Engineer

Code review

👓 another person examining a change before it is merged

What it is

A code review is the step in which someone other than the author reads a change before it becomes part of the codebase. Google's guide to its engineering practices defines it as code being examined by anyone other than the people who wrote it, and the chapter on the subject in *Software Engineering at Google* adds that this often happens before the code enters the codebase. The plural, code reviews, is how most teams talk about the practice; the names change with the tool. On GitHub it is a pull request review, at Google it is a precommit review, and in the Linux kernel it happens by email, where a reviewer who is satisfied offers a Reviewed-by tag for the patch.

Google's guide lists what a reviewer looks at: design, functionality, complexity, tests, naming, comments, style and documentation. Behind each word is a question. Does the code do what its author meant it to do, and is that good for the people who use it? Could it be simpler? Are there automated tests, and are they correct and sensibly designed? Has the documentation that describes it been updated?

The mechanics are much the same everywhere. The author sends a diff with a description, reviewers comment on individual lines, the author answers or uploads a new version, and the round repeats until a reviewer accepts the change. At Google that acceptance is called LGTM, short for *looks good to me*, and the book describes it as a permissions bit: only one LGTM is required by default, and together with approval from a code owner and a readability check it allows the change to be committed. GitHub ends each review with one of three decisions. Comment gives feedback that neither approves nor asks for changes, Approve says the change can be merged as it stands, and Request changes marks points the author has to deal with first. Anyone who can read the repository can

review and comment, and its administrators can make approval a condition of merging. Google's guide calls each change a CL, which is why one of its pages is titled Small CLs.

Use the concept in a decision

Put "What is a code review?" to work in a real decision. We can help relate the concept to the system you are building or buying. One team taking an idea from a sketch to something your customers can log into, instead of handing it between three suppliers.

Explore our services Get in touch →

Test the assumptions behind the term

Use the explanation and linked work to frame questions for your project. We can examine the assumptions with you before a tool is chosen. Led a GIS map application to 10x revenue growth — 10x revenue growth, led from the map product

Led a GIS map application to 10x revenue growth Get in touch →

Why it matters

The book lists six benefits of a well-run review. It checks that the code is correct, makes sure other engineers can understand the change, enforces consistency across the codebase, promotes team ownership, spreads knowledge and leaves a historical record. Correctness is the obvious one, and the authors cite a study at IBM, published in 1986, which found that defects discovered earlier took less time to fix. The book adds the condition that matters: the time invested in review paid off only while the process stayed lightweight. Static analysis and automated tests now carry much of the correctness work, which the book says has lessened the need to rely on people for it.

Comprehension is the benefit no tool replaces. However good an engineer is, they cannot read their own work as a stranger would, and a reviewer can; review is often the first time anyone but the author checks that a change makes sense. That matters because code is read far more often than it is written. The cultural benefits are quieter. Review reminds an engineer that the code is not theirs but part of a shared enterprise, it lets experts pass on what they know about an area, and because every change keeps its review, anyone can later find out when a pattern arrived and read the review that brought it in.

Review also costs time, and slow review costs more. Google's guide on speed says that when reviews are slow, the whole team moves more slowly, developers begin to resent the process, and code health suffers, because there is pressure to let weaker changes through and cleanups get discouraged. Most complaints that a reviewer is too strict, it adds, go away when the same reviewer simply answers faster. Its standard for approving is pragmatic: once a change clearly

leaves the code health of the system better than it was, reviewers should lean towards approving it, even if it is not perfect. The guide calls this the senior principle among its rules and puts the reason in one line: “there is no such thing as ‘perfect’ code—there is only better code”.

What it looks like in practice

Take a shop whose prices are stored in whole pence. An author adds a spring campaign and sends this change for review:

```
--- a/shop/pricing.py
+++ b/shop/pricing.py
@@ -8,2 +8,4 @@ def order_total(basket):
     subtotal = sum(line.price * line.quantity for line in basket.lines)
+    if basket.voucher == "SPRING":
+        subtotal = subtotal * 0.9
     return subtotal + shipping_for(basket)
```

The reviewer leaves four comments:

```
shop/pricing.py, line 9
  The other vouchers and their rates live in VOUCHERS in
  shop/vouchers.py. Could SPRING go there too? Then the next
  campaign is a change to data, not to this function.

shop/pricing.py, line 10
  Prices are whole pence everywhere else in this module, and
  multiplying by 0.9 can leave a fraction of a penny. The other
  discounts go through round_money() in shop/money.py.

shop/pricing.py, line 11
  Nit: shipping is left out of the discount. If that is on
  purpose, a short comment saying so would help the next reader.

General
  There is no test with a voucher on the basket. Could you add
  one for SPRING and one showing that an unknown code changes
  nothing?
```

Each comment is about the code rather than the person, and each says why. The first two and the last carry no label, so they are expected to be dealt with; the third starts with Nit:, which Google’s guide uses for a small point the author ought to fix but that changes little. The guide’s other labels are Optional, or

Consider, for a good idea that is not required, and FYI, for a thought the author is not expected to act on in this change. Without labels, it warns, authors may read every comment as mandatory. On GitHub this round ends with Request changes. The author moves SPRING into the table, routes the discount through the rounding helper, adds the comment and both tests, and asks for another look; the second round ends with Approve, or at Google with LGTM. The guide asks reviewers to answer a review request within one business day at most, and not to break off a focused task to do it but to wait for a natural break.

None of the comments is about formatting, import order or whether the tests pass, because a machine should have said that before the reviewer opened the diff. The book describes presubmits that run when a change is first sent for review and can reject it before anyone reads it; the kernel asks authors to run its style checker, `scripts/checkpatch.pl`, before they submit. The guard engine built for a repository, from 2025, takes that idea a long way: 268 registered commit checks, 277 lint rules and 15 custom ESLint rules, plus 146 tests of the guards themselves. The guards check things such as whether a migration can be rolled back and whether a translation key exists in both locales, and any commit marked as a fix has to carry a test that fails without it. The checks encoded mechanical rules, so reviewers could spend more time on design. It also states the cost: committing is slow, and a badly written guard is infuriating to work around.

Where it goes wrong

The change is too big to read. Google’s guide calls 100 lines usually a reasonable size for a change and 1,000 lines usually too large, while leaving the call to the reviewer, and notes that 200 lines in one file might be fine when the same 200 lines spread across 50 files would not be. Small changes are reviewed more quickly and more thoroughly and are less likely to introduce bugs; the book adds that reviewers can rightly reject a change as too large for a single review. Split the work: stack small changes on top of each other, send groups of files to different reviewers, keep refactorings apart from feature changes and bug fixes, and keep each change’s tests in the same change.

The review sits for days. Every day a change waits delays the work built on it, and a reviewer who answers every few days with major requests makes the process feel harsher than it is. Answer within one business day; the book’s expectation is feedback within 24 working hours, and if the full review has to wait, a reviewer should say they have seen the change. Send all the comments at once rather than piecemeal, since few things annoy an author more than a new round of unrelated feedback after the last one was addressed.

Taste is presented as a requirement. A reviewer who prefers another loop or another name and blocks the change over it trains authors to stop improving the code. Google's guide sets the order: facts and data count for more than opinion or personal taste, on matters of style the style guide has the last word, and when several approaches are equally sound, the author chooses. Label a preference as Nit or Optional, so that the author can tell it apart from a requirement.

The comment is about the person. The guide's own example of a bad comment asks why the author used threads when there is obviously no benefit from concurrency; the good version says the concurrency model adds complexity with no performance benefit the reviewer can see, so the code should be single-threaded. Comment on the code, never on the developer, and explain the reasoning behind the request.

People check what a machine could. Reviewers who spend their attention on spacing and import order miss the design question that only a person can answer. Move every rule that can be written down into a linter, a formatter or a presubmit check, and leave the reviewer the judgement calls.

The answer stays in the review tool. A reviewer asks what a block of code is for, the author explains in the thread, the reviewer approves, and the next person to read the code never sees the explanation. Google's guide says such a question should usually lead to the code being rewritten more clearly, or at least to a comment in the code. The kernel makes the same point to authors: when a reviewer's question changes no code, the answer should, as a rule, end up in a comment or in the changelog.

Further reading

- Introduction — the opening page of Google's code review guide: what reviewers look for, how to pick reviewers and when pair programming counts as review.
- The Standard of Code Review — the senior principle, mentoring through review, and how to resolve conflicts between author and reviewer.
- Small CLs — why small changes get better reviews, how big is too big, and four ways to split a change.
- Code Review — Tom Manshreck and Caitlin Sadowski's chapter of *Software Engineering at Google*: the benefits of review, LGTM and ownership, and best practices.

- Pull request reviews — GitHub's reference for reviewing, approving and requesting changes, and for requiring approvals before a merge.
- Submitting patches: the essential guide to getting your code into the kernel — how the Linux kernel reviews by email, what a Reviewed-by tag states, and how authors should answer reviewers.

Glossary

Choose for your actual constraints

Describe the workflow, the existing interfaces and the users who need the change. We can define a testable delivery and a handover that lets your team maintain it.

See what it costs Get in touch →

Make the terminology practical

Ask how "What is a code review?" applies to your organisation. A concrete example from your current system is enough to start the discussion.

Explore our services

welcome@engineer.company +45 50 10 61 84 Copenhagen, Denmark

Take this page with you

Contact card

Carefully Crafted With Love by D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Copyright © 2025 – present · Engineer ApS · All rights reserved.

Printed from https://engineer.company/glossary/code-review/