 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.

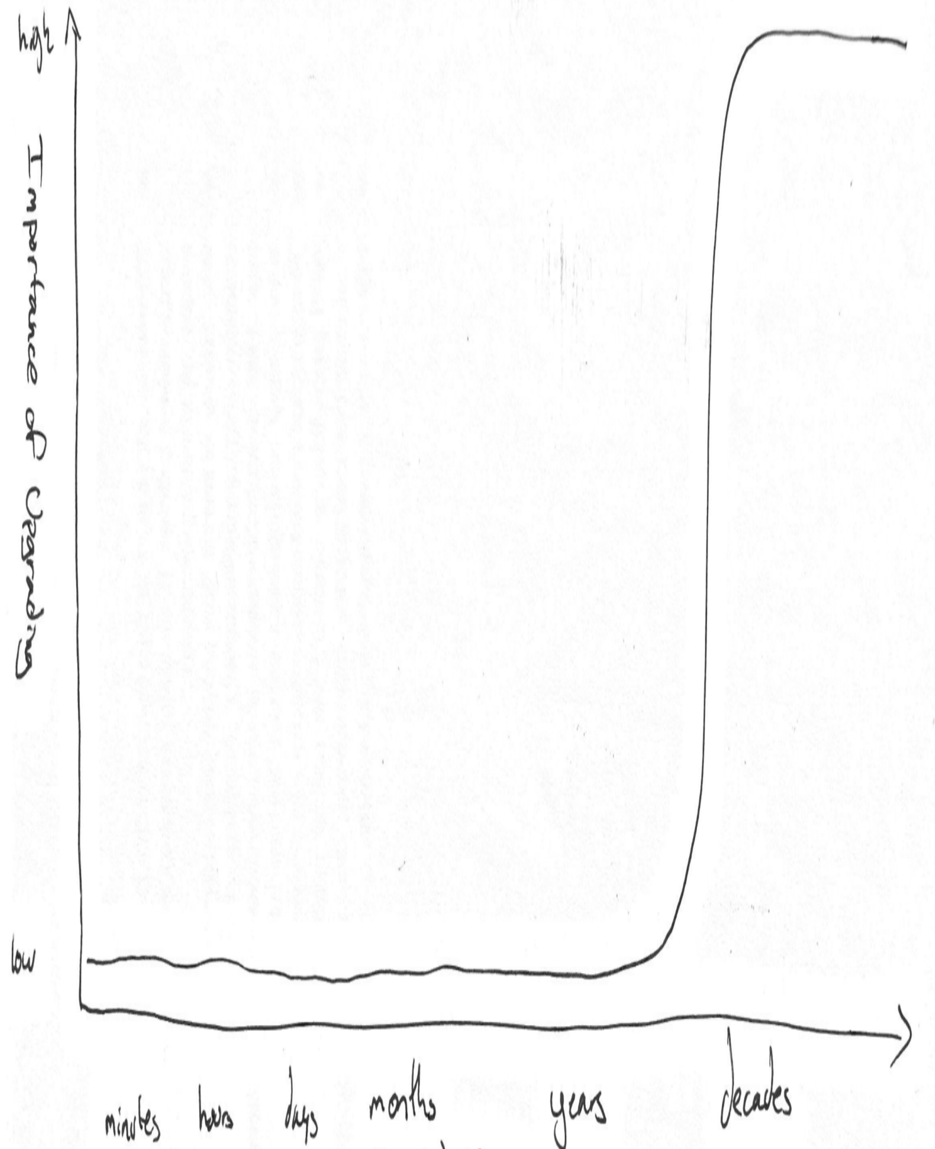

Figure 1-1. Lifespan and the Importance of Upgrades

These two points on the expected lifespan 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 start to react to changing externalities[6]. 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 lifespan of the project in question.

Getting through not only that first big upgrade, but getting to the point 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 lifespan? 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, as 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” vs “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 have to be aware of Hyrum’s Law[7], 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, then your API will be easy to change. Given enough time and enough users, even the most innocuous change *will* break something[8] - your analysis of the value of that change has to incorporate the difficulty in sussing out 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 may seem unremarkable, but over the past decade or two, the computing industry’s experience using such types has evolved:

- Hash flooding[9] attacks provide an increased incentive for non-deterministic 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 which depend on the order that 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?” they 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 may be used by other code that implicitly creates such a dependency. For example, if your library serializes values into an RPC (Remote Procedure Call) response, the RPC caller may 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 may 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 have to weigh the tradeoffs 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” vs. a “works indefinitely” mentality, we can extract some clear relationships. Looking at code as an artifact with a (highly) variable lifetime requirement, we can start to categorize programming styles: code that depends on brittle and unpublished features of its dependencies is likely to be described as “hacky” or “clever”, while 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 lifespan 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 may 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 may need to be changed.” If you’ve got a project written in pure C with no external dependencies (or only external dependencies that promise great long-term stability like POSIX), you may 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 may change to support new features, and affect downstream users. Security problems are disclosed in all manner of technology, from processors to networking libraries to application code. *Every* piece of technology that your project depends upon has some (hopefully small) risk of containing critical bugs and security vulnerabilities that may come to light only after you've started relying on it. If you are incapable of deploying a patch for Heartbleed[10] or mitigating speculative execution problems like Spectre and [11] 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 may 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 may 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 above 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 affect us directly or only manifest in 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 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 sysadmin 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 & Efficiency

As noted in the Site Reliability Engineering (SRE) book[12], 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 across 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[13]. Eventually, time will take hold and something unexpected will arise that you absolutely must change. When your project doubles in scope and you have 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 which 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 may come where 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[14]- 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 has to 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 100x. When we are 10x larger, did we add 10x more work on our sample engineer to keep up with? Does the amount of work our engineer has to 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'll discuss deprecation much more in "Deprecation," 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. In order 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 may 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 may impact 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 have to do the work to migrate their internal users to new versions themselves, or do the update in-place in backwards-compatible fashion. This policy, which we've called the "Churn Rule", scales better: dependent projects are no longer spending ever-increasing 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 sub-problem. 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 may 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 re-syncing and testing. Such branch management can be made to work for a small organization juggling 5-10 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 - we’ll discuss that in the “Version Control.”

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 super-linear 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[15]. 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 have 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’ve got 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. This is covered more in “Knowledge Sharing.”

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 is it an operation 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 mis-compilations, 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 only been exposed to 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[16]. In the end, the 2006 compiler upgrade was extremely painful. Many Hyrum’s Law problems, both big and small, had crept into the codebase and served to ossify 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 where most code has gone through several upgrades, it stops depending on the nuances of the underlying implementation, and instead the actual abstraction guaranteed by the language or operating system. 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, since 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[17].

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 have 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 once 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.[18] 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, as 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, 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 only discovered 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.

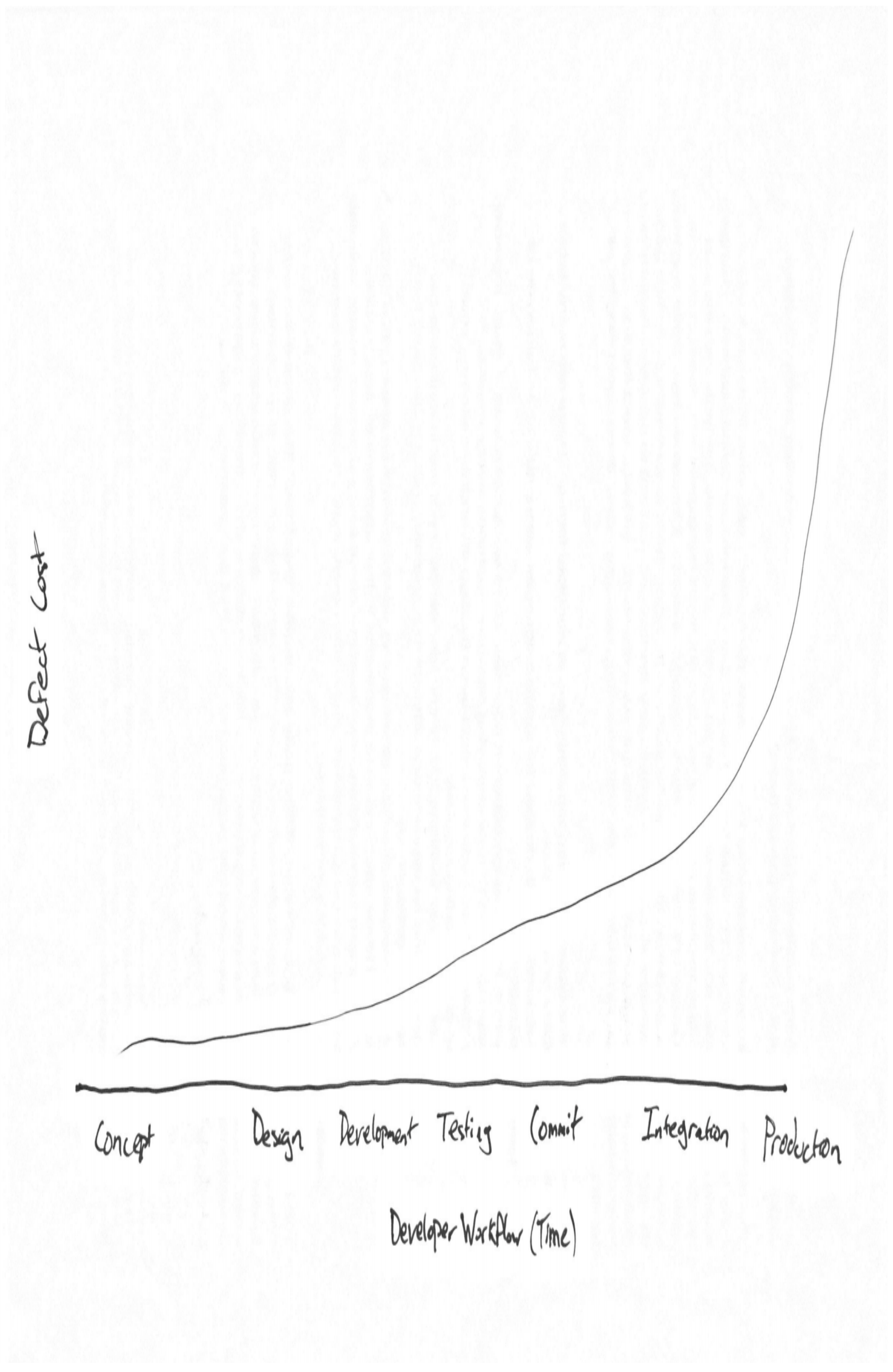

Figure 1-2. Timeline of the developer workflow

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 & 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, then 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 may involve any or all of the below:

- 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, 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 "Engineering for Equity."

In addition to the above costs (or our estimate of them), there are biases: status quo bias, loss aversion, etc. When we evaluate cost, we need to keep all of the above 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 10-20% 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 dried up. 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 handful 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 tradeoffs 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 may 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 tradeoffs. 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 (see “Measuring Engineering Productivity”).

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

- We are doing this because we have to (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.”[19]

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 we’re evaluating tradeoffs 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 2 weeks changing this linked-list into a higher-performance structure, I’m going to use 5GiB more production RAM but save 2K 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 2 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 (see “Measuring Engineering Productivity”). 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 harder to manage.

Example: Distributed Builds

Consider your build. According to completely unscientific Twitter polling, something like 60-70% of developers build locally, even with today’s large complicated builds. This leads directly to non-jokes 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, etc. These resource costs were particularly troublesome: of course we want people to have as fast a build as possible … but at the same time, most of the time a high-performance desktop development machine will still sit idly. This doesn't feel like the right 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. "Build Systems" will go into more of the details on our approach to distributed builds and the relevant tradeoffs.

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 where consumption could run rampant, and almost nobody was incentivized to keep an eye on build bloat. This is reminiscent of Jevon's Paradox[20] - 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, not all of these costs were foreseen. Having blazed ahead, we found ourselves in a situation where 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 tradeoff 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 may 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 importantly, 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 re-using 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 lifespan 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 may 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 lifespan 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 that 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 “How to Lead a Team “and “Leading at Scale.”

Software Engineering vs. Programming

When presented with our distinction between software engineering and programming, some readers will 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 "Larger Scope Testing") and continuous deployment practices (See "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 "Dependency Management") 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 lifespan of that code.
- There is at least a factor of 100,000x between the lifespans 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 lifespan of the code, we are capable of responding to changes in dependencies, technology, or

product requirements. We may choose not to 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 may 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 "Nothing is Certain Except Death, Taxes and a Short Mobile App Lifespan", Appcelerator, https://devblog.axway.com/mobile-apps/nothing-is-certain-except-death-taxes-and-a-short-mobile-app-lifespan-2/

6 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 5 years. Somewhere between 5-10 years seems like a conservative estimate for this transition in general.

7 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.

8 "Workflow", http://xkcd.com/1172/

9 A type of 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.

10 "The Heartbleed Bug" http://heartbleed.com/

11 "Meltdown and Spectre" https://meltdownattack.com/

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

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

14 https://en.wikipedia.org/wiki/Boiling_frog#As_metaphor

15 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.”

16 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.

17 Ibid. Chapter 5: Eliminating Toil.

18 In our experience, an average 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.

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

20 https://en.wikipedia.org/wiki/Jevons_paradox

Chapter 2. Build Systems & 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 build code quickly and reliably. The effort has been so successful that Blaze, the main component of the build system, has been re-implemented several different times by ex-Googlers who have left the company[2]. In 2015, Google finally open-sourced an implementation of Blaze named Bazel[3].

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 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 "Continuous Deployment").

- Developer changes are automatically tested when they're sent for code review (see "Code Review Tooling") 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 harder to submit breaking changes.
- Authors of low-level libraries are able to test their changes across the entire codebase, ensuring 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. Large-scale changes are discussed more in "Large Scale Changes".

All of this is possible only because of Google's investment in its build system. Though Google may 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?

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 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 get more complicated quickly once 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 whole 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 right 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 your hobby project starts out simple enough that you can build it using just a compiler, but you start 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 right order. This helps out for a while, but pretty soon you start running into even more problems:

1. It gets tedious. As your system gets more complex, you start spending almost as much time working on your build scripts as on real code. Debugging shell scripts is painful, and more and more hacks start getting layered on top of one another.
2. It’s slow. In order 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.
3. 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[4]. 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…
4. Disaster! Your hard drive crashes, and now you need to re-create your whole 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 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?
5. Despite the problems, your project is successful enough that you’re able to start hiring more engineers. Now you realize that it doesn’t take a disaster for the previous problems to come up - you have 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.
6. 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.

7. Builds get 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 co-worker 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 may 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 once you need to coordinate across multiple developers and their machines, even a perfect build script isn't enough since 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 script-based DIY approach we were working on above: they run the same compilers under the hood, and you need to understand those underlying tools to be able to tell 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 above problems, one theme repeats over and over: managing your own code is fairly straightforward, but managing its dependencies is much harder (see "Dependency Management") 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** describing how to perform the build. Take this example from the Ant manual[5]:

```
            <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[6]). 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 plugins 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):

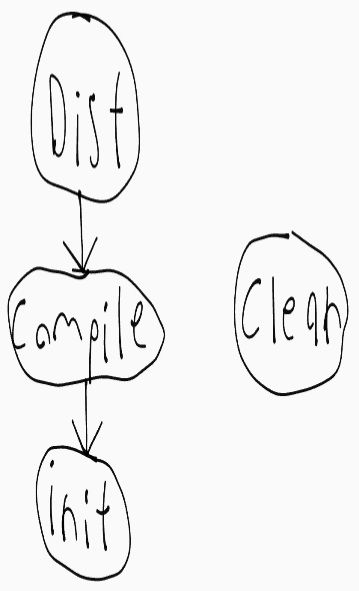

Figure 2-1. An acyclic graph showing dependencies

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 in order:

1. Loads a file named build.xml in the current directory and parses it to create the graph structure above.
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 since all of that task's dependencies have been run.

7. Executes the commands defined in the `dist` task since 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/*
```

Once 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 figuring out 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

Since 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 get 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**. Since 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. Since 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 get to task A more quickly? 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 hard-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, where 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 reasons described above. But unfortunately, task-based build systems struggle here too. Since tasks can do anything, there's no way in general to check if they've already been done. Many tasks simply take a set of source files and run a compiler to create a set of binaries, and so don't need to be re-run 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. In order to guarantee correctness, the system typically has to re-run 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 re-run. 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 gets 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 re-run 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 hard 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"). Since 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. In order 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: since 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 multi-core machine, and is only possible since 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 re-used. 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`. Since 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 gets even harder when your project uses languages that require different tools based on which platform they're being built on or compiled for (e.g. Windows vs. 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 have 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[7]. 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[8].

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, as 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[9]. 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, since 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 have to download dependencies (whether tools or libraries) from external sources rather than building them directly. 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[10]: 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[11]. 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 re-downloads 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 start failing if you don’t have another copy of that dependency available. To avoid this problem, we recommend that, for any non-trivial 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 over two billion lines of code, chains of dependencies can get 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**, where 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 towards by defining an artifact-based build system.

REMOTE CACHING

The simplest type of distributed build is one that only leverages **remote caching**. The system is shown in Figure 18-2:

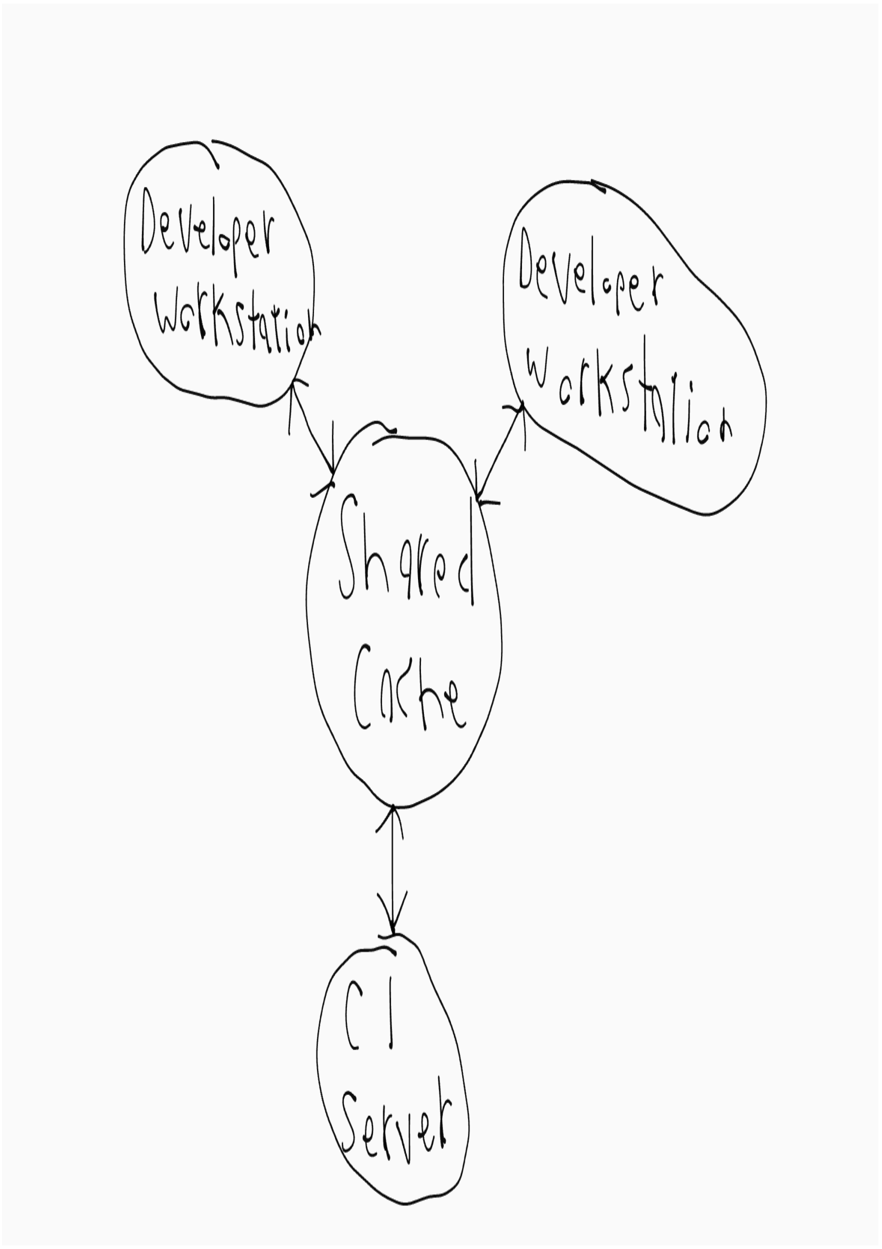

Figure 2-2. A distributed build showing remote caching

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.

In order 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[12]. 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**, where the actual work of doing the build can be spread across any number of workers. A remote execution system is shown in Figure 18-3:

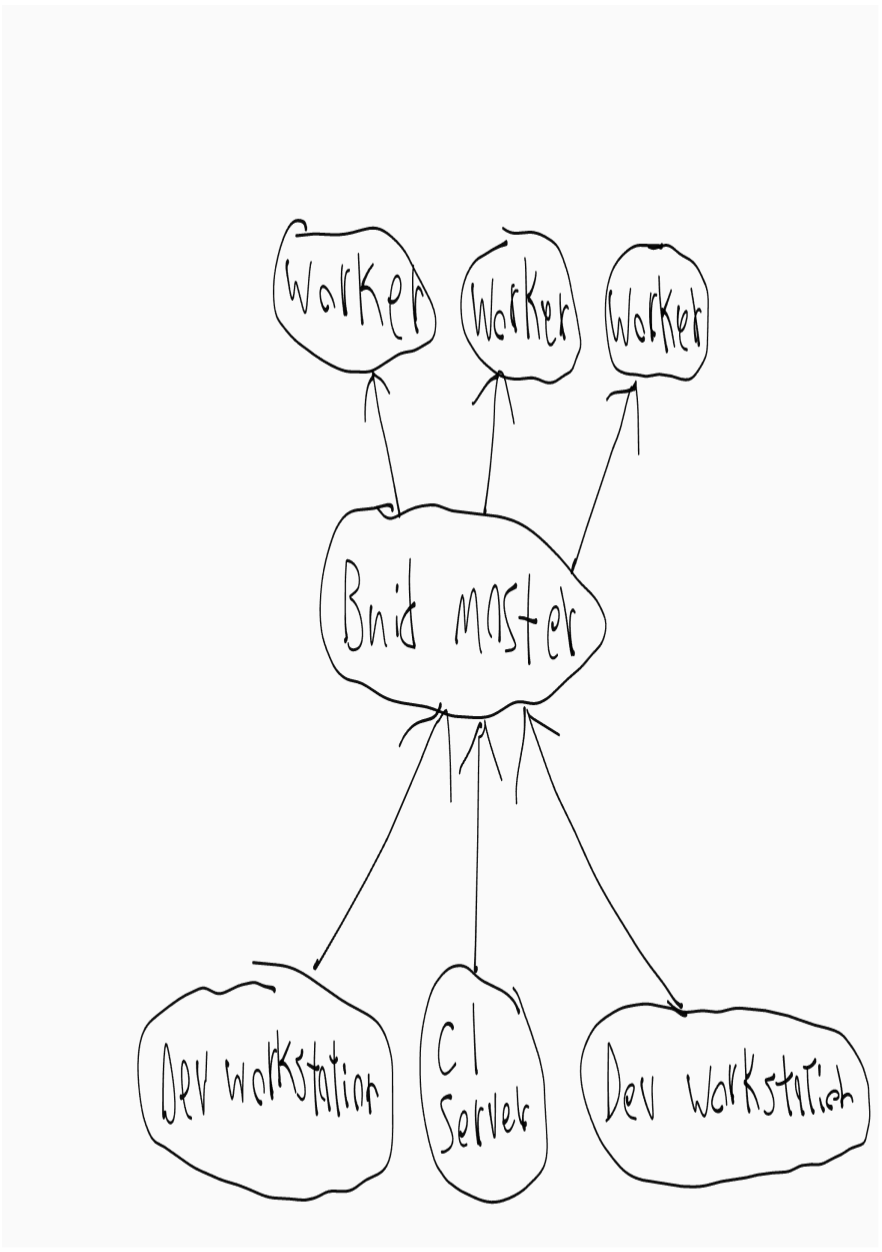

Figure 2-3. A remote execution system

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 may 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.

In order 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, as each step may 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

Google has been using a distributed build system that leverages both remote caching and remote execution since 2008. The system is illustrated in Figure 18-4:

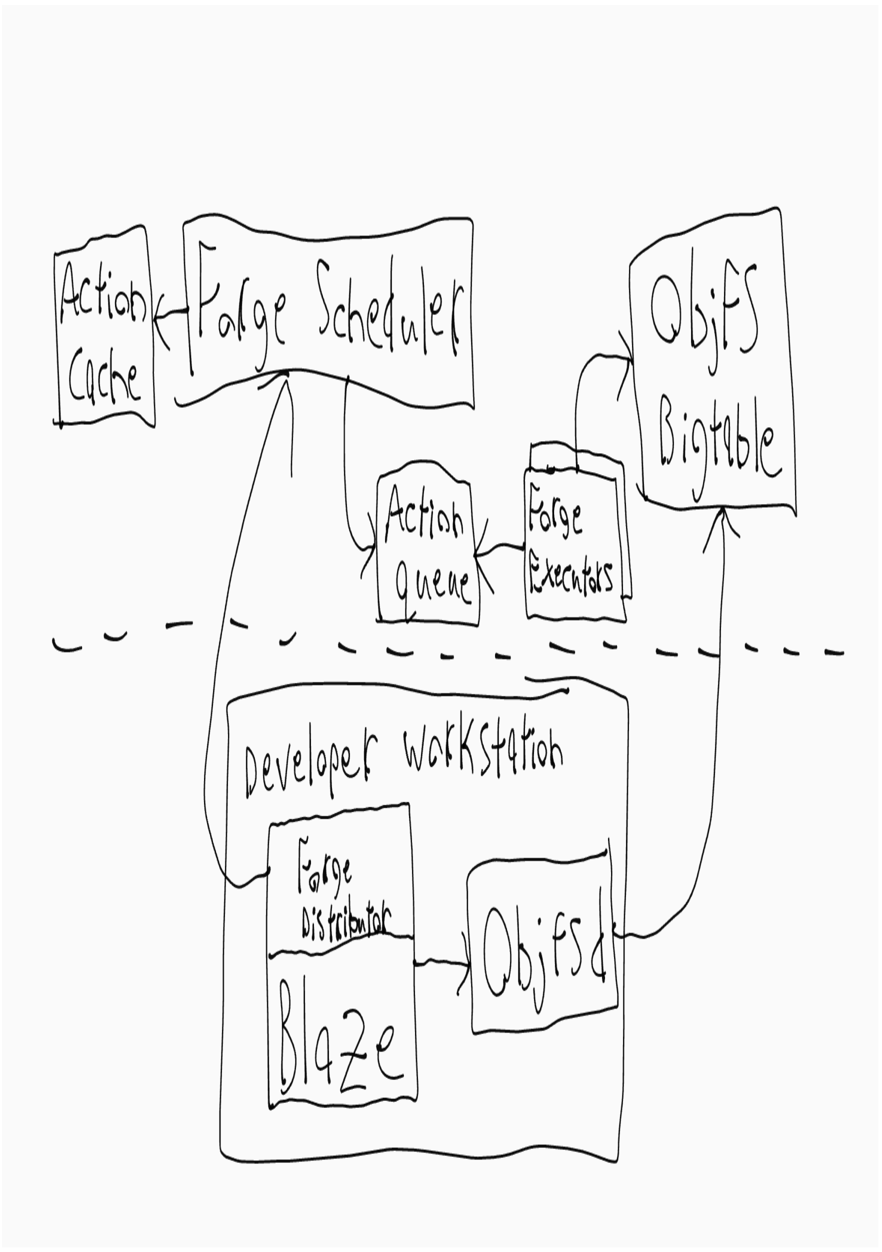

Figure 2-4. Google's distributed build system

Google's remote cache is called ObjFS. It consists of a backend that stores build outputs in Bigtables[13] 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[14] 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 about millions of builds executing millions of test cases and producing petabytes of build outputs from billions of lines of source code every *day*[15]. 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 start 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, 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 a project's build system can be expensive, and that cost grows the larger the project gets. 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 structured 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 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 have to spend more work 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 may own several hundred targets within their 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[16]). Languages with weaker packaging conventions will frequently define multiple targets per BUILD file.

The benefits of smaller build targets really start to show at scale, as they lead to faster distributed builds and a less frequent need to rebuild targets. The advantages get even more compelling once testing enters the picture, as finer-grained targets means that the build system can be much smarter about running only a limited subset of tests that could be affected by any given change. Since 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[17].

Minimizing Module Visibility

Bazel and other build systems allow each target to specify a visibility, a property which 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 the can only be referenced 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, then 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 only make targets public 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 only have 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 repo.

One issue that should be handled carefully with regards to internal dependencies is how to treat **transitive dependencies** (shown in 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?

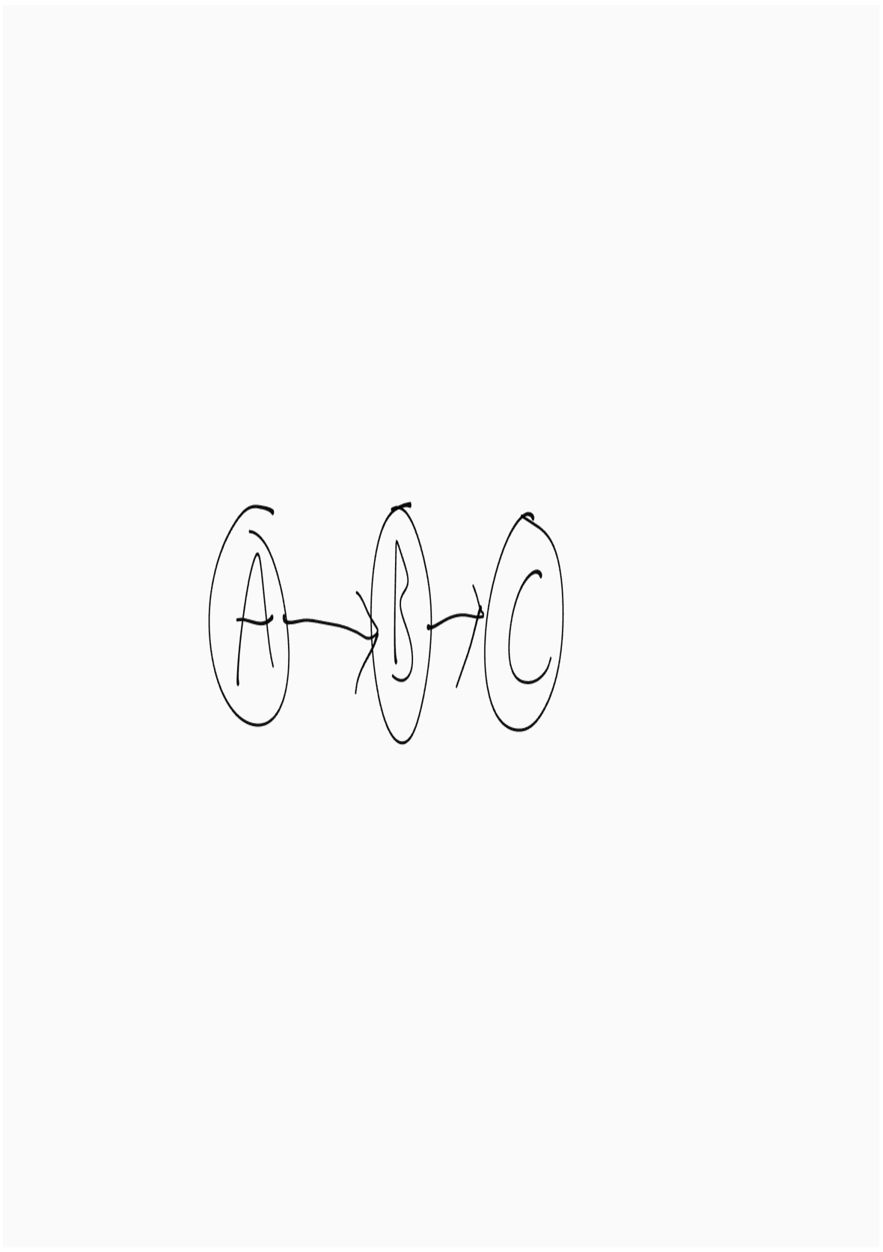

Figure 2-5. Transitive dependencies

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 started to see problems. Suppose 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 if 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 multi-year effort, but it was well worth it - our builds are now much faster since targets have fewer unnecessary dependencies[18], 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 deps 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[19].

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 vs. 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[20] 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 non-trivial 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 may be subtle behavior or performance changes that are impossible to track down.

In contrast, since 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 to 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, and so we enforce a strict **one-version rule**[21] 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 as 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, since 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.

The one-version rule is discussed further in the context of a large monorepo in “Dependency Management.”

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 - since different libraries may 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 starts 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[22] 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 updated manually to adjust the versions of each dependency.

Yet again, the choice here is one between convenience and scalability. Small projects may 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 may 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 harder, since 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 above. Such a build system will save the resulting artifacts from every build in 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 the performance benefits of dependending 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, as 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 **one-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.

1 In an internal survey, 83% of Googlers reported being satisfied with the build system, making it the 4th 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://bazel.build

4 https://xkcd.com/1168/

5 https://ant.apache.org/manual/using.html

6 Ant uses the word "target" to represent what we call a "task" in this chapter, and uses the word "task" to refer to what we'll call "commands".

7 https://docs.bazel.build/versions/master/toolchains.html

8 https://docs.bazel.build/versions/master/skylark/rules.html

9 https://blog.bazel.build/2017/08/25/introducing-sandboxfs.html

10 Such "software supply chain" attacks are becoming more common: https://blog.sonatype.com/2018-state-of-the-software-supply-chain-report

11 Go recently added preliminary support for modules using the exact same system: https://github.com/golang/go/wiki/Modules

12 https://docs.bazel.build/versions/master/remote-caching.html

13 https://research.google.com/archive/bigtable-osdi06.pdf

14 http://google-engtools.blogspot.com/2011/10/build-in-cloud-distributing-build.html

15 https://docs.google.com/presentation/d/1l6Xyt0DtH7OIp04tzM3WjzDTHt0cl_IbuGToBD_yPYM/edit#slide=id.g3a89e6b664_1_2027

16 https://www.pantsbuild.org/build_files.html

17 https://github.com/bazelbuild/bazel-gazelle

18 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 "Large Scale Changes" for more information about how Google makes large-scale changes like this.

19 https://blog.bazel.build/2017/06/28/sjd-unused_deps.html

20 https://semver.org

21 https://opensource.google.com/docs/thirdparty/oneversion/

22 https://docs.bazel.build/versions/master/generate-workspace.html

Chapter 3. 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 may 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 harder 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, as well as providing some real-life examples of how and where we've used these approaches. While 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, a large-scale change is any set of changes which 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, a large-scale change 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.

Large-scale changes at Google are almost always generated using automated tooling. Reasons for making a large-scale change vary, but the changes themselves generally fall into a few basic categories:

- Cleaning up common anti-patterns 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, it is useful for their customers to have insight in to 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 may be broader motivating causes behind specific large-scale changes. For example, a new language standard may introduce a more efficient idiom for accomplishing a given task, an internal library interface may change, or a new compiler release may require fixing existing problems which would be flagged as errors by the new release. The majority of large-scale changes across Google actually have near-zero functional impact: they tend to be widespread textual updates for clarity, optimization or future-compatibility. But large-scale changes 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 may 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 large-scale changes, as an enabler for the many teams doing infrastructure work. We've also found that efficient tooling also helps engineers performing smaller changes. The same tools which make changing thousands of files efficient, also scale down to tens-of-files reasonably well.

Who Deals with Large-Scale Changes?

As indicated above, the infrastructure teams that build and manage our systems are responsible for much of the work of performing large scale changes, but the tools and resources are available across the company. If you skipped the opening chapter "What is Software Engineering," function or system and dictate that everybody who uses the old one move to the updated analogue? While this may 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 which 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 which infrastructure teams already have. Centralization also allows for faster recovery when faced with errors, as 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 large-scale changes, 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]. While a new system may 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 which own the systems requiring large-scale changes 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 who has a vested interest in removing the old system responsible for the migration effort helps ensure that it actually gets done. While funding and staffing a team to run these kinds of migrations can seem like an additional cost, it is actually just internalizing the externalities which an unfunded mandate creates, with the additional benefits of economies of scale.

FILLING POTHOLES

While the large-scale change 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 migrating 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.

While fixing this single-character change might seem pointless, particularly across a codebase the size of Google's, the maturity of our large-scale change 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 Google uses to actually effect large-scale changes, 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 which 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 may be completely infeasible even at small scale when using a set of distributed or federated repositories.

Technical Limitations

To start with, most version control systems have operations which scale linearly with the size of a change. Your system may be able to handle small commits (e.g., on the order of tens of files) just fine, but may not have sufficient memory or processing power to atomically commit thousands of files at once. In centralized version control systems, 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 may not be just "hard" or "unwise" to make a large change atomically: it may simply be impossible with a given set of infrastructure. 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 may be able to sneak in a change which touches every file in the repository on a weekend when nobody is doing development. Or you may 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 site reliability engineers who run Google's production services have a mantra: "No Haunted Graveyards". A haunted graveyard in this sense is a system which is so ancient, obtuse or complex that no one dares enter it. Haunted graveyards are often business critical systems which 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 which 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 which may 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 large-scale change 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 which they use. From a large-scale change 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 or not 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

Large-scale changes only really work 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 right code transformations to the correct places. If your organization has many different version control systems, 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 pre-submit 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 the change has an associated bug. Many of these checks are relevant for teams writing new features, but for large-scale changes, they just add additional irrelevant complexity.

We've decided to embrace some of this complexity, such as running pre-submit tests, by making it standard across our codebase. For other inconsistencies we advise teams to omit their special checks when parts of large-scale changes touch their project code. Most teams are happy to help, given the benefit these kinds of changes are to their projects.

Much of the benefits of consistency for humans mentioned in the "Style Guides" chapter also applies for automated tooling.

Testing

Every change should be tested (a process we'll talk about more below), but the larger the change the harder it is to actually test it appropriately. Google's continuous integration system will run not only the tests immediately impacted by a change, but also any tests which 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 twenty-five files is pretty straightforward; finding one in a ten-thousand file change is like the proverbial needle in a haystack.

The tradeoff in this decision is that smaller changes will cause the same tests to be run multiple times, particularly tests which 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 tradeoff we're willing to make. That same tradeoff might not hold for all organizations, but it is worth examining what the right balance is for yours.

TESTING LARGE SCALE CHANGES

By Adam Bender

Today it is common for a double-digit percentage (10% -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 a 1000 different places across hundreds of different products and services[7]. After the change is written, you have to coordinate code reviews across dozens of teams. Lastly, once 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 re-run 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 a large scale change can be a slow and frustrating process. Once 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 re-running 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 the whole process down. 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 once 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:

1. LSCs tend to be pure refactorings and therefore very narrow in scope, preserving local semantics.
2. Individual changes are often more simple 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 once and doesn't require 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 1000 randomly-selected tests.
2. Gather up all the changes that passed their 1000 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 may mean running every single test in Google's repository. This process can take more than six hours to complete.
4. For each non-flaky test that fails, re-run 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 "Code Review," all changes need to be reviewed before submission, and this policy applies even for large-scale changes. 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, see below). As we discuss below, tooling can often help in this space, but for some classes of changes, we still want humans to explicitly verify they are correct. Breaking a large-scale change 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 they get 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 which the language now could detect. std::unique_ptr was strictly better than scoped_ptr, yet Google's codebase had over 500,000 references to scoped_ptr scattered amongst millions of source files. Moving to the more modern type required the largest large-scale change 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 over 700 independent changes, touching over 15,000 files *per day*. Today we sometimes manage 10x that throughput, having refined our practices and improved our tooling.

Like almost all large-scale changes, 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 which 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 only possible because of our technology and tooling for large-scale changes.

Large-Scale Change Infrastructure

Google has invested in a significant amount of infrastructure to make large-scale changes possible. This infrastructure includes tooling for change creation, change management, change review, and testing. However, perhaps the most important support for large-scale changes has been the evolution of cultural norms around large-scale changes and the oversight given to them. While the sets of technical and social tools may differ for your organization, the general principles should be the same.

Policies and Culture

As we've described in "Version Control," Google stores the bulk of its source code in a single monolithic repository, 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 large-scale change 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. While 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 light-weight approval process for teams and individuals seeking to make large-scale changes across Google. This process is overseen by a group of experienced engineers 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 large-scale changes, 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 may 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 large-scale changes. While it is important for code owners to have a sense of responsibility for their software, they also needed to learn that large-scale changes 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 large-scale changes. 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 large-scale change, 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 they don't hold a veto over the broader large-scale change. Over time, we've found that a good FAQ and a solid historic track record of improvements have generated widespread endorsement of large-scale changes throughout Google.

Codebase Insight

In order to do large-scale changes, 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[12] 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 they can be incorporated into refactoring tools. (For further examples, see "Code Search" and "Static Analysis.")

We also use compiler-based indices to run abstract syntax tree-based analysis and transformations over our codebase. Tools such as ClangMR[13], JavacFlume or Refaster[14], which can perform transformations in a highly parallelizable way, depend on these insights as part of their function. For smaller changes, authors may 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 it's human effort scale sublinearly with the codebase: 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 her change covers all the cases she's 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 below when describing our large-scale change process. In many respects, Rosie is not just a tool, but an entire platform for making large-scale changes at Google-scale, which gives 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 "Testing Overview," tests are one of the important ways that we validate our software will behave as expected. This is particularly important when applying changes which 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 large-scale changes differs slightly from that of normal changes, while still using the same underlying continuous integration infrastructure. Testing large-scale changes 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 may contain arbitrary files, we don't use the standard project-based pre-submit tests. Instead, we run each shard over the transitive closure of every test it may affect, which we discussed above.

Language Support

Large-scale changes 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. In languages which lack these features, it is often difficult to migrate systems incrementally.[15]

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 which we can use to build tools to effect large-scale changes, 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 harder to maintain. While 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 large-scale change infrastructure. Since we work towards 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 have to worry about formatting specifics. Applying automated formatting, such as google-java-format[16] or clang-format[17] 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

Large-scale changes 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”[18].

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. Since the only requirement for the vulnerability was having the wrong library somewhere in its classpath, anything which 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 large-scale change process. By using tools such as BigQuery[19], volunteers identified affected projects, and sent over 2,600 patches to upgrade their versions of the Commons library to one which addressed Mad Gadget. Instead of automated tools managing the process, over fifty humans made this large-scale change work.

The Large-Scale Change Process

With these pieces of infrastructure in place, we can now talk about the process for actually making a large-scale change. 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.

Once the change is approved, the author can move forward in getting her change submitted. Historically, the committee has been very liberal with their approval[20], and often gives approval not just for a specific change but also for a broad set of related changes. Committee members may, 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 large-scale change authors. The committee is also empowered as the escalation body for concerns or conflicts about a large-scale change: 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, a large-scale change author will start to produce the actual code edits. Sometimes, these can be generated comprehensively into a single large global change which 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[21] or textual merge conflicts occur in the changed code. Occasionally, in the rare case where technical tools aren’t able to generate the global change, we have sharded change generation across humans (see accompanying sidebar “Operation Rosehub”). While 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 “Style Guides”[22].

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 VS. 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 may 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 finally gets 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 large-scale changes 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, which 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 hard 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 continuous integration framework. We run every test which depends on the files in a given change transitively, which often creates high load on our continuous integration system.

This may 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 which impact 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 “Testing Overview,” which often don’t harm the teams which write and maintain them, are particularly difficult for large-scale change authors. While fairly low impact for individual teams, flaky tests can seriously impact the throughput of a large-scale change system. Automatic flake detection and elimination systems help with this issue, but it can be a constant effort to ensure teams which write flaky tests are the ones who bear their costs.

In our experience with large-scale changes 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 may cause a regression that is only detected 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 large-scale change process, individual shards should be committable independently. This means 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 "Code Review" 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 which 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 pre-commit tools, which may perform additional checks. For large-scale changes, we selectively disable certain checks such as those for non-standard change description formatting. While useful for individual changes on specific projects, such checks are a source of heterogeneity across the codebase, and can add significant friction to the large-scale change process. This heterogeneity is a barrier to scaling our processes and systems, and large-scale change tools and authors can't be expected to understand special policies for each team.

We also aggressively ignore pre-submit check failures which pre-exist 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 large-scale changes across Google's codebase. Local code owners are responsible for having no pre-existing 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 large-scale changes with the same rigor as regular changes - they trust the engineers generating large-scale changes 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 only given 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 may 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 which meet their expectations. Thus, they only have to manually examine a small subset which 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 pre-commit 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 which 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 large-scale changes have different definitions of "done", which may vary from completely removing an old system, to migrating only high-value references and leaving old ones to organically disappear.[23] In almost all cases, it's important to have a system which prevents additional introductions of the symbol or system which the large-scale change worked hard to remove. At Google, we use the Tricorder framework mentioned in "Static Analysis" and "Code Review Tooling" 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 "Deprecation."

Conclusion

Large-scale changes 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 large-scale change 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 both 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

- A large-scale change process makes it possible to rethink the immutability of certain technical decisions.

- Traditional models of refactoring break at large scales.
- Making large-scale changes means making a habit of making large-scale changes.

---

1 See "Version Control" for some ideas about why

2 It's possible in this federated world to just 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 See "Deprecation" for a further discussion about this practice

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 has to 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 "Large Scale Changes" ever executed removed over 1 BILLION lines of code from the repository over the course of three days. This was largely to remove an obsolete part of the repo that had been migrated to a new home, but still - how confident do you have to be to delete 1 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 only performed 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 https://kythe.io/

13 https://ieeexplore.ieee.org/abstract/document/6676954

14 https://dl.acm.org/citation.cfm?id=2541355

15 In fact, Go recently introduced these kinds of language features specifically to support large-scale refactorings (see https://talks.golang.org/2016/refactor.article).

16 https://github.com/google/google-java-format

17 https://clang.llvm.org/docs/ClangFormat.html

18 https://opensource.googleblog.com/2017/03/operation-rosehub.html

19 https://cloud.google.com/bigquery

20 The only kinds of changes the committee has outright rejected as ones 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 large-scale changes has dropped, the threshold for approval has as well.

21 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.

22 In actuality, this is the reasoning behind the original work on clang-format for C++.

23 Sadly, the systems we most want to organically decompose are those which are the most resilient to doing so. They are the plastic six-pack rings of the code ecosystem.

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.