Chapter 22. Large-Scale Changes

Written by Hyrum Wright

Edited by Lisa Carey

Think for a moment about your own codebase. How many files can you reliably update in a single, simultaneous commit? What are the factors that constrain that number? Have you ever tried committing a change that large? Would you be able to do it in a reasonable amount of time in an emergency? How does your largest commit size compare to the actual size of your codebase? How would you test such a change? How many people would need to review the change before it is committed? Would you be able to roll back that change if it did get committed? The answers to these questions might surprise you (both what you *think* the answers are and what they actually turn out to be for your organization).

At Google, we've long ago abandoned the idea of making sweeping changes across our codebase in these types of large atomic changes. Our observation has been that, as a codebase and the number of engineers working in it grows, the largest atomic change possible counterintuitively *decreases*—running all affected presubmit checks and tests becomes difficult, to say nothing of even ensuring that every file in the change is up to date before submission. As it has become more difficult to make sweeping changes to our codebase, given our general desire to be able to continually improve underlying infrastructure, we've had to develop new ways of reasoning about large-scale changes and how to implement them.

In this chapter, we'll talk about the techniques, both social and technical, that enable us to keep the large Google codebase flexible and responsive to changes in underlying infrastructure. We'll also provide some real-life examples of how and where we've used these approaches. Although your codebase might not look like Google's, understanding these principles and adapting them locally will help your development organization scale while still being able to make broad changes across your codebase.

What Is a Large-Scale Change?

Before going much further, we should dig into what qualifies as a large-scale change (LSC). In our experience, an LSC is any set of changes that are logically related but cannot practically be submitted as a single atomic unit. This might be because it touches so many files that the underlying tooling can't commit them all at once, or it might be because the change is so large that it would always have merge conflicts. In many cases, an LSC is dictated by your repository topology: if your organization uses a collection of distributed or federated repositories,[1] making atomic changes across them might not even be technically possible.[2] We'll look at potential barriers to atomic changes in more detail later in this chapter.

LSCs at Google are almost always generated using automated tooling. Reasons for making an LSC vary, but the changes themselves generally fall into a few basic categories:

- Cleaning up common antipatterns using codebase-wide analysis tooling
- Replacing uses of deprecated library features
- Enabling low-level infrastructure improvements, such as compiler upgrades
- Moving users from an old system to a newer one[3]

The number of engineers working on these specific tasks in a given organization might be low, but it is useful for their customers to have insight into the LSC tools and process. By their very nature, LSCs will affect a large number of customers, and the LSC tools easily scale down to teams making only a few dozen related changes.

There can be broader motivating causes behind specific LSCs. For example, a new language standard might introduce a more efficient idiom for accomplishing a given task, an internal library interface might change, or a new compiler release might require fixing existing problems that would be flagged as errors by the new release. The majority of LSCs across Google actually have near-zero functional impact: they tend to be widespread textual updates for clarity, optimization, or future compatibility. But LSCs are not theoretically limited to this behavior-preserving/refactoring class of change.

In all of these cases, on a codebase the size of Google's, infrastructure teams might routinely need to change hundreds of thousands of individual references to the old pattern or symbol. In the largest cases so far, we've touched millions of references, and we expect the process to continue to scale well. Generally, we've found it advantageous to invest early and often in tooling to enable LSCs for the many teams doing infrastructure work. We've also found that efficient tooling also helps engineers performing smaller changes. The same tools that make changing thousands of files efficient also scale down to tens of files reasonably well.

Who Deals with LSCs?

As just indicated, the infrastructure teams that build and manage our systems are responsible for much of the work of performing LSCs, but the tools and resources are available across the company. If you skipped Chapter 1, you might wonder why infrastructure teams are the ones responsible for this work. Why can't we just introduce a new class, function, or system and dictate that everybody who uses the old one move to the updated analogue? Although this might seem easier in practice, it turns out not to scale very well for several reasons.

First, the infrastructure teams that build and manage the underlying systems are also the ones with the domain knowledge required to fix the hundreds of thousands of references to them. Teams that consume the infrastructure are unlikely to have the context for handling many of these migrations, and it is globally inefficient to expect them to each relearn expertise that infrastructure teams already have. Centralization also allows for faster recovery when faced with errors because errors generally fall into a small set of categories, and the team running the migration can have a playbook—formal or informal—for addressing them.

Consider the amount of time it takes to do the first of a series of semi-mechanical changes that you don't understand. You probably spend some time reading about the motivation and nature of the change, find an easy example, try to follow the provided suggestions, and then try to apply that to your local code. Repeating this for every team in an organization greatly increases the overall cost of execution. By making only a few centralized teams responsible for LSCs, Google both internalizes those costs and drives them down by making it possible for the change to happen more efficiently.

Second, nobody likes unfunded mandates.[4] Even though a new system might be categorically better than the one it replaces, those benefits are often diffused across an organization and thus unlikely to matter enough for individual teams to want to update on their own initiative. If the new system is important enough to migrate to, the costs of migration will be borne somewhere in the organization. Centralizing the migration and accounting for its costs is almost always faster and cheaper than depending on individual teams to organically migrate.

Additionally, having teams that own the systems requiring LSCs helps align incentives to ensure the change gets done. In our experience, organic migrations are unlikely to fully succeed, in part because engineers tend to use existing code as examples when writing new code. Having a team that has a vested interest in removing the old system responsible for the migration effort helps ensure that it actually gets done. Although funding and staffing a team to run these kinds of migrations can seem like an additional cost, it is actually just internalizing the externalities that an unfunded mandate creates, with the additional benefits of economies of scale.

CASE STUDY: FILLING POTHOLES

Although the LSC systems at Google are used for high-priority migrations, we've also discovered that just having them available opens up opportunities for various small fixes across our codebase, which just wouldn't have been possible without them. Much like transportation infrastructure tasks consist of building new roads as well as repairing old ones, infrastructure groups at Google spend a lot of time fixing existing code, in addition to developing new systems and moving users to them.

For example, early in our history, a template library emerged to supplement the C++ Standard Template Library. Aptly named the Google Template Library, this library consisted of several header files' worth of implementation. For reasons lost in the mists of time, one of these header files was named *stl_util.h* and another was named *map-util.h* (note the different separators in the file names). In addition to driving the consistency purists nuts, this difference also led to reduced productivity, and engineers had to remember which file used which separator, and only discovered when they got it wrong after a potentially lengthy compile cycle.

Although fixing this single-character change might seem pointless, particularly across a codebase the size of Google's, the maturity of our LSC tooling and process enabled us to do it with just a couple weeks' worth of background-task effort. Library authors could find and apply this change en masse without having to bother end users of these files, and we were able to quantitatively reduce the number of build failures caused by this specific issue. The resulting increases in productivity (and happiness) more than paid for the time to make the change.

As the ability to make changes across our entire codebase has improved, the diversity of changes has also expanded, and we can make some engineering decisions knowing that they aren't immutable in the future. Sometimes, it's worth the effort to fill a few potholes.

Barriers to Atomic Changes

Before we discuss the process that Google uses to actually effect LSCs, we should talk about why many kinds of changes can't be committed atomically. In an ideal world, all logical changes could be packaged into a single atomic

commit that could be tested, reviewed, and committed independent of other changes. Unfortunately, as a repository—and the number of engineers working in it—grows, that ideal becomes less feasible. It can be completely infeasible even at small scale when using a set of distributed or federated repositories.

Technical Limitations

To begin with, most Version Control Systems (VCSs) have operations that scale linearly with the size of a change. Your system might be able to handle small commits (e.g., on the order of tens of files) just fine, but might not have sufficient memory or processing power to atomically commit thousands of files at once. In centralized VCSs, commits can block other writers (and in older systems, readers) from using the system as they process, meaning that large commits stall other users of the system.

In short, it might not be just "difficult" or "unwise" to make a large change atomically: it might simply be impossible with a given 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 might be able to sneak in a change that touches every file in the repository on a weekend when nobody is doing development. Or you might have an informal system of grabbing the global repository lock by passing a virtual (or even physical!) token around your development team. At a large, global company like Google, these approaches are just not feasible: somebody is always making changes to the repository.

With few files in a change, the probability of merge conflicts shrinks, so they are more likely to be committed without problems. This property also holds for the following areas as well.

No Haunted Graveyards

The SREs who run Google's production services have a mantra: "No Haunted Graveyards." A haunted graveyard in this sense is a system that is so ancient, obtuse, or complex that no one dares enter it. Haunted graveyards are often business-critical systems that are frozen in time because any attempt to change them could cause the system to fail in incomprehensible ways, costing the business real money. They pose a real existential risk and can consume an inordinate amount of resources.

Haunted graveyards don't just exist in production systems, however; they can be found in codebases. Many organizations have bits of software that are old and unmaintained, written by someone long off the team, and on the critical path of some important revenue-generating functionality. These systems are also frozen in time, with layers of bureaucracy built up to prevent changes that might cause instability. Nobody wants to be the network support engineer II who flipped the wrong bit!

These parts of a codebase are anathema to the LSC process because they prevent the completion of large migrations, the decommissioning of other systems upon which they rely, or the upgrade of compilers or libraries that they use. From an LSC perspective, haunted graveyards prevent all kinds of meaningful progress.

At Google, we've found the counter to this to be good, ol'-fashioned testing. When software is thoroughly tested, we can make arbitrary changes to it and know with confidence whether those changes are breaking, no matter the age or complexity of the system. Writing those tests takes a lot of effort, but it allows a codebase like Google's to evolve over long periods of time, consigning the notion of haunted software graveyards to a graveyard of its own.

Heterogeneity

LSCs really work only when the bulk of the effort for them can be done by computers, not humans. As good as humans can be with ambiguity, computers rely upon consistent environments to apply the proper code transformations to the correct places. If your organization has many different VCSs, Continuous Integration (CI) systems, project-specific tooling, or formatting guidelines, it

is difficult to make sweeping changes across your entire codebase. Simplifying the environment to add more consistency will help both the humans who need to move around in it and the robots making automated transformations.

For example, many projects at Google have presubmit tests configured to run before changes are made to their codebase. Those checks can be very complex, ranging from checking new dependencies against a whitelist, to running tests, to ensuring that the change has an associated bug. Many of these checks are relevant for teams writing new features, but for LSCs, they just add additional irrelevant complexity.

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

NOTE

Many of the benefits of consistency for humans mentioned in Chapter 8 also apply to automated tooling.

Testing

Every change should be tested (a process we'll talk about more in just a moment), but the larger the change, the more difficult it is to actually test it appropriately. Google's CI system will run not only the tests immediately impacted by a change, but also any tests that transitively depend on the changed files.[6] This means a change gets broad coverage, but we've also observed that the farther away in the dependency graph a test is from the impacted files, the more unlikely a failure is to have been caused by the change itself.

Small, independent changes are easier to validate, because each of them affects a smaller set of tests, but also because test failures are easier to diagnose and fix. Finding the root cause of a test failure in a change of 25 files is pretty straightforward; finding 1 in a 10,000-file change is like the proverbial needle in a haystack.

The trade-off in this decision is that smaller changes will cause the same tests to be run multiple times, particularly tests that depend on large parts of the codebase. Because engineer time spent tracking down test failures is much more expensive than the compute time required to run these extra tests, we've made the conscious decision that this is a trade-off we're willing to make. That same trade-off might not hold for all organizations, but it is worth examining what the proper balance is for yours.

Adam Bender

Today it is common for a double-digit percentage (10% to 20%) of the changes in a project to be the result of LSCs, meaning a substantial amount of code is changed in projects by people whose full-time job is unrelated to those projects. Without good tests, such work would be impossible, and Google's codebase would quickly atrophy under its own weight. LSCs enable us to systematically migrate our entire codebase to newer APIs, deprecate older APIs, change language versions, and remove popular but dangerous practices.

Even a simple one-line signature change becomes complicated when made in a thousand different places across hundreds of different products and services.[7] After the change is written, you need to coordinate code reviews across dozens of teams. Lastly, after reviews are approved, you need to run as many tests as you can to be sure the change is safe.[8] We say "as many as you can," because a good-sized LSC could trigger a rerun of every single test at Google, and that can take a while. In fact, many LSCs have to plan time to catch downstream clients whose code backslides while the LSC makes its way through the process.

Testing an LSC can be a slow and frustrating process. When a change is sufficiently large, your local environment is almost guaranteed to be permanently out of sync with head as the codebase shifts like sand around your work. In such circumstances, it is easy to find yourself running and rerunning tests just to ensure your changes continue to be valid. When a project has flaky tests or is missing unit test coverage, it can require a lot of manual intervention and slow down the entire process. To help speed things up, we use a strategy called the TAP (Test Automation Platform) train.

RIDING THE TAP TRAIN

The core insight to LSCs is that they rarely interact with one another, and most affected tests are going to pass for most LSCs. As a result, we can test more than one change at a time and reduce the total number of tests executed. The train model has proven to be very effective for testing LSCs.

The TAP train takes advantage of two facts:

- LSCs tend to be pure refactorings and therefore very narrow in scope, preserving local semantics.

- Individual changes are often simpler and highly scrutinized, so they are correct more often than not.

The train model also has the advantage that it works for multiple changes at the same time and doesn't require that each individual change ride in isolation.[9]

The train has five steps and is started fresh every three hours:

1. For each change on the train, run a sample of 1,000 randomly-selected tests.
2. Gather up all the changes that passed their 1,000 tests and create one uber-change from all of them: "the train."
3. Run the union of all tests directly affected by the group of changes. Given a large enough (or low-level enough) LSC, this can mean running every single test in Google's repository. This process can take more than six hours to complete.
4. For each nonflaky test that fails, rerun it individually against each change that made it into the train to determine which changes caused it to fail.
5. TAP generates a report for each change that boarded the train. The report describes all passing and failing targets and can be used as evidence that an LSC is safe to submit.

Code Review

Finally, as we mentioned in Chapter 9, all changes need to be reviewed before submission, and this policy applies even for LSCs. Reviewing large commits can be tedious, onerous, and even error prone, particularly if the changes are generated by hand (a process you want to avoid, as we'll discuss shortly). In just a moment, we'll look at how tooling can often help in this space, but for some classes of changes, we still want humans to explicitly verify they are correct. Breaking an LSC into separate shards makes this much easier.

CASE STUDY: SCOPED_PTR TO STD::UNIQUE_PTR

Since its earliest days, Google's C++ codebase has had a self-destructing smart pointer for wrapping heap-allocated C++ objects and ensuring that they are destroyed when the smart pointer goes out of scope. This type was called `scoped_ptr` and was used extensively throughout Google's codebase to ensure that object lifetimes were appropriately managed. It wasn't perfect, but given the limitations of the then-current C++ standard (C++98) when the type was first introduced, it made for safer programs.

In C++11, the language introduced a new type: `std::unique_ptr`. It fulfilled the same function as `scoped_ptr`, but also prevented other classes of bugs that the language now could detect. `std::unique_ptr` was strictly better than `scoped_ptr`, yet Google's codebase had more than 500,000 references to `scoped_ptr` scattered among millions of source files. Moving to the more modern type required the largest LSC attempted to that point within Google.

Over the course of several months, several engineers attacked the problem in parallel. Using Google's large-scale migration infrastructure, we were able to change references to `scoped_ptr` into references to `std::unique_ptr` as well as slowly adapt `scoped_ptr` to behave more closely to `std::unique_ptr`. At the height of the migration process, we were consistently generating, testing and committing more than 700 independent changes, touching more than 15,000 files *per day*. Today, we sometimes manage 10 times that throughput, having refined our practices and improved our tooling.

Like almost all LSCs, this one had a very long tail of tracking down various nuanced behavior dependencies (another manifestation of Hyrum's Law), fighting race conditions with other engineers, and uses in generated code that weren't detectable by our automated tooling. We continued to work on these manually as they were discovered by the testing infrastructure.

`scoped_ptr` was also used as a parameter type in some widely used APIs, which made small independent changes difficult. We contemplated writing a call-graph analysis system that could change an API and its callers, transitively, in one commit, but were concerned that the resulting changes would themselves be too large to commit atomically.

In the end, we were able to finally remove `scoped_ptr` by first making it a type alias of `std::unique_ptr` and then performing the textual substitution between the old alias and the new, before eventually just removing the old `scoped_ptr` alias. Today, Google's codebase benefits from using the same standard type as the rest of the C++ ecosystem, which was possible only because of our technology and tooling for LSCs.

LSC Infrastructure

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

Policies and Culture

As we've described in Chapter 16, Google stores the bulk of its source code in a single monolithic repository (monorepo), and every engineer has visibility into almost all of this code. This high degree of openness means that any engineer can edit any file and send those edits for review to those who can approve them. However, each of those edits has costs, both to generate as well as review.[10]

Historically, these costs have been somewhat symmetric, which limited the scope of changes a single engineer or team could generate. As Google's LSC tooling improved, it became easier to generate a large number of changes very cheaply, and it became equally easy for a single engineer to impose a burden on a large number of reviewers across the company. Even though we want to encourage widespread improvements to our codebase, we want to make sure there is some oversight and thoughtfulness behind them, rather than indiscriminate tweaking.[11]

The end result is a lightweight approval process for teams and individuals seeking to make LSCs across Google. This process is overseen by a group of experienced engineers who are familiar with the nuances of various languages, as well as invited domain experts for the particular change in

question. The goal of this process is not to prohibit LSCs, but to help change authors produce the best possible changes, which make the most use of Google's technical and human capital. Occasionally, this group might suggest that a cleanup just isn't worth it: for example, cleaning up a common typo without any way of preventing recurrence.

Related to these policies was a shift in cultural norms surrounding LSCs. Although it is important for code owners to have a sense of responsibility for their software, they also needed to learn that LSCs were an important part of Google's effort to scale our software engineering practices. Just as product teams are the most familiar with their own software, library infrastructure teams know the nuances of the infrastructure, and getting product teams to trust that domain expertise is an important step toward social acceptance of LSCs. As a result of this culture shift, local product teams have grown to trust LSC authors to make changes relevant to those authors' domains.

Occasionally, local owners question the purpose of a specific commit being made as part of a broader LSC, and change authors respond to these comments just as they would other review comments. Socially, it's important that code owners understand the changes happening to their software, but they also have come to realize that they don't hold a veto over the broader LSC. Over time, we've found that a good FAQ and a solid historic track record of improvements have generated widespread endorsement of LSCs throughout Google.

Codebase Insight

To do LSCs, we've found it invaluable to be able to do large-scale analysis of our codebase, both on a textual level using traditional tools, as well as on a semantic level. For example, Google's use of the semantic indexing tool Kythe provides a complete map of the links between parts of our codebase, allowing us to ask questions such as "Where are the callers of this function?" or "Which classes derive from this one?" Kythe and similar tools also provide programmatic access to their data so that they can be incorporated into refactoring tools. (For further examples, see Chapters 17 and 20.)

We also use compiler-based indices to run abstract syntax tree-based analysis and transformations over our codebase. Tools such as ClangMR, JavacFlume, or Refaster, which can perform transformations in a highly parallelizable way, depend on these insights as part of their function. For smaller changes, authors

can use specialized, custom tools, `perl` or `sed`, regular expression matching, or even a simple shell script.

Whatever tool your organization uses for change creation, it's important that its human effort scale sublinearly with the codebase; in other words, it should take roughly the same amount of human time to generate the collection of all required changes, no matter the size of the repository. The change creation tooling should also be comprehensive across the codebase, so that an author can be assured that their change covers all of the cases they're trying to fix.

As with other areas in this book, an early investment in tooling usually pays off in the short to medium term. As a rule of thumb, we've long held that if a change requires more than 500 edits, it's usually more efficient for an engineer to learn and execute our change-generation tools rather than manually execute that edit. For experienced "code janitors," that number is often much smaller.

Change Management

Arguably the most important piece of large-scale change infrastructure is the set of tooling that shards a master change into smaller pieces and manages the process of testing, mailing, reviewing, and committing them independently. At Google, this tool is called Rosie, and we discuss its use more completely in a few moments when we examine our LSC process. In many respects, Rosie is not just a tool, but an entire platform for making LSCs at Google scale. It provides the ability to split the large sets of comprehensive changes produced by tooling into smaller shards, which can be tested, reviewed, and submitted independently.

Testing

Testing is another important piece of large-scale-change–enabling infrastructure. As discussed in Chapter 11, tests are one of the important ways that we validate our software will behave as expected. This is particularly important when applying changes that are not authored by humans. A robust testing culture and infrastructure means that other tooling can be confident that these changes don't have unintended effects.

Google's testing strategy for LSCs differs slightly from that of normal changes while still using the same underlying CI infrastructure. Testing LSCs

means not just ensuring the large master change doesn't cause failures, but that each shard can be submitted safely and independently. Because each shard can contain arbitrary files, we don't use the standard project-based presubmit tests. Instead, we run each shard over the transitive closure of every test it might affect, which we discussed earlier.

Language Support

LSCs at Google are typically done on a per-language basis, and some languages support them much more easily than others. We've found that language features such as type aliasing and forwarding functions are invaluable for allowing existing users to continue to function while we introduce new systems and migrate users to them non-atomically. For languages that lack these features, it is often difficult to migrate systems incrementally.[1][2]

We've also found that statically typed languages are much easier to perform large automated changes in than dynamically typed languages. Compiler-based tools along with strong static analysis provide a significant amount of information that we can use to build tools to affect LSCs and reject invalid transformations before they even get to the testing phase. The unfortunate result of this is that languages like Python, Ruby, and JavaScript that are dynamically typed are extra difficult for maintainers. Language choice is, in many respects, intimately tied to the question of code lifespan: languages that tend to be viewed as more focused on developer productivity tend to be more difficult to maintain. Although this isn't an intrinsic design requirement, it is where the current state of the art happens to be.

Finally, it's worth pointing out that automatic language formatters are a crucial part of the LSC infrastructure. Because we work toward optimizing our code for readability, we want to make sure that any changes produced by automated tooling are intelligible to both immediate reviewers and future readers of the code. All of the LSC-generation tools run the automated formatter appropriate to the language being changed as a separate pass so that the change-specific tooling does not need to concern itself with formatting specifics. Applying automated formatting, such as google-java-format or clang-format, to our codebase means that automatically produced changes will "fit in" with code written by a human, reducing future development friction. Without automated

formatting, large-scale automated changes would never have become the accepted status quo at Google.

CASE STUDY: OPERATION ROSEHUB

LSCs have become a large part of Google's internal culture, but they are starting to have implications in the broader world. Perhaps the best known case so far was "Operation RoseHub."

In early 2017, a vulnerability in the Apache Commons library allowed any Java application with a vulnerable version of the library in its transitive classpath to become susceptible to remote execution. This bug became known as the Mad Gadget. Among other things, it allowed an avaricious hacker to encrypt the San Francisco Municipal Transportation Agency's systems and shut down its operations. Because the only requirement for the vulnerability was having the wrong library somewhere in its classpath, anything that depended on even one of many open source projects on GitHub was vulnerable.

To solve this problem, some enterprising Googlers launched their own version of the LSC process. By using tools such as BigQuery, volunteers identified affected projects and sent more than 2,600 patches to upgrade their versions of the Commons library to one that addressed Mad Gadget. Instead of automated tools managing the process, more than 50 humans made this LSC work.

The LSC Process

With these pieces of infrastructure in place, we can now talk about the process for actually making an LSC. This roughly breaks down into four phases (with very nebulous boundaries between them):

1. Authorization
2. Change creation
3. Shard management
4. Cleanup

Typically, these steps happen after a new system, class, or function has been written, but it's important to keep them in mind during the design of the new

system. At Google, we aim to design successor systems with a migration path from older systems in mind, so that system maintainers can move their users to the new system automatically.

Authorization

We ask potential authors to fill out a brief document explaining the reason for a proposed change, its estimated impact across the codebase (i.e., how many smaller shards the large change would generate), and answers to any questions potential reviewers might have. This process also forces authors to think about how they will describe the change to an engineer unfamiliar with it in the form of an FAQ and proposed change description. Authors also get "domain review" from the owners of the API being refactored.

This proposal is then forwarded to an email list with about a dozen people who have oversight over the entire process. After discussion, the committee gives feedback on how to move forward. For example, one of the most common changes made by the committee is to direct all of the code reviews for an LSC to go to a single "global approver." Many first-time LSC authors tend to assume that local project owners should review everything, but for most mechanical LSCs, it's cheaper to have a single expert understand the nature of the change and build automation around reviewing it properly.

After the change is approved, the author can move forward in getting their change submitted. Historically, the committee has been very liberal with their approval,[13] and often gives approval not just for a specific change, but also for a broad set of related changes. Committee members can, at their discretion, fast-track obvious changes without the need for full deliberation.

The intent of this process is to provide oversight and an escalation path, without being too onerous for the LSC authors. The committee is also empowered as the escalation body for concerns or conflicts about an LSC: local owners who disagree with the change can appeal to this group who can then arbitrate any conflicts. In practice, this has rarely been needed.

Change Creation

After getting the required approval, an LSC author will begin to produce the actual code edits. Sometimes, these can be generated comprehensively into a single large global change that will be subsequently sharded into many

smaller independent pieces. Usually, the size of the change is too large to fit in a single global change, due to technical limitations of the underlying version control system.

The change generation process should be as automated as possible so that the parent change can be updated as users backslide into old uses[14] or textual merge conflicts occur in the changed code. Occasionally, for the rare case in which technical tools aren't able to generate the global change, we have sharded change generation across humans (see "Case Study: Operation RoseHub"). Although much more labor intensive than automatically generating changes, this allows global changes to happen much more quickly for time-sensitive applications.

Keep in mind that we optimize for human readability of our codebase, so whatever tool generates changes, we want the resulting changes to look as much like human-generated changes as possible. This requirement leads to the necessity of style guides and automatic formatting tools (see Chapter 8).[15]

Sharding and Submitting

After a global change has been generated, the author then starts running Rosie. Rosie takes a large change and shards it based upon project boundaries and ownership rules into changes that *can* be submitted atomically. It then puts each individually sharded change through an independent test-mail-submit pipeline. Rosie can be a heavy user of other pieces of Google's developer infrastructure, so it caps the number of outstanding shards for any given LSC, runs at lower priority, and communicates with the rest of the infrastructure about how much load it is acceptable to generate on our shared testing infrastructure.

We talk more about the specific test-mail-submit process for each shard below.

CATTLE VERSUS PETS

We often use the "cattle and pets" analogy when referring to individual machines in a distributed computing environment, but the same principles can apply to changes within a codebase.

At Google, as at most organizations, typical changes to the codebase are handcrafted by individual engineers working on specific features or bug fixes. Engineers might spend days or weeks working through the creation, testing, and review of a single change. They come to know the change intimately, and are proud when it is finally committed to the main repository. The creation of such a change is akin to owning and raising a favorite pet.

In contrast, effective handling of LSCs requires a high degree of automation and produces an enormous number of individual changes. In this environment, we've found it useful to treat specific changes as cattle: nameless and faceless commits that might be rolled back or otherwise rejected at any given time with little cost unless the entire herd is affected. Often this happens because of an unforeseen problem not caught by tests, or even something as simple as a merge conflict.

With a "pet" commit, it can be difficult to not take rejection personally, but when working with many changes as part of a large-scale change, it's just the nature of the job. Having automation means that tooling can be updated and new changes generated at very low cost, so losing a few cattle now and then isn't a problem.

Testing

Each independent shard is tested by running it through TAP, Google's CI framework. We run every test that depends on the files in a given change transitively, which often creates high load on our CI system.

This might sound computationally expensive, but in practice, the vast majority of shards affect fewer than one thousand tests, out of the millions across our codebase. For those that affect more, we can group them together: first running the union of all affected tests for all shards, and then for each individual shard running just the intersection of its affected tests with those that failed the first run. Most of these unions cause almost every test in the

codebase to be run, so adding additional changes to that batch of shards is nearly free.

One of the drawbacks of running such a large number of tests is that independent low-probability events are almost certainties at large enough scale. Flaky and brittle tests, such as those discussed in Chapter 11, which often don't harm the teams that write and maintain them, are particularly difficult for LSC authors. Although fairly low impact for individual teams, flaky tests can seriously affect the throughput of an LSC system. Automatic flake detection and elimination systems help with this issue, but it can be a constant effort to ensure that teams that write flaky tests are the ones that bear their costs.

In our experience with LSCs as semantic-preserving, machine-generated changes, we are now much more confident in the correctness of a single change than a test with any recent history of flakiness—so much so that recently flaky tests are now ignored when submitting via our automated tooling. In theory, this means that a single shard can cause a regression that is detected only by a flaky test going from flaky to failing. In practice, we see this so rarely that it's easier to deal with it via human communication rather than automation.

For any LSC process, individual shards should be committable independently. This means that they don't have any interdependence or that the sharding mechanism can group dependent changes (such as to a header file and its implementation) together. Just like any other change, large-scale change shards must also pass project-specific checks before being reviewed and committed.

Mailing reviewers

After Rosie has validated that a change is safe through testing, it mails the change to an appropriate reviewer. In a company as large as Google, with thousands of engineers, reviewer discovery itself is a challenging problem. Recall from Chapter 9 that code in the repository is organized with OWNERS files, which list users with approval privileges for a specific subtree in the repository. Rosie uses an owners detection service that understands these OWNERS files and weights each owner based upon their expected ability to review the specific shard in question. If a particular owner proves to be

unresponsive, Rosie adds additional reviewers automatically in an effort to get a change reviewed in a timely manner.

As part of the mailing process, Rosie also runs the per-project precommit tools, which might perform additional checks. For LSCs, we selectively disable certain checks such as those for nonstandard change description formatting. Although useful for individual changes on specific projects, such checks are a source of heterogeneity across the codebase and can add significant friction to the LSC process. This heterogeneity is a barrier to scaling our processes and systems, and LSC tools and authors can't be expected to understand special policies for each team.

We also aggressively ignore presubmit check failures that preexist the change in question. When working on an individual project, it's easy for an engineer to fix those and continue with their original work, but that technique doesn't scale when making LSCs across Google's codebase. Local code owners are responsible for having no preexisting failures in their codebase as part of the social contract between them and infrastructure teams.

Reviewing

As with other changes, changes generated by Rosie are expected to go through the standard code review process. In practice, we've found that local owners don't often treat LSCs with the same rigor as regular changes—they trust the engineers generating LSCs too much. Ideally these changes would be reviewed as any other, but in practice, local project owners have come to trust infrastructure teams to the point where these changes are often given only cursory review. We've come to only send changes to local owners for which their review is required for context, not just approval permissions. All other changes can go to a "global approver": someone who has ownership rights to approve *any* change throughout the repository.

When using a global approver, all of the individual shards are assigned to that person, rather than to individual owners of different projects. Global approvers generally have specific knowledge of the language and/or libraries they are reviewing and work with the large-scale change author to know what kinds of changes to expect. They know what the details of the change are and what potential failure modes for it might exist and can customize their workflow accordingly.

Instead of reviewing each change individually, global reviewers use a separate set of pattern-based tooling to review each of the changes and automatically approve ones that meet their expectations. Thus, they need to manually examine only a small subset that are anomalous because of merge conflicts or tooling malfunctions, which allows the process to scale very well.

Submitting

Finally, individual changes are committed. As with the mailing step, we ensure that the change passes the various project precommit checks before actually finally being committed to the repository.

With Rosie, we are able to effectively create, test, review, and submit thousands of changes per day across all of Google's codebase and have given teams the ability to effectively migrate their users. Technical decisions that used to be final, such as the name of a widely used symbol or the location of a popular class within a codebase, no longer need to be final.

Cleanup

Different LSCs have different definitions of "done," which can vary from completely removing an old system to migrating only high-value references and leaving old ones to organically disappear.[16] In almost all cases, it's important to have a system that prevents additional introductions of the symbol or system that the large-scale change worked hard to remove. At Google, we use the Tricorder framework mentioned in Chapters 20 and 19 to flag at review time when an engineer introduces a new use of a deprecated object, and this has proven an effective method to prevent backsliding. We talk more about the entire deprecation process in Chapter 15.

Conclusion

LSCs form an important part of Google's software engineering ecosystem. At design time, they open up more possibilities, knowing that some design decisions don't need to be as fixed as they once were. The LSC process also allows maintainers of core infrastructure the ability to migrate large swaths of Google's codebase from old systems, language versions, and library idioms to new ones, keeping the codebase consistent, spatially and temporally. And all

of this happens with only a few dozen engineers supporting tens of thousands of others.

No matter the size of your organization, it's reasonable to think about how you would make these kinds of sweeping changes across your collection of source code. Whether by choice or by necessity, having this ability will allow greater flexibility as your organization scales while keeping your source code malleable over time.

TL;DRs

- An LSC process makes it possible to rethink the immutability of certain technical decisions.
- Traditional models of refactoring break at large scales.
- Making LSCs means making a habit of making LSCs.

1 For some ideas about why, see Chapter 16.

2 It's possible in this federated world to say "we'll just commit to each repo as fast as possible to keep the duration of the build break small!" But that approach really doesn't scale as the number of federated repositories grows.

3 For a further discussion about this practice, see Chapter 15.

4 By "unfunded mandate," we mean "additional requirements imposed by an external entity without balancing compensation." Sort of like when the CEO says that everybody must wear an evening gown for "formal Fridays" but doesn't give you a corresponding raise to pay for your formal wear.

5 See *https://ieeexplore.ieee.org/abstract/document/8443579*.

6 This probably sounds like overkill, and it likely is. We're doing active research on the best way to determine the "right" set of tests for a given change, balancing the cost of compute time to run the tests, and the human cost of making the wrong choice.

7 The largest series of LSCs ever executed removed more than one billion lines of code from the repository over the course of three days. This was largely to remove an obsolete part of the repository that had been migrated to a new home; but still, how confident do you have to be to delete one billion lines of code?

8 LSCs are usually supported by tools that make finding, making, and reviewing changes relatively straight forward.

9 It is possible to ask TAP for single change “isolated” run, but these are very expensive and are performed only during off-peak hours.

10 There are obvious technical costs here in terms of compute and storage, but the human costs in time to review a change far outweigh the technical ones.

11 For example, we do not want the resulting tools to be used as a mechanism to fight over the proper spelling of “gray” or “grey” in comments.

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

13 The only kinds of changes that the committee has outright rejected have been those that are deemed dangerous, such as converting all `NULL` instances to `nullptr`, or extremely low-value, such as changing spelling from British English to American English, or vice versa. As our experience with such changes has increased and the cost of LSCs has dropped, the threshold for approval has as well.

14 This happens for many reasons: copy-and-paste from existing examples, committing changes that have been in development for some time, or simply reliance on old habits.

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

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