O'REILLY®

Software Engineering at Google

Lessons Learned from Programming Over Time

Curated by Titus Winters, Tom Manshreck & Hyrum Wright

CHAPTER 1

What Is Software Engineering?

Written by Titus Winters
Edited by Tom Manshreck

Nothing is built on stone; all is built on sand, but we must build as if the sand were stone.
—Jorge Luis Borges

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

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

One way to see the impact of time on a program is to think about the question, "What is the expected life span[1] of your code?" Reasonable answers to this question

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?

vary by roughly a factor of 100,000. It is just as reasonable to think of code that needs to last for a few minutes as it is to imagine code that will live for decades. Generally, code on the short end of that spectrum is unaffected by time. It is unlikely that you need to adapt to a new version of your underlying libraries, operating system (OS), hardware, or language version for a program whose utility spans only an hour. These short-lived systems are effectively "just" a programming problem, in the same way that a cube compressed far enough in one dimension is a square. As we expand that time to allow for longer life spans, change becomes more important. Over a span of a decade or more, most program dependencies, whether implicit or explicit, will likely change. This recognition is at the root of our distinction between software engineering and programming.

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

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

Team organization, project composition, and the policies and practices of a software project all dominate this aspect of software engineering complexity. These problems are inherent to scale: as the organization grows and its projects expand, does it become more efficient at producing software? Does our development workflow

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

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

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

become more efficient as we grow, or do our version control policies and testing strategies cost us proportionally more? Scale issues around communication and human scaling have been discussed since the early days of software engineering, going all the way back to the *Mythical Man Month*.[5] Such scale issues are often matters of policy and are fundamental to the question of software sustainability: how much will it cost to do the things that we need to do repeatedly?

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

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

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

5 Frederick P. Brooks Jr. *The Mythical Man-Month: Essays on Software Engineering* (Boston: Addison-Wesley, 1995).

Time and Change

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

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

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

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

6 Appcelerator, "Nothing is Certain Except Death, Taxes and a Short Mobile App Lifespan," Axway Developer blog, December 6, 2012.

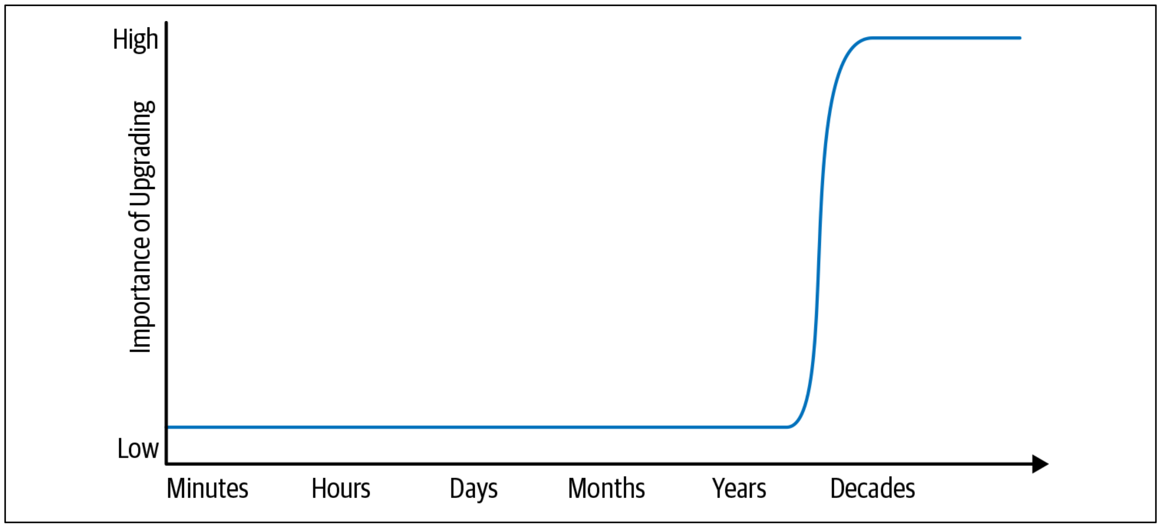

Figure 1-1. Life span and the importance of upgrades

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

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

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

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

Getting through not only that first big upgrade, but getting to the point at which you can reliably stay current going forward, is the essence of long-term sustainability for your project. Sustainability requires planning and managing the impact of required change. For many projects at Google, we believe we have achieved this sort of sustainability, largely through trial and error.

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

Hyrum's Law

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

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

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

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

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

9 See "Workflow," an *xkcd* comic.

Example: Hash Ordering

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

```
>>> for i in {"apple", "banana", "carrot", "durian", "eggplant"}: print(i)
...
durian
carrot
apple
eggplant
banana
```

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

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

As a result, if you ask any expert "Can I assume a particular output sequence for my hash container?" that expert will presumably say "No." By and large that is correct, but perhaps simplistic. A more nuanced answer is, "If your code is short-lived, with no changes to your hardware, language runtime, or choice of data structure, such an assumption is fine. If you don't know how long your code will live, or you cannot promise that nothing you depend upon will ever change, such an assumption is incorrect." Moreover, even if your own implementation does not depend on hash container order, it might be used by other code that implicitly creates such a dependency. For example, if your library serializes values into a Remote Procedure Call (RPC) response, the RPC caller might wind up depending on the order of those values.

This is a very basic example of the difference between "it works" and "it is correct." For a short-lived program, depending on the iteration order of your containers will not cause any technical problems. For a software engineering project, on the other hand, such reliance on a defined order is a risk—given enough time, something will

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

make it valuable to change that iteration order. That value can manifest in a number of ways, be it efficiency, security, or merely future-proofing the data structure to allow for future changes. When that value becomes clear, you will need to weigh the trade-offs between that value and the pain of breaking your developers or customers.

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

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

Why Not Just Aim for "Nothing Changes"?

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

As with effectively everything else in this book, it depends. We'll readily commit to "For most projects, over a long enough time period, everything underneath them might need to be changed." If you have a project written in pure C with no external dependencies (or only external dependencies that promise great long-term stability, like POSIX), you might well be able to avoid any form of refactoring or difficult upgrade. C does a great job of providing stability—in many respects, that is its primary purpose.

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

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

Concerns like those just mentioned are why there are large risks for long-term projects that haven't invested in sustainability. We must be capable of responding to these sorts of issues and taking advantage of these opportunities, regardless of whether they directly affect us or manifest in only the transitive closure of technology we build upon. Change is not inherently good. We shouldn't change just for the sake of change. But we do need to be capable of change. If we allow for that eventual necessity, we should also consider whether to invest in making that capability cheap. As every system administrator knows, it's one thing to know in theory that you can recover from tape, and another to know in practice exactly how to do it and how much it will cost when it becomes necessary. Practice and expertise are great drivers of efficiency and reliability.

Scale and Efficiency

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

11 Beyer, B. et al. *Site Reliability Engineering: How Google Runs Production Systems*. (Boston: O'Reilly Media, 2016).

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

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

Finally, the most precious asset of a software organization—the codebase itself—also needs to scale. If your build system or version control system scales superlinearly over time, perhaps as a result of growth and increasing changelog history, a point might come at which you simply cannot proceed. Many questions, such as "How long does it take to do a full build?", "How long does it take to pull a fresh copy of the repository?", or "How much will it cost to upgrade to a new language version?" aren't actively monitored and change at a slow pace. They can easily become like the metaphorical boiled frog; it is far too easy for problems to worsen slowly and never manifest as a singular moment of crisis. Only with an organization-wide awareness and commitment to scaling are you likely to keep on top of these issues.

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

Policies That Don't Scale

With a little practice, it becomes easier to spot policies with bad scaling properties. Most commonly, these can be identified by considering the work imposed on a single

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

engineer and imagining the organization scaling up by 10 or 100 times. When we are 10 times larger, will we add 10 times more work with which our sample engineer needs to keep up? Does the amount of work our engineer must perform grow as a function of the size of the organization? Does the work scale up with the size of the codebase? If either of these are true, do we have any mechanisms in place to automate or optimize that work? If not, we have scaling problems.

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

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

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

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

Policies That Scale Well

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

One of our favorite internal policies is a great enabler of infrastructure teams, protecting their ability to make infrastructure changes safely. "If a product experiences outages or other problems as a result of infrastructure changes, but the issue wasn't surfaced by tests in our Continuous Integration (CI) system, it is not the fault of the infrastructure change." More colloquially, this is phrased as "If you liked it, you should have put a CI test on it," which we call "The Beyoncé Rule."[13] From a scaling perspective, the Beyoncé Rule implies that complicated, one-off bespoke tests that aren't triggered by our common CI system do not count. Without this, an engineer on an infrastructure team could conceivably need to track down every team with any affected code and ask them how to run their tests. We could do that when there were a hundred engineers. We definitely cannot afford to do that anymore.

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

Example: Compiler Upgrade

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

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

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

The most storied compiler upgrade in Google's history took place all the way back in 2006. At that point, we had been operating for a few years and had several thousand engineers on staff. We hadn't updated compilers in about five years. Most of our engineers had no experience with a compiler change. Most of our code had been exposed to only a single compiler version. It was a difficult and painful task for a team of (mostly) volunteers, which eventually became a matter of finding shortcuts and simplifications in order to work around upstream compiler and language changes that we didn't know how to adopt.[14] In the end, the 2006 compiler upgrade was extremely painful. Many Hyrum's Law problems, big and small, had crept into the codebase and served to deepen our dependency on a particular compiler version. Breaking those implicit dependencies was painful. The engineers in question were taking a risk: we didn't have the Beyoncé Rule yet, nor did we have a pervasive CI system, so it was difficult to know the impact of the change ahead of time or be sure they wouldn't be blamed for regressions.

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

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

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

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

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

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

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

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

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

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

15 Beyer et al. *Site Reliability Engineering: How Google Runs Production Systems*, Chapter 5, "Eliminating Toil."

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

Shifting Left

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

This term seems to have originated from arguments that security mustn't be deferred until the end of the development process, with requisite calls to "shift left on security." The argument in this case is relatively simple: if a security problem is discovered only after your product has gone to production, you have a very expensive problem. If it is 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.

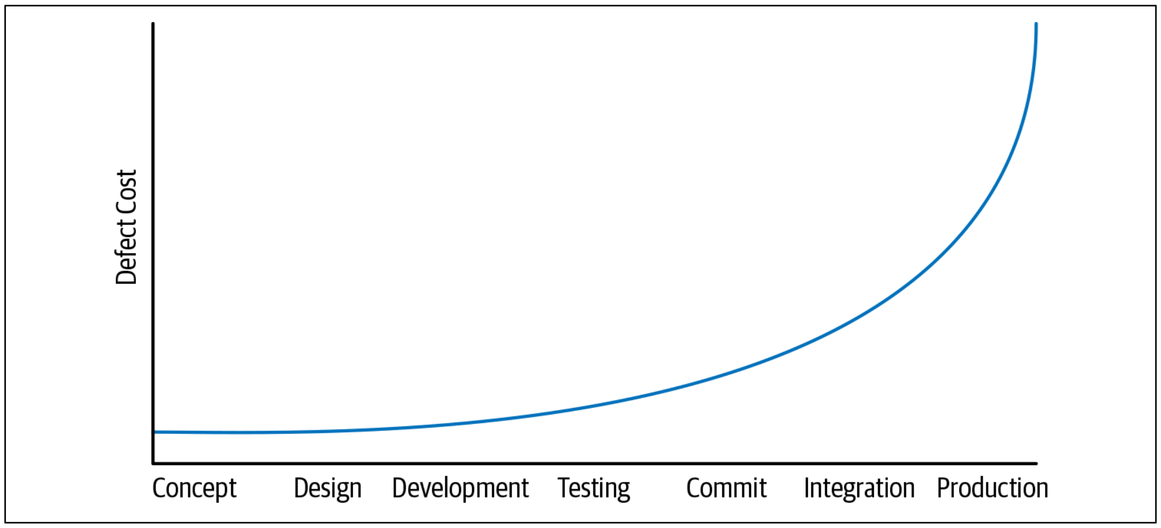

Figure 1-2. Timeline of the developer workflow

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, and 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, so we can assume a defense-in-depth approach, hopefully catching as many defects on the left side of the graph as possible.

Trade-offs and Costs

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

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

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

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

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

Example: Markers

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

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

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

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

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

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

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

Inputs to Decision Making

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

- All of the quantities involved are measurable or can at least be estimated. This usually means that we're evaluating trade-offs between CPU and network, or dollars and RAM, or considering whether to spend two weeks of engineer-time in order to save *N* CPUs across our datacenters.
- Some of the quantities are subtle, or we don't know how to measure them. Sometimes this manifests as "We don't know how much engineer-time this will take." Sometimes it is even more nebulous: how do you measure the engineering cost of a poorly designed API? Or the societal impact of a product choice?

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

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

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

Example: Distributed Builds

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

Back in the mid-2000s, Google relied purely on a local build system: you checked out code and you compiled it locally. We had massive local machines in some cases (you could build Maps on your desktop!), but compilation times became longer and longer as the codebase grew. Unsurprisingly, we incurred increasing overhead in personnel costs due to lost time, as well as increased resource costs for larger and more powerful local machines, and so on. These resource costs were particularly troublesome: of course we want people to have as fast a build as possible, but most of the time, a high-performance desktop development machine will sit idle. This doesn't feel like the proper way to invest those resources.

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

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

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

Example: Deciding Between Time and Scale

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

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

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

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

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

Revisiting Decisions, Making Mistakes

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

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

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

Software Engineering Versus Programming

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

Of course not. Our point is not that software engineering is superior, merely that these represent two different problem domains with distinct constraints, values, and best practices. Rather, the value in pointing out this difference comes from recognizing that some tools are great in one domain but not in the other. You probably don't need to rely on integration tests (see Chapter 14) and Continuous Deployment (CD) practices (see Chapter 24) for a project that will last only a few days. Similarly, all of our long-term concerns about semantic versioning (SemVer) and dependency management in software engineering projects (see Chapter 21) don't really apply for short-term programming projects: use whatever is available to solve the task at hand.

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

Conclusion

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

TL;DRs

- "Software engineering" differs from "programming" in dimensionality: programming is about producing code. Software engineering extends that to include the maintenance of that code for its useful life span.
- There is a factor of at least 100,000 times between the life spans of short-lived code and long-lived code. It is silly to assume that the same best practices apply universally on both ends of that spectrum.
- Software is sustainable when, for the expected life span of the code, we are capable of responding to changes in dependencies, technology, or product requirements. We may choose to not change things, but we need to be capable.
- Hyrum's Law: with a sufficient number of users of an API, it does not matter what you promise in the contract: all observable behaviors of your system will be depended on by somebody.
- Every task your organization has to do repeatedly should be scalable (linear or better) in terms of human input. Policies are a wonderful tool for making process scalable.
- Process inefficiencies and other software-development tasks tend to scale up slowly. Be careful about boiled-frog problems.
- Expertise pays off particularly well when combined with economies of scale.
- "Because I said so" is a terrible reason to do things.
- Being data driven is a good start, but in reality, most decisions are based on a mix of data, assumption, precedent, and argument. It's best when objective data makes up the majority of those inputs, but it can rarely be *all* of them.
- Being data driven over time implies the need to change directions when the data changes (or when assumptions are dispelled). Mistakes or revised plans are inevitable.