How to Review Code - Tips and Best Practices

https://google.github.io/eng-practices/review/reviewer/

Terminology

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

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

The Standard of Code Review

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

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

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

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

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

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

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

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

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

A key point here is that there is no such thing as "perfect" code—there is only *better* code. Reviewers should not require the author to polish every tiny piece of a CL before granting approval. Rather, the reviewer should balance out the need to make forward progress compared to the importance of the changes they are suggesting. Instead of seeking perfection, what a reviewer should seek is *continuous improvement*. A CL that, as a whole, improves the maintainability, readability, and understandability of the system shouldn't be delayed for days or weeks because it isn't "perfect."

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

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

Mentoring

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

Principles

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

Resolving Conflicts

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

When coming to consensus becomes especially difficult, it can help to have a face-to-face meeting or a video conference between the reviewer and the author, instead of just trying to resolve the

conflict through code review comments. (If you do this, though, make sure to record the results of the discussion as a comment on the CL, for future readers.)

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

Next: What to look for in a code review

What to look for in a code review

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

Design

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

Functionality

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

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

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

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

Complexity

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

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

Tests

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

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

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

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

Naming

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

Comments

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

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

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

Style

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

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

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

Consistency

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

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

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

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

Documentation

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

Every Line

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

If it's too hard for you to read the code and this is slowing down the review, then you should let the developer know that and wait for them to clarify it before you try to review it. At Google, we hire great software engineers, and you are one of them. If you can't understand the code, it's very likely that other developers won't either. So you're also helping future developers understand this code, when you ask the developer to clarify it.

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

Exceptions

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

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

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

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

Context

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

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

Good Things

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

Summary

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

- The code is well-designed.
- The functionality is good for the users of the code.
- Any UI changes are sensible and look good.
- Any parallel programming is done safely.

- The code isn't more complex than it needs to be.
- The developer isn't implementing things they *might* need in the future but don't know they need now.
- Code has appropriate unit tests.
- Tests are well-designed.
- The developer used clear names for everything.
- Comments are clear and useful, and mostly explain *why* instead of *what*.
- Code is appropriately documented (generally in g3doc).
- The code conforms to our style guides.

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

Next: Navigating a CL in Review

Navigating a CL in review

Summary

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

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

Step One: Take a broad view of the change

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

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

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

If you get more than a few CLs that represent changes you don't want to make, you should consider re-working your team's development process or the posted process for external contributors so that

there is more communication before CLs are written. It's better to tell people "no" before they've done a ton of work that now has to be thrown away or drastically re-written.

Step Two: Examine the main parts of the CL

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

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

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

- Developers often mail a CL and then immediately start new work based on that CL while they wait for review. If there are major design problems in the CL you're reviewing, they're also going to have to re-work their later CL. You want to catch them before they've done too much extra work on top of the problematic design.
- Major design changes take longer to do than small changes. Developers nearly all have deadlines; in order to make those deadlines and still have quality code in the codebase, the developer needs to start on any major re-work of the CL as soon as possible.

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

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

Next: Speed of Code Reviews

Speed of Code Reviews

Why Should Code Reviews Be Fast?

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

When code reviews are slow, several things happen:

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

How Fast Should Code Reviews Be?

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

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

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

Speed vs. Interruption

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

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

Fast Responses

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

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

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

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

Cross-Time-Zone Reviews

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

LGTM With Comments

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

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

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

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

Large CLs

If somebody sends you a code review that is so large you're not sure when you will be able to have time to review it, your typical response should be to ask the developer to split the CL into several smaller CLs that build on each other, instead of one huge CL that has to be reviewed all at once. This is usually possible and very helpful to reviewers, even if it takes additional work from the developer.

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

Code Review Improvements Over Time

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

Emergencies

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

Next: How to Write Code Review Comments

How to write code review comments

Summary

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

Courtesy

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

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

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

Explain Why

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

Giving Guidance

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

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

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

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

Label comment severity

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

Here are some examples:

Nit: This is a minor thing. Technically you should do it, but it won't hugely impact things.

Optional (or Consider): I think this may be a good idea, but it's not strictly required.

FYI: I don't expect you to do this in this CL, but you may find this interesting to think about for the future.

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

Accepting Explanations

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

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

Next: Handling Pushback in Code Reviews

Handling pushback in code reviews

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

Who is right?

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

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

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

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

Upsetting Developers

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

Cleaning It Up Later

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

If a CL introduces new complexity, it must be cleaned up before submission unless it is an emergency. If the CL exposes surrounding problems and they can't be addressed right now, the

developer should file a bug for the cleanup and assign it to themselves so that it doesn't get lost. They can optionally also write a TODO comment in the code that references the filed bug.

General Complaints About Strictness

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

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

Resolving Conflicts

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