

```
static int __init dummy_init_one(void)
{
    struct net_device *dev_dummy;
    int err;

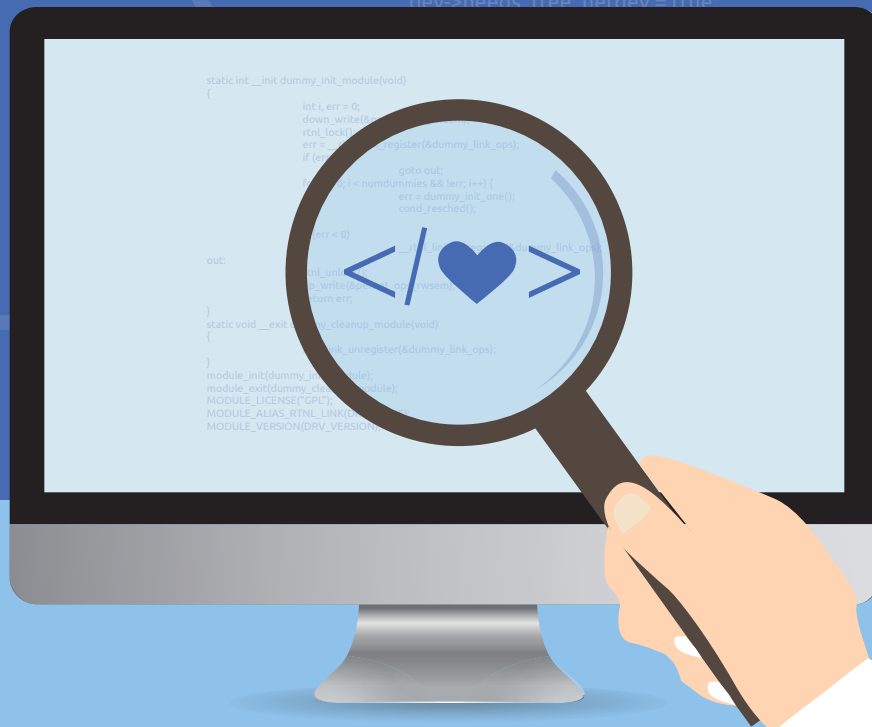
    dev_dummy = alloc_netdev(0, "dummy%d", NET_NAME_ENUM, dummy_set-
up);
    if (!dev_dummy)
        return -ENOMEM;

    dev_dummy->rtnl_link_ops = &dummy_link_ops;
    err = register_netdevice(dev_dummy);
    if (err < 0)
        goto out;
    return 0;
err:
    free_netdev(dev_dummy);
    return err;
}
```

CODE REVIEW GUIDELINES

```
static void dummy_setup(struct net_device *dev)
{
    ether_setup(dev);

    /* Initialize the device structure. */
    dev->netdev_ops = &dummy_netdev_ops;
    dev->ethtool_ops = &dummy_ethtool_ops;
    dev->needs_free_netdev = true;
```



flanksource

TABLE OF CONTENTS

Introduction	3
Preamble	3
Bad review	3
What is code review for?	4
Standards.....	5
When should a pull request be approved?.....	5
How quickly should a pull request be attended to?	11
What should reviewers be looking for?.....	13
Supporting documentation	18
Kindness.....	19
Understanding others	19
How to comment on code?.....	20
How to prepare code?	32
How to respond to comments?.....	34
Supporting documentation	40
Strictness	41
Plane crash hospitals vs. car crash hospitals	41
The value of strictness.....	43
Supporting documentation	46
Organisation.....	47
Management	47
Team/technical leads.....	52
Rules	57
Tools	58
Supporting documentation	62

Introduction

Preamble

Scottish thinker David Hume once said, “Truth springs from argument amongst friends”. We might read Hume so:

- when we place each other’s ideas under scrutiny – when we expose them to question and challenge – what ideas emerge from that crucible are better; and
- it is among friends the first is true – among those who hold warmth for each other, who provide safety, who set aside ego, who engage in collaborative inquiry.

In software development, one place for this “argument” is code review. Review may not be a search for truth, but it is for value, correctness, commonality and cohesion; and as with truth, uncovering these requires detail-oriented scrutiny and interpersonal kindness.

Bad review

Those lucky to have worked only in high-performing environments may find what follows a mystery – to which we say: “long may your puzzlement persist”. The rest have encountered some of the following frustrations in review:

- long cycle times – i.e., the gap between submitting a review and getting feedback,
- multiplied by the number of back-and-forths necessary for alignment;
- the time burden of review, which can be especially heavy on leads;
- interpersonal conflict;
- unwillingness of co-workers to explain their code;
- pressure to get code into main/master branch once in review;
- inability to ask for changes, either because co-workers believe their work is entitled to acceptance once they make a PR, or because estimates didn’t factor in the possibility of necessary adjustments to work;
- inability to get useful comments, either because coworkers have no time for proper review or because they lack expertise;
- difficulty in reviewing code for which one lacks context;

- tendency of teammates to defer work raised in a pull request, which can often mean it's never done – this is, for example, how test coverage degrades and bad patterns are introduced;
- inability to capture follow-up work;
- lack of clear responsibility for reviewing, merging or approving;
- unwillingness in co-workers to tidy or refactor code they didn't write, even when proximate to code they're touching; and
- neglect of documentation.

The processes and norms in this guide are designed to address these problems.

What is code review for?

For many, the answer is “catching bugs” – this guide takes a different view.

Quality assurance is desirable, but in a mature environment with effective CI/CD pipelines, the ultimate responsibility for catching bugs is in unit, integration and end-to-end tests. Teams that are building out automated systems might temporarily rely on review for bug catching; but, in high-performing teams, review serves the following purposes:

- 1 training and knowledge sharing;
- 2 team cohesion;
- 3 architectural and pattern alignment; and
- 4 de-siloing;

These four outcomes do much work in solving our problems.

Standards

“ When bad code gets into source control: it increases the likelihood that someone will find it and copy it. Once it's been copied one time, that increases the likelihood it will be copied again, and so on. Before you know it, the bad code has infiltrated all parts of the repository and is hard to extract. The problem grows as the number of people committing code to the same repository grows. When you have a team of a significant size, managing what ends up in source control becomes incredibly important because it will multiply quickly. As such, you want to make sure the code that gets checked in is as high quality as possible and represents what you want others to do.”

The Bunny Theory of Code, Nicholas C. Zakas

When should a pull request be approved?



In general, reviewers should favor approving a pull request where:

- 1 it is in a state where it definitely improves the overall code health of the system being worked on;
- 2 where reasonable, all opportunities for training and consensus development have been taken and results captured;
- 3 it fulfils the architectural ambitions and meets any standards, definitions of done, or the requirements of other checklists and processes;
- 4 review has been appropriately thorough and the reviewer understands the changes;
- 5 it has no easy-to-fix, clear and obvious errors without guaranteed remediation path;
- 6 all reasonable change requests have been addressed;
- 7 all comments have been acknowledged and appropriately addressed;
- 8 all work arising from the review is captured; and
- 9 no automated checks are failing.

These conditions do not apply in a production emergency, but it is expected that code which is pushed to resolve an emergency will be revisited.



PR is in a state where it definitely improves the overall code health of the system being worked on

Teams face a balance between correctness and progress. Strictness is valuable, but perfection isn't possible; the perfect should not be the enemy of the good. A pull request must simply improve the maintainability, readability, and understandability of the system; where it has been produced with skill and diligence and adds desirable features, merging shouldn't be delayed for days or weeks.

The two exceptions relate to training:

- 1 when training a new developer in how to prepare a review to standard, in which case strict, detail-oriented guidance is an important training tool; and
- 2 when training a team into desired architectural and functional patterns in the early stages of a project.

In these cases, it may be reasonable to hold off merging for longer.



Where reasonable, all opportunities for training and consensus development have been taken and results captured

When a pull request introduces a new architecture or is inconsistent with existing code, review can evoke questions like:

- “Is this how we should do this everywhere?”
- “Will this pattern serve our long term goals?”
- “How do we want to handle problems like these going forward?”

When such questions arise, it can be useful to ask a team to come to a view on the right pattern. Usually, unless a member is empowered to make a call, it is best to include technical or architectural leads in discussions.

These discussions might be deferred: when the questions raised are far reaching, it can be best to meet to agree on a design that answers them.

Decisions should be captured in a decision log and/or contribution guide, along with any supporting documentation.

Where time allows, reviewees and reviewers should take opportunities to include in discussions members who otherwise lack context. Proactive sharing and inclusion lays groundwork for later review and other contributions by the included member.



PR fulfils the architectural ambitions and meets any standards, definitions of done, or the requirements of other checklists and processes

Code must fulfill the ambitions of the project. A pull request may be rejected for reasons other than quality – it may add features or architecture that the reviewer does not believe belong.

Many teams have standards, architectural decisions records, checklists and definitions of done that support code quality, bring coherence to a codebase and ensure things are not missed.

Definitions of done often address questions like:

- Does it appropriately support local development?
- Is it deployable?
- Has it been appropriately tested?
- Has documentation been updated?
- Have appropriate automated tests been added?

And standards might provide guidance on:

- how the codebase uses certain libraries and the integration patterns; and
- the conventions for branching, branch naming, commit messages and commit organisation.

The relevant guidance should be easily available, well organised and clear in its scope of application.



PR has been appropriately thorough and the reviewer understands the changes

Code review is not a rubber stamp process. Reviewers must be knowledgeable enough to understand the implications of changes and must be diligent and fine-grained in working through reviews.

Team members generally have different but overlapping areas of expertise and different levels of seniority, and at times it is difficult to find a reviewer who is knowledgeable about the code/systems being changed.

When such a reviewer cannot be found, reviewees should be prepared to teach their code. The onus is on them to ensure whomever reviews is provided with what context they need. Walking the reviewer step-by-step through changes and answering questions until the reviewer is satisfied will usually be sufficient.



PR has no easy-to-fix, clear and obvious errors without guaranteed remediation path

This requirement can be a source of conflict because of review cycle times. A spelling error is an easy-to-fix, clear and obvious error, but waiting an hour more to merge because of such an error is painful if changes void approvals.

Where not a functional error (which cannot be merged), the pull request should only be approved:

- once there is a branch in waiting with a ready patch; or
- where there is a process to ensure that such a patch will be merged no later than COB.

In the second case, it should be considered serious if a promise to remediate goes undischarged in time.



All reasonable change requests have been addressed

What count as reasonable change requests are driven by product, logic, standards and any other guidelines contained herein.

A note on scope is important: it is essential to the health of a project that developers opportunistically improve code and clear cruft. Inconsistency is a major source of cognitive load in even medium-sized codebases.

Without planning, opportunistic fixes are the only opportunity developers have to improve code hygiene when that poor hygiene is not manifesting (yet) in bugs.

It is reasonable to request changes of the following kinds:

- to the whole codebase, where the PR introduces a new pattern, name or system and:
 - it is consensus that it is a good change, widely desired;
 - consistency is valuable and inconsistency would degrade code health; and
 - it is only incrementally more work to change the codebase to consistently use the new standard;
- to improve code health proximate to required changes;
- to refactor a function, module or class in which changes have been made; and
- to add commentary to provide clarity on the purpose of code and/or to provide links to documentation.

Requests for forward-looking code hygiene should be accepted especially where the cost is incremental.



All comments have been acknowledged and appropriately addressed

All comments must be acknowledged and appropriately addressed by:

- responding directly; or
- resolving the issue thread.

Issue threads can always be resolved by the initiating reviewer, but threads should only be resolved by the reviewee under two conditions:

- 1 the reviewee believes they have addressed the issue such that the reviewer could not in good faith contest; and
- 2 the change has already been pushed.

Otherwise:

- where comments are implicit change requests (e.g., “I think you meant to use X” or “perhaps such and such an approach would be more viable”); or
- where comment are direct questions,

a reply is required, even to explain why the change will not be made.

Where a reviewee and reviewer have had an in-person (or video-call) review, comments may be resolved with permission of the reviewer.

A pull request should not, in general, be merged while there are open issue threads. The exception is if both:

- the reviewee believes they have addressed the issue such that the reviewer could not in good faith contest; and
- the reviewer is not reasonably contactable or cannot attend to resolving threads before COB.



NOTE

“could not in good faith contest” is a very strong condition requiring high confidence.



All work arising from the review is captured

During a review, reviewers may notice work in need of doing. It may be that:

- The reviewers realise that the scope of work required to handle that work goes beyond the ambit of the current pull request; and
- while reading surrounding code or understanding context, work not previously considered comes to the reviewers' attention.

This work should be captured (e.g., in tickets, issues, or however the organisation tracks tasks) **before the review is merged** and, if possible, links to tasks should be placed in the comment thread that elicited the work.

As with “PR has no easy-to-fix, clear and obvious errors without a guaranteed remediation path”, some judgment should be applied in blocking a merge/approval when work has not been captured, but approval should only be given where there is a clear promise to make the capture before COB. As in that case, it should be considered serious if a promise to remediate goes undischarged in time.



IMPORTANT

When deciding if work should be tasked out or addressed in the pull request, consider whether doing the work will not be quicker than putting the task through a full planning cycle, remembering that costs include:

- the time to capture the task and spec it;
- the time to do a new review of the fixes;
- the time for a new developer to understand the problem and set up to investigate it; and
- the time taken to discuss the task in planning multiplied by the number of people in the meeting.

Organisational pressure can mean this decision may need approval; those managing sprint expectations must be sensitive to the trade off.



No automated checks are failing

While repositories should be set up to prevent merges when tests are failing, pull requests should not be approved until all gates are passing.

How quickly should a pull request be attended to?



Team members should respond to requests for review **shortly after they come in** and as close to immediately as is reasonable – unless in the middle of a focused task, such as writing code. Even then, reviews should be undertaken as soon **as a natural breakpoint arises**, such as when a coding task or test is completed or a developer pauses to make coffee.

Long cycle time - the gap between submitting a review and getting feedback, multiplied by the number of back-and-forths necessary to get code aligned – is a major input into review processes failing and, as consequence, is a major contributor to poor code health. It cannot be emphasised enough: **reduced cycle time is a core input to project health**.

Why?

Here is what the [Google Engineering Practice documentation](#) has to say about speed of review:



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, change list – being a pull request] 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.”

All things equal, developers should prioritise team velocity over personal velocity, and managers should incentivise this behaviour. High responsiveness from members means a pull request may get multiple reviews in a day.



First response and partial reviews

Reviews may be incomplete. What's important is that the reviewee receives feedback quickly.

The reviewee must quickly be able to return to preparing their code. Reviewers should not make reviewees wait for large comment sets when reviewees have not yet received feedback, unless a partial response would be misleading. As a matter of politeness, reviewers should explain that their review is partial.

Reviewers should attempt to give fast first-pass reviews. A first pass may include:

- a read of whether the solution is sensible or not – if it's clearly going wrong, the reviewer should catch this here and not request a larger set of changes;
- notes on obvious mistakes and code hygiene; and
- notes on missing features, tests and/or mismatches with requirements.

Partial review is especially valuable when the best person to conduct the review is not immediately available.



IMPORTANT

A first-pass partial review does not substitute for a full review. A reviewee's experience through review should not involve being bumped from partial review to partial review. A first-pass review buys time for a more fulsome review, and that must follow quickly – especially as more thorough review may invalidate requests made with less knowledge.



Maximum response time

The maximum response time for review should be one business day, barring unusual circumstances. Some techniques that can be used to support this goal:

- At start of day, team leads should assign outstanding reviews to the team.
- Review request notifications – and, depending on the level of signal to noise: comment, status update and interaction notifications – should be published to actively monitored communication channels, such as, for example, Slack, Discord or MS Teams.
- Ensuring that all members are sharing the burden of reviewing.
- Having a team norm of welcoming reminders and requests from team members about code they need reviewed – and not considering them nags.

What should reviewers be looking for?



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.”

[The Standard of Code Review](#), Google Engineering Practice documentation

It’s difficult to define what makes good software and there are more opinions than developers. Reviewers must be careful they do not mistake their opinions for well-founded principles – or indeed well-founded principles for mere opinions.

Teams must implement standards, adopt paradigms and methodologies, create designs, implement architecture, follow style guides and write product definitions to fill in the granular details of the desirable and undesirable for their projects.

A high-level checklist may be useful for ensuring reviews are complete and thorough. The following checklist covers a lot of ground:

- Functionality
- Standards and architecture
- Cruft and technical debt
- Complexity, clarity & performance
- Product fit
- Documentation & comments
- Testing
- Deployability
- Security, privacy and compliance
- Altruism
- Observability



Functionality

Reviewers should check that code does what it intends. In a mature environment, much validation is automated, and tests would be submitted alongside code, including end-to-end, browser and UI tests, where relevant.

Nevertheless, some kinds of functionality should be properly validated by the reviewer. From [What to look for in a code review](#), Google Engineering Practices:



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 [change list (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.)”

Code should also be robust and resistant to error, and reviewers should be alert to code that is missing appropriate error handling.



Standards and architecture

Reviewers should check that code meets any standards set out in architectural guidance, style guides or definitions of done and that it conforms with the paradigms and methodologies adopted in the project.



Cruft and technical debt

Reviewers should check that code does not introduce avoidable [cruft](#) or [technical debt](#).



Complexity, clarity & performance

Reviewers should check that code:

- **Is as locally simple as possible without sacrificing global simplicity.** Simpler code is better code; however, sometimes stomaching some complexity advances overall simplicity. For example, the use of state machines, which incur a high initial learning curve, but provide predictable, tractable and declarative state management.
- **Achieves the correct balance between performance and clarity.** Micro-optimisation is considered an anti-pattern, as developer time is more expensive than processor time. Where such a trade-off exists, the correct balance between clarity and performance often means giving up performance; however, reviewers should understand what code is performance sensitive and/or what lies in hot paths, and should ensure the correct balance is being struck.
- **Where performance is a requirement, meets the performance requirements.** Performance can mean many things. For example, availability is a kind of performance.

“Elegant” is often a word for code that perfectly balances all these concerns (among others).

Reviewers should be asking themselves whether code is elegant.



Product fit

Reviews should ensure code advances project goals and meets product requirements.



Documentation and comments

Reviewers should check that documentation is updated, is clear, and that comments are of high quality. High-quality comments give context, explain purpose and provide resources. Comments should not restate what code does – they should explain why it exists.

Marc Brooker, in [Code Only Says What it Does](#), quotes another developer in explaining the inspiration for the post:

“ Since most software doesn’t have a formal spec, most software “is what it does”, there’s an incredible pressure to respect authorial intent when editing someone else’s code. You don’t know which quirks are load-bearing.”

They write:



Code says what it does. That's important for the computer, because code is the way that we ask the computer to do something. It's OK for humans, as long as we never have to modify or debug the code. As soon as we do, we have a problem. Fundamentally, debugging is an exercise in changing what a program does to match what it should do. It requires us to know what a program should do, which isn't captured in the code.

Sometimes that's easy: What it does is crash, what it should do is not crash. Outside those trivial cases, discovering intent is harder.

[...]

This is a major problem with code: You don't know which quirks are load-bearing. You may remember, or be able to guess, or be able to puzzle it out from first principles, or not care, but all of those things are slow and error-prone."

Documentation has a wider purpose than comments, but documents that explain why things are some way provide readers with a larger universe of understandings than those which simply record facts.



Testing

Where testing is not covered under "Standards and architecture", reviewers should ensure that code is properly tested with automated tests.



Deployability

Reviewers should ensure code remains deployable in all relevant contexts, including for local development purposes.



Security, privacy and compliance

Reviewers should watch for code containing clear vulnerabilities or exposing private information. Secrets should not be checked into repositories and, should a check-in occur, remediation should be undertaken.

Reviewers should be aware of any relevant laws and regulations and should consider code in light of them. Regulations often cover matters like:

- use of personal information
- privacy
- accessibility

In Europe and South Africa, the [General Data Protection Regulation](#) and [Protection of Personal Information Act 4 of 2013](#) govern sharing of personal data, respectively.

Users also have privacy concerns not governed by regulation. In general, regulations are developed in response to invasive practices that alarm users once they discover them. Reviewers should be alert for features that users would consider inappropriate or invasive or alarming, should they discover them.

Many countries also have robust requirements for systems to cater to people with disabilities and failing to cater can incur liability.



Altruism

Reviewers should check that easy opportunities for improving the codebase have not been missed.



Observability

Reviewers attend to the need for insight into their code, and should ensure that new code provides logging and is integrated properly with monitoring and tracing systems.

Supporting documentation

When should a pull request be approved?

- [*The Bunny Theory of Code*](#), Nicholas C. Zakas
- [*The Standard of Code Review*](#), Google Engineering Practices

How quickly should a pull request be attended to?

- [*Speed of Code Review*](#), Google Engineering Practices

What should reviewers be looking for?

- [*The Standard of Code Review*](#), Google Engineering Practices
- [*What to look for in a code review*](#), Google Engineering Practices
- [*Code Only Says What it Does*](#), Marc Brooker
- [*Maslow's pyramid of code review*](#), Charles-Axel Dein
- [*Code Review Best Practices - Lessons from the Trenches*](#), Drazen Zaric
- [*Giving better code reviews*](#), Joel Kemp

Kindness

“You are not your code”, the aphorism goes. The truth is that it is difficult to have professional pride without investing some identity in one’s work. One may not be one’s code, but it’s likely one desires the respect of colleagues, and that one would feel diminished were it a fiction.

Being treated as if they were their code is an experience many developers have gone through, and treating others the same is a mistake many have made.

There is some value in the advice to guard oneself (“have a thick skin”). Tone is difficult in text and colleagues are often busy and under pressure – some just find writing difficult or are introverts and can be terse without meaning insult. It’s a kindness to offer understanding. By adopting review norms, approaches and methods designed to make everyone feel human, we raise the bar for everyone.

Understanding others



People are different. It’s important to hold this front of mind. Some have anxiety; some trauma; some are loud; some quiet; some are bold; some have to be drawn out of their shells.

While this guide offers strategies for how to behave in review, it acknowledges that some will have an easier time following than others. We should respect those who struggle – both as reviewers and reviewees – though they must still make the effort.

Given the coverage given to reviewers, it’s easy for reviewees to see this guide as a manifesto for their treatment by reviewers. That would be an error. Kindness and respect are owed to everyone, and reviewees should not underestimate the risks reviewers take writing their views. Nobody wants to upset anyone else.



Hurt

If one gets hurt in review, either as a reviewer or reviewee, often (not always) the best thing to do is reach out to the other and tell them. This can be hard. It’s scary to be vulnerable with no guarantee our hurt will be respected.

The truth is that nine times out of ten when one gets hurt in review, there’s misunderstanding – either the reviewer wrote inelegantly or the reviewee reads their fears into the text. Resolving can be as simple and difficult as saying:

“I’m finding the way things are going in review distressing and hurtful. I doubt that was your intent, so do you mind if we have a quick chat about it?”

Resolving hurt openly builds trust and both parties must be ready. Do this out of band – there is no need to be on record. Send a message or make a call.



Gratitude

Reviewers work hard at reviews. Reviewees work hard to prepare properly for reviews. Everyone deserves gratitude for the ordinary kindnesses they do and everyone should practice saying “thank you!”.



Anger

The first thing to do when feeling angry is [give it five minutes](#).

Code review can induce anger. It’s **never** acceptable to let that anger flow into the review. If anyone is so angry during review they cannot shrug it off, calm down, and continue politely, there is an issue that requires a mediated meeting.

How to comment on code



This guide cannot guarantee for readers that their colleagues will be competent and diligent or act in good faith. That guide is atop the stone of truth, tucked behind the elixir of youth in the paradise wing of the cave of wonders.

That colleagues arrive with **competence and goodwill** must be the starting **assumption**. Developers arriving at review concerned about the competence and goodwill of colleagues need a conversation with their manager – code review is not for those worries.

Taking this assumption as bedrock, team members should be able to write comments that promote collaboration, safety, team cohesion and high-quality code.

Assuming competence and goodwill makes it easier to avoid cruelty: don’t shame people; don’t use strong negative language; don’t be sarcastic; and don’t assign negative traits to them.

Sometimes it's harder to be kind than to avoid cruelty. This guide outlines ways of being generous with, and kind and respectful to, reviewees, and provides examples for those less naturally gregarious.



Assume the reviewee is knowledgeable

The reviewee already knows their code, and has wrestled with the problems it is intended to solve – the reviewer is approaching it fresh. When reviewers don't understand the purpose of code, choices can seem strange and code oddly structured. In this case, reviewers should say something tonally equivalent to:

"I'm not sure about this but maybe I'm missing something.
Could you help me understand the choices you made here?"

It's good to provide more information, and the next example matches the above in outcome:

"I'm not sure about these function parameters.
I understood that we were trying to ensure all the functions in this module
shared the same signature.

This was the post that Zimkita shared around last week and
I thought we had some consensus on it www.techin.com ↗

Also, we made an entry into our decision log here ↗

But I'm also not that familiar with the problem
space you're dealing with, so please help
me out if I'm missing something here!"

A few features worth highlighting:

- the problem is identified and, if the reviewer is correct and the code isn't up to scratch, this will lead to the code being changed, but without anyone feeling attacked;
- it is assumed that were the reviewee in possession of correct information they would make the correct decision;
- the reviewer proactively finds links/references to help the reviewee;
- the reviewer always speaks about their understandings rather than about the reviewee;
- the reviewer keeps exposing their thinking, so the reviewee has a clear understanding of their mind for any response; and
- by highlighting how they may have missed something, the reviewer shows humility and ensures they will find out the truth if they are in fact missing the point.

It's not always necessary to say so much. At times, pairing a statement with a question will do:

"This function duplicates functionality found in [file/module].
Is this intentional?"



Explain first

Offering reasons in review does two things. First, reasons help teach the reviewee the **why** of the change request for the next time. Second, it allows the reviewee to defend their change if they believe they have already considered reviewer concerns – otherwise the review takes an extra cycle for the reviewee to ask for the reason. For example:

"We have agreed that all config files should go in the config folder at root so that our deployment scripts know where they are.
Please won't you move this over?"

versus

"Move this to the config folder in root".

To the first, the reviewee may reply:

“This isn’t a deployment config.
I’ve just used this config file to pull out hardcoded data
that this function needs, so I think it’s best to keep it local.”

Explaining one’s reasons is also a show of interpersonal respect. It treats a person as someone who would make similar decisions given the same information - i.e., as an equal.



NOTE

Reviewers should feel free to be matter of fact when drawing attention to something they have strong confidence the reviewee has just missed. This requires a judgment of appropriateness – and reviewers should be open to hearing they have missed the mark.

When a reviewee has missed removing code they know they must, it’s acceptable to say:

“Please remove.”



Use a feedback ladder

Often one wants to make comments in review that are non-blocking, such as:


- showing someone how something could be done for the future; or
- asking questions for interest and knowledge.

Using a common language for such comments can be a useful way of reducing frustration, as **comments should generally be assumed to be blocking**.

A simple scheme, and one that is a good starting point for a team, is to use “nb” and “nit” to denote comments that are non-blocking.

So, for example:

“[nit]: For future, rather put this in its own commit”

“[nb]: This blog post outlines a pretty neat way of handling this problem, if you run into it again: www.scint.com 

“[nb]: For my knowledge, what does this syntax do?”

Some comments require a meeting to be created or work captured before approval. A more advanced feedback ladder for these might work as follows:

“[capture]: We need to make sure we come back and refactor this. Won't you please write up the relevant task? Task link in the response comment please.”

“[meeting]: Happy to let this through for now, but we should have a team meeting on what we want to do going forward. Please schedule and link in?”

There are many ways to construct a feedback ladder – [Netlify has a feedback ladder](#) that makes use of the metaphor of a beach house and the various things that can blow into the house and cause problems, resulting in different levels of emergency.

Whatever style a team chooses, it should support any strictness and standards requirements.



Mention the positives

Saying nice things is important, but can sometimes feel awkward. Praising someone for expected behaviour can feel condescending, and one certainly shouldn't praise worries.


Nevertheless, reviewers should find opportunities for both praise and gratitude. Saying “thanks!” when a reviewee acts on a comment or discussion helps reinforce that practice.

Reviewers should say “this is nice work!” when they encounter an elegant piece of code. Bonds of mutual respect make teamwork easier and improve the dignity of those who hold them.

When asking for a change, reviewers might still praise what they found. An otherwise clever solution that's just not to standard might be met with:

“Ah, this is very clever.

It almost makes me wish we hadn't already gone in a different direction.

Sadly, we're covered this already here: www.orgmgmt.com 

Overall/general comments provide really good opportunities for pleasantness:

“Thanks for your hard work on this!

I really like the overall direction this is going in. I've just left some detailed comments on sections I think we should look at.

Give me a shout if you have any questions.”

It's important for reviewers to comment positively on changes made as they wished:

“Nice! This is exactly what I was getting at.”

“Thanks for fixing this up!”



Speak personally or about the code

Reviewers can reduce tension in reviews by speaking in first person or about the code, but never about the reviewee:

I'm confused here. Won't you help me?



is superior to saying

This is confusing. Won't you help me?



and much, much better than

You're being confusing. Won't you help me?



while the worst is

"You're always so confusing. Won't you help me?"



The first situates the problem with the reviewer; the second focuses on the code; the third makes an accusation; while the fourth makes an ongoing claim about the person.



Suggest alternatives or improvements

Often a reviewer will:

- know a way to improve code;
- know an alternative method with advantages; and
- know of others who have solved similar problems.


Reviewers should make thoughtful suggestions about the code they're reading:

"So, this is one way to do it, but we might run into problems with this structure down the line if we want to expose this state. Have a look at this: www.technews.com . The way they construct the function lets them keep things private for now but it's pretty easy to modify. Have a think about it and let me know."

"I saw this blog the other day that had a really elegant approach to these kinds of problems. Do you think this might work well for this case? www.codebase.com "

When offering improvements to code, reviewers should be generous with their reasoning:


“I see what you’re getting at here, but I still think we can do better. Let me try to explain and we’ll see if I’m making any sense.


Using the following syntax at line 10: www.socialtech.com 


Will let us extract the values we need without going through the second loop.

It’s slightly more memory intensive, but references should be garbage collected pretty quickly so I’m not especially worried about that.

Then, if we structure the data link like so: www.abccode.com 

We’ll be able to share the interface with our module in www.int.com/devops , which means we’ll be able to do some nice fluent chaining/piping.

Also, if we declare the constants at the top of the file rather than inline in the function (on line 72), we should be able to export them nicely for the module in www.vectorlus.com/moduledev , which probably shouldn’t have been the owner in the first place, and refactor that.

There’s a little trick we can do with that mapping on line 22. I quickly prototyped it in codesandbox just so you can see the simple case. Here’s the live code: www.codebase.com 

Let me know what you think!
Happy to pop on a call if you run into trouble.”



NOTE


While reviewers are expected to be helpful, they are not expected to redesign or reimplement code. Reviewers should offer maximal guidance, but should never feel trapped into doing work for someone else. Everyone loses when a reviewer does the job of two people and the reviewee never learns better.




Point out the standards

Sometimes how code should look, or what should be included in a pull request, is clear and on record. Most often, the reviewee hasn't fulfilled a requirement because they didn't know about it or they forgot.

Development work requires judgment. Following a standard off a cliff is never useful. Understanding this, lets us phrase our comments firmly but with humility:

"I think this might be covered by the standards.
See www.tronicselect.com/pages/s1/file/7 

"I know you're passionate about this, but our standards take a different approach. See www.softint.com/pages/s1/file/7 .
If there's something that's missing there or you think this is an edge case, let's have a chat about that after, or maybe raise it to the team."



Ask questions

Reviewers must be appropriately thorough and understand any changes, so they should ask questions when they need the expertise necessary to make a review valuable.

It can be hard to admit ignorance – for everyone. Junior developers worry about whether they're cut out for the work or that they're annoying, while more senior developers worry that ignorance is a sign they don't deserve their position.

Everyone will at some point need to know something, but asking questions is the most effective way to find things out. Some valuable questions:

- 1 "You're making use of this "connect()" function all over. I looked at the codebase and it's used quite a lot. I don't really understand it. Mind talking me through how it fits in?"

- 2 “I’ve noticed Sibo and Orli using \$(!!) in their scripting also. I can’t seem to find it online. Mind just filling me in?”
- 3 “I see you’ve separated out that config into a yml file rather than json. I’m curious, why did you decide to do it that way?”
- 4 “I see you have strong behavioral tests for this, so I’m comfortable that it works, but how does this syntax propagate the error?”
- 5 “As an aside, do you know what kind of runtime our pods use by default? I see you commented that bash wouldn’t work. What would?”

And some observations:

- The examples are in decreasing order of importance to a reviewer, but all are valid comments. Specifically, (1) and (2) are essential for the reviewer to complete their task, though the scope of explanation required is different.
- In (3), there might be a reason or not. The reviewee might not have considered the alternatives. By asking, the reviewer might cause the reviewee to reconsider.

When the reviewer receives an answer, it might be that the reviewer knows a better way. Without asking, the reviewer would not know enough to suggest it.

- In (4), depending on how wild the function being tested is, and how good the tests, it might be acceptable for a reviewer to approve the request without knowing the syntax.

This can often present when there is a gap between reviewers and reviewees in language knowledge. A front-end browser engineer will write idiomatic, terse JavaScript at a higher level than a full-stack C# developer, even if that C# developer can fill in.

By answering, the reviewee contributes to the reviewer’s skills, making them a better collaborator in future. The same in (5) – by mentoring coworkers, the whole team is enlarged; time spent coaching is an investment with high returns.

- Using a feedback ladder can make it easier for reviewees to understand what is required. See “Use a feedback ladder” above.



NOTE

When a reviewer asks a question and the reviewee provides an explanation that clarifies the code, that explanation/change should be added as a comment and not left in the review tool.

Comments should be beginner friendly – they should make it easy for a **new but broadly competent reader** (imagine a knowledgeable developer working in their least fluent language and unfamiliar with the architecture) to understand what’s going on – and they should explain the purpose of the code or offer further reading, not restate what code does.



Be willing to talk

The intent of a review comments system is two-fold:

- 1 to highlight problems, ask questions and receive answers in a well-threaded and locally scoped asynchronous manner that is broadly sharable and can be collaborated on by multiple parties; and
- 2 to keep a public record of decisions and discussions so that anyone who needs to revisit a pull request can understand the thinking of those involved.

What a review comments system is not:

- a recorded public display of misunderstandings, disagreements, talking past each other and missing the point – preserved forever as a relentless drawn out back and forth.

It's fine to register disagreements in comments, but as soon as it's clear that:

- there isn't a clear path to resolution;
- that there is a misunderstanding;
- that people are speaking past each other; or
- that someone doesn't understand the discussion,

it's no longer worth the time and it's best to resolve in person. A reviewer or reviewee might trigger this by saying:

"Let's try clear this up on a quick call."

Whatever comes out of an in-person meeting should be recorded by the reviewee (unless a reviewer is more appropriate) in the comments. The reviewee might add the following to thread:

"From discussions, the new intention is to go ahead with an architecture meeting rather than attempting to resolve in this pull request. @kassandra_gitlab2 to schedule."

Reviewers should be willing to talk in person since responding well in text can be burdensome, especially when understanding an answer requires context.

As reviewers are encouraged to ask questions, they should make sure to receive answers in ways that are not overly burdensome to reviewees.



Be willing to pair

Often a reviewer will make a suggestion that the reviewee doesn't know how to fulfill timeously. When a reviewee is having difficulty, reviewers should be willing to help reviewees move remediation efforts along quickly, which also helps reviewees gain knowledge.



Ask for help and be willing to offer it

Reviewers will often discover code that they are not the best person to assess. When this happens, reviewers should request help:

“@jiang, @amir, this section that handles the concurrency is beyond my expertise. Could you weigh in on this discussion?”

When pulled into reviews like this, developers should respond as if they are reviewers.



Encourage tool use

Encourage developers to use automated tools to keep code at a high level. Reviewers should be generous with time to help reviewees configure their IDEs if suggesting tools they know.

When multiple tools with different outputs are in use by a team (e.g., two autoformatters with differing views), it can cause cycles of changes as developers with different tools do work, creating noise on diffs. In this situation, it's best to achieve consensus on a single tool.



Provide resources

Reviewers should provide links to readings, documents, standards, other lines of codes, definitions of done and anything relevant whenever possible. Reviewers need not trawl the net to find a link, but they should be helpful and reduce the cost of search for reviewees, and reviewees should feel free to ask for documentation.



Identify root causes

Errors may arise from issues in the organisation, such as shortfalls in training, onboarding, information capture and dissemination. Review is an opportunity for remedy:

- a developer guide might be missing how to run automated tests in a local environment;
- a standards document might be out of date or difficult to find; or
- decisions made in a meeting may not have been shared with everyone.

How to prepare code



Kindness flows in many directions. We should expect kindness from our reviewers, and we should show them kindness by taking only what valuable time we need. Everyone should feel supported and respected. Reviewers help achieve this by preparing code properly.



NOTE

Draft pull requests do not have to match this standard, but it's still the case that well-prepped code is easier on the reviewer, and the reviewee should be proactive here.



Include batteries

Anyone in the team should be able to check out, run, and make a pull request of their own with code in review. It must compile (if it compiles), tests must not be broken, cli commands must be updated and local development must be supported.

Reviewees should ensure they've checked all relevant standards, design documentation, guidelines and definitions of done before submitting work for review. This will be hardest for newer developers, and team members should be approachable in getting the code to standard quickly.

Once a developer has been on project for a while, they should ensure that they cover as much ground as possible before code is up for review.



Learn from past reviews

When a new developer arrives at a review, or when they have never worked at an organisation with a strong review culture, it's normal and expected for everyone else in the team, especially their lead, to put in work with them and offer mentorship.

Developers should be deliberate and conscientious to match more standards and expectations in each successive review. People don't expect perfection, just a clear improvement path.



Get feedback early

Review is an important moment for code. It's the first time it's made public, but that doesn't mean it's the first time it should be seen. It can pay off for developers to discuss their approach with a colleague before starting.

Developers should push code to branches early or set up draft pull requests (which can also be especially valuable for checking automated gates will pass) and ask colleagues more experienced colleagues how well they are doing.

By the time code is fully public, there should be little chance it's wildly off base.



Annotate work

It can be useful to annotate work post request by using comment/issue threads on the review UI to give context for decisions and help guide a reviewer.

Providing context and documentation for a reviewer can be valuable, especially by adding specification or task information usually captured in issues, tickets, designs and product specs.

A reviewee writing a high-level description of what they are trying to achieve can ensure the reviewer understands why they even have a request in front of them to begin with.

Annotations should only contain information that helps with the pull request – even just for a specific reviewer with a particular set of knowledge; when developers find themselves explaining the wider purpose of code in annotations or complex ideas in links to documentation, that means a comment in code should be preferred.



Look at the diff

Post request, reviewees should scan their diff for obvious errors. Reviewing text in a new environment (different typeface, background colour, layout, etc.) can prevent the mental short-cutting that occurs when one reviews their own work.



Small requests/structured commits

Small requests/structured commits should be preferred, all things being equal.

While difficult or impossible at times – especially in the early days of a project, often a request will implement a full system – smaller commits/pull requests are easier for reviewers.

Where easy to merge multiple pull requests, it is often best to split code between them. Where less easy, structuring commits becomes more important. The following is derived from *Small CLs*, Google Engineering Practices:

Small, simple pull requests/well structured commits are:

- Easier to review more quickly.
- Easier to review more thoroughly.
- Less likely to introduce bugs.
- Less wasted work if they are rejected.
- Easier to merge.
- Easier to design well.
- Less blocking on reviews.
- Simpler to roll back.
- Clearer in 'git blame' annotation/git history.



NOTE

Because of these benefits, reviewers may reject a pull request for being too large.

Standards may well say something about structuring requests and commits, but in general it is best for each commit to serve a narrow purpose, with a good commit message that explains it clearly.

Because opportunistic refactoring is so important in keeping a code base healthy, team leads must ensure everyone is sufficiently skilled with git to rework commits into desired shape without difficulty. If developers avoid refactoring work because they're unsure how to rework their git history, this requires intervention.

How to respond to comments



Be charitable

Reviewees should adopt the best case interpretation of reviewer comments. It might be wrong, but there is kindness in offering the benefit of the doubt. Usually, asking for clarification helps, as it's very rare a reviewer is deliberately mean. Giving a reviewer an opportunity to remake their point is better than reacting:

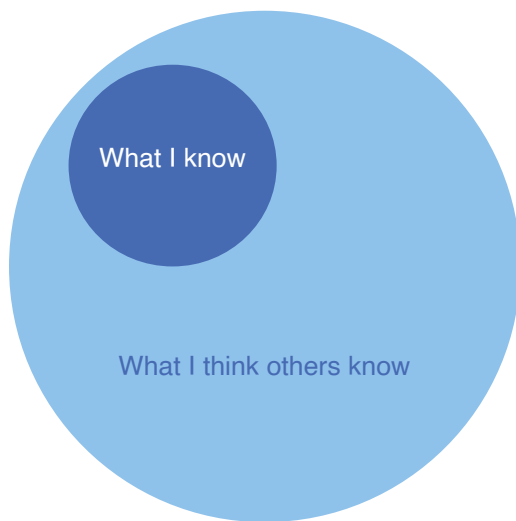
“Could you elaborate here?
I’m not sure I’m understanding the point being made.”



Expect changes

Exposing oneself to critique is hard, and can lead to a feeling of inadequacy known as imposter syndrome.

IMPOSTOR SYNDROME



REALITY

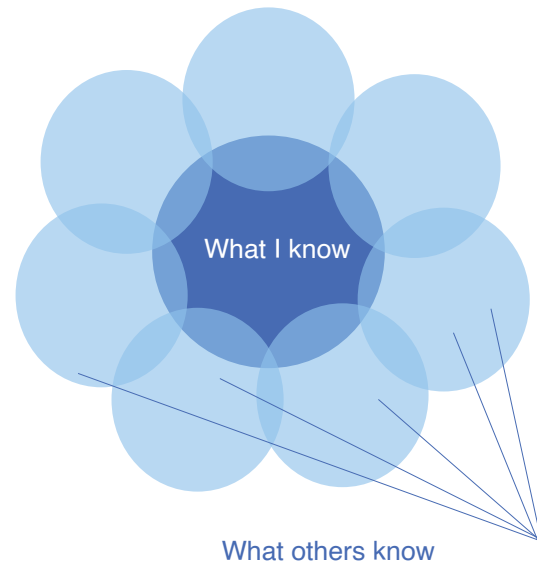


Image [source](#)

As illustrated, feelings of inadequacy can be fueled by reviewers commenting from multiple positions of expertise while reviewees are exposed in the full state of their knowledge (or ignorance).

By making review a safe place for ignorance and mistakes, we leverage all the expertise at our disposal.

The best teams maximise strengths and minimise weaknesses. When knowledge/expertise is brought to bear on code, it’s inevitable some of what was missed will be uncovered.

Reviewees should see their pull requests as the first step in a multi-step program, expect comments and changes, and see these as the review process working to:

- raise the quality of the codebase;
- share knowledge;
- develop team understanding; and
- help them improve.



Treat critique as a gift

Developers should be grateful for opportunities to learn – and few are as powerful as direct feedback on work. When reviewers are generous with information, thought, time and explanation, reviewees should respond with gratitude:

“Thanks for all this guidance.
Appreciate the feedback!”

“Thanks for the catch!”



Put the team first

Sometimes reviewees will be asked to tidy up code in proximity to the code they’re submitting or to make changes for consistency.

While this can be frustrating, some problems only come to light while code is under observation – and logging future work can cost time in progressing tickets through management processes.

Reviewees should graciously accept that altruistic changes may be requested. Being held up by changes one is not responsible for can be frustrating, having a clean, healthy code base to work in is payment.

Reviewers want the best version of the code in the codebase – when reviewees are unprepared to receive changes or react with frustration, it can be difficult and painful for reviewers to fulfil their role. It is a kindness to the reviewer to accept comments with grace.



Ask follow up questions

Reviewers can be unclear. If a reviewee is struggling with a comment, they should feel free to ask for clarification or information:

“I’m not sure I really understand what’s been requested here.
@arjun99, you’re making a point about the performance of this function, but I don’t really know how else it might be implemented.
Do you have any readings or examples you could send my way?”

Reviewees are entitled to clear instruction on requests, and errors or failures of clarity by a reviewer should be remedied.




Advocate for yourself

Reviewees should not fear disagreeing with reviewers. Reviewees are often best positioned to understand code and mostly have wide experience and expertise of their own. The three requirements are that reviewees:

- 1 disagree politely;
- 2 expose their reasoning; and
- 3 take reviewer comments/suggestions seriously.

By exposing their reasoning, reviewees ensure they're providing reviewers with enough information to evaluate their response:

"Thanks for the feedback! I'd like to advocate for the way I've done this a little:

1) We have competing guidelines for this case, and I felt that the guidelines on performance here have more bearing than the programming paradigm guide (which you've brought up), as this is a hot path. See: www.techcom.com 

2) I can probably work with your request on lines 14–25, but not sure how to make the codes on lines 46–92 performant without this technique. Open to ideas though."

Responding this way shows reviewers that their comments and suggestions have been thoughtfully considered rather than dismissed.



Respect reviewer time

Reviewers should, generally speaking, be willing to get on a call and explain themselves or be part of a pairing session. To show respect, reviewees should make sure they avoid acting helplessly. This can be unexpectedly tricky – when someone with more knowledge comes to a problem, it can be easy to defer.

To ensure reviewer time is well spent, reviewees should make sure they:

- are ready to run any code under discussion;
- have read or at least surveyed any resources provided by the reviewer;
- have done at least some research in any direction they've been directed; and
- are ready and willing to take notes and capture any new work arising from the conversation.



Be methodical

When updating a pull request and re-presenting it for review, reviewees should ensure that they have responded to all comments by:

- making the requested changes;
- responding with questions; or
- or responding with discussion/explanation of why they feel changes are not necessary.

Reviewees should avoid the situation **where they appear to believe their work is complete but comments are unaddressed**, as this can be frustrating for reviewers. Reviewees should treat comments as a to-do checklist where each item must be checked off before moving on.



NOTE

Reviewees should follow the guidance under “Standards – When should a pull request be approved? – All comments have been acknowledged and appropriately addressed” when deciding how to resolve a comment.



Clean it up now

It can be frustrating when under pressure to feel like scope is increasing with each comment. Sometimes, it's appropriate to capture new work in a new task/ticket.

Most often, work arising out of a pull request should be handled immediately, except in an emergency. When evaluating whether to log a new ticket/task or to handle it immediately, remember that **pushing tickets through the full cycle of write up, grooming and reproduction is very expensive**, as many people are involved.

It's often most effective for the developer with immediate context to handle an issue right away – and reviewees should be willing. From [Handling pushback in code reviews](#), Google Engineering Practices:



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 [change list (CL)] in. So they say they will clean something up in a later CL, and thus you should [looks good to me (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."

It is essential that management support this practice.



Embrace detail

Reviewees can find it frustrating to receive "small" comments on reviews, as if an issue shouldn't be big enough to elicit a comment. Calling out small errors can seem pedantic, but quality is a habit and standards die the death of a thousand cuts. It's the job of the reviewer to guard the long-term quality of the codebase, and it's important reviewees understand and embrace this.

Supporting documentation

Understanding others

- [*Give it five minutes*](#), Jason Fried

How to prepare code for review

- [*A Guide for Code Authors*](#), Chromium Project
- [*Small CLs*](#), Google Engineering Practices

How to write code review comments

- [*How to write code review comments*](#), Google Engineering Practices
- [*A Guide for Code Reviewers*](#), Chromium Project
- [*Feedback Ladders: How We Encode Code Reviews at Netlify*](#),
L. Cohn-Wein, K. Lavavej, S.Wang
- [*Code Review Guidelines for Humans*](#), Phillip Hauer

How to respond to code review comments

- [*Handling pushback in code reviews*](#), Google Engineering Practices

Strictness

Plane crash hospitals vs. car crash hospitals



On March 24, a pilot named Andreas Lubitz took off from Barcelona flying Germanwings Flight 9525 [...] Lubitz deliberately crashed the Airbus into the French Alps, about 100 miles northwest of the coastal city of Nice. All 144 passengers and six crew members died.

The easy response to this tragedy would have been fatalism: It's extraordinarily rare for a pilot to commit murder-suicide in the cockpit, but if he does, there's really no way to stop him. But that wasn't Lufthansa's response. [...]

On [March 27](#), three days after the crash, Lufthansa issued a policy change: It would require every flight to have two pilots in the cockpit instead of one. Other European airlines quickly followed suit, implementing similar protocol.

[...] Whenever a plane crashes, whether by mechanical failure or operator air, airlines and regulatory bodies immediately assume something went wrong — something that needs to be fixed on every plane that will ever fly again.”

Do no harm, Sarah Kliff

Some organisations deal with catastrophes as car crashes — acceptable risks. Others deal with them as air crashes — disasters that can never again occur. Most organisations are between.

Air-crash organisations use good processes to ensure outcomes - they make policy. In the article, Kliff relates how Peter Pronovost, a critical-care physician at Johns Hopkins University, took an air-crash approach and implemented a checklist reducing “central line infections at the Johns Hopkins University surgical intensive care unit [...] by 50 percent within three months of initiating the checklist. By six months, they were down 70 percent.”

Central line infections are infections of central line catheters: “doctors insert millions of them into patients each year. Because they run straight to the heart, central lines are the fastest, most effective method of delivering often lifesaving medication. But if bacteria manages to get into the central line — when a nurse changes a dressing or injects a medication — it can quickly become a bloodstream infection. At best, these infections cause suffering for already-sick patients. At worst, they kill them.”

Five steps to prevent central line infections

- 1** Wash hands using soap or alcohol prior to placing the catheter.



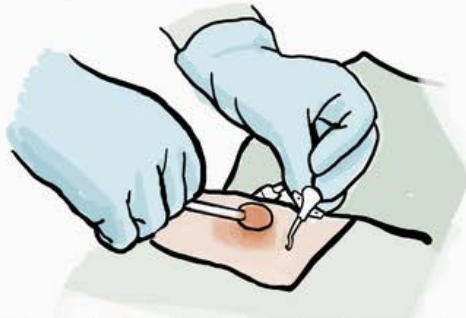
- 2** Wear sterile gloves, hat, mask and gown.



- 3** Completely cover the patient with sterile drapes. Avoid placing the catheter in the groin, if possible.



- 4** Clean the insertion site on the patient's skin with chlorhexidine antiseptic solution.



- 5** Remove catheters when they are no longer needed.



SOURCE: *Safe Patients, Smart Hospitals*. Peter Pronovost

Vox

All Pronovost had done was “create a simple five-item checklist that centered on obsessive, meticulous cleanliness when inserting the central line and changing the dressing”.

This is an astonishing result from so simple a procedure. Before Pronovost’s checklist, central line infections had simply been considered a cost of doing business in American hospitals.

What software teams can learn from Pronovost and air-crash organisations is the value of a good, targeted process in avoiding seemingly unavoidable problems.

The value of strictness



In *Is High Quality Software Worth the Cost?*, Martin Fowler writes on the topic of internal quality:

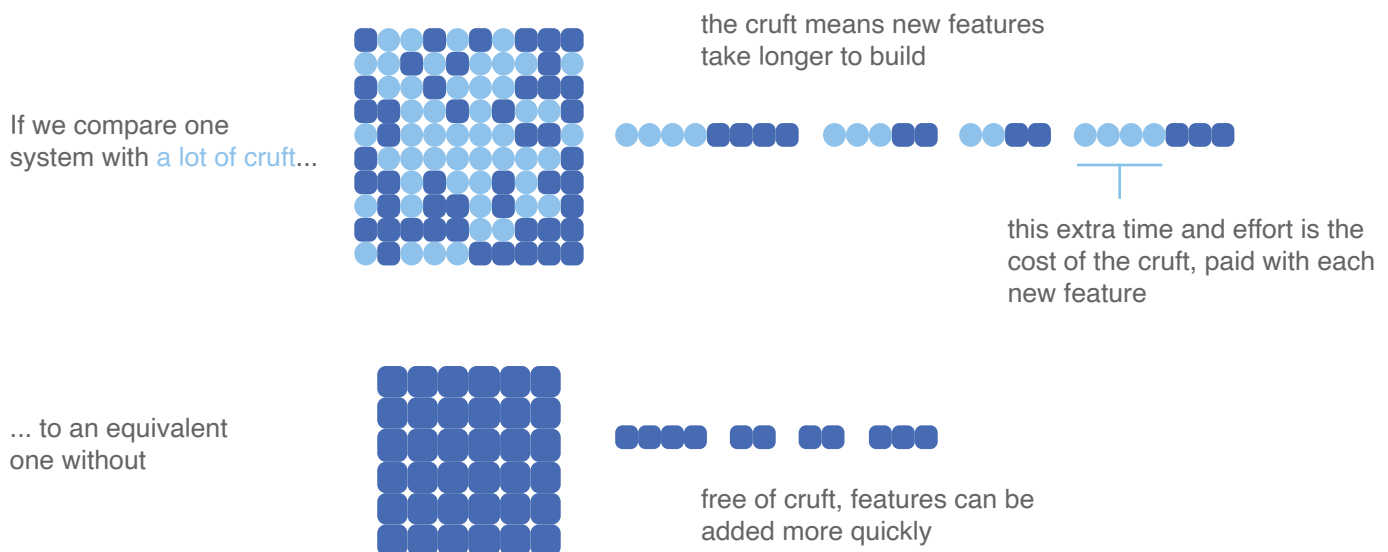
“[Why] is it that software developers make an issue out of internal quality? Programmers spend most of their time modifying code. Even in a new system, almost all programming is done in the context of an existing code base. When I want to add a new feature to the software, my first task is to figure out how this feature fits into the flow of the existing application. I then need to change that flow to let my feature fit in. I often need to use data that’s already in the application, so I need to understand what the data represents, how it relates to the data around it, and what data I may need to add for my new feature.

All of this is about me understanding the existing code. But it’s very easy for software to be hard to understand. Logic can get tangled, the data can be hard to follow, the names used to refer to things may have made sense to Tony six months ago, but are as mysterious to me as his reasons for leaving the company. All of these are forms of what developers refer to as cruft - the difference between the current code and how it would ideally be.”



The difference between the current code and how it would ideally be

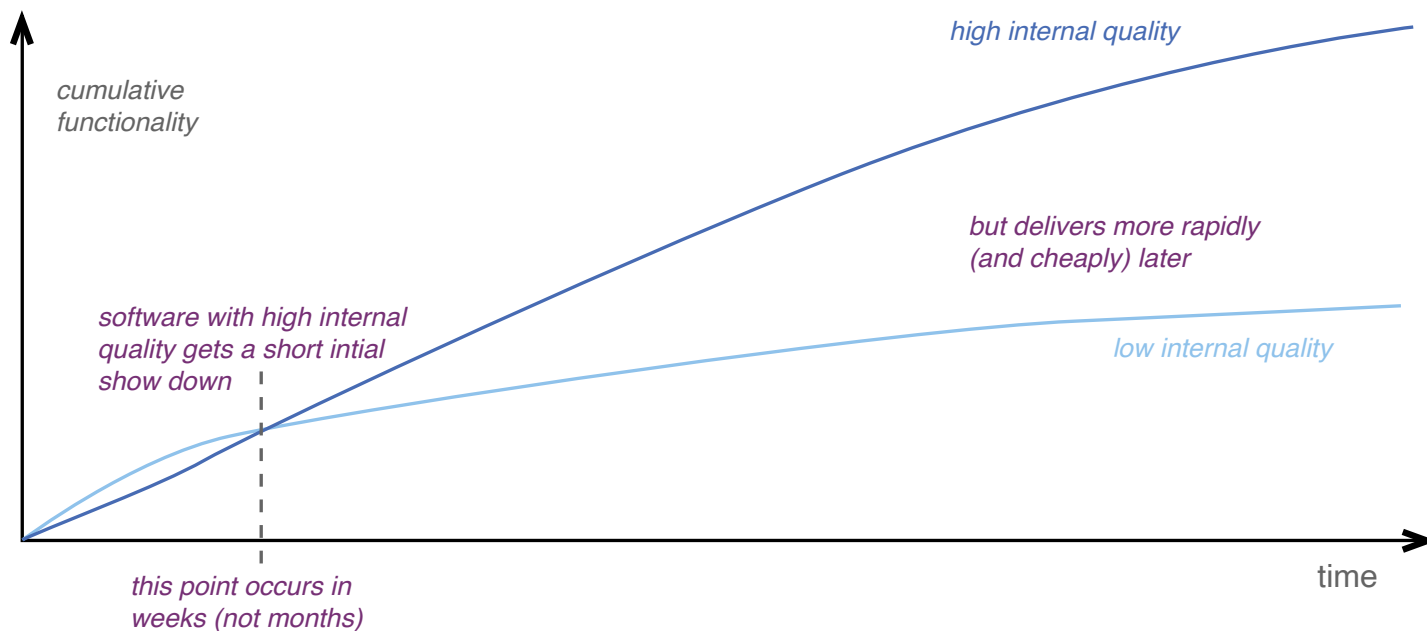
Fowler gives the following graphic to illustrate cruft accumulating in a codebase:



The central point: it’s cheaper to work in a code base without cruft.

This isn't **always** true. There are times when acquiring cruft is valuable. The old metaphor of technical debt looms large here – debt can be used as leverage. However, debt comes with interest payments – cruft.

Fowler believes, and this guide agrees, that the point at which cruft begins to cost arrives **quickly** – it isn't a theoretical long-term concern but an immediate threat, with little room for overspending. They offer the following pseudograph as a way of thinking about the impact cruft has on the quality and speed of software production.



Strictness to avoid cruft

Strictness is a way to manage a technical debt budget proactively. Quality is a habit, and cannot be tacked on at the end of a project. It arises from the day-to-day decisions of software teams – from their discipline and coordination, knowledge and attention to detail.

Cruft, like debt, is not merely bad code: mostly it's the cognitive overload of handling inconsistent architecture, folder structures, unused code, unclear variable names, missing documentation, code repetition, low test coverage, flakey tests, inadequate tooling and poor functional patterns – it is, at heart, the accumulation of poor attention to detail.

Comments at review provide a coordination point for teaching and sharing, ensuring that the widest group of developers understand the greatest amount of the product – not just its functionality, but also its organisation.

Strictness sets an aspirational standard for new developers, for themselves and for those training them, and helps them enter an environment that is well-organised and easy to navigate.



Strictness as an educational model

Strictness is a virtue of what might be called an open standard educational model. Open standard models present the full weight of requirements to learners up front.

Open standard model

Open standard models are not perfect, but they have some benefits, for example:

- learners are able to self-direct improvement in the absence of a teacher; and
- learners have a clear standard against which to measure themselves and know when they are doing well.

A downside is that the standard itself can feel overwhelming and discouraging.

Hidden standard model

By contrast, a hidden standard model protects the learner from being overwhelmed and intimidated by only introducing the next stage of development when the learner has mastered the previous one. The downsides are:

- the learner is always at the mercy of the teacher adding standards; and
- the learner is not able to educate themselves in their own time.

This guide takes the view that a supportive open standards model is superior when:

- new learners are not pressured to complete work on their own but are incrementally given guidance to bring work to standard; and
- work is not admitted until it is correct.



Strictness as an accelerator

Teams often feel pressure to cut corners, and strictness can feel the opposite of what's sensible. However, leniency can provide a false economy.

Look again at Fowler's pseudograph. While it seems as though one is saving time being lenient, instead one is costing oneself time in the very near future. By being strict in review, and making sure the greatest educational juice is squeezed from each opportunity, teams can produce a codebase that is easy to work and train a team well suited to maintaining it.

While strictness in review may initially result in large reviews with many comments, over time teams understand what is required and will naturally produce at a faster pace. Simply put, with investment, teams get both faster and better.

Supporting documentation

Plane crash hospitals vs. car crash hospitals

- *Do no harm*, Sarah Kliff

The value of strictness

- *Is High Quality Software Worth the Cost?*, Martin Fowler

Organisation

Management



A broad peer review program can only succeed in a work culture that values quality in its many dimensions, including freedom from defects, satisfaction of customer needs and business objectives, timeliness of delivery, and the possession of desirable product functionality and attributes. Recognizing that team success depends on helping each other do the best job possible, members of a healthy culture prefer to have peers – not customers – find defects. They understand that reviews are not meant to identify scapegoats for quality problems. Having a coworker locate a defect is regarded as a “good catch,” not a personal failing. In fact, reviews can motivate authors to practice superior craftsmanship because they know their colleagues will closely examine their work.”

Humanizing Peer Reviews, Karl E. Wiegers



Creating a culture of safety



Fear invites wrong figures. Bearers of bad news fare badly. To keep his job, anyone may present to his boss only good news.”

W.E. Demming

A culture of safety requires more than the mere absence of blame, sanction and unconstructive criticism. Rather, a culture of safety is created when individuals feel empowered to take an interpersonal risk, such as asking a question, seeking feedback, reporting a mistake or proposing a new idea.

Systems like the Westrum Organization Typology (below) place high value on organisations that have the qualities associated with psychological safety, naming them “Generative”, and those without them “Pathological”. [The Google Cloud Architecture Guide](#) makes explicit use of the model saying “a high-trust, generative culture predicts software delivery and organizational performance in technology.”

Pathological	Bureaucratic	Generative
<i>Power oriented</i>	<i>Rule oriented</i>	<i>Performance oriented</i>
Low Co-operation	Modest cooperation	High cooperation
Messengers shot	Messengers neglected	Messengers trained
Responsibilities shirked	Narrow responsibilities	Risks are shared
Bridging discouraged	Bridging tolerated	Bridging encouraged
Failure leads to Scapegoating	Failure leads to justice	Failure leads to inquiry
Novelty crushed	Novelty leads to problems	Novelty implemented

Westrum Organization Typology – *A typology of organisational cultures*, R. Westrum

This guide is built on similar principles, and is designed for cooperative work, removing fear of reporting problems, sharing of risks and bridging. At the heart of its process is the notion that **failure leads to inquiry**. When reviewers detect errors, they should treat them as an opportunity to explore.



Drift into failure

One of the purposes of this guide is to prevent a drift into failure by setting clear and explicit standards for developers.

Sidney Dekker [has argued that management pressure on processes can lead to a “drift into failure”](#). Organisations that lack explicit goals and values will must deal with decisions made in the face of conflicting priorities that are antithetical to global ends. Two kinds of drift are highlighted:

- 1 drift towards failure (lowering the margin of safety); or
- 2 procedural drift (deviation from normal).

When an organisation is silent on standards, random, natural drift from shifting implicit norms and outcome bias (past success justifying old practices) can take over and drive new norms.

Procedures must be well considered. Where rules are impossible, developers will naturally find workarounds and adopt these as local norms. Pressure on developers to ship where procedures do not support working to standards will naturally cause standards to be abandoned.

Understanding the role of the planning fallacy

The planning fallacy or optimism bias [is the tendency of people to underestimate tasks](#). One cause may be that estimation can only be done with reference to knowns but that actual work must, in its course, discover unknowns, which necessarily alter estimates. Whatever the true cause, it seems like a fact of life – and even those aware of the planning fallacy find it difficult to adjust for it.

Development must not be a bed of Procrustes. In the myth, “Procrustes (“the stretcher [who hammers out the metal]”) was a rogue smith and bandit from Attica who attacked people by stretching them or cutting off their legs, so as to force them to fit the size of an iron bed. The word “Procrustean” is thus used to describe situations where an arbitrary standard is used to measure success, while completely disregarding obvious harm that results from the effort.” ([Wikipedia – Procrustes](#)).

To avoid being Procrustean, there must always be room for discovering otherwise unforeseen tasks. In this guide, review is often the moment of discovery, and properly supporting this role means supporting the rescoping of tasks on discovery, where necessary.

Refusing re-scoping asks for two impossible things to occur at once, and teams have to either abandon standards or their compliance with expectations. Generally, teams will choose the second because of the personal cost of choosing the first, leading to a drift toward failure.



Showing commitment



Without visible and sustained commitment to peer reviews from management, only those practitioners who believe reviews are important will perform them. Management commitment goes far beyond providing verbal support or giving team members permission to hold reviews. [The list below] lists several clear signs of management commitment to peer reviews. If too many of these indicators are missing in your organization, your review program will likely struggle.”

Humanizing Peer Reviews, Karl E. Wieggers

Managers are often under extreme pressure or have heavy incentives to make decisions with short-term payoffs, which can lead to drift into failure. Without development management support, development team drift is inevitable.

Nothing in this guide will work without management support. To help managers understand what they must do to help, the following list is adapted from Wieggers’ entitled “Ten Signs of Management Commitment to Peer Reviews”.

Providing resources and time to develop, implement, and sustain effective review.

The primary resources are time and flexibility. At first, leads and developers will need permission to slow down so that they can learn these processes – and, in general, individuals will need permission to rescope work on the fly based on discovery in review – and suffer no penalties for making these decisions.

Setting policies, expectations, and goals about review practice.

This guide itself serves as an example for these.

Ensuring project schedules include time needed to perform reviews.

This guide asserts that project velocity will be greater if followed, and narrowly resolved measures of goal achievement – ticket completion within a sprint, for example – should be relaxed to advance pro-social goals. Timekeeping must permit review time as acceptable work.

Making training available to developers and attending training themselves.

Leads must schedule training time with their reports to make sure they are capable reviewers. Where there is wider training or seminars, development managers should attend.

Never using review results to evaluate performance of individuals.

If reviews are to be the place of free inquiry and learning, the outcomes of a review should not be metricised to evaluate performance.

It is acceptable for reviews to be used to support a conclusion about how a developer works or contributes to the team, but it should never be the case that comment or issues are tabulated, categorised and counted in a way that extracts them from their context and compares them, and they should not be used to set a threshold.

For example, it would be impermissible to consider a review with 10 comments a problem but not one with 9 comments, or to say that the developers with the most comments on their reviews should be penalised. It's plausible, for example, that developers with the most comments on their work are doing the most interesting work.

At worst, reviews may be used as textual support for an argument that draws on many other lines of evidence to support good contextual readings.

Holding people accountable for participating in and contributing constructively to reviews.

This document is a guide for how to constructively contribute to reviews, but if developers feel free to shirk the responsibilities outlined herein, the system will tend towards dysfunction, as those who care about the process will become increasingly burdened with the work of maintaining it.

Publicly rewarding the early adopters of reviews to reinforce desired behaviors.

Be generous with praise and reward for those who make the desired changes quickly and who defend processes and standards.

Running interference with managers or customers who challenge the need for reviews.

Ensure that a team has sufficient time and resources to achieve all that is hoped for them. Other managers, and sometimes customers, might not have the same vision, and teams can feel pressured to fulfil everyone's expectations. By running interference, managers show teams that they can follow their path in safety.

Respecting the judgment of an inspection team's appraisal of a document's quality.

When development managers second guess the judgment of their leads, or override them, they subvert the very processes they're supposed to be supporting. At best, their technical leads lose authority and the capacity to direct day-to-day development. At worst genuine errors can be inserted by someone who is not accountable, leading to learned helplessness.

Following up on how review is working, its costs and its benefits.

If teams value a process, development managers should know. They should certainly know if a process is a hindrance if sufficient time, resources and support have been given to trying it out. Asking team members directly about the costs/benefits they experience will give managers the knowledge to modify the process, if required – or defend it.



Measurement

Goodhart's law states that “any observed statistical regularity will tend to collapse once pressure is placed upon it for control purposes.”, or more colloquially, “when a measure becomes a target, it ceases to be a good measure.”

When introducing measurement, managers must ensure that they do not punish the behaviour they wish to see. Here we can turn to [Humanizing Peer Reviews](#) for Goodhart's law in action:



[T]he development manager announced that finding more than five bugs during an inspection would count against the author at performance evaluation time. [...] This misapplication of inspection data could lead to numerous dysfunctional outcomes:

1. Developers might not submit their work for inspection to avoid being punished for their results. They might refuse to inspect a peer's work to avoid contributing to someone else's punishment.

2. Inspectors might not point out defects during the inspection, instead reporting them to the author offline so they aren't tallied against the author. This undermines the open focus on quality that should characterize peer review.

3. Inspection teams might endlessly debate whether something really is a defect, because defects count against the author while issues or simple questions do not.

4. The team's inspection culture will develop an unstated goal of finding few defects, rather than revealing as many as possible.

5. Authors might inspect very small pieces of work so they don't find more than five bugs in any one inspection. This leads to inefficient and time-wasting inspections."

The way out of Goodhart's dilemma is to measure **that which would anyway be true when things are successful**. For example, some true measures of the standard of a (feature team) codebase are:

- How quickly can new features be added, adjusting for the fundamental size/complexity of the features?
- How quickly can the team move checked in code to production?
- How often do deployments cause failures?

In all cases, a lower number is always better, and trying to game it would almost always result in true gains.

Team/technical leads

Whether or not the system in this guide succeeds depends on management support, with team and technical leads in second place.



Model the standards

To get the best results, team and technical leads must model the standards to bring others on board. They should be the most knowledgeable about preparing code for review, must show kindness and patience, and should ensure their reviews follow standards and checklists.

Leads might initially ask a team member to keep them on the straight and narrow. If a lead adopts this strategy, they must ensure they empower that person to enforce standards.



Train reviewers

Team and technical leads must train reviewers to standard. Reviewers need their reviews reviewed, especially at the start of a project.

Leads should work with new reviewers on early reviews to check for missed issues, failures to apply standards, architectural alignment and high-quality comments. New developers generally arrive at code unsure of standards and conventions and, while able to offer advice on narrower issues, miss those more project or organisation specific.

When reviewing a reviewer, leads should conduct a review in parallel with a reviewer, so both review at the same time. When both finish, they should compare and discuss differences.

Leads should use judgment on which reviews to use for training. It will generally not be useful for a few lines to change a config, unless that touches all the larger systems in play.



Auditing reviews

Training reviewers is important, but without auditing it can be hard to stay aligned - natural drift is inevitable.

For new developers, after initial training, leads should sit in on about 1 in 3 **large or consequential** reviews – with a cadence of about 1 in 5 with more senior developers.

This is a starting point and leads must develop an audit cadence that suits them. Starting at a high rate before tapering off is useful, as early alignment pays high dividends. Leads should know the quality of review done in their teams, and the audit cadence should ensure that.

High quality reviews enable delegation by team leads. Trust that team members are operating at a high standard, empowers those members to advance the project without checking in.



Always be reviewing

For team and technical leads, review is an absolutely crucial responsibility. It is so important **that needing to work through pending reviews should be considered good reason for team and technical leads to reject or cancel meetings.**

It's not unreasonable at the start of a project that as much as two thirds of a team or technical lead's time is taken with code review, especially with pairing and in person reviews. Team and technical leads will often serve as the third reviewer and as the tiebreaker for discussions in reviews involving others where their being unable to receive feedback will be blocking.



Ensure fairness

Reviews are work. They're time consuming and require attention to detail. It's important that some team members – especially good reviewers – don't acquire the burden of doing everyone's reviews.

Training reviewers and auditing reviews is a way of ensuring fairness. When there are many good reviewers, the most diligent developers do not think "if you want it done right, do it yourself". A high common standard makes it easy to allocate a review to the next in turn.

Leads should ensure team members are fulfilling their share of review, as well as ensuring there is a ceiling to how long code can go without review.

Two strategies for leads:

- 1 if team members are not claiming reviews, finding reviewers as reviews come in; and
- 2 starting the day by allocating dangling reviews to make sure they don't sit longer.



Handling resistance

Resistance to the review process can have many sources. Leads should listen to any concerns and address team worries – that may mean adjusting a process; that may mean expanding on how reviews achieves some long-term goal.

High production pressure

The most common reason review practices fail is high production pressure. It's a lot for team members to resist pressure without support. Team leads should advocate for the benefits of the process and should do their best to protect team members.

Senior developer feels they have no peer reviewer

As knowledge is often siloed, senior developers may feel having juniors review their code is rubber stamping it. Leads should respond by emphasising the educational and desiloing gains of review.

The expectation is that senior developers will explain their code to more junior partners – that they will teach their solution; over time, more junior members will gain the context and expertise to make their reviews more valuable.



Leads might say: “yes, the review isn’t of the highest quality, but that if it isn’t done, the problem that reviews feel like rubber stamps will persist, whereas if evIEWS are used for training, the team will be better for it in the medium term”.

New developers can also play an important role in uncovering where code should be simplified or better documented.

Bikeshedding

Arguments over seemingly trivial issues can be a source of pain in review. Sometimes this is a symptom of inadequate tooling – for example, if linting and formatting tooling are poorly integrated into workflow, discussions can arise over actions that should be automated.

For more difficult cases, reviewers and leads should be quick to find consensus and include a record in a log. Consensus does not have to be right, but it should be followed until changed.

Disagreements about standards should result in a discussion aimed at finding consensus. Where consensus is hard or unnecessary, the team lead should make the call.

Difficulty complying with standards

Resistance to review can arise out of the strictness and granularity of standards. Where there are code formatting rules or linter-enforced practices, developers can feel overwhelmed by the things that they have to consider.

Fulfilling the following criteria is important:

- code-standards tooling should be automated;
- tooling should be IDE integrated with autofix where useful;
- where autofixes are not possible, automated fixes should be scripted, if feasible;
- linters must support override snippets;
- compliance with standards should be gated in CI;
- CI and local development must share configuration; and
- configuration must be kept in the repo.

Tooling that checks standards should, to the greatest extent possible, get out of the way. Developers should only be left to make decisions on issues that cannot be automated.

A developer has been running a repo/project solo

Sometimes a developer has been responsible for a project for a long time and has run things as they like. It's important to remember, in this case that, when adding requirements or making changes, one is treading on their territory.

While we may not like to think of code as territory, some sense of ownership is expected from people who invest in their work. Sensitivity to this is important, and leads asking for changes should be willing to discuss their ambitions at length – and listen where the developer has good reasons for doing things differently.

Style versus substance

There are more opinions about code than developers. Leads should be quick to tie-break disagreements field by mere stylistic preference.

The boundaries between style and substance are not firm, however. For example, the overall stylistic coherence of a codebase contributes to how hard it is navigating the whole of it.

Team leads are well within their rights to set a paradigm for the project: for example: object-oriented, functional, functional reactive, event or stream driven. When developers are writing code that is out of paradigm, it's best that leads explain their long-term vision.

Fear of being exposed

“ Asking your colleagues to point out errors in your work is a learned—not instinctive—behavior. We all take pride in the work we do and the products we create. We don’t like to admit that we make mistakes, we don’t realize how many we make, and we don’t like to ask other people to find them. Holding successful peer reviews requires us to overcome this natural resistance to outside critique of our work.

Humanizing Peer Reviews, Karl E. Wiegers

It can be difficult for those who came of age as developers in companies with punitive cultures to move past fear of blame and sanction.

Leads should be sensitive to this and make sure that their developers understand that making themselves vulnerable is on behalf of the team. Repeatedly emphasising that everyone is on the same side and that the most important thing is the quality of the code should help sensitive developers feel like they are not under a spotlight.

Rules



Approval rules

There should be at least one (non-author) reviewer of code and this rule should be set on the repo as a merge guard. If a team is large enough, say five or more developers, then this number should be two.

In most cases, one approver is an effective number; however, more may be useful at the start of the project where technical leads and architects might want an overview of all code.

Approval should void on changes

When code is approved and then changes, approval should be revoked. Approvers are responsible for the code they approve, and a means for changes to be made that they’re in principle unaware of should be avoided.



Approval and review groups

Those managing the repo should set up approval groups for developers to quickly add the correct subset of reviewers. If code is strictly business logic, it may make no sense to include the devops team. If code touches deployment, it may make sense.

Because there is such a strong injunction to attend to reviews quickly, it's important that people can tell immediately whether or not they're a person whose response is needed.



Assignees versus Reviewers

It can be useful to distinguish between assignees and reviewers.

On our account, an assignee is ultimately responsible for merging. They can be the reviewer (but may also delegate), and should ideally be a senior member. Reviewers should be knowledgeable people whose input into the code will improve it or those who need to be aware of changes (or who need to be introduced to the system in the first place).



Reviewers

Where reviews might have only one approver, it is good practice for there to be many eyes on code. Reviewees should do what they are able to involve more than one colleague, and reviewers should be willing to be the second or third reviewer.

Consequential changes especially should have multiple reviewers.

Tools



The most important tool for review is the review tool UI (e.g., GitLab, GitHub, Bitbucket). Developers should be familiar with its functionality, such as how to use comments, make suggestions, compare two arbitrary branches, set reviewers and assignees, find branches and commits, create pull requests and monitor CI/CD pipelines.

There are also other kinds of tools especially useful in the review context. Gates, for example, are blockers on CI and represent automated checks for standards.



Gates

Gates are blockers on CI and represent automated checks that code is up to scratch. There are many kinds of gates, including unit tests, end-to-end tests, static analysis, security scanning and more. A code quality standards document will generally lay out what's blocking.

Two kinds of gates useful in the review context are: auto formatters and linters – what we might call quick-response gates: low-cost, high-speed checks that rule out obvious errors and enforce scriptable standards. Many platforms permit these checks to be run first and block more costly tests on failure.

The worst possible way to spend time in a development team is arguing over minor stylistic points. Good gates and effective IDE integration are really important for preventing tedious [bikeshedding](#) – checks for compliance with style and format standards should be automated and autofix tooling should be configured and maintained. If a standard can be easily automated, it should be.

Autoformatting

The most useless way for development teams to spend their time is in discussing the format of code. **Almost any standard** is better than having even one conversation about it.

Autoformatters are different from linters in that autoformatters should never require developer decisions opinion, whereas linters often raise things for thought.

Language communities have different ways of solving auto formatting issues, some better than others (this guide does not vouch for the quality of these options other than prettier with eslint):

<i>Language</i>	<i>Solution</i>
go	gofmt, goimports
JavaScript	eslint, with eslint-plugin-prettier
Python	black
rust	rustfmt
C#	ReSharper
Java	Prettier-Java

Teams should invest in integrating these tools with their IDEs so they provide automatic format-on-save functionality. Tools that require code to be shaped in a particular way, must fulfil two strong requirements:

- 1 CI must break when code that isn't properly shaped hits the remote repo; and
- 2 configuration must be in code and must be the same locally and in CI.

The second condition should warn developers away from storing config in their IDEs – configuration belongs to the project.

Linting

Linters, by contrast, should highlight potential bugs and standards violations. Where possible, lint rules should be fixable automatically and scriptably, but it's sensible for a linter to highlight something a developer needs to consider.

Language communities have different ways of performing linting (usually with plugins – again, this guide uses the following as examples and does not vouch for them):

<i>Language</i>	<i>Solution</i>
go	golangci-lint , glint
JavaScript	eslint
Python	pylama , flake8
rust	rust lints you (OK, rust-clippy)
C#	StyleCopAnalyzers
Java	SpotBugs , PMD , Error Prone , CheckStyle

These tools should be integrated with IDEs and must support the same configuration locally as in CI.



Reading

The [github1s](#) project makes it possible to view any GitHub repo in a live VS Code instance by replacing the “github” in the URL with “github1s”.



Live code

Providing live code examples during review can be extremely useful. The following tools permit permanently hosted shareable links.

Replit	Shareable (with sign up) sandboxed REPLs for almost every language.
CodeSandbox , StackBlitz	For web, both provide high quality development environments.



Spell checker

Spelling errors are errors, and taking pride in one's work means being detail oriented. To prevent errors and raise the general quality of work, developers should make use of spell checkers.

<i>IDE</i>	<i>Plugin</i>
VS Code	Code Spell Checker
Visual Studio	Visual Studio Spell Checker (VS2017 and Later)
Jetbrains IDEs	Built-in and configurable
vim	Config

Supporting documentation

Management

- *Drift into Failure: From Hunting Broken Components to Understanding Complex Systems*, Sidney Dekker
- *Intuitive Prediction: Biases and Corrective Procedures*, Kahneman & Tversky
- *A typology of organisational cultures*, R. Westrum
- *DevOps culture: Westrum organizational culture*, Google Cloud Architecture Guide
- *The State Of DevOps Report 2019*, Google
- *Humanizing Peer Reviews*, Karl E. Wiegers

Team/technical leads

- *Humanizing Peer Reviews*, Karl E. Wiegers

