Code Reviews

Good Practices for Collaborative Teams.

The merge review process is a common part of the software development lifecycle. This guide covers practices that can help improve the process and make it smoother and more enjoyable for your team.

Start →

⚠️ Links may not work in some browsers

Contents

Introduction	03
The Basics	05
Your Motivation	08
Empathy & Compassion	11
The Review	14
For Everyone	42

This guide was put together by the team at **#muster** to help dev teams work better together —making code reviews easier, more effective, and more enjoyable for everyone involved. We hope you find it valuable for your team!

—Shri, Vince & Kate

Intro

This document started its life as a “Best Practices” document, but “Best Practice” only applies within a specific idealised scenario.

Accepting the reality of life, the document is about good practices. All of the practices here will bring benefits. Some of them will bring more benefit than others, but by the same token, some of them are more work to implement.

You will have to decide how you want to take this journey of improvement of the review process. It is far better to implement one change that is sustainable than try to implement 5 changes that fall by the wayside in a month or two.

Slow and steady wins the race

—Aesop

When to review?

While there is good reason to gate the review on merge, this is not always the best method. There is no reason for it to be the only method either.

The review that happens at each stage has different priorities as well.

Before writing code

It's a good idea to come up with a plan of action for how you are going to solve the problem. You can come up with the initial solution collaboratively. Alternatively, once you have a viable solution, run your team through it to get additional opinions and perspectives. Course correction here is the easiest and the lowest cost.

The main purpose of the review here is to ensure that the approach is sound, no critical options or possibilities are missed, and more importantly that the correct problem is being solved.

Draft reviews

If there is some complexity it may be worth getting some additional perspectives once you have some tests and rough code. The draft review may also be valuable if you are new to the team or to software engineering.

The review at this stage again focuses on the overall approach and ensuring that there are no gaps in communication or understanding.

Ready to merge

Most reviews happen when a small enough bit of work is complete.

The main outcome here should be to ensure that the change does not negatively impact the code base and that it adds value.

The main question to be asked is: Is it good enough to merge?

There could be changes and fixes that could be completed in subsequent merges and that is to be expected.

After the merge

There is no reason to ignore a review because it has been merged. The best time to find an issue is when the code is being written. The second best time is before it impacts the customer.

Once the code has been merged, there is less time pressure and the review can happen with more rigour. Any improvements that come out of the review can be completed in subsequent merges.

Pull/Merge Request

These days, code reviews are usually done through Pull Requests (by GitHub, Bitbucket, Gitea, etc.) or Merge Requests (by GitLab).

Pull request, as a concept makes more sense in open source software where someone forks a repo and requests their changes to be "pulled" into the original repository.

Within the context of a team where the requests are internal, the term merge request is easier to understand.

Both the terms are generally used interchangeably.

While there are lot of team benefits from doing reviews, what are the benefits to **you**?

As a Requester

Getting your code reviewed isn't about getting it "checked" or "graded." It is about a collaborative effort to ensure that the quality of the code is as high as is reasonably possible.

Part of the benefit to you is that there are likely to be fewer defects in the code you write. Nobody wants the customer to be impacted by a bug (particularly a critical one at the weekend) and this is one way to try and mitigate that risk.

No matter how long you have been writing software, there is a good chance that you can learn something new from someone else reviewing your code. Continuous improvement applies not just to the code, but also to our abilities.

It is also an opportunity to mentor more junior members of your team. Learning to mentor others is an important skill in becoming a better engineer.

Asking for someone to review your code is a privilege. There is no need to be entirely responsible for the code you write. Lean on your team.

Encourage your team to ask questions. Perhaps you were too close to the code and didn't realise that some of it isn't easy to understand. You'll appreciate making it easier to understand when you come back to it in six months or a year.

As a Reviewer

Depending on your level of experience and your role, there are many benefits from reviewing code.

Reviewing code is undoubtedly a privilege and a valuable activity for the reviewer. When defects are found, it impacts the whole team and usually also impacts the customer. In a lot of cases, a code review could have picked it up and prevented it altogether. The time spent in code reviews is paid back many times over in the time saved hunting bugs and in the related restorative work. It will reduce stress and effort for future you and your team.

Having a better understanding of the code base as a whole has its own benefits. It is also an opportunity to align on tools and libraries in use.

There are further benefits based on your experience and role

Junior Developers

Of all the roles, the junior developer perhaps stands to gain the most from code reviews. It is worth taking every opportunity to do reviews. These are excellent opportunities to see how others solve problems and to ask questions.

Reviews aren't just about making software better. They are also about making sure that the code is easy to understand and to follow - for everyone. If you are unsure about something, ask the question. Maybe the code needs to be simplified.

Quality Assurance/ Product Owner/ Scrum Master

If you use Behaviour Driven Development (BDD)—and you probably should—it would make sense for the whole team to do some code reviews including the Product Owner. This opportunity is really valuable for the PO and anyone in Quality Assurance to close any gaps in understanding of what the code should be doing.

If you are not using BDD, it is still a good idea to try and make the tests, particularly the functional ones as easy to understand as possible. Peer review is an opportunity for the whole team to check their alignment. The better you are able to make use of it, the more value it will bring.

Senior/Lead Developers

The biggest advantage for senior/lead developers is the opportunity to check that the code being written is in line with your expectations and direction. Reviews are the most valuable opportunities to ensure this.

Unless you are in a small team, it would make sense to focus on the big picture and on mentoring the more junior members of the team to pick up the smaller details in the review.

Your experience will help you pick up potential issues that others may miss, and it is critical to flag these. In identifying these at this stage instead of in production, you are saving yourself a great deal of pain in the future.

Empathy & Compassion

Peer reviews are a social and collaborative activity. It is important that you bear this in mind when doing reviews. No matter what the issue or disagreement, **put the people first**. It is more important that you collaborate to a reasonable solution rather than getting the code into some "perfect state."

Remember, that one of the purposes of code reviews is for the whole team to pull in the same direction. Explain, clarify and discuss solutions and alternative ways of doing something. Ultimately, any action should be agreed together as a team as much as possible.

Priorities in communication

Be Kind

It is difficult to know what others are thinking, or what they are going through. No matter how you feel about someone's work, express it with kindness and compassion.

Be Polite

Regardless of who you are or who the other team member is, it is important to be polite. Being rude damages credibility and team cohesion.

Show Respect

Another valuable trait in any team member, the absence of which results in losing credibility, team cohesion and morale.

Talk about the code, not the person, and about how it could be improved. Avoid value judgments and make your statements as objective as possible.

No matter who found an issue or how embarrassing it might seem, it is more important to improve the code. Appreciate the input so that they are motivated to find issues for you next time.

Consider the **team and its members to be the most valuable**, and through a better understanding of each other and collaboration, aim to achieve the outcome of improved code.

The Review

The review is a whole team process and starts long before the request comes in.

As the team

Be holistic

As humans, we are prone to bias, errors, and unverified claims. Peer reviews can provide differing views which serve to question and dispel at least some of these qualities and helps to improve rigour, credibility, and diverse perspectives. These benefits can go beyond code.

The more holistically you can consider your reviews, the more value you can get out of it. Consider the impact of code everywhere, not just in the context that it was built.

The impact

How does the code impact:

- Infrastructure
- Costs
- Customer / User
- Support & Maintenance
- Marketing
- Other areas of the business

How can you ensure that code review takes all the relevant areas into consideration? What else could benefit from review?

Defining and continually improving processes that support a holistic approach is important.

If you don't already have an agreed set of guidelines and expectations, you should work towards getting that in place.

✨ The following areas should be considered.

Coding standards

- Use pre-existing ones if possible
- Branch naming (and validation)
- Rules for TODO's and other tags
- PR naming strategy: e.g. treat the PR title as the merge commit title

Git guidelines

- Conventional commits
- How much to curate and fixup commits (or whether to squash)
- Rebasing vs merging

Safeguards

- Modular Components to minimise merge conflicts
- Automated testing

Knowledge sharing

- How will you avoid single person dependencies for code reviews?
- How will you mitigate knowledge silos?

Continued...

Review process

- When is a PR in draft state and when is it ready?
- How should branch protection work?
- Target time until first review?
- Target time to merge per PR?
- What's the maximum size of a PR? (recommendations later)
- What constitutes a request for change instead of approval? i.e. what blocks a merge?
- When do you request a re-review?
- What kind of changes can happen in a subsequent PR? (How) do you track those requests?
- Who commits? Merge, Squash or Rebase? It usually makes sense for the requester to do the merge as a way for them to close off the work they have completed.

Review templates

- Who does reviews? How many reviews are required?

Communication

- Who needs to be included
- What areas of impact needs to be considered?

Ownership

- Which team is responsible for which parts of the code (set up CODEOWNERS)

Once you have agreed as much of the processes as possible, the next step is to automate what you can.

⚠️ Note: this not real code. Or as cool as DreamBerd 😉

```
# set standards and test against them
#
# e.g. pr size <= 200 or branch name in form: <project>/<ticket>
run validations
    validate branch name
    validate commit message
    validate pr size
```

```
# set rules for reviewing code
#
# e.g. 2 reviews required before merging to main
configure branch protection
    set minimum reviewers
    prevent push to main
```

```
# check the code based on your style configuration
run linters
```

Continued...

```
# auto-detect common errors in the code
configure error detection
    check for print statements
    check newline at end of file
    check for whitespace
```

```
# if relevant automatically assign reviewers to the PR
assign reviewers
```

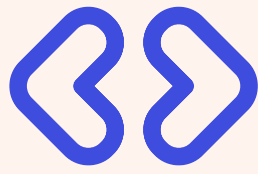
```
# build and run all tests
build
run tests
```

```
# deploy a test instance to check against relevant resources
deploy test instance
```

```
# static analysis to run checks without code execution
analyse code
    check vulnerabilities
    check errors
    check syntax
```

```
# check for dependency updates
scan dependencies
```

Automating these processes will make pull requests quicker and allow reviewers to focus on the more important issues. It also helps the requester to “self-serve” code quality improvements.



Metrics

Collecting the metrics of your development process is at least as important as all the metrics you collect from running systems and is often ignored in development teams.

The metrics you collect can help you identify process issues, improve and measure progress.

✨ Some of the valuable metrics you can collect are:

Size of PR (lines of code)

Time to first review

Reviews per PR

Cycle time (PR open to merge)

Defects (and seriousness)

Continuous Improvement

Another key part that should be done as a team is continuous improvement. Having an intention to monitor and continuously improve the process is a key part of agile and is no different when it comes to peer review.

The metrics you collect (see Metrics) should help with this. Metrics aren't everything though—you could bring cycle time down to zero by not doing a review, but that will almost certainly increase defects.

As the one requesting a review, the largest burden of the process falls on you. The goal is to get high quality reviews from a reasonable number of relevant people.

To achieve this, you want to make the review as easy as possible for the reviewers. One important aspect of the process is making sure the feature you are building is the feature that was asked for. The second part of this is validating that you are building it the way it is expected.

Once alignment has been confirmed we can move on to general code quality, bug finding, maintainability and the other benefits the review brings.

Think ahead while you're preparing a feature.

Before you start on your feature, it's worth considering **how you are going to split it into PR's.** Unless it's a tiny feature, you will likely want to do multiple self contained small PR's to complete the feature.

If the implementation of the feature is nontrivial, another thing you might want to do before you start is to work out a rough plan of how you are going to implement it, ideally collaboratively with the team. This can help validate alignment, both on what you're going to implement and how.

Each subsequent PR is an opportunity for further validation. If you need course correction, **the earlier it is flagged, the easier it is.**

While you're building, it's worth considering what will go in the PR description.

- What specific part of the feature is getting implemented?
- What screenshots would represent the functionality?
- Are there screen recordings that could help to understand functionality?
- What's the impact of this specific change?
- Are there any risks?
- What should the reviewer test? How?

It may help to start collecting some of this into a draft request, while you are working through the implementation.

With small code reviews, the reviewers will find it easier to hold the whole change in their mind, which makes it easier to reason about what the change does and its impact.

✨ Small reviews result in...

i.e. quicker merges and development of features

How small you ask? There is evidence that the ideal PR is 50 lines (or fewer), which might be too small. It is a good idea (1), (2), however, to aim for PR's to be **no more than 200 - 400 lines**. The limit will depend on factors like how verbose your language is (we're looking at you Java!).

Split the PR if it is too large, does more than "one thing" (e.g. two features, or a feature and a bug fix), or has high complexity, risk, or impact.

Creating a Pull Request...

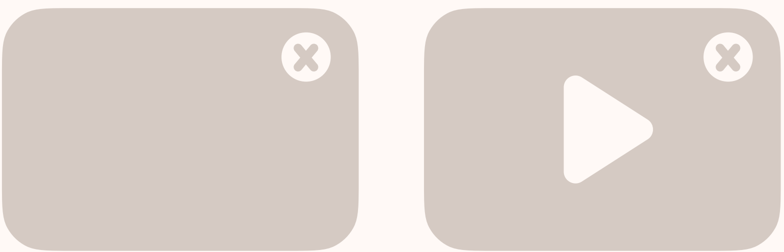
Ensure the PR has **more context than might be necessary**, particularly if there could be reviews from people who may not be fully aware of the motivations behind the change. Ideally the reviewers are already aware of the feature, the motivations and the approach for solving the problem before they ever get the request.

If the reviewers know to expect a PR and have a rough idea of what to expect from it, it will be easier for them to plan for it and to do the review.

Continued...

muster commented 1 minute ago

- Link to the related issue/feature/story, including:
  - Why is this being done?
  - What is the expected end result?
- What part of is being tackled, including:
  - What has already been completed
  - What's next (if relevant)
- What questions should the reviewer answer for you?
- Detailed instructions on how to test it
- Identified risks (if any)
- The impact of the change, including:
  - Infrastructure
  - Costs
  - Maintenance
  - Support
  - Customers
  - Data
  - Regulations (e.g. GDPR)



screenshots and videos of how it works and how to test the change

Continued...

The smaller the change the easier to be thorough on all the above.

Before assigning any reviewers, the final step is to **do a Self Review.** Put yourself in the shoes of a reviewer and do a review of the whole PR, starting with the above. It is better if a reviewer finds an issue instead of your customer. Similarly, it is better if you find an issue instead of a reviewer!

While waiting for the review, it is a good idea to do any pending reviews so that you can help unblock others work. If there are none that you can contribute to, you could use stacked PRs (aviator.co can help with this) to carry on with your work. Avoid building up multiple stacks though, as each stack will require additional effort to keep in line with "upstream" changes.

A good review is one which catches issues that you hadn't thought about. Better to **find them now than in production.** Appreciate the time and effort that the reviewers have put in to help you out.

Prioritise, discuss and implement the changes. Focus on the changes that are easy to implement, then the ones that are crucial to be able to merge the PR. Take a note of the remaining changes for subsequent PRs. It is a good idea to *fixup* with git to curate your commits rather than having commits that fix stuff from review.

Depending on your team guidelines, request re-reviews as required, but ensure that you are not using your reviewers as someone who checks your work. It is after all, a peer review.

The PR has been approved! So just merge it right?

Before you merge, ensure that nobody is currently in the process of reviewing your code. Reviewing code can take time and it can be frustrating if a PR is closed while in the middle of a review—particularly since Github can discard all comments that the reviewer was making! It is important to communicate with your team to avoid gaps in communication. The PR process **does not remove the requirements for communication.**

Once you have a request ready to merge, go ahead and follow the team guidelines on merging.

It is a good idea to **appreciate the reviewers for their input and feedback**. Software development is easier when done collaboratively. When your reviewers feel like their contributions are valued, they are more likely to enjoy doing reviews for you.

As the Reviewer

Reviewing code is hard, but is worth the effort. Here are some strategies to make reviews easier as well as more effective.

The quicker you can do the review, the better. The person who has requested the review could be (partially) blocked until the review is complete. This requirement, of course, has to be balanced with maintaining focus for yourself and minimising interruptions.

Reviews are not about checking someone else's work or about grading it. It is about learning (for both of you), collaborating and improvement (code and your abilities). Bearing this in mind, LGTM (Looks Good To Me) reviews add little value.

If you do not spot any areas for improvement, consider pointing out areas that stood out for you, or that you learned from. Review is collaborative and appreciating someone's work is an important part of collaboration and team building.

Goals

Why are you doing the code review?

If you are a more junior developer, or new to the team, you may be looking to learn about the code base or engineering in general. When you learn something, consider adding a comment. These comments can help to reinforce your learning, validate your understanding and appreciate the requester. **Code reviews aren't only about improvement.**

More senior developers may be more interested in preventing defects and maintaining or improving the quality of the code base. Mentoring and coaching could also be a part of that. Bear in mind that mentoring and coaching can also apply to other reviewers.

Having an idea of what you are looking to get out of the review is important. Remember that while doing a code review is hard, it is also a **valuable opportunity: to learn, to contribute and to collaborate** on curating the source code, which is an important asset owned by your team.

Before jumping into the review...

It is a good idea to limit a review session to no more than an hour. "Vigilance decrement" is the decline characterised by reduced accuracy and slower response times when performing tasks that require sustained attention. In fact, the clock test, a 1948 study showed a decline in ability by 10 - 15% after 30 minutes.

If a PR looks like it'll take longer than an hour to review, it may be better to ask them to break it up.

Another option might be to split the review into multiple sessions.

Carrying out the review...

If you are not going to be able to do a review or a re-review, let the requester know so that they can carry on. If there is something which is better reviewed synchronously, request a call/meeting as relevant. Sometimes a quick 5min call can save a 20min commenting conversation, not to mention potential misunderstandings.

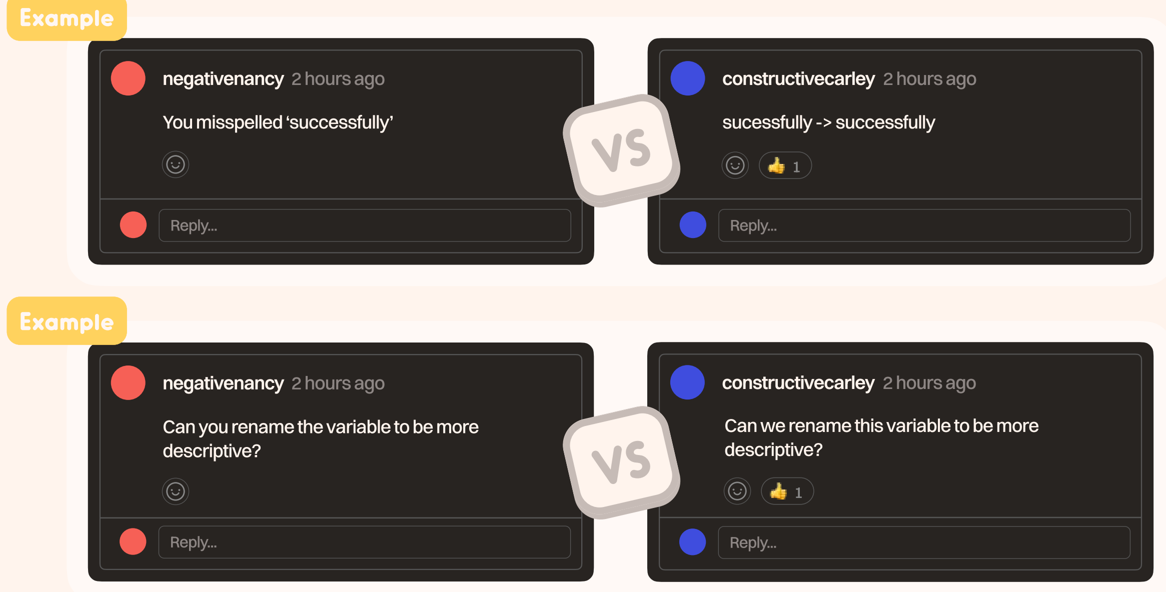
It is a good idea to pull the changes yourself and test it. It can help identify any assumptions about developer environments, locally installed packages and bugs that the requester was perhaps too close to see.

Communication

Being constructive in your communication goes without saying and helps to build a collaborative process focused on improvement.

Avoid "You"

To make reviews easier, it is a good idea to talk about the work and not the person. One straightforward strategy you can use is to avoid the word "You."



Using '**we**' also helps to reinforce the collaborative team nature of a review.

Continued...

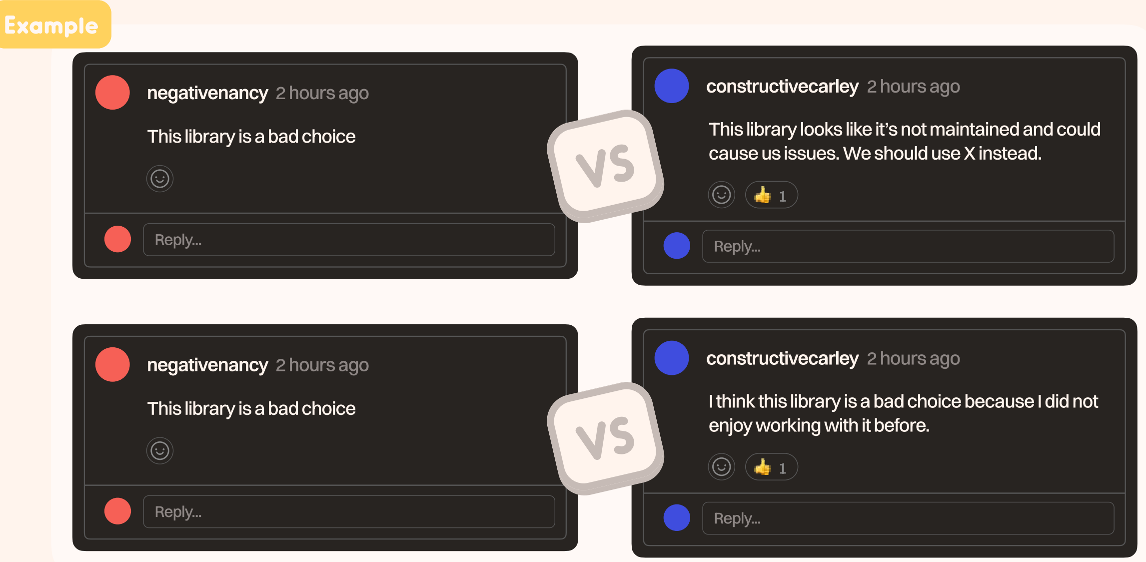
There are of course exceptions, particularly when you want their opinion, but be conscious that it does not come across as passive aggressive.

What do you think about X?

It might take a little while to break this habit, but it is worthwhile. If you would like to know more about this, you can look up Transactional Analysis (TA), a psychoanalytic theory and therapy that can help with communication.

Avoid value judgements

Whether something is good, bad, right, or wrong is usually too subjective to be constructive or valuable. Focusing on the why can be more helpful.



By making it clear that it is an opinion, it makes it easier to talk about.

Continued...

Next

Ask questions

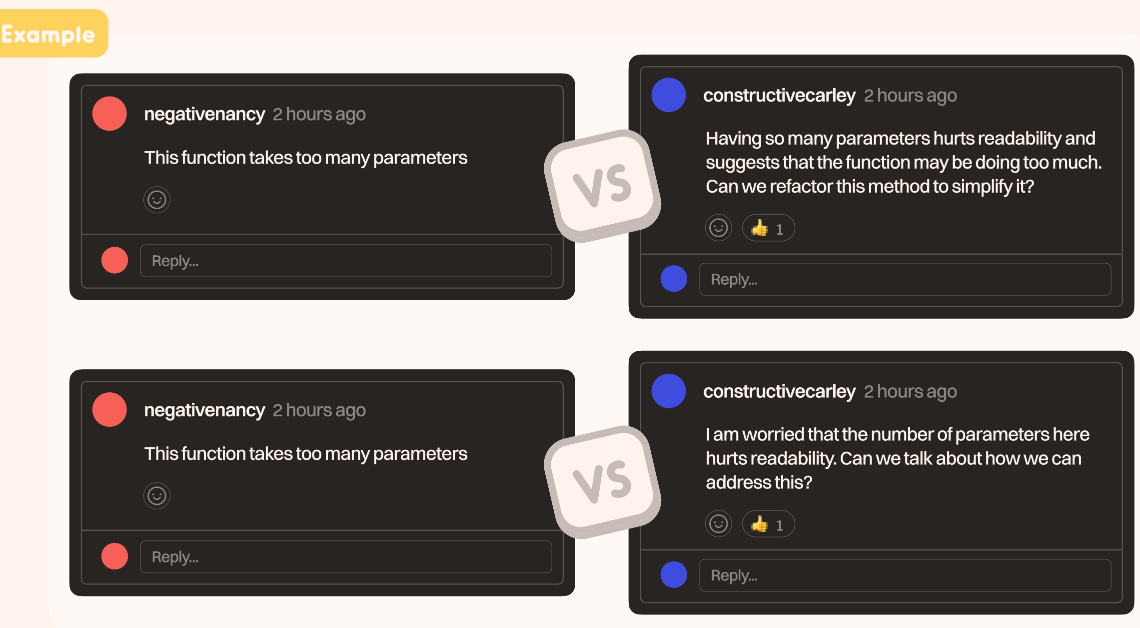
Regardless of seniority or experience, if you do not understand how or why something is the way it is, ask the question.

Code should be readable and easy to understand for everyone. The fact that you had to ask might be indication that the code needs to change to be easier to understand. It is usually better to make the code easier to understand than to add a comment with explanations.

Be specific and offer alternatives

When talking about improvements, strive to be specific and to offer alternatives.

Example



If you do not have a solution, it would be better to open a dialogue.

Collaboration is a powerful tool when it comes to improving quality of code and working out effective solutions.

When you see something valuable, or if someone has gone an extra mile, call it out and appreciate them. Code reviews aren't only about opportunities for improving the code. It's also **opportunities to improve the team cohesion, wellbeing and morale.** By calling out good work, you will also be encouraging it.

Consider phrasing nontrivial comments as invitations to discussion rather than being prescriptive. Collaborating to a mutually agreed outcome can provide an improved solution and can also lead to better ways of working.

If you can prioritise your feedback by category of importance, it can help the requester focus on the most relevant items first.

🚫 Blocker: Needs to be addressed before merge

⚠️ Follow Up: Needs to be addressed but can be in later PR

🔔 Minor: Could be resolved in a subsequent PR if time allows

While your focus ability is still high, **start with the big picture items.** Are there any blockers? Is there any reason this bit of code cannot or should not be merged? Are there any missing items?

After that, **focus on the smaller items.** As you get more senior, you may want to consider focusing less on smaller issues. Your time may be better spent encouraging and helping the more junior developers on the team to do reviews and find and report those. If you find most or all of the issues, your team may be inclined to leave the reviews to you.

If you come across code or changes that could benefit from review by people not currently on the list, it is a good idea to let the requester know and to bring them in. The requester may have a reason for not including them, so it may be worth checking with them first.

Try to consider the impact this change could have.

Complexity
Could anything be simplified, or even better designed away?

Alternative solutions
Assume the author has considered them but might be worth a discussion.

Dependencies
The fewer the dependencies, the better. Can some be removed?

Biases & Assumptions
Is there anything the requester is too close to see?

It may be impossible to consider everything, but it is good to be mindful of the impact the change could have. It is also a good idea to document any potential risks and possible mitigation.

After a PR has been merged... is that it?

Just because a PR has been merged does not mean it can no longer benefit from review. Reviewing PRs after they have been merged provides the same benefits. It is a good idea to work out how that process would work since platforms like GitHub are not built to support this workflow as well as the gated merge.

From a lean methodology perspective, it would make sense to automate as many of the checks as possible and automatically merge on pass. All reviews would then be completed post-merge. This process would have the benefit that all features are providing value at the earliest possible point.

It is important to **integrate your code continuously**. This process reduces waste by:

- Making merges easier (fewer conflicts)
- Potential defects are found sooner, and are easier to resolve
- Validating the direction of the feature as well as the method (i.e. where are we going and how)
- Allow reuse of code earlier

It may help to think of development more like **crafting or curating software, collaboratively, together.**

Be Gracious

No matter your role in the team or how long you have been a part of it, the cohesion of the team and the wellbeing of its members is crucial to its effectiveness. The quality of what you build is also important, but by and large, it is easier to fix software than it is to fix issues within a team.

Be mindful of actions that may negatively impact your or your teams wellbeing and cohesion. Be aware of your ego and if it is getting in the way of collaboration.

Be kind, courteous and considerate—Be gracious!

Click me

Get involved!

muster.chat/get-involved

Finish