# Chapter 9. Code Review

*Written by Tom Manshreck and Caitlin Sadowski*

*Edited by Lisa Carey*

Code review is a process in which code is reviewed by someone other than the author, often before the introduction of that code into a codebase. Although that is a simple definition, implementations of the process of code review vary widely throughout the software industry. Some organizations have a select group of "gatekeepers" across the codebase that review changes. Others delegate code review processes to smaller teams, allowing different teams to require different levels of code review. At Google, essentially every change is reviewed before being committed, and every engineer is responsible for initiating reviews and reviewing changes.

Code reviews generally require a combination of a process and a tool supporting that process. At Google, we use a custom code review tool, Critique, to support our process.[1] Critique is an important enough tool at Google to warrant its own chapter in this book. This chapter focuses on the process of code review as it is practiced at Google rather than the specific tool, both because these foundations are older than the tool and because most of these insights can be adapted to whatever tool you might use for code review.

NOTE

For more information on Critique, see Chapter 19.

Some of the benefits of code review, such as detecting bugs in code before they enter a codebase, are well established[2] and somewhat obvious (if imprecisely measured). Other benefits, however, are more subtle. Because the code review process at Google is so ubiquitous and extensive, we've noticed many of these more subtle effects, including psychological ones, which provide many benefits to an organization over time and scale.

## Code Review Flow

Code reviews can happen at many stages of software development. At Google, code reviews take place before a change can be committed to the

codebase; this stage is also known as a *precommit review*. The primary end goal of a code review is to get another engineer to consent to the change, which we denote by tagging the change as "looks good to me." We use this "looks good to me" as a necessary permissions "bit" (combined with other bits noted below) to allow the change to be committed.

A typical code review at Google goes through the following steps:

1. A user writes a change to the codebase in their workspace. This *author* then creates a snapshot of the change: a patch and corresponding description that are uploaded to the code review tool. This change produces a *diff* against the codebase, which is used to evaluate what code has changed.
2. The author can use this initial patch to apply automated review comments or do self-review. When the author is satisfied with the diff of the change, they mail the change to one or more reviewers. This process notifies those reviewers asking them to view and comment on the snapshot.
3. *Reviewers* open the change in the code review tool and post comments on the diff. Some comments request explicit resolution. Some are merely informational.
4. The author modifies the change and uploads new snapshots based on the feedback and then replies back to the reviewers. Steps 3 and 4 may be repeated multiple times.
5. After the reviewers are happy with the latest state of the change, they agree to the change and accept it by marking it as "looks good to me." Only one "looks good to me" is required by default, although convention might request that all reviewers agree to the change.
6. After a change is marked "looks good to me," the author is allowed to commit the change to the codebase, provided they *resolve all comments* and that the change is *approved*. We'll cover approval in the next section.

We'll go over this process in more detail later in this chapter.

## Code Is a Liability

It's important to remember (and accept) that code itself is a liability. It might be a necessary liability, but by itself, code is simply a maintenance task to someone somewhere down the line. Much like the fuel that an airplane

carries, it has weight, though it is, of course, necessary for that airplane to fly.[3]

New features are often necessary, of course, but care should be taken before developing code in the first place to ensure that any new feature is warranted. Duplicated code not only is a wasted effort, it can actually cost more in time than not having the code at all; changes that could be easily performed under one code pattern often require more effort when there is duplication in the codebase. Writing entirely new code is so frowned upon, that some of us have a saying "If you're writing it from scratch, you're doing it wrong!"

This is especially true of library or utility code. Chances are, if you are writing a utility, someone else somewhere in a codebase the size of Google's has probably done something similar. Tools such as Chapter 17 are therefore critical for both finding such utility code and preventing the introduction of duplicate code. Ideally, this research is done beforehand, and a design for anything new has been communicated to the proper groups before any new code is written.

Of course, new features are often necessary. New projects happen, new techniques are introduced, new components are needed, and so on. All that said, a code review is not an occasion to rehash or debate previous design decisions. Design decisions often take time, requiring the circulation of design proposals, debate on the design in API reviews or some similar meetings, and so forth. and perhaps the development of prototypes. As much as a code review of entirely new code should not come out of the blue, the code review process itself should also not be viewed as an opportunity to revisit previous decisions.

## How Code Review Works at Google

We've pointed out roughly how the typical code review process works, but the devil is in the details. This section outlines in specific how code review works at Google and how these practices allow it to scale properly over time.

There are three aspects of review that require "approval" for any given change at Google:

- A correctness and comprehension check from another engineer that the code is appropriate and does what the author claims it does. This is often a team member, though it does not need to be. This is reflected in the "looks good to me" permissions "bit," which will be set after a peer reviewer agrees that the code "looks good" to them.

- Approval from one of the code owners that the code is appropriate for this particular part of the codebase (and can be checked into a particular directory). This approval might be implicit if the author is such an owner. Google's codebase is a tree structure with hierarchical owners of particular directories. (See XREF(Source Control)). Owners act as gatekeepers for their particular directories. A change might be proposed by any engineer and "looks good to me'ed" by any other engineer, but an owner of the directory in question must also *approve* this addition to their part of the codebase. Such an owner might be a tech lead or other engineer deemed expert in that particular area of the codebase. It's generally up to each team to decide how broadly or narrowly to assign ownership privileges.
- Approval from someone with language "readability"[4] that the code conforms to the language's style and best practices, checking whether the code is written in the manner we expect. This approval, again, might be implicit if the author has such readability. These engineers are pulled from a company-wide pool of engineers who have been granted readability in that programming language.

Although this level of control sounds onerous—and, admittedly, it sometimes is—most reviews have one person assuming all three roles, which speeds up the process quite a bit. Importantly, the author can also assume the latter two roles, needing only an "looks good to me" from another engineer to check code into their own codebase, provided they already have readability in that language (which owners often do).

NOTE

For more information on readability, see Chapter 3.

These requirements allow the code review process to be quite flexible. A tech lead who is an owner of a project and has that code's language readability can submit a code change with only a "looks good to me" from another engineer. An intern without such authority can submit the same change to the same codebase, provided they get approval from an owner with language readability. The three aforementioned permission "bits" can be combined in any combination. An author can even request more than one "looks good to me" from separate people by explicitly tagging the change as wanting a "looks good to me" from all reviewers.

In practice, most code reviews that require more than one approval usually go through a two-step process: gaining a "looks good to me" from a peer

engineer, and then seeking approval from appropriate code owner/readability reviewer(s). This allows the two roles to focus on different aspects of the code review and saves review time. The primary reviewer can focus on code correctness and the general validity of the code change; the code owner can focus on whether this change is appropriate for their part of the codebase without having to focus on the details of each line of code. An approver is often looking for something different than a peer reviewer, in other words. After all, someone is trying to check in code to their project/directory. They are more concerned with questions such as: "Will this code be easy or difficult to maintain?" "Does it add to my technical debt?" "Do we have the expertise to maintain it within our team?"

If all three of these types of reviews can be handled by one reviewer, why not just have those types of reviewers handle all code reviews? The short answer is scale. Separating the three roles adds flexibility to the code review process. If you are working with a peer on a new function within a utility library, you can get someone on your team to review the code for code correctness and comprehension. After several rounds (perhaps over several days), your code satisfies your peer reviewer and you get a "looks good to me." Now, you need only get an OWNER of the library (and owners often have appropriate readability) to approve the change.

## Ownership

*By Hyrum Wright*

When working on a small team in a dedicated repository, it's common to grant the entire team access to everything in the repository. After all, you know the other engineers, the domain is narrow enough that each of you can be experts, and small numbers constrain the effect of potential errors.

As the team grows larger, this approach can fail to scale. The result is either a messy repository split or a different approach to recording who has what knowledge and responsibilities in different parts of the repository. At Google, we call this set of knowledge and responsibilities *ownership* and the people to exercise them *owners*. This concept is different than possession of a collection of source code, but rather implies a sense of stewardship to act in the company's best interest with a section of the codebase. (Indeed, "stewards" would almost certainly be a better term if we had it to do over again.)

Specially named OWNERS files list usernames of people who have ownership responsibilities for a directory and its children. These files may also contain references to other OWNERS files or external access control lists, but eventually they resolve to a list of individuals. Each subdirectory may also contain a separate OWNERS file, and the relationship is hierarchically additive: a given file is generally owned by the union of the members of all the OWNERS files above it in the directory tree. OWNERS files may have as many entries as teams like, but we encourage a relatively small and focused list to ensure responsibility is clear.

Ownership of Google's code conveys approval rights for code within one's purview, but these rights also come with a set of responsibilities, such as understanding the code that is owned or knowing how to find somebody who does. Different teams have different criteria for granting ownership to new members, but we generally encourage them not to use ownership as a rite of initiation, and encourage departing members to yield ownership as soon as is practical.

This distributed ownership structure enables many of the other practices we've outlined in this book. For example, the set of people in the root OWNERS file can act as global approvers for large-scale changes (See Chapter 22), without having to bother local teams. Likewise, OWNERS files act as a kind of documentation, making it easy for people and tools to find those responsible for a given piece of code just by walking up the directory tree. When new projects are created, there's no central authority that has to register new ownership privileges: a new OWNERS file is sufficient.

This ownership mechanism is simple, yet powerful and has scaled well over the past two decades. It is one of the ways that Google ensures that tens of thousands of engineers can operate efficiently on billions of lines of code in a single repository.

## Code Review Benefits

Across the industry, code review itself is not controversial, though it is far from a universal practice. Many (maybe even most) other companies and open source projects have some form of code review, and most view the process as important as a sanity check on the introduction of new code into a codebase. Software engineers understand some of the more obvious benefits of code review, even if they might not personally think it applies in all cases.

But at Google, this process is generally more thorough and more wide-spread than at most other companies.

Google's culture, like that of a lot of software companies, is based on giving engineers wide latitude in how they do their jobs. There is a recognition that strict processes tend not to work well for a dynamic company needing to respond quickly to new technologies, and that bureaucratic rules tend not to work well with creative professionals. Code review, however, is a mandate, one of the few blanket processes in which all software engineers at Google must participate. Google requires code review for almost[5] every code change to the codebase, no matter how small. This mandate does have a cost and effect on engineering velocity given that it does slow down the introduction of new code into a codebase and can impact time-to-production for any given code change. (Both of these are common complaints by software engineers of strict code review processes.) Why, then, do we require this process? Why do we believe that this is a long-term benefit?

A well-designed code review process and a culture of taking code review seriously provides the following benefits:

- Checks code correctness
- Ensures the code change is comprehensible to other engineers
- Enforces consistency across the codebase
- Psychologically promotes team ownership
- Enables knowledge sharing
- Provides a historical record of the code review itself

Many of these benefits are critical to a software organization over time, and many of them are beneficial to not only the author but also the reviewers. The following sections go into more specifics for each of these items.

## Code Correctness

An obvious benefit of code review is that it allows a reviewer to check the "correctness" of the code change. Having another set of eyes look over a change helps ensure that the change does what was intended. Reviewers typically look for whether a change has proper testing, is properly designed and functions correctly and efficiently. In many cases, checking code correctness is checking whether the particular change can introduce bugs into the codebase.

Many reports point to the efficacy of code review in the prevention of future bugs in software. A study at IBM found that discovering defects earlier in a process, unsurprisingly, led to less time required to fix them later on.[6] The investment in the time for code review saved time otherwise spent in testing, debugging, and performing regressions, provided that the code review process itself was streamlined to keep it lightweight. This latter point is important; code review processes that are heavyweight, or don't scale properly, become unsustainable.[7] We will get into some best practices for keeping the process lightweight later in this chapter.

To prevent the evaluation of correctness from becoming more subjective than objective, authors are generally given deference to their particular approach, whether it be in the design or the function of the introduced change. A reviewer shouldn't propose alternatives because of personal opinion. Reviewers can propose alternatives, but only if they improve comprehension (by being less complex, for example) or functionality (by being more efficient, for example). In general, engineers are encouraged to approve changes that improve the codebase rather than wait for consensus on a more "perfect" solution. This focus tends to speed up code reviews.

As tooling becomes stronger, many correctness checks are performed automatically through techniques such as static analysis and automated testing (though tooling might never completely obviate the value for human-based inspection of code). (See Chapter 20 for more information). Though this tooling has its limits, it has definitely lessoned the need to rely on human-based code reviews for checking code correctness.

That said, checking for defects during the initial code review process is still an integral part of a general "shift left" strategy, aiming to discover and resolve issues at the earliest possible time, so that they don't require escalated costs and resources further down in the development cycle. A code review is not a panacea, nor the only check for such correctness, but is an element of a defense-in-depth against such problems in software. As a result, code review does not need to be "perfect" to achieve results.

Surprisingly enough, checking for code correctness is not the primary benefit Google accrues from the process of code review. Checking for code correctness generally ensures that a change works, but more importance is attached to ensuring that a code change is understandable and makes sense over time and as the codebase itself scales. To evaluate those aspects, we need to look at other factors other than whether the code is simply logically "correct" or understood.

## Comprehension of Code

A code review typically is the first opportunity for someone other than the author to inspect a change. This perspective allows a reviewer to do something that even the best engineer cannot do: provide feedback unbiased by an author's perspective. *A code review is often the first test of whether a given change is understandable to a broader audience*. This perspective is vitally important because code will be read many more times than it is written, and understanding and comprehension are critically important.

It is often useful to find a reviewer who has a different perspective from the author, especially a reviewer who might need, as part of their job, to maintain or use the code being proposed within your change. Unlike the deference reviewers should give authors regarding design decisions, it's often useful to treat questions on code comprehension using the maxim "the customer is always right." In some respect, any questions you get now will be multiplied many-fold over time, so view each question on code comprehension as valid. This doesn't mean that you need to change your approach or your logic in response to the criticism, but it does mean that you might need to explain it more clearly.

Together, the code correctness and code comprehension checks are the main criteria for an "looks good to me" from another engineer, which is one of the approval bits needed for an approved code review. When an engineer marks a code review as "looks good to me", they are saying that the code does what it says, and that it is understandable. Google, however, also requires that the code be sustainably maintained, so we have additional approvals needed for code in certain cases.

## Code Consistency

At scale, code that you write will be depended on, and eventually maintained, by someone else. Many others will need to read your code and understand what you did. Others (including automated tools) might need to refactor your code long after you've moved to another project. Code therefore needs to conform to some standards of consistency so that it can be understood and maintained. Code should also avoid being overly complex; simpler code is easier for others to understand and maintain, as well. Reviewers can assess how well this code lives up the standards of the codebase itself during code review. A code review therefore should act to ensure *code health*.

It is for maintainability that the "looks good to me" state of a code review (indicating code correctness and comprehension) is separated from that of readability approval. Readability approvals can be granted only by individuals who have successfully gone through the process of code readability training in a particular programming language. For example, Java code requires approval from an engineer who has "Java readability."

A readability approver is tasked with reviewing code to ensure that it follows agreed-on best practices for that particular programming language, is consistent with the codebase for that language within Google's code repository, and avoids being overly complex. Code that is consistent and simple is easier to understand and easier for tools to update when it comes time for refactoring, making it more resilient. If a particular pattern is always done in one fashion in the codebase, it's easier to write a tool to refactor it.

Additionally, code might be written only once, but it will be read dozens, hundreds, or even thousands of times. Having code that is consistent across the codebase improves comprehension for all of engineering, and this consistency even affects the process of code review itself. Consistency sometimes clashes with functionality; a readability reviewer may prefer a less complex change that may not be functionally "better," but is easier to understand.

With a more consistent codebase, it is easier for engineers to step in and review code on someone else's projects. Engineers might occasionally need to look outside the team for help in a code review. Being able to reach out and ask experts to review the code, knowing they can expect the code itself to be consistent, allows those engineers to focus more properly on code correctness and comprehension.

## Psychological and Cultural Benefits

Code review also has important cultural benefits: it reinforces to software engineers that code is not "theirs" but in fact part of a collective enterprise. Such psychological benefits can be subtle but are still important. Without code review, most engineers would naturally gravitate toward personal style and their own approach to software design. The code review process forces an author to not only let others have input, but to compromise for the sake of the greater good.

It is human nature to be proud of one's craft and to be reluctant to open up one's code to criticism by others. It is also natural to be somewhat reticent to welcome critical feedback about code that one writes. The code review

process provides a mechanism to mitigate what might otherwise be an emotionally charged interaction. Code review, when it works best, provides not only a challenge to an engineer's assumptions, but also does so in a prescribed, neutral manner, acting to temper any criticism which might otherwise be directed to the author if provided in an unsolicited manner. After all, the process *requires* critical review (we in fact call our code review tool "Critique"), so you can't fault a reviewer from doing their job and being critical. The code review process itself, therefore, can act as the "bad cop," whereas the reviewer can still be seen as the "good cop."

Of course, not all, or even most, engineers, need such psychological devices. But buffering such criticism through the process of code review often provides a much gentler introduction for most engineers to the expectations of the team. Many engineers joining Google, or a new team, are intimidated by code review. It is easy to think that any form of critical review reflects negatively on a person's job performance. But over time, almost all engineers come to expect to be challenged when sending a code review and come to value the advice and questions offered through this process (though, admittedly, this sometimes takes a while).

Another psychological benefit of code review is validation. Even the most capable engineers can suffer from imposter syndrome and be too self-critical. A process like code review acts as validation and recognition for one's work. Often, the process involves an exchange of ideas and knowledge sharing (covered in the next section), which benefits both the reviewer and the reviewee. As an engineer grows in their domain knowledge, it's sometimes difficult for them to get positive feedback on how they improve. The process of code review can provide that mechanism.

The process of initiating a code review also forces all authors to take a little extra care with their changes. Many software engineers are not perfectionists; most will admit that code that "gets the job done" is better than code that is perfect but that takes too long to develop. Without code review, it's natural that many of us would cut corners, even with the full intention of correcting such defects later. "Sure, I don't have all of the unit tests done, but I can do that later." A code review forces an engineer to resolve those issues before sending the change. Collecting the components of a change for code review psychologically forces an engineer to make sure that all of their ducks are in a row. The little moment of reflection that comes before sending off your change is the perfect time to read through your change and make sure you're not missing anything.

## Knowledge Sharing

One of the most important, but underrated, benefits of code review is in knowledge sharing. Most authors pick reviewers who are experts, or at least knowledgeable, in the area under review. The review process allows reviewers to impart domain knowledge to the author, allowing the reviewer(s) to offer suggestions, new techniques, or advisory information to the author. (Reviewers can even mark some comments "FYI," requiring no action; they are simply added as an aid to the author.) Authors who become particularly proficient in an area of the codebase will often become owners, as well, who then in turn will be able to act as reviewers for other engineers.

Part of the code review process of feedback and confirmation involves asking questions on why the change is done in a particular way. This exchange of information facilitates knowledge sharing. In fact, many code reviews involve an exchange of information both ways: the authors as well as the reviewers can learn new techniques and patterns from code review. At Google, reviewers may even directly share suggested edits with an author within the code review tool itself.

An engineer might may not read every email sent to them, but they tend to respond to every code review sent. This knowledge sharing can occur across time zones and projects, as well, using Google's scale to disseminate information quickly to engineers in all corners of the codebase. Code review is a perfect time for knowledge transfer: it is timely and actionable. (Many engineers at Google "meet" other engineers first through their code reviews!)

Given the amount of time Google engineers spend in code review, the knowledge accrued is quite significant. A Google engineer's primary task is still programming, of course, but a large chunk of their time is still spent in code review. The code review process provides one of the primary ways that software engineers interact with one another and exchange information about coding techniques. Often, new patterns are advertised within the context of code review, sometimes through refactorings such as large-scale changes.

Moreover, because each change becomes part of the codebase, code review acts as a historical record. Any engineer can inspect the Google codebase and determine when some particular pattern was introduced and bring up the actual code review in question. Often, that archeology provides insights to many more engineers than the original author and reviewer(s).

# Code Review Best Practices

Code review can, admittedly, introduce friction and delay to an organization. Most of these issues are not problems with code review, per se, but with their chosen implementation of code review. Keeping the code review process running smoothly at Google is no different, and it requires a number of best practices to ensure that code review is worth the effort put into the process. Most of those practices emphasize keeping the process nimble and quick so that code review can scale properly.

## Be Polite and Professional

As pointed out in the Culture section of this book, Google heavily fosters a culture of trust and respect. This filters down into our perspective on code review. A software engineer needs a "looks good to me" from only one other engineer to satisfy our requirement on code comprehension, for example. Many engineers make comments and "looks good to me" a change, with the understanding that the change can be submitted after those changes are made, without any additional rounds of review. That said, code reviews can introduce anxiety and stress to even the most capable engineers. It is critically important to keep all feedback and criticism firmly in the professional realm.

In general, reviewers should defer to authors on particular approaches, and only point out alternatives if the author's approach is deficient. If an author can demonstrate that several approaches are equally valid, the reviewer should accept the preference of the author. Even in those cases, if defects are found in an approach, consider the review a learning opportunity (for both sides!). All comments should remain strictly professional. Reviewers should be careful about jumping to conclusions based on a code author's particular approach. It's better to ask questions on why something was done the way it was before assuming that approach is wrong.

Reviewers should be prompt with their feedback. At Google, we expect feedback from a code review within 24 (working) hours. If a reviewer is unable to complete a review in that time, it's good practice (and expected) to respond that they've at least seen the change and will get to the review as soon as possible. Reviewers should avoid responding to the code review in piecemeal fashion. Few things annoy an author more than getting feedback from a review, addressing it, and then continuing to get unrelated further feedback in the review process.

As much as we expect professionalism on the part of the reviewer, we expect professionalism on the part of the author, as well. Remember that you are not your code, and that this change you propose is not "yours" but the team's. After you check that piece of code into the codebase, it is no longer yours in any case. Be receptive to questions on your approach, and be prepared to explain why you did things in certain ways. Remember that part of the responsibility of an author is to make sure this code is understandable and maintainable for the future.

It's important to treat each reviewer comment within a code review as a TODO item; a particular comment might not need to be accepted without question, but it should at least be addressed. If you disagree with a reviewer's comment, let them know, and let them know why and don't mark a comment as resolved until each side has had a chance to offer alternatives. One common way to keep such debates civil if an author doesn't agree with a reviewer is to offer an alternative and ask the reviewer to PTAL (please take another look). Remember that code review is a learning opportunity, for both the reviewer and the author. That insight often helps to mitigate any chances for disagreement.

By the same token, if you are an owner of code and responding to a code review within your codebase, be amenable to changes from an outside author. As long as the change is an improvement to the codebase, you should still give deference to the author that the change indicates something that could and should be improved.

## Write Small Changes

Probably the most important practice to keep the code review process nimble is to keep changes small. A code review should ideally be easy to digest and focus on a single issue, both for the reviewer and the author. Google's code review process discourages massive changes consisting of fully formed projects, and reviewers can rightfully reject such changes as being too large for a single review. Smaller changes also prevent engineers from wasting time waiting for reviews on larger changes, reducing downtime. These small changes have benefits further down in the software development process, as well. It is far easier to determine the source of a bug within a change if that particular change is small enough to narrow it down.

That said, it's important to acknowledge that a code review process that relies on small changes is sometimes difficult to reconcile with the introduction of major new features. A set of small, incremental code changes can be easier to

digest individually, but more difficult to comprehend within a larger scheme. Some engineers at Google admittedly are not fans of the preference given to small changes. Techniques exist for managing such code changes (development on integration branches, management of changes using a diff base different than HEAD), but those techniques inevitably involve more overhead. Consider the optimization for small changes just that: an optimization, and allow your process to accommodate the occasional larger change.

"Small" changes should generally be limited to about 200 lines of code. A small change should be easy on a reviewer and, almost as important, not be so cumbersome that additional changes are delayed waiting for an extensive review. Most changes at Google are expected to be reviewed within about a day.[8] (This doesn't necessarily mean that the review is over within a day, but that initial feedback is provided within a day.) About 35% of the changes at Google are to a single file.[9] Being easy on a reviewer allows for quicker changes to the codebase and benefits the author, as well. The author wants a quick review; waiting on an extensive review for a week or so would likely impact follow-on changes. A small initial review also can prevent much more expensive wasted effort on an incorrect approach further down the line.

Because code reviews are typically small, it's common for almost all code reviews at Google to be reviewed by one and only one person. Were that not the case—if a team were expected to weigh in on all changes to a common codebase—there is no way the process itself would scale. By keeping the code reviews small, we enable this optimization. It's not uncommon for multiple people to comment on any given change—most code reviews are sent to a team member, but also CC'd to appropriate teams—but the primary reviewer is still the one whose "looks good to me" is desired, and only one "looks good to me" is necessary for any given change. Any other comments, though important, are still optional.

Keeping changes small also allows the "approval" reviewers to more quickly approve any given changes. They can quickly inspect whether the primary code reviewer did due diligence and focus purely on whether this change augments the codebase while maintaining code health over time.

## Write Good Change Descriptions

A change description should indicate its type of change on the first line, as a summary. The first line is prime real estate and is used to provide summaries within the code review tool itself, to act as the subject line in any associated

emails, and become the visible line Google engineers see in a history summary within CodeSearch, so that first line is important.

Although the first line should be a summary of the entire change, the description should still go into detail on what is being changed *and why*. A description of "Bug fix" is not helpful to a reviewer or a future code archeologist. If several related modifications were made in the change (while still keeping it on message and small) enumerate them within a list. The description is the historical record for this change and tools such as CodeSearch allow you to find who wrote what line in any particular change in the codebase. Drilling down into the original change is often useful when trying to fix a bug.

Descriptions aren't the only opportunity for adding documentation to a change. When writing a public API, you generally don't want to leak implementation details, but by all means do so within the actual implementation, where you should comment liberally. If a reviewer does not understand why you did something, even if it is correct, it is a good indicator that such code needs better structure or better comments (or both). If, during the code review process, a new decision is reached, update the change description, or add appropriate comments within the implementation. A code review is not just something that you do in the present time; it is something you do to record what you did for posterity.

## Keep Reviewers to a Minimum

Most code reviews at Google are reviewed by precisely one reviewer.[10] Because the code review process allows the bits on code correctness, owner acceptance, and language readability to be handled by one individual, the code review process scales quite well across an organization the size of Google.

There is a tendency within the industry, and within individuals, to try to get additional input (and unanimous consent) from a cross-section of engineers. After all, each additional reviewer can add their own particular insight to the code review in question. But we've found that this leads to diminishing returns; the most important "looks good to me" is the first one, and subsequent ones don't add as much as you might think to the equation. The cost of additional reviewers quickly outweighs their value.

The code review process is optimized around the trust we place in our engineers to do the right thing. In certain cases, it can be useful to get a

particular change reviewed by multiple people, but even in those cases, those reviewers should focus on different aspects of the same change.

### Automate Where Possible

Code review is a human process, and that human input is important, but if there are components of the code process that can be automated, try to do so. Opportunities to automate mechanical human tasks should be explored; investments in proper tooling reap dividends. At Google, our code review tooling allows authors to automatically submit and automatically sync changes to the source control system upon approval (usually used for fairly simple changes).

One of the most important technological improvements regarding automation over the past few years is automatic static analysis of a given code change. (See Chapter 20). Rather than require authors to run tests, linters, or formatters, the current Google code review tooling provides most of that utility automatically through what is known as *presubmits*. A presubmit process is run when a change is initially sent to a reviewer. Before that change is sent, the presubmit process can detect a variety of problems with the existing change, reject the current change (and prevent sending an awkward email to a reviewer), and ask the original author to fix the change first. Such automation not only helps out with the code review process itself; it allows the reviewers to focus on more important concerns than formatting.

## Types of Code Reviews

All code reviews are not alike! Different types of code review require different levels of focus on the various aspects of the review process. Code changes at Google generally fall into one of the following buckets (though there is sometimes overlap):

- Greenfield reviews and new feature development
- Behavioral changes, improvements, and optimizations
- Bug fixes and rollbacks
- Refactorings and large-scale changes

### Greenfield Code Reviews

The least common type of code review is that of entirely new code, a so-called "greenfield" review. A greenfield review is the most important time to

evaluate whether the code will stand the test of time: that it will be easier to maintain as time and scale change the underlying assumptions of the code. Of course, the introduction of entirely new code should not come as a surprise. As mentioned earlier in this chapter, code is a liability, so the introduction of entirely new code should generally solve a real problem rather than simply provide yet another alternative. At Google, we generally require new code and/or projects to undergo an extensive design review, apart from a code review. A code review is not the time to debate design decisions already made in the past (and by the same token, a code review is not the time to introduce the design of a proposed API).

To ensure that code is sustainable, a greenfield review should ensure that an API matches an agreed design (which may require reviewing a design document) and is tested *fully*, with all API endpoints having some form of unit test, and that those tests fail when the code's assumptions change. (See Chapter 11). The code should also have proper owners (one of the first reviews in a new project is often of a single OWNERS file for the new directory), be sufficiently commented, and provide supplemental documentation, if needed. A greenfield review might also necessitate the introduction of a project into the continuous integration system. (See Chapter 23).

## Behavioral Changes, Improvements, and Optimizations

Most changes at Google generally fall into the broad category of modifications to existing code within the codebase. These additions may include modifications to API endpoints, improvements to existing implementations, or optimizations for other factors such as performance. Such changes are the bread and butter of most software engineers.

In each of these cases, the guidelines that apply to a greenfield review also apply: is this change necessary, and does this change improve the codebase? Some of the best modifications to a codebase are actually deletions! Getting rid of dead or obsolete code is one of the best ways to improve the overall code health of a codebase.

Any behavioral modifications should necessarily include revisions to appropriate tests for any new API behavior. Augmentations to the implementation should be tested in a Continuous Integration (CI) system to ensure that those modifications don't break any underlying assumptions of the existing tests. As well, optimizations should of course ensure that they don't affect those tests and might need to include performance benchmarks

for the reviewers to consult. Some optimizations might require benchmark tests, as well.

## Bug Fixes and Rollbacks

Inevitably, you will need to submit a change for a bug fix to your codebase. *When doing so, avoid the temptation to address other issues*. Not only does this risk increasing the size of the code review, it also makes it more difficult to perform regression testing or for others to rollback your change. A bug fix should focus solely on fixing the indicated bug, and (usually) updating associated tests to catch the error that occurred in the first place.

Addressing the bug with a revised test is often necessary. The bug surfaced because existing tests were either inadequate, or the code had certain assumptions which were not met. As a reviewer of a bug fix, it is important to ask for updates to unit tests if applicable.

Sometimes, a code change in a codebase as large as Google's causes some dependency to fail that was either not detected properly by tests, or unearths an untested part of the codebase. In those cases, Google allows such changes to be "rolled back," usually by the affected downstream customers. A rollback consists of a change that essentially undoes the previous change. Such rollbacks can be created in seconds because they just revert the previous change to a known state, but still require a code review.

It also becomes critically important that any change that could cause a potential rollback (and that includes all changes!) be as small and atomic as possible, so that a rollback, if needed, does not cause further breakages on other dependencies that can be difficult to untangle. At Google, we've seen developers start to depend on new code very quickly after it is submitted, and rollbacks sometimes break these developers as a result. Small changes help to mitigate these concerns, both because of their atomicity, and because reviews of small changes tend to be done quickly.

## Refactorings and Large-Scale Changes

Many changes at Google are automatically generated: the author of the change isn't a person, but a machine. We discuss more about the large-scale change (LSC) process in Chapter (Chapter 22), but even machine-generated changes require review. In cases where the change is considered low risk, they are reviewed by designated reviewers, who have approval privileges for our entire codebase. But for cases in which the change might be risky or

otherwise requires local domain expertise, individual engineers might be asked to review automatically generated changes as part of their normal workflow.

At first look, a review for an automatically generated change should be handled the same as any other code review: the reviewer should check for correctness and applicability of the change. However, we encourage reviewers to limit comments in the associated change and only flag concerns which are specific to their code, not the underlying tool or LSC generating the changes. While the specific change might be machine generated, the overall process generating these changes has already been reviewed, and individual teams cannot hold a veto over the process, or it would not be possible to scale such changes across the organization. If there is a concern about the underlying tool or process, reviewers can escalate out of band to an LSC oversight group for more information.

We also encourage reviewers of automatic changes to avoid expanding their scope. When reviewing a new feature or a change written by a teammate, it is often reasonable to ask the author to address related concerns within the same change, so long as the request still follows the earlier advice to keep the change small. This does not apply to automatically generated changes, because the human running the tool might have hundreds of changes in flight, and even a small percentage of changes with review comments or unrelated questions limits the scale at which the human can effectively operate the tool.

## Conclusion

Code review is one of the most important and critical processes at Google. Code review acts as the glue connecting engineers with one another, and the code review process is the primary developer workflow upon which almost all other processes must hang, from testing to static analysis to CI. A code review process must scale appropriately, and for that reason best practices, including small changes and rapid feedback and iteration, are important to maintain developer satisfaction and appropriate production velocity.

## TL;DRs

- Code review has many benefits, including ensuring code correctness, comprehension, and consistency across a codebase.

- Always check your assumptions through someone else: optimize for the reader.
- Provide the opportunity for critical feedback while remaining professional.
- Code review is important for knowledge sharing throughout an organization.
- Automation is critical for scaling the process.
- The code review itself provides a historical record.

1 We also use Gerrit to review Git code, primarily for our open source projects. However, Critique is the primary tool of a typical software engineer at Google.

2 McConnell, Steve. *Code Complete*. Microsoft Press, 2004.

3 *https://twitter.com/TitusWinters/status/1132640942878613515*

4 At Google, "readability" does not refer simply to comprehension, but to the set of styles and best practices that allow code to be maintainable to other engineers. See Chapter 3.

5 Some changes to documentation and configurations might not require a code review, but often it is still preferable to obtain such a review.

6 "Advances in Software Inspection," *IEEE Transactions on Software Engineering*, SE-12(7): 744–751, July 1986. Granted, this study took place before robust tooling and automated testing had become so important in the software development process, but the results still seem relevant in the modern software age.

7 Rigby, Peter C. and Christian Bird. 2013. Convergent software peer review practices. In FSE.

8 "Modern Code Review: A Case Study at Google" Caitlin Sadowski, Emma Söderberg, Luke Church, Michal Sipko, and Alberto Bacchelli.

9 Ibid.

10 "Modern Code Review: A Case Study at Google" Caitlin Sadowski, Emma Söderberg, Luke Church, Michal Sipko, and Alberto Bacchelli.