Google Generic Coding Good Practices Guidelines

September 4, 2024

Abstract

This document outlines the engineering practices followed at Google, which encompass guidelines applicable to all programming languages and projects. These practices, derived from extensive experience and expertise, aim to enhance the quality and efficiency of code reviews and version control processes. Key sections include detailed guides for code reviewers and change authors, offering insights into conducting effective code reviews and submitting high-quality changelists. The document also clarifies common terminologies, such as "changelist" (CL) and "LGTM" (Looks Good to Me), to bridge any understanding gaps for external audiences. This resource is intended to provide valuable best practices that can be adopted by open-source projects and other organizations.

Contents

1 Google Engineering Practices Documentation **3**
- 1.1 Terminology . . . 3
- 1.2 License . . . 3

2 Writing good CL descriptions **3**
- 2.1 First Line . . . 3
- 2.2 Body is Informative . . . 4
- 2.3 Bad CL Descriptions . . . 4
- 2.4 Good CL Descriptions . . . 4
  - 2.4.1 Functionality change . . . 4
  - 2.4.2 Refactoring . . . 5
  - 2.4.3 Small CL that needs some context . . . 5
- 2.5 Using tags . . . 5
- 2.6 Generated CL descriptions . . . 6
- 2.7 Review the description before submitting the CL . . . 6

3 How to handle reviewer comments **6**
- 3.1 Don't Take it Personally . . . 6
- 3.2 Fix the Code . . . 6
- 3.3 Think Collaboratively . . . 7
- 3.4 Resolving Conflicts . . . 7

4 The CL author's guide to getting through code review **7**

5 Small CLs **7**
- 5.1 Why Write Small CLs? . . . 7
- 5.2 What is Small? . . . 8
- 5.3 When are Large CLs Okay? . . . 8
- 5.4 Writing Small CLs Efficiently . . . 9
- 5.5 Splitting CLs . . . 9
  - 5.5.1 Stacking Multiple Changes on Top of Each Other . . . 9
  - 5.5.2 Splitting by Files . . . 9
  - 5.5.3 Splitting Horizontally . . . 9
  - 5.5.4 Splitting Vertically . . . 9
  - 5.5.5 Splitting Horizontally & Vertically . . . 10
- 5.6 Separate Out Refactorings . . . 10
- 5.7 Keep related test code in the same CL . . . 10
- 5.8 Don't Break the Build . . . 10
- 5.9 Can't Make it Small Enough . . . 11

6 Emergencies **11**
- 6.1 What Is An Emergency? . . . 11
- 6.2 What Is NOT An Emergency? . . . 11
- 6.3 What Is a Hard Deadline? . . . 11
- 6.4 Introduction . . . 12
- 6.5 What Do Code Reviewers Look For? . . . 12
  - 6.5.1 Picking the Best Reviewers . . . 12
  - 6.5.2 In-Person Reviews (and Pair Programming) . . . 12
- 6.6 See Also . . . 13

7 How to write code review comments **13**
- 7.1 Summary . . . 13
- 7.2 Courtesy . . . 13

7.3 Explain Why . . . 13
7.4 Giving Guidance . . . 13
7.5 Label comment severity . . . 14
7.6 Accepting Explanations . . . 14

8 How to do a code review **14**

9 What to look for in a code review **14**
9.1 Design . . . 14
9.2 Functionality . . . 15
9.3 Complexity . . . 15
9.4 Tests . . . 15
9.5 Naming . . . 15
9.6 Comments . . . 16
9.7 Style . . . 16
9.8 Consistency . . . 16
9.9 Documentation . . . 16
9.10 Every Line . . . 16
9.10.1 Exceptions . . . 17
9.11 Context . . . 17
9.12 Good Things . . . 17
9.13 Summary . . . 17

10 Navigating a CL in review **18**
10.1 Summary . . . 18
10.2 Step One: Take a broad view of the change . . . 18
10.3 Step Two: Examine the main parts of the CL . . . 18
10.4 Step Three: Look through the rest of the CL in an appropriate sequence . . . 19

11 Handling pushback in code reviews **19**
11.1 Who is right? . . . 19
11.2 Upsetting Developers . . . 19
11.3 Cleaning It Up Later . . . 19
11.4 General Complaints About Strictness . . . 20
11.5 Resolving Conflicts . . . 20

12 Speed of Code Reviews **20**
12.1 Why Should Code Reviews Be Fast? . . . 20
12.2 How Fast Should Code Reviews Be? . . . 20
12.3 Speed vs. Interruption . . . 21
12.4 Fast Responses . . . 21
12.5 Cross-Time-Zone Reviews . . . 21
12.6 LGTM With Comments . . . 21
12.7 Large CLs . . . 21
12.8 Code Review Improvements Over Time . . . 22
12.9 Emergencies . . . 22

13 The Standard of Code Review **22**
13.1 Mentoring . . . 23
13.2 Principles . . . 23
13.3 Resolving Conflicts . . . 23

1 Google Engineering Practices Documentation

Google has many generalized engineering practices that cover all languages and all projects. These documents represent our collective experience of various best practices that we have developed over time. It is possible that open source projects or other organizations would benefit from this knowledge, so we work to make it available publicly when possible.

Currently this contains the following documents:

- Google's Code Review Guidelines, which are actually two separate sets of documents:
  - The Code Reviewer's Guide
  - The Change Author's Guide

1.1 Terminology

There is some Google-internal terminology used in some of these documents, which we clarify here for external readers:

- **CL**: Stands for "changelist", which means one self-contained change that has been submitted to version control or which is undergoing code review. Other organizations often call this a "change", "patch", or "pull-request".
- **LGTM**: Means "Looks Good to Me". It is what a code reviewer says when approving a CL.

1.2 License

2 Writing good CL descriptions

A CL description is a public record of change, and it is important that it communicates:

1. **What** change is being made? This should summarize the major changes such that readers have a sense of what is being changed without needing to read the entire CL.
2. **Why** are these changes being made? What contexts did you have as an author when making this change? Were there decisions you made that aren't reflected in the source code? etc.

The CL description will become a permanent part of our version control history and will possibly be read by hundreds of people over the years.

Future developers will search for your CL based on its description. Someone in the future might be looking for your change because of a faint memory of its relevance but without the specifics handy. If all the important information is in the code and not the description, it's going to be a lot harder for them to locate your CL.

And then, after they find the CL, will they be able to understand *why* the change was made? Reading source code may reveal what the software is doing but it may not reveal why it exists, which can make it harder for future developers to know whether they can move Chesterton's fence.

A well-written CL description will help those future engineers – sometimes, including yourself!

2.1 First Line

- Short summary of what is being done.
- Complete sentence, written as though it was an order.
- Follow by empty line.

The **first line** of a CL description should be a short summary of *specifically* **what** *is being done by the CL*, followed by a blank line. This is what appears in version control history summaries, so it should be informative

enough that future code searchers don't have to read your CL or its whole description to understand what your CL actually *did* or how it differs from other CLs. That is, the first line should stand alone, allowing readers to skim through code history much faster.

Try to keep your first line short, focused, and to the point. The clarity and utility to the reader should be the top concern.

By tradition, the first line of a CL description is a complete sentence, written as though it were an order (an imperative sentence). For example, say "**Delete** the FizzBuzz RPC and **replace** it with the new system." instead of "**Deleting** the FizzBuzz RPC and **replacing** it with the new system." You don't have to write the rest of the description as an imperative sentence, though.

2.2 Body is Informative

The first line should be a short, focused summary, while the rest of the description should fill in the details and include any supplemental information a reader needs to understand the changelist holistically. It might include a brief description of the problem that's being solved, and why this is the best approach. If there are any shortcomings to the approach, they should be mentioned. If relevant, include background information such as bug numbers, benchmark results, and links to design documents.

If you include links to external resources consider that they may not be visible to future readers due to access restrictions or retention policies. Where possible include enough context for reviewers and future readers to understand the CL.

Even small CLs deserve a little attention to detail. Put the CL in context.

2.3 Bad CL Descriptions

"Fix bug" is an inadequate CL description. What bug? What did you do to fix it? Other similarly bad descriptions include:

- "Fix build."
- "Add patch."
- "Moving code from A to B."
- "Phase 1."
- "Add convenience functions."
- "kill weird URLs."

Some of those are real CL descriptions. Although short, they do not provide enough useful information.

2.4 Good CL Descriptions

Here are some examples of good descriptions.

2.4.1 Functionality change

Example:

> RPC: Remove size limit on RPC server message freelist.
>
> Servers like FizzBuzz have very large messages and would benefit from reuse. Make the freelist larger, and add a goroutine that frees the freelist entries slowly over time, so that idle servers eventually release all freelist entries.

The first few words describe what the CL actually does. The rest of the description talks about the problem being solved, why this is a good solution, and a bit more information about the specific implementation.

2.4.2 Refactoring

Example:

> Construct a Task with a TimeKeeper to use its TimeStr and Now methods.
>
> Add a Now method to Task, so the borglet() getter method can be removed (which was only used by OOMCandidate to call borglet's Now method). This replaces the methods on Borglet that delegate to a TimeKeeper.
>
> Allowing Tasks to supply Now is a step toward eliminating the dependency on Borglet. Eventually, collaborators that depend on getting Now from the Task should be changed to use a TimeKeeper directly, but this has been an accommodation to refactoring in small steps.
>
> Continuing the long-range goal of refactoring the Borglet Hierarchy.

The first line describes what the CL does and how this is a change from the past. The rest of the description talks about the specific implementation, the context of the CL, that the solution isn't ideal, and possible future direction. It also explains *why* this change is being made.

2.4.3 Small CL that needs some context

Example:

> Create a Python3 build rule for status.py.
>
> This allows consumers who are already using this as in Python3 to depend on a rule that is next to the original status build rule instead of somewhere in their own tree. It encourages new consumers to use Python3 if they can, instead of Python2, and significantly simplifies some automated build file refactoring tools being worked on currently.

The first sentence describes what's actually being done. The rest of the description explains *why* the change is being made and gives the reviewer a lot of context.

2.5 Using tags

Tags are manually entered labels that can be used to categorize CLs. These may be supported by tools or just used by team convention.

For example:

- "[tag]"
- "[a longer tag]"
- "#tag"
- "tag:"

Using tags is optional.

When adding tags, consider whether they should be in the body of the CL description or the first line. Limit the usage of tags in the first line, as this can obscure the content.

Examples with and without tags:

```
// Tags are okay in the first line if kept short.
[banana] Peel the banana before eating.

// Tags can be inlined in content.
Peel the #banana before eating.

// Tags are optional.
Peel the banana before eating.
```

```
// Multiple tags are acceptable if kept short.
#banana #apple: Assemble a fruit basket.

// Tags can go anywhere in the CL description.
> Assemble a fruit basket.
>
> #banana #apple

// Too many tags (or tags that are too long) overwhelm the first line.
//
// Instead, consider whether the tags can be moved into the description body
// and/or shortened.
[banana peeler factory factory][apple picking service] Assemble a fruit basket.
```

2.6 Generated CL descriptions

Some CLs are generated by tools. Whenever possible, their descriptions should also follow the advice here. That is, their first line should be short, focused, and stand alone, and the CL description body should include informative details that help reviewers and future code searchers understand each CL's effect.

2.7 Review the description before submitting the CL

CLs can undergo significant change during review. It can be worthwhile to review a CL description before submitting the CL, to ensure that the description still reflects what the CL does.

Next: Small CLs

3 How to handle reviewer comments

When you've sent a CL out for review, it's likely that your reviewer will respond with several comments on your CL. Here are some useful things to know about handling reviewer comments.

3.1 Don't Take it Personally

The goal of review is to maintain the quality of our codebase and our products. When a reviewer provides a critique of your code, think of it as their attempt to help you, the codebase, and Google, rather than as a personal attack on you or your abilities.

Sometimes reviewers feel frustrated and they express that frustration in their comments. This isn't a good practice for reviewers, but as a developer you should be prepared for this. Ask yourself, "What is the constructive thing that the reviewer is trying to communicate to me?" and then operate as though that's what they actually said.

Never respond in anger to code review comments. That is a serious breach of professional etiquette that will live forever in the code review tool. If you are too angry or annoyed to respond kindly, then walk away from your computer for a while, or work on something else until you feel calm enough to reply politely.

In general, if a reviewer isn't providing feedback in a way that's constructive and polite, explain this to them in person. If you can't talk to them in person or on a video call, then send them a private email. Explain to them in a kind way what you don't like and what you'd like them to do differently. If they also respond in a non-constructive way to this private discussion, or it doesn't have the intended effect, then escalate to your manager as appropriate.

3.2 Fix the Code

If a reviewer says that they don't understand something in your code, your first response should be to clarify the code itself. If the code can't be clarified, add a code comment that explains why the code is there. If a

comment seems pointless, only then should your response be an explanation in the code review tool.

If a reviewer didn't understand some piece of your code, it's likely other future readers of the code won't understand either. Writing a response in the code review tool doesn't help future code readers, but clarifying your code or adding code comments does help them.

3.3 Think Collaboratively

Writing a CL can take a lot of work. It's often really satisfying to finally send one out for review, feel like it's done, and be pretty sure that no further work is needed. It can be frustrating to receive comments asking for changes, especially if you don't agree with them.

At times like this, take a moment to step back and consider if the reviewer is providing valuable feedback that will help the codebase and Google. Your first question to yourself should always be, "Do I understand what the reviewer is asking for?"

If you can't answer that question, ask the reviewer for clarification.

And then, if you understand the comments but disagree with them, it's important to think collaboratively, not combatively or defensively:

```
Bad: "No, I'm not going to do that."
```

```
Good: "I went with X because of [these pros/cons] with [these tradeoffs]
My understanding is that using Y would be worse because of [these reasons].
Are you suggesting that Y better serves the original tradeoffs, that we should
weigh the tradeoffs differently, or something else?"
```

Remember, **courtesy and respect should always be a first priority**. If you disagree with the reviewer, find ways to collaborate: ask for clarifications, discuss pros/cons, and provide explanations of why your method of doing things is better for the codebase, users, and/or Google.

Sometimes, you might know something about the users, codebase, or CL that the reviewer doesn't know. Fix the code where appropriate, and engage your reviewer in discussion, including giving them more context. Usually you can come to some consensus between yourself and the reviewer based on technical facts.

3.4 Resolving Conflicts

Your first step in resolving conflicts should always be to try to come to consensus with your reviewer. If you can't achieve consensus, see The Standard of Code Review, which gives principles to follow in such a situation.

4 The CL author's guide to getting through code review

The pages in this section contain best practices for developers going through code review. These guidelines should help you get through reviews faster and with higher-quality results. You don't have to read them all, but they are intended to apply to every Google developer, and many people have found it helpful to read the whole set.

- Writing Good CL Descriptions
- Small CLs
- How to Handle Reviewer Comments

See also How to Do a Code Review, which gives detailed guidance for code reviewers.

5 Small CLs

5.1 Why Write Small CLs?

Small, simple CLs are:

- **Reviewed more quickly.** It's easier for a reviewer to find five minutes several times to review small CLs than to set aside a 30 minute block to review one large CL.
- **Reviewed more thoroughly.** With large changes, reviewers and authors tend to get frustrated by large volumes of detailed commentary shifting back and forth—sometimes to the point where important points get missed or dropped.
- **Less likely to introduce bugs.** Since you're making fewer changes, it's easier for you and your reviewer to reason effectively about the impact of the CL and see if a bug has been introduced.
- **Less wasted work if they are rejected.** If you write a huge CL and then your reviewer says that the overall direction is wrong, you've wasted a lot of work.
- **Easier to merge.** Working on a large CL takes a long time, so you will have lots of conflicts when you merge, and you will have to merge frequently.
- **Easier to design well.** It's a lot easier to polish the design and code health of a small change than it is to refine all the details of a large change.
- **Less blocking on reviews.** Sending self-contained portions of your overall change allows you to continue coding while you wait for your current CL in review.
- **Simpler to roll back.** A large CL will more likely touch files that get updated between the initial CL submission and a rollback CL, complicating the rollback (the intermediate CLs will probably need to be rolled back too).

Note that **reviewers have discretion to reject your change outright for the sole reason of it being too large.** Usually they will thank you for your contribution but request that you somehow make it into a series of smaller changes. It can be a lot of work to split up a change after you've already written it, or require lots of time arguing about why the reviewer should accept your large change. It's easier to just write small CLs in the first place.

5.2 What is Small?

In general, the right size for a CL is **one self-contained change**. This means that:

- The CL makes a minimal change that addresses **just one thing**. This is usually just one part of a feature, rather than a whole feature at once. In general it's better to err on the side of writing CLs that are too small vs. CLs that are too large. Work with your reviewer to find out what an acceptable size is.
- The CL should include related test code.
- Everything the reviewer needs to understand about the CL (except future development) is in the CL, the CL's description, the existing codebase, or a CL they've already reviewed.
- The system will continue to work well for its users and for the developers after the CL is checked in.
- The CL is not so small that its implications are difficult to understand. If you add a new API, you should include a usage of the API in the same CL so that reviewers can better understand how the API will be used. This also prevents checking in unused APIs.

There are no hard and fast rules about how large is "too large." 100 lines is usually a reasonable size for a CL, and 1000 lines is usually too large, but it's up to the judgment of your reviewer. The number of files that a change is spread across also affects its "size." A 200-line change in one file might be okay, but spread across 50 files it would usually be too large.

Keep in mind that although you have been intimately involved with your code from the moment you started to write it, the reviewer often has no context. What seems like an acceptably-sized CL to you might be overwhelming to your reviewer. When in doubt, write CLs that are smaller than you think you need to write. Reviewers rarely complain about getting CLs that are too small.

5.3 When are Large CLs Okay?

There are a few situations in which large changes aren't as bad:

- You can usually count deletion of an entire file as being just one line of change, because it doesn't take the reviewer very long to review.

- Sometimes a large CL has been generated by an automatic refactoring tool that you trust completely, and the reviewer's job is just to verify and say that they really do want the change. These CLs can be larger, although some of the caveats from above (such as merging and testing) still apply.

5.4 Writing Small CLs Efficiently

If you write a small CL and then you wait for your reviewer to approve it before you write your next CL, then you're going to waste a lot of time. So you want to find some way to work that won't block you while you're waiting for review. This could involve having multiple projects to work on simultaneously, finding reviewers who agree to be immediately available, doing in-person reviews, pair programming, or splitting your CLs in a way that allows you to continue working immediately.

5.5 Splitting CLs

When starting work that will have multiple CLs with potential dependencies among each other, it's often useful to think about how to split and organize those CLs at a high level before diving into coding.

Besides making things easier for you as an author to manage and organize your CLs, it also makes things easier for your code reviewers, which in turn makes your code reviews more efficient.

Here are some strategies for splitting work into different CLs.

5.5.1 Stacking Multiple Changes on Top of Each Other

One way to split up a CL without blocking yourself is to write one small CL, send it off for review, and then immediately start writing another CL *based* on the first CL. Most version control systems allow you to do this somehow.

5.5.2 Splitting by Files

Another way to split up a CL is by groupings of files that will require different reviewers but are otherwise self-contained changes.

For example: you send off one CL for modifications to a protocol buffer and another CL for changes to the code that uses that proto. You have to submit the proto CL before the code CL, but they can both be reviewed simultaneously. If you do this, you might want to inform both sets of reviewers about the other CL that you wrote, so that they have context for your changes.

Another example: you send one CL for a code change and another for the configuration or experiment that uses that code; this is easier to roll back too, if necessary, as configuration/experiment files are sometimes pushed to production faster than code changes.

5.5.3 Splitting Horizontally

Consider creating shared code or stubs that help isolate changes between layers of the tech stack. This not only helps expedite development but also encourages abstraction between layers.

For example: You created a calculator app with client, API, service, and data model layers. A shared proto signature can abstract the service and data model layers from each other. Similarly, an API stub can split the implementation of client code from service code and enable them to move forward independently. Similar ideas can also be applied to more granular function or class level abstractions.

5.5.4 Splitting Vertically

Orthogonal to the layered, horizontal approach, you can instead break down your code into smaller, full-stack, vertical features. Each of these features can be independent parallel implementation tracks. This enables some tracks to move forward while other tracks are awaiting review or feedback.

Back to our calculator example from Splitting Horizontally. You now want to support new operators, like multiplication and division. You could split this up by implementing multiplication and division as separate verticals or sub-features, even though they may have some overlap such as shared button styling or shared validation logic.

5.5.5 Splitting Horizontally & Vertically

To take this a step further, you could combine these approaches and chart out an implementation plan like this, where each cell is its own standalone CL. Starting from the model (at the bottom) and working up to the client:

Table 1: : : multiplication : | Model | Add proto definition | Add proto definition |

Layer	Feature: Multiplication	Feature: Division
Client	Add button	Add button
API	Add endpoint	Add endpoint
Service	Implement transformations	Share transformation logic with

5.6 Separate Out Refactorings

It's usually best to do refactorings in a separate CL from feature changes or bug fixes. For example, moving and renaming a class should be in a different CL from fixing a bug in that class. It is much easier for reviewers to understand the changes introduced by each CL when they are separate.

Small cleanups such as fixing a local variable name can be included inside of a feature change or bug fix CL, though. It's up to the judgment of developers and reviewers to decide when a refactoring is so large that it will make the review more difficult if included in your current CL.

5.7 Keep related test code in the same CL

CLs should include related test code. Remember that smallness here refers the conceptual idea that the CL should be focused and is not a simplistic function on line count.

Tests are expected for all Google changes.

A CL that adds or changes logic should be accompanied by new or updated tests for the new behavior. Pure refactoring CLs (that aren't intended to change behavior) should also be covered by tests; ideally, these tests already exist, but if they don't, you should add them.

Independent test modifications can go into separate CLs first, similar to the refactorings guidelines. That includes:

- Validating pre-existing, submitted code with new tests.
  - Ensures that important logic is covered by tests.
  - Increases confidence in subsequent refactorings on affected code. For example, if you want to refactor code that isn't already covered by tests, submitting test CLs *before* submitting refactoring CLs can validate that the tested behavior is unchanged before and after the refactoring.
- Refactoring the test code (e.g. introduce helper functions).
- Introducing larger test framework code (e.g. an integration test).

5.8 Don't Break the Build

If you have several CLs that depend on each other, you need to find a way to make sure the whole system keeps working after each CL is submitted. Otherwise you might break the build for all your fellow developers for a few minutes between your CL submissions (or even longer if something goes wrong unexpectedly with your later CL submissions).

5.9 Can't Make it Small Enough

Sometimes you will encounter situations where it seems like your CL *has* to be large. This is very rarely true. Authors who practice writing small CLs can almost always find a way to decompose functionality into a series of small changes.

Before writing a large CL, consider whether preceding it with a refactoring-only CL could pave the way for a cleaner implementation. Talk to your teammates and see if anybody has thoughts on how to implement the functionality in small CLs instead.

If all of these options fail (which should be extremely rare) then get consent from your reviewers in advance to review a large CL, so they are warned about what is coming. In this situation, expect to be going through the review process for a long time, be vigilant about not introducing bugs, and be extra diligent about writing tests.

Next: How to Handle Reviewer Comments

6 Emergencies

Sometimes there are emergency CLs that must pass through the entire code review process as quickly as possible.

6.1 What Is An Emergency?

An emergency CL would be a **small** change that: allows a major launch to continue instead of rolling back, fixes a bug significantly affecting users in production, handles a pressing legal issue, closes a major security hole, etc.

In emergencies we really do care about the speed of the entire code review process, not just the speed of response. In this case *only*, the reviewer should care more about the speed of the review and the correctness of the code (does it actually resolve the emergency?) than anything else. Also (perhaps obviously) such reviews should take priority over all other code reviews, when they come up.

However, after the emergency is resolved you should look over the emergency CLs again and give them a more thorough review.

6.2 What Is NOT An Emergency?

To be clear, the following cases are *not* an emergency:

- Wanting to launch this week rather than next week (unless there is some actual hard deadline for launch such as a partner agreement).
- The developer has worked on a feature for a very long time and they really want to get the CL in.
- The reviewers are all in another timezone where it is currently nighttime or they are away on an off-site.
- It is the end of the day on a Friday and it would just be great to get this CL in before the developer leaves for the weekend.
- A manager says that this review has to be complete and the CL checked in today because of a soft (not hard) deadline.
- Rolling back a CL that is causing test failures or build breakages.

And so on.

6.3 What Is a Hard Deadline?

A hard deadline is one where **something disastrous would happen** if you miss it. For example:

- Submitting your CL by a certain date is necessary for a contractual obligation.
- Your product will completely fail in the marketplace if not released by a certain date.

- Some hardware manufacturers only ship new hardware once a year. If you miss the deadline to submit code to them, that could be disastrous, depending on what type of code you're trying to ship.

Delaying a release for a week is not disastrous. Missing an important conference might be disastrous, but often is not.

Most deadlines are soft deadlines, not hard deadlines. They represent a desire for a feature to be done by a certain time. They are important, but you shouldn't be sacrificing code health to make them.

If you have a long release cycle (several weeks) it can be tempting to sacrifice code review quality to get a feature in before the next cycle. However, this pattern, if repeated, is a common way for projects to build up overwhelming technical debt. If developers are routinely submitting CLs near the end of the cycle that "must get in" with only superficial review, then the team should modify its process so that large feature changes happen early in the cycle and have enough time for good review.

6.4 Introduction

A code review is a process where someone other than the author(s) of a piece of code examines that code.

At Google, we use code review to maintain the quality of our code and products.

This documentation is the canonical description of Google's code review processes and policies.

This page is an overview of our code review process. There are two other large documents that are a part of this guide:

- **How To Do A Code Review**: A detailed guide for code reviewers.
- **The CL Author's Guide**: A detailed guide for developers whose CLs are going through review.

6.5 What Do Code Reviewers Look For?

Code reviews should look at:

- **Design**: Is the code well-designed and appropriate for your system?
- **Functionality**: Does the code behave as the author likely intended? Is the way the code behaves good for its users?
- **Complexity**: Could the code be made simpler? Would another developer be able to easily understand and use this code when they come across it in the future?
- **Tests**: Does the code have correct and well-designed automated tests?
- **Naming**: Did the developer choose clear names for variables, classes, methods, etc.?
- **Comments**: Are the comments clear and useful?
- **Style**: Does the code follow our style guides?
- **Documentation**: Did the developer also update relevant documentation?

See **How To Do A Code Review** for more information.

6.5.1 Picking the Best Reviewers

In general, you want to find the *best* reviewers you can who are capable of responding to your review within a reasonable period of time.

The best reviewer is the person who will be able to give you the most thorough and correct review for the piece of code you are writing. This usually means the owner(s) of the code, who may or may not be the people in the OWNERS file. Sometimes this means asking different people to review different parts of the CL.

If you find an ideal reviewer but they are not available, you should at least CC them on your change.

6.5.2 In-Person Reviews (and Pair Programming)

If you pair-programmed a piece of code with somebody who was qualified to do a good code review on it, then that code is considered reviewed.

You can also do in-person code reviews where the reviewer asks questions and the developer of the change speaks only when spoken to.

6.6 See Also

- How To Do A Code Review: A detailed guide for code reviewers.
- The CL Author's Guide: A detailed guide for developers whose CLs are going through review.

7 How to write code review comments

7.1 Summary

- Be kind.
- Explain your reasoning.
- Balance giving explicit directions with just pointing out problems and letting the developer decide.
- Encourage developers to simplify code or add code comments instead of just explaining the complexity to you.

7.2 Courtesy

In general, it is important to be courteous and respectful while also being very clear and helpful to the developer whose code you are reviewing. One way to do this is to be sure that you are always making comments about the *code* and never making comments about the *developer*. You don't always have to follow this practice, but you should definitely use it when saying something that might otherwise be upsetting or contentious. For example:

Bad: "Why did **you** use threads here when there's obviously no benefit to be gained from concurrency?"

Good: "The concurrency model here is adding complexity to the system without any actual performance benefit that I can see. Because there's no performance benefit, it's best for this code to be single-threaded instead of using multiple threads."

7.3 Explain Why

One thing you'll notice about the "good" example from above is that it helps the developer understand *why* you are making your comment. You don't always need to include this information in your review comments, but sometimes it's appropriate to give a bit more explanation around your intent, the best practice you're following, or how your suggestion improves code health.

7.4 Giving Guidance

In general it is the developer's responsibility to fix a CL, not the reviewer's. You are not required to do detailed design of a solution or write code for the developer.

This doesn't mean the reviewer should be unhelpful, though. In general you should strike an appropriate balance between pointing out problems and providing direct guidance. Pointing out problems and letting the developer make a decision often helps the developer learn, and makes it easier to do code reviews. It also can result in a better solution, because the developer is closer to the code than the reviewer is.

However, sometimes direct instructions, suggestions, or even code are more helpful. The primary goal of code review is to get the best CL possible. A secondary goal is improving the skills of developers so that they require less and less review over time.

Remember that people learn from reinforcement of what they are doing well and not just what they could do better. If you see things you like in the CL, comment on those too! Examples: developer cleaned up a messy algorithm, added exemplary test coverage, or you as the reviewer learned something from the CL. Just as with all comments, include why you liked something, further encouraging the developer to continue good practices.

7.5 Label comment severity

Consider labeling the severity of your comments, differentiating required changes from guidelines or suggestions.

Here are some examples:

> Nit: This is a minor thing. Technically you should do it, but it won't hugely impact things.
>
> Optional (or Consider): I think this may be a good idea, but it's not strictly required.
>
> FYI: I don't expect you to do this in this CL, but you may find this interesting to think about for the future.

This makes review intent explicit and helps authors prioritize the importance of various comments. It also helps avoid misunderstandings; for example, without comment labels, authors may interpret all comments as mandatory, even if some comments are merely intended to be informational or optional.

7.6 Accepting Explanations

If you ask a developer to explain a piece of code that you don't understand, that should usually result in them **rewriting the code more clearly**. Occasionally, adding a comment in the code is also an appropriate response, as long as it's not just explaining overly complex code.

Explanations written only in the code review tool are not helpful to future code readers. They are acceptable only in a few circumstances, such as when you are reviewing an area you are not very familiar with and the developer explains something that normal readers of the code would have already known.

Next: Handling Pushback in Code Reviews

8 How to do a code review

The pages in this section contain recommendations on the best way to do code reviews, based on long experience. All together they represent one complete document, broken up into many separate sections. You don't have to read them all, but many people have found it very helpful to themselves and their team to read the entire set.

- The Standard of Code Review
- What to Look For In a Code Review
- Navigating a CL in Review
- Speed of Code Reviews
- How to Write Code Review Comments
- Handling Pushback in Code Reviews

See also the CL Author's Guide, which gives detailed guidance to developers whose CLs are undergoing review.

9 What to look for in a code review

Note: Always make sure to take into account The Standard of Code Review when considering each of these points.

9.1 Design

The most important thing to cover in a review is the overall design of the CL. Do the interactions of various pieces of code in the CL make sense? Does this change belong in your codebase, or in a library? Does it integrate well with the rest of your system? Is now a good time to add this functionality?

9.2 Functionality

Does this CL do what the developer intended? Is what the developer intended good for the users of this code? The "users" are usually both end-users (when they are affected by the change) and developers (who will have to "use" this code in the future).

Mostly, we expect developers to test CLs well-enough that they work correctly by the time they get to code review. However, as the reviewer you should still be thinking about edge cases, looking for concurrency problems, trying to think like a user, and making sure that there are no bugs that you see just by reading the code.

You *can* validate the CL if you want—the time when it's most important for a reviewer to check a CL's behavior is when it has a user-facing impact, such as a **UI change**. It's hard to understand how some changes will impact a user when you're just reading the code. For changes like that, you can have the developer give you a demo of the functionality if it's too inconvenient to patch in the CL and try it yourself.

Another time when it's particularly important to think about functionality during a code review is if there is some sort of **parallel programming** going on in the CL that could theoretically cause deadlocks or race conditions. These sorts of issues are very hard to detect by just running the code and usually need somebody (both the developer and the reviewer) to think through them carefully to be sure that problems aren't being introduced. (Note that this is also a good reason not to use concurrency models where race conditions or deadlocks are possible—it can make it very complex to do code reviews or understand the code.)

9.3 Complexity

Is the CL more complex than it should be? Check this at every level of the CL—are individual lines too complex? Are functions too complex? Are classes too complex? "Too complex" usually means **"can't be understood quickly by code readers."** It can also mean **"developers are likely to introduce bugs when they try to call or modify this code."**

A particular type of complexity is **over-engineering**, where developers have made the code more generic than it needs to be, or added functionality that isn't presently needed by the system. Reviewers should be especially vigilant about over-engineering. Encourage developers to solve the problem they know needs to be solved *now*, not the problem that the developer speculates *might* need to be solved in the future. The future problem should be solved once it arrives and you can see its actual shape and requirements in the physical universe.

9.4 Tests

Ask for unit, integration, or end-to-end tests as appropriate for the change. In general, tests should be added in the same CL as the production code unless the CL is handling an emergency.

Make sure that the tests in the CL are correct, sensible, and useful. Tests do not test themselves, and we rarely write tests for our tests—a human must ensure that tests are valid.

Will the tests actually fail when the code is broken? If the code changes beneath them, will they start producing false positives? Does each test make simple and useful assertions? Are the tests separated appropriately between different test methods?

Remember that tests are also code that has to be maintained. Don't accept complexity in tests just because they aren't part of the main binary.

9.5 Naming

Did the developer pick good names for everything? A good name is long enough to fully communicate what the item is or does, without being so long that it becomes hard to read.

9.6 Comments

Did the developer write clear comments in understandable English? Are all of the comments actually necessary? Usually comments are useful when they **explain why** some code exists, and should not be explaining *what* some code is doing. If the code isn't clear enough to explain itself, then the code should be made simpler. There are some exceptions (regular expressions and complex algorithms often benefit greatly from comments that explain what they're doing, for example) but mostly comments are for information that the code itself can't possibly contain, like the reasoning behind a decision.

It can also be helpful to look at comments that were there before this CL. Maybe there is a TODO that can be removed now, a comment advising against this change being made, etc.

Note that comments are different from *documentation* of classes, modules, or functions, which should instead express the purpose of a piece of code, how it should be used, and how it behaves when used.

9.7 Style

We have style guides at Google for all of our major languages, and even for most of the minor languages. Make sure the CL follows the appropriate style guides.

If you want to improve some style point that isn't in the style guide, prefix your comment with "Nit:" to let the developer know that it's a nitpick that you think would improve the code but isn't mandatory. Don't block CLs from being submitted based only on personal style preferences.

The author of the CL should not include major style changes combined with other changes. It makes it hard to see what is being changed in the CL, makes merges and rollbacks more complex, and causes other problems. For example, if the author wants to reformat the whole file, have them send you just the reformatting as one CL, and then send another CL with their functional changes after that.

9.8 Consistency

What if the existing code is inconsistent with the style guide? Per our code review principles, the style guide is the absolute authority: if something is required by the style guide, the CL should follow the guidelines.

In some cases, the style guide makes recommendations rather than declaring requirements. In these cases, it's a judgment call whether the new code should be consistent with the recommendations or the surrounding code. Bias towards following the style guide unless the local inconsistency would be too confusing.

If no other rule applies, the author should maintain consistency with the existing code.

Either way, encourage the author to file a bug and add a TODO for cleaning up existing code.

9.9 Documentation

If a CL changes how users build, test, interact with, or release code, check to see that it also updates associated documentation, including READMEs, g3doc pages, and any generated reference docs. If the CL deletes or deprecates code, consider whether the documentation should also be deleted. If documentation is missing, ask for it.

9.10 Every Line

In the general case, look at *every* line of code that you have been assigned to review. Some things like data files, generated code, or large data structures you can scan over sometimes, but don't scan over a human-written class, function, or block of code and assume that what's inside of it is okay. Obviously some code deserves more careful scrutiny than other code—that's a judgment call that you have to make—but you should at least be sure that you *understand* what all the code is doing.

If it's too hard for you to read the code and this is slowing down the review, then you should let the developer know that and wait for them to clarify it before you try to review it. At Google, we hire great software

engineers, and you are one of them. If you can't understand the code, it's very likely that other developers won't either. So you're also helping future developers understand this code, when you ask the developer to clarify it.

If you understand the code but you don't feel qualified to do some part of the review, make sure there is a reviewer on the CL who is qualified, particularly for complex issues such as privacy, security, concurrency, accessibility, internationalization, etc.

9.10.1 Exceptions

What if it doesn't make sense for you to review every line? For example, you are one of multiple reviewers on a CL and may be asked:

- To review only certain files that are part of a larger change.
- To review only certain aspects of the CL, such as the high-level design, privacy or security implications, etc.

In these cases, note in a comment which parts you reviewed. Prefer giving LGTM with comments .

If you instead wish to grant LGTM after confirming that other reviewers have reviewed other parts of the CL, note this explicitly in a comment to set expectations. Aim to respond quickly once the CL has reached the desired state.

9.11 Context

It is often helpful to look at the CL in a broad context. Usually the code review tool will only show you a few lines of code around the parts that are being changed. Sometimes you have to look at the whole file to be sure that the change actually makes sense. For example, you might see only four new lines being added, but when you look at the whole file, you see those four lines are in a 50-line method that now really needs to be broken up into smaller methods.

It's also useful to think about the CL in the context of the system as a whole. Is this CL improving the code health of the system or is it making the whole system more complex, less tested, etc.? **Don't accept CLs that degrade the code health of the system.** Most systems become complex through many small changes that add up, so it's important to prevent even small complexities in new changes.

9.12 Good Things

If you see something nice in the CL, tell the developer, especially when they addressed one of your comments in a great way. Code reviews often just focus on mistakes, but they should offer encouragement and appreciation for good practices, as well. It's sometimes even more valuable, in terms of mentoring, to tell a developer what they did right than to tell them what they did wrong.

9.13 Summary

In doing a code review, you should make sure that:

- The code is well-designed.
- The functionality is good for the users of the code.
- Any UI changes are sensible and look good.
- Any parallel programming is done safely.
- The code isn't more complex than it needs to be.
- The developer isn't implementing things they *might* need in the future but don't know they need now.
- Code has appropriate unit tests.
- Tests are well-designed.
- The developer used clear names for everything.
- Comments are clear and useful, and mostly explain *why* instead of *what*.
- Code is appropriately documented (generally in g3doc).

- The code conforms to our style guides.

Make sure to review **every line** of code you've been asked to review, look at the **context**, make sure you're **improving code health**, and compliment developers on **good things** that they do.

Next: Navigating a CL in Review

10 Navigating a CL in review

10.1 Summary

Now that you know what to look for, what's the most efficient way to manage a review that's spread across multiple files?

1. Does the change make sense? Does it have a good description?
2. Look at the most important part of the change first. Is it well-designed overall?
3. Look at the rest of the CL in an appropriate sequence.

10.2 Step One: Take a broad view of the change

Look at the CL description and what the CL does in general. Does this change even make sense? If this change shouldn't have happened in the first place, please respond immediately with an explanation of why the change should not be happening. When you reject a change like this, it's also a good idea to suggest to the developer what they should have done instead.

For example, you might say "Looks like you put some good work into this, thanks! However, we're actually going in the direction of removing the FooWidget system that you're modifying here, and so we don't want to make any new modifications to it right now. How about instead you refactor our new BarWidget class?"

Note that not only did the reviewer reject the current CL and provide an alternative suggestion, but they did it *courteously*. This kind of courtesy is important because we want to show that we respect each other as developers even when we disagree.

If you get more than a few CLs that represent changes you don't want to make, you should consider re-working your team's development process or the posted process for external contributors so that there is more communication before CLs are written. It's better to tell people "no" before they've done a ton of work that now has to be thrown away or drastically re-written.

10.3 Step Two: Examine the main parts of the CL

Find the file or files that are the "main" part of this CL. Often, there is one file that has the largest number of logical changes, and it's the major piece of the CL. Look at these major parts first. This helps give context to all of the smaller parts of the CL, and generally accelerates doing the code review. If the CL is too large for you to figure out which parts are the major parts, ask the developer what you should look at first, or ask them to split up the CL into multiple CLs.

If you see some major design problems with this part of the CL, you should send those comments immediately, even if you don't have time to review the rest of the CL right now. In fact, reviewing the rest of the CL might be a waste of time, because if the design problems are significant enough, a lot of the other code under review is going to disappear and not matter anyway.

There are two major reasons it's so important to send these major design comments out immediately:

- Developers often mail a CL and then immediately start new work based on that CL while they wait for review. If there are major design problems in the CL you're reviewing, they're also going to have to re-work their later CL. You want to catch them before they've done too much extra work on top of the problematic design.

- Major design changes take longer to do than small changes. Developers nearly all have deadlines; in order to make those deadlines and still have quality code in the codebase, the developer needs to start on any major re-work of the CL as soon as possible.

10.4 Step Three: Look through the rest of the CL in an appropriate sequence

Once you've confirmed there are no major design problems with the CL as a whole, try to figure out a logical sequence to look through the files while also making sure you don't miss reviewing any file. Usually after you've looked through the major files, it's simplest to just go through each file in the order that the code review tool presents them to you. Sometimes it's also helpful to read the tests first before you read the main code, because then you have an idea of what the change is supposed to be doing.

Next: Speed of Code Reviews

11 Handling pushback in code reviews

Sometimes a developer will push back on a code review. Either they will disagree with your suggestion or they will complain that you are being too strict in general.

11.1 Who is right?

When a developer disagrees with your suggestion, first take a moment to consider if they are correct. Often, they are closer to the code than you are, and so they might really have a better insight about certain aspects of it. Does their argument make sense? Does it make sense from a code health perspective? If so, let them know that they are right and let the issue drop.

However, developers are not always right. In this case the reviewer should further explain why they believe that their suggestion is correct. A good explanation demonstrates both an understanding of the developer's reply, and additional information about why the change is being requested.

In particular, when the reviewer believes their suggestion will improve code health, they should continue to advocate for the change, if they believe the resulting code quality improvement justifies the additional work requested. **Improving code health tends to happen in small steps.**

Sometimes it takes a few rounds of explaining a suggestion before it really sinks in. Just make sure to always stay polite and let the developer know that you *hear* what they're saying, you just don't *agree*.

11.2 Upsetting Developers

Reviewers sometimes believe that the developer will be upset if the reviewer insists on an improvement. Sometimes developers do become upset, but it is usually brief and they become very thankful later that you helped them improve the quality of their code. Usually, if you are polite in your comments, developers actually don't become upset at all, and the worry is just in the reviewer's mind. Upsets are usually more about the way comments are written than about the reviewer's insistence on code quality.

11.3 Cleaning It Up Later

A common source of push back is that developers (understandably) want to get things done. They don't want to go through another round of review just to get this CL in. So they say they will clean something up in a later CL, and thus you should LGTM *this* CL now. Some developers are very good about this, and will immediately write a follow-up CL that fixes the issue. However, experience shows that as more time passes after a developer writes the original CL, the less likely this clean up is to happen. In fact, usually unless the developer does the clean up *immediately* after the present CL, it never happens. This isn't because developers are irresponsible, but because they have a lot of work to do and the cleanup gets lost or forgotten in the press of other work. Thus, it is usually best to insist that the developer clean up their CL *now*, before the code is in the codebase and "done." Letting people "clean things up later" is a common way for codebases to degenerate.

If a CL introduces new complexity, it must be cleaned up before submission unless it is an emergency. If the CL exposes surrounding problems and they can't be addressed right now, the developer should file a bug for the cleanup and assign it to themselves so that it doesn't get lost. They can optionally also write a TODO comment in the code that references the filed bug.

11.4 General Complaints About Strictness

If you previously had fairly lax code reviews and you switch to having strict reviews, some developers will complain very loudly. Improving the speed of your code reviews usually causes these complaints to fade away.

Sometimes it can take months for these complaints to fade away, but eventually developers tend to see the value of strict code reviews as they see what great code they help generate. Sometimes the loudest protesters even become your strongest supporters once something happens that causes them to really see the value you're adding by being strict.

11.5 Resolving Conflicts

If you are following all of the above but you still encounter a conflict between yourself and a developer that can't be resolved, see The Standard of Code Review for guidelines and principles that can help resolve the conflict.

12 Speed of Code Reviews

12.1 Why Should Code Reviews Be Fast?

At Google, we optimize for the speed at which a team of developers can produce a product together, as opposed to optimizing for the speed at which an individual developer can write code. The speed of individual development is important, it's just not *as* important as the velocity of the entire team.

When code reviews are slow, several things happen:

- **The velocity of the team as a whole is decreased.** Yes, the individual who doesn't respond quickly to the review gets other work done. However, new features and bug fixes for the rest of the team are delayed by days, weeks, or months as each CL waits for review and re-review.
- **Developers start to protest the code review process.** If a reviewer only responds every few days, but requests major changes to the CL each time, that can be frustrating and difficult for developers. Often, this is expressed as complaints about how "strict" the reviewer is being. If the reviewer requests the *same* substantial changes (changes which really do improve code health), but responds *quickly* every time the developer makes an update, the complaints tend to disappear. **Most complaints about the code review process are actually resolved by making the process faster.**
- **Code health can be impacted.** When reviews are slow, there is increased pressure to allow developers to submit CLs that are not as good as they could be. Slow reviews also discourage code cleanups, refactorings, and further improvements to existing CLs.

12.2 How Fast Should Code Reviews Be?

If you are not in the middle of a focused task, **you should do a code review shortly after it comes in.**

One business day is the maximum time it should take to respond to a code review request (i.e., first thing the next morning).

Following these guidelines means that a typical CL should get multiple rounds of review (if needed) within a single day.

12.3 Speed vs. Interruption

There is one time where the consideration of personal velocity trumps team velocity. **If you are in the middle of a focused task, such as writing code, don't interrupt yourself to do a code review.** Research has shown that it can take a long time for a developer to get back into a smooth flow of development after being interrupted. So interrupting yourself while coding is actually *more* expensive to the team than making another developer wait a bit for a code review.

Instead, wait for a break point in your work before you respond to a request for review. This could be when your current coding task is completed, after lunch, returning from a meeting, coming back from the breakroom, etc.

12.4 Fast Responses

When we talk about the speed of code reviews, it is the *response* time that we are concerned with, as opposed to how long it takes a CL to get through the whole review and be submitted. The whole process should also be fast, ideally, but **it's even more important for the *individual responses* to come quickly than it is for the whole process to happen rapidly.**

Even if it sometimes takes a long time to get through the entire review *process*, having quick responses from the reviewer throughout the process significantly eases the frustration developers can feel with "slow" code reviews.

If you are too busy to do a full review on a CL when it comes in, you can still send a quick response that lets the developer know when you will get to it, suggest other reviewers who might be able to respond more quickly, or provide some initial broad comments. (Note: none of this means you should interrupt coding even to send a response like this—send the response at a reasonable break point in your work.)

It is important that reviewers spend enough time on review that they are certain their "LGTM" means "this code meets our standards." However, individual responses should still ideally be fast.

12.5 Cross-Time-Zone Reviews

When dealing with time zone differences, try to get back to the author while they have time to respond before the end of their working hours. If they have already finished work for the day, then try to make sure your review is done before they start work the next day.

12.6 LGTM With Comments

In order to speed up code reviews, there are certain situations in which a reviewer should give LGTM/Approval even though they are also leaving unresolved comments on the CL. This should be done when at least one of the following applies:

- The reviewer is confident that the developer will appropriately address all the reviewer's remaining comments.
- The comments don't *have* to be addressed by the developer.
- The suggestions are minor, e.g. sort imports, fix a nearby typo, apply a suggested fix, remove an unused dep, etc.

The reviewer should specify which of these options they intend, if it is not otherwise clear.

LGTM With Comments is especially worth considering when the developer and reviewer are in different time zones and otherwise the developer would be waiting for a whole day just to get "LGTM, Approval".

12.7 Large CLs

If somebody sends you a code review that is so large you're not sure when you will be able to have time to review it, your typical response should be to ask the developer to split the CL into several smaller CLs that

build on each other, instead of one huge CL that has to be reviewed all at once. This is usually possible and very helpful to reviewers, even if it takes additional work from the developer.

If a CL *can't* be broken up into smaller CLs, and you don't have time to review the entire thing quickly, then at least write some comments on the overall design of the CL and send it back to the developer for improvement. One of your goals as a reviewer should be to always unblock the developer or enable them to take some sort of further action quickly, without sacrificing code health to do so.

12.8 Code Review Improvements Over Time

If you follow these guidelines and you are strict with your code reviews, you should find that the entire code review process tends to go faster and faster over time. Developers learn what is required for healthy code, and send you CLs that are great from the start, requiring less and less review time. Reviewers learn to respond quickly and not add unnecessary latency into the review process. But **don't compromise on the code review standards or quality for an imagined improvement in velocity**—it's not actually going to make anything happen more quickly, in the long run.

12.9 Emergencies

There are also emergencies where CLs must pass through the *whole* review process very quickly, and where the quality guidelines would be relaxed. However, please see What Is An Emergency? for a description of which situations actually qualify as emergencies and which don't.

Next: How to Write Code Review Comments

13 The Standard of Code Review

The primary purpose of code review is to make sure that the overall code health of Google's code base is improving over time. All of the tools and processes of code review are designed to this end.

In order to accomplish this, a series of trade-offs have to be balanced.

First, developers must be able to *make progress* on their tasks. If you never submit an improvement to the codebase, then the codebase never improves. Also, if a reviewer makes it very difficult for *any* change to go in, then developers are disincentivized to make improvements in the future.

On the other hand, it is the duty of the reviewer to make sure that each CL is of such a quality that the overall code health of their codebase is not decreasing as time goes on. This can be tricky, because often, codebases degrade through small decreases in code health over time, especially when a team is under significant time constraints and they feel that they have to take shortcuts in order to accomplish their goals.

Also, a reviewer has ownership and responsibility over the code they are reviewing. They want to ensure that the codebase stays consistent, maintainable, and all of the other things mentioned in "What to look for in a code review."

Thus, we get the following rule as the standard we expect in code reviews:

In general, reviewers should favor approving a CL once it is in a state where it definitely improves the overall code health of the system being worked on, even if the CL isn't perfect.

That is *the* senior principle among all of the code review guidelines.

There are limitations to this, of course. For example, if a CL adds a feature that the reviewer doesn't want in their system, then the reviewer can certainly deny approval even if the code is well-designed.

A key point here is that there is no such thing as "perfect" code—there is only *better* code. Reviewers should not require the author to polish every tiny piece of a CL before granting approval. Rather, the reviewer should balance out the need to make forward progress compared to the importance of the changes they are suggesting. Instead of seeking perfection, what a reviewer should seek is *continuous improvement*. A CL

that, as a whole, improves the maintainability, readability, and understandability of the system shouldn't be delayed for days or weeks because it isn't "perfect."

Reviewers should *always* feel free to leave comments expressing that something could be better, but if it's not very important, prefix it with something like "Nit:" to let the author know that it's just a point of polish that they could choose to ignore.

Note: Nothing in this document justifies checking in CLs that definitely *worsen* the overall code health of the system. The only time you would do that would be in an emergency.

13.1 Mentoring

Code review can have an important function of teaching developers something new about a language, a framework, or general software design principles. It's always fine to leave comments that help a developer learn something new. Sharing knowledge is part of improving the code health of a system over time. Just keep in mind that if your comment is purely educational, but not critical to meeting the standards described in this document, prefix it with "Nit:" or otherwise indicate that it's not mandatory for the author to resolve it in this CL.

13.2 Principles

- Technical facts and data overrule opinions and personal preferences.
- On matters of style, the style guide is the absolute authority. Any purely style point (whitespace, etc.) that is not in the style guide is a matter of personal preference. The style should be consistent with what is there. If there is no previous style, accept the author's.
- **Aspects of software design are almost never a pure style issue or just a personal preference.** They are based on underlying principles and should be weighed on those principles, not simply by personal opinion. Sometimes there are a few valid options. If the author can demonstrate (either through data or based on solid engineering principles) that several approaches are equally valid, then the reviewer should accept the preference of the author. Otherwise the choice is dictated by standard principles of software design.
- If no other rule applies, then the reviewer may ask the author to be consistent with what is in the current codebase, as long as that doesn't worsen the overall code health of the system.

13.3 Resolving Conflicts

In any conflict on a code review, the first step should always be for the developer and reviewer to try to come to consensus, based on the contents of this document and the other documents in The CL Author's Guide and this Reviewer Guide.

When coming to consensus becomes especially difficult, it can help to have a face-to-face meeting or a video conference between the reviewer and the author, instead of just trying to resolve the conflict through code review comments. (If you do this, though, make sure to record the results of the discussion as a comment on the CL, for future readers.)

If that doesn't resolve the situation, the most common way to resolve it would be to escalate. Often the escalation path is to a broader team discussion, having a Technical Lead weigh in, asking for a decision from a maintainer of the code, or asking an Eng Manager to help out. **Don't let a CL sit around because the author and the reviewer can't come to an agreement.**

Next: What to look for in a code review