How To Do Code Reviews Properly

Why Do We Need to Do Code Reviews?

An essential step in the software development lifecycle is code review. It enables developers to enhance code quality significantly. It resembles the authoring of a book. First, the author writes the story, which is edited to ensure no mistakes like mixing up "you're" with "yours." Code review, in this context, refers to examining and assessing other people's code. It is based on **the Pull request model**, popularized by open source.
A code review has different benefits:

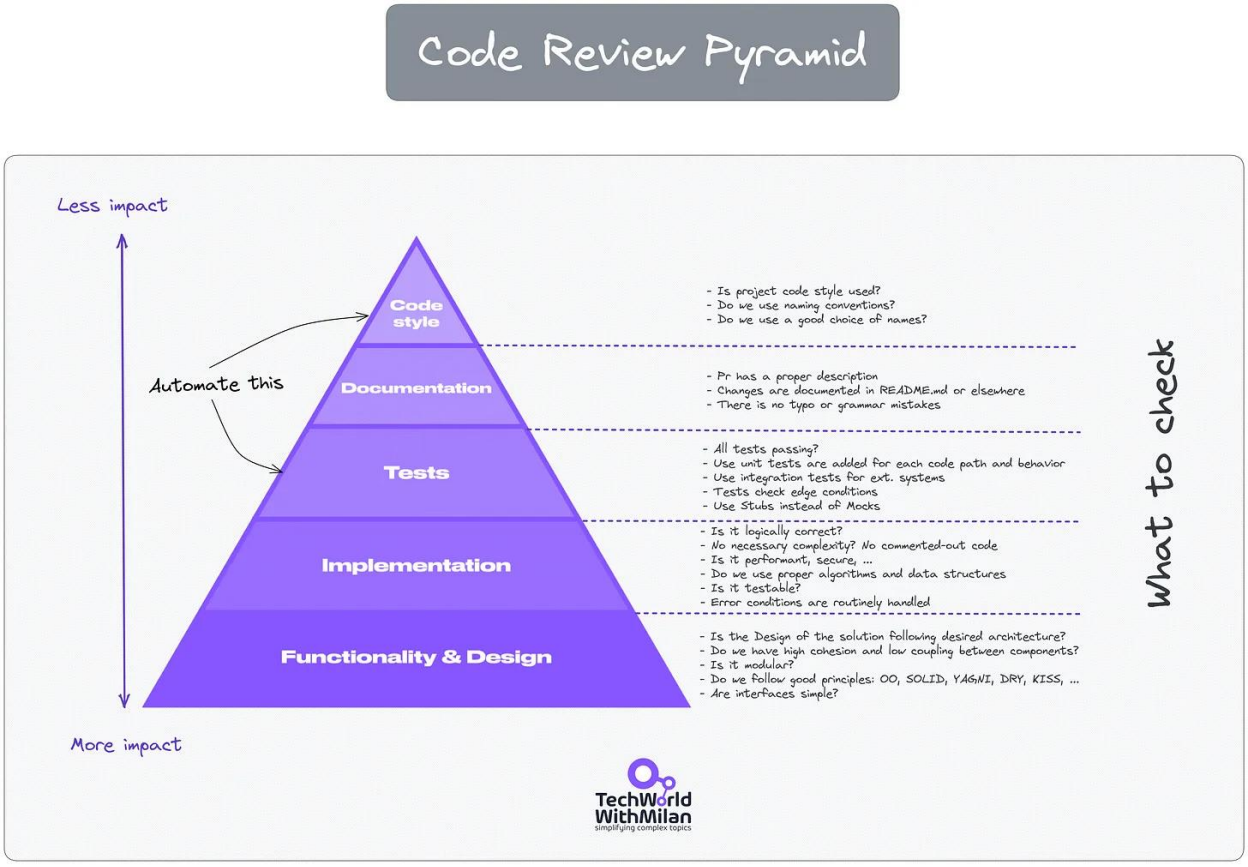
- It **ensures consistency in design** and implementation,
- **Optimizes code for better performance**,
- It is an **opportunity to learn**,
- **Knowledge sharing** and mentoring, and
- **Promotes team cohesion**.

What To Check In Code Reviews?

What should you look for in a code review? There are different things that we could check, from the different **levels of importance** and **possibility of automation**:
Try to look for things such as:

1. **Functionality & Design** - Here, we need to find the answer to questions such as: does this change follow our desired architecture, or does it integrate well with the rest of the system? Does it have high cohesion and low coupling between components? Does it follow sound principles like OO, SOLID, DRY, KISS, YAGNI, and more?
2. **Implementation** - Here, we check if the solution is logically correct (it does change what the developer intended), if this code is more complex than it should be, etc. **Is the code needed?** We also check whether we use good design patterns. And the critical stuff, such as **API entry point**.
3. **Testing** - Here, we check whether all tests pass, do we use unit tests for all code paths and behaviors, and integration tests for ext. systems (DBs, file system, etc.). Do we check all edge conditions and code coverage from 60-80%?

4. **Documentation** - Is our PR description added? Is our solution documented in the README.md in the repo or elsewhere updated?
5. **Code Styles** - Do we follow our project code styles? Do you know if naming is good? Did the developer choose exact names for classes, methods, etc.? Is the code readable?



Some Good Practices When Doing a Code Review

Here are some good practices when doing a code review:

1.Try to review your code first.

Before sending a code to your colleagues, try to read and understand it first. Search for parts that confuse you. Seeing your code outside an IDE often helps to view it as something "new" and **avoid operational blindness**.

2.Write a short description of what has changed.

This should explain what changes were at a high level and why those changes were made.

3.Automate what can be automated.

Leave everything that can be automated to the system, such as checking for successful builds (CI), style changes (linters), automated tests, and static code analysis (e.g., with SonarQube). We have integrated it **on the PR level**, so it will run on every PR and give blockers for code merge. This will enable us to remove **unnecessary discussions** and leave room for more important ones.

4.Do a kick-off for more significant stories with your team members.

If you are starting to work on something more significant, especially design-wise, try first to **do a kick-off with the code owner** or someone who will be your code reviewer. This will enable an agreement before the implementation and reduce the effort on the PR review level with no surprises.

5.Don’t rush

You need to understand what has changed—every line of it. Read multiple times if required, class by class. One should look at **every line of code** that is assigned for review. Some codes need more thought and some less, but it's a decision we must make during a review. Make yourself available for verbal discussion if you need to. Yet, try to make it under one hour. Everything more than this is not effective.

6.Don’t sit with the author

Try not to review the code with the author if you’re not working in a pairing mode (check the last section, “Better alternatives to code review”). Why? Because the author can influence the reviewer.

7.Comment with kindness

Never mention the person (you), always focus on changes as questions or suggestions, and leave at least one positive comment. Explain the "why" in your words and advise on how to improve it.

8.Approve PR when it's good enough.

Don't strive for perfection, but hold to high standards. Don't be a nitpicker.

9.Make reviews manageable in size.

We should limit the number of lines of code for review in one sitting. PR should contain **as few changed files as possible**. I prefer more minor incremental changes to significant sweeping changes. Our brains cannot process so much information at once. The **ideal number of LOC is 200 to 400 lines** of the core at one time, usually 60 to 90 minutes. If you have an enormous task, refine it into smaller sub-tasks that can be quickly reviewed.

10.Use checklists

Another thing you can do here is to have a **checklist**, which can be used to go through all aspects of a code review before adding reviewers. We use **pull request templates** in Markdown on Azure DevOps, which are applied to the description field when a pull request is created. E.g.

Thank you for your contribution to this repo. Before submitting this PR, please make sure the:

- [ ] Your code builds clean without any errors or warnings
- [ ] You are using the appropriate design
- [ ] You have added unit tests and are all green

These templates could be different for a branch or can be optional, too.

11.Use tools

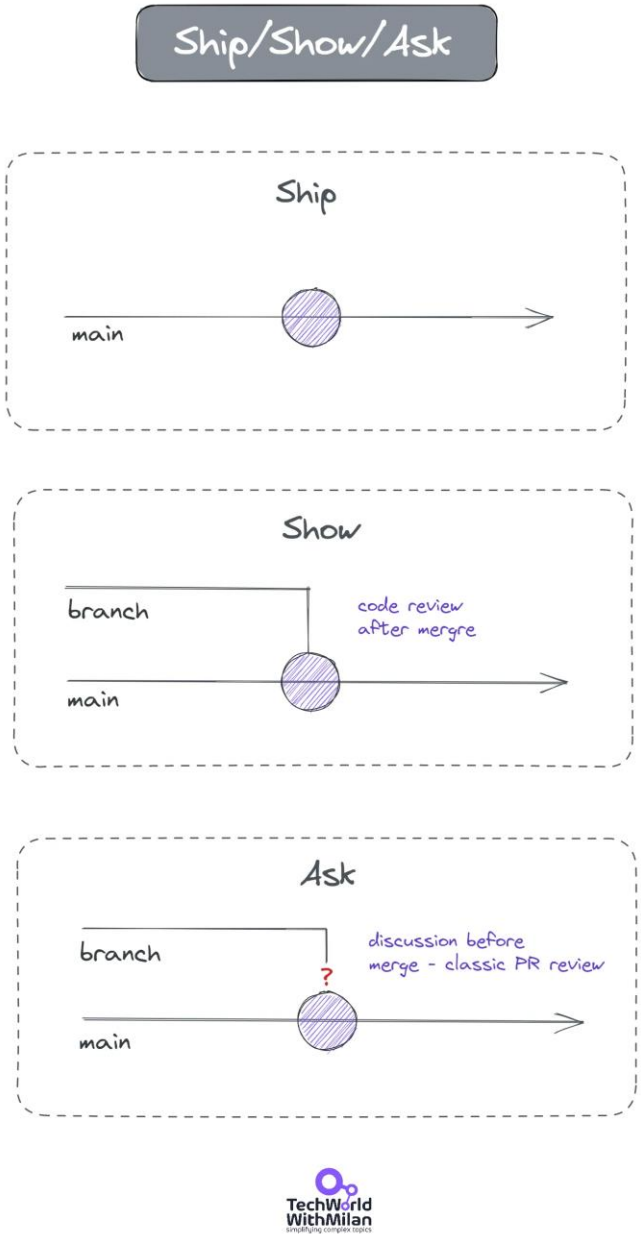
For all code reviews, you should use some **tools**, such as **BitBucket, Azure DevOps, GitHub, or GitLab**. For example, **Microsoft** has used an internal tool called **CodeFlow** for years, which supports developers and guides them through all code review steps. It helps during the

preparation of the code, automatically notifies reviewers, and has a rich commenting and discussion functionality. In the later years, they switched to GitHub Pull Requests. **Google** is also using two kinds of solutions for code reviews. They use the **Gerrit code review tool** for open-source code, yet they use an internal tool called Critique for internal code

How To Enable Continuous Integration with Pull Requests?

With Pull Requests, we lost the ability to have a proper continuous integration process in a way that **delayed integration due to code reviews**. So here comes a **"Ship/Show/Ask" branching strategy**. The thing is that **not all pull requests need code reviews**.
So, whenever we make a change, we have three options:

- **Ship** - Small changes that don't need people's review can be pushed directly to the main branch. We have some build pipelines running on the main brunch, which run tests and other checks, so it is a safety net for our changes. **Some examples are** fixing a typo, increasing the minor dependency version, and updating documentation.
- **Show** - Here, we want to show what has been done. When you have a branch, you open a Pull Request and merge it without a review. Yet, you still want people to be notified of the change (to review it later), but don't expect essential discussions. **Some examples are** local refactoring, fixing a bug, and adding a test case.
- **Ask** - Here, we make our changes and open a Pull Request while waiting for feedback. We do this because we want a proper review in case we need clarification on our approach. **This is a classical way of making Pull Requests. Examples include** Adding a new feature, major refactoring, and proof of concept.



A Better Alternative To Code Reviews

There is another alternative to classic PR reviews that could help you gain more efficiency and speed in your coding process. It is based on a model other than Pull Request, called **Trunk-based development**. Here, you synchronously have code reviews. In this way, all developers work on the mainline bra nch, **frequently committing to it**. An example of such practice is a **Collaborative programming** approach (Pair and Mob programming), introduced as an **Extreme Programming** technique by Kent Beck in the ‘90s.

Pair programming and mob programming are collaborative programming approaches that involve two or more developers working together on a single task, sharing ideas and thoughts while writing code. Here, one developer acts as a **driver** (writes code), while the other plays a **navigator** role (ensures the code accuracy). They **switch positions** from time to time during the process.
These methods offer **several benefits** compared to classic code reviews:

- **Real-time feedback**: Pair and mob programming allow developers to receive immediate feedback on their code, enabling them to address issues and improve. This contrasts with classic code reviews, where feedback might be delayed until the code review stage.
- **Enhanced knowledge sharing**: Collaborative programming enables developers to learn from each other's experience and knowledge, leading to better code and skill development. This is particularly helpful when working with new technologies or for newbies. Also, learning is **more easily spread among team members**, reducing the risk of a single developer becoming a bottleneck.
- **Faster problem-solving** encourages developers to work together to solve problems, leading to quicker and more efficient solutions. This can help to reduce development time and improve project outcomes.
- **Increased focus and productivity**: Working closely with another developer can help to maintain focus and reduce distractions.
- **Improved code quality:** When multiple developers work together, they are more likely to catch errors and design issues early in development.

This approach **works best when you have a co-located team of mostly senior developers who must iterate fast**. However, the Pull Request Model works better if you have a team of juniors or have a more complex product where more than one person needs to review the code.

---

code review process:

The code review process:

for maintaining code quality and ensuring smooth project progress. Recognizing when the process is flawed is the first step toward improvement.

Many signals allow you to understand if a code review process is flawed. Some of them are objective, and others are more subjective.

Signs of a bad code review process include frequently shipping bugs, rushed reviews with few comments, and team members feeling anxious about receiving feedback.

If PRs are often blocked over minor issues, require numerous review cycles, or involve continuous long discussion threads, it clearly indicates that the process needs refinement.

Conversely, a good code review process creates a more positive and productive environment where knowledge is shared inside the team.

Two possible ways a team can approach code reviews are synchronous or asynchronous.

Synchronous reviews, such as pair reviewing, benefit fragile or significant, complex changes. However, this approach is not ideal for identifying flaws and can hinder individual productivity.

Asynchronous reviews are generally better and can significantly enhance individual productivity. They allow reviewers to focus on the code, ensuring thoughtful and clear feedback.

This method also simulates how developers will see the code for the first time, providing valuable fresh perspectives.

Code review as an author:

To enhance the code review process as an author, it's important to establish clear expectations and provide thorough yet concise explanations.

It's best to create small pull requests (PRs), which are easier to review, less prone to bugs, and more time-effective. It's also crucial to ensure that the PR is ready for deployment by confirming that all builds are successful and that required tests are included.

A well-prepared PR should clearly explain the purpose of the code and the reasoning behind its implementation. The description should briefly outline the code's functionality and provide insight into the decisions and trade-offs that were made.

It can also be beneficial to include supplementary information, such as related tickets, test coverage, screenshots, rollback safety measures, and backward compatibility. Additionally,

providing links to relevant resources or documentation can offer reviewers additional context and help them better understand the changes.

Another important thing is to choose reviewers who are familiar with the codebase and can facilitate the resolution of any conflicts to ensure the PR gets merged.

Code review as a reviewer:

It is important for reviewers to understand what to look for and how to write effective comments.

As a reviewer, you should begin by gathering context from the review description and related tickets. Before going through the code changes, you should also ensure that the pull request (PR) complies with the team's guidelines on size and scope, that all builds and tests are successful, and that there are no merge conflicts.

The best approach is to review the code with a fresh perspective, checking for flaws inside and outside the diff. This includes looking for missed edge cases, potential optimizations, and any side effects on other parts of the system.

It's crucial to ensure that the changes are backward compatible and that the code is not over-engineered.

Effective comments should focus on the code and be constructive and clear. It's important to explain any suggested changes and offer possible solutions.

When making non-blocking suggestions, prefacing comments with "Nit, non-blocker" helps convey the nature of the feedback. As a rule of thumb, avoid commenting on style issues, as linters can handle those.

Leaving positive comments on elegant solutions or insightful approaches is also important to encourage and motivate the authors, especially if they are more junior engineers.

Write better code:

Independently of the process, writing better code reduces the number of review cycles and streamlines the overall development process

I won't make a long list, but there are three things to keep in mind to write better code.

First, show empathy to the future readers of your code. Value the reader's time more than yours and prioritize readability over convenience.

Second, be opinionated. Try to have a clear opinion on what constitutes good code and be able to justify your decisions. Reading books like "Clean Code" or "A Philosophy of Software Design" can help with that.

Lastly, be intentional when writing code. Ensure that every line of code serves a purpose and clearly communicates its intent.