International Journal of Emerging Research in Engineering and Technology
Pearl Blue Research Group| Volume 6 Issue 2 PP 71-80, 2025
ISSN: 3050-922X | https://doi.org/10.63282/3050-922X.IJERET-V6I2P109

Original Article

Code Reviews That Don't Suck: Tips for Reviewers and Submitters

Bhavitha Guntupalli [1], Venkata ch [2]
[1] ETL/Data Warehouse Developer at Blue Cross Blue Shield of Illinois, USA.
[2] Software Developer at Northern Trust Bank, USA.

Received On: 22/03/2025 ***Revised On: 17/04/2025*** ***Accepted On: 01/04/2025*** ***Published On: 20/05/2025***

Abstract: *In collaborative software development, code reviews are essential; yet, oftentimes they seem difficult, useless, or even hostile. This paper presents acceptable suggestions to increase the relevance and efficiency of code reviews, therefore addressing the typical problems found by submitters and reviewers. The statistics mostly support a paradigm change: seeing code review as a developmental conversation among peers rather than as a gatekeeping tool. This entails focusing on clarity rather than wit for reviewers, replacing help with sarcasm, and stressing the goal of the code rather than only its syntax. For submitters, it means aggressively clarifying design decisions, embracing comments with their openness, and viewing review as an opportunity for development rather than just a procedural need. The article offers ideas and techniques to handle these recurring problems, including unclear remarks, too much inspection, long review times, and false expectations. Typical motifs are clarity, empathy, technique, and tools. Empathy is learning the humanity concealed under every set of rules. Clarity is writing even for your future self clear, understandable comments and commitments. The procedure includes well-defined policies and timetables to help assessments stay on their intended route or avoid stretching too long. From linters to review templates, tooling can help to reduce friction and direct focus on these critical issues. Regardless of your experience level as an engineer, this piece offers sharp analysis and useful advice to turn code reviews from unwelcome chores into major team building, trust, and technical knowledge possibilities.*

Keywords: *Code review, pull request, software development, peer review, code quality, submitter tips, reviewer guidelines, DevOps, continuous integration, engineering best practices, collaborative programming, version control, GitHub, merge request, productivity.*

1. Introduction

Code reviews are very important in today's software engineering. If you work for a big company on ancient systems or a startup on new ones, code reviews are a great method to make sure your code is good. They make it easy to change code, find errors before production starts, and hold team members accountable for their work. They are more than just a technical review; they are a big part of engineering culture since they help people learn, mentor, and work together. Code reviews are good for the code and the people who work on it when they are done right. If done wrong, they could look like bureaucratic obstacles or fights between people with enormous egos, which would be bad for everyone.

1.1. A Brief History of Code Review Practices

It's not a new notion to evaluate codes. It derives from the time of early mainframes, when tests were done in controlled, formal settings known as "inspections." Most of the procedures were mechanical and focused on checklists. The most important items were meticulous analysis and documentation. As software development changed from waterfall to agile to DevOps, code reviews changed too. Version control technologies like Git and platforms like GitLab, GitHub, and Bitbucket made things easier, more consistent, and sometimes even asynchronous. Right now, the pull request (PR)-based review is the most used standard. It campaigned for constant integration methods, comments in line, and peer reviews. This shift made things run more smoothly, but it also caused new difficulties, such as informality, inconsistency, and disputes between workers, which often made the benefits of code reviews less evident.

1.2. Why Code Reviews Often "Suck"

Many engineers find code reviews unpleasant even with their best of intentions. Tickets can create challenges in the development process since they lie waiting for clearance by any one person. Reviewers may rapidly evaluate their work or exhibit too much pedantry, stressing aesthetic problems and neglecting structural defects. Sometimes they follow personal tastes as rigid guidelines. On the other hand, submitters may become defensive, judgment-oriented, or terrified about the predicted encounters. Reviewing takes gatekeeping as its first importance above development. Ego, tone, and ambiguity make things worse. Saying something like "this is incorrect" lacks weight in the absence of context or alternatives.

Particularly in a sprint cycle with high stress, direct or imprecise comments could be interpreted as an insult. Add remote teams, diverse time zones, and changing criteria to generate a formula for malfunction.

Code Reviews That Don't Suck
Tips for Reviewers and Submitters

Fig 1: Effective Code Review Process: Best Practices for Reviewers and Submitters

1.3. Setting the Stage for Better Reviews

It is not necessary this way, though. Code reviews shouldand should be cooperative, polite, and quick. They are about creating a shared knowledge of the codebase, uncovering better solutions, and training less experienced developers, not only about spotting mistakes. Fixing code reviews fixes the processfrom tooling to communication styles to review techniques. Empathy first drives it: realizing that every comment has a human on the opposite side. It develops clearly with language that guides rather than inflames. It scales with process by means of agreed rules and expectations meant to lower uncertainty.

Toolinglinters, templates, bots, and integrations that simplify the humdrum so individuals may concentrate on what countshelps to assist this as well. We will discuss in the parts that follow how both reviewers and submitters could make little but significant changes to make code reviews suck less and turn into a force multiplier for team cohesion, individual development, and product quality. Whether you want your team to work more cooperatively or you have review burnout, it's time to consider how we review codeand why.

2. Foundations of a Good Code Review

Code reviews are cultural events influencing team chemistry, cooperation, and development instead of merely technical procedures. When done consciously, they are indispensable members of good engineering teams. This part examines the fundamental concepts of effective code reviews, emphasizes the relevant values they support, specifies the features of quality, and looks at the pitfalls many teams come across.

2.1. Purpose of Code Reviews

Code reviews have mainly three connected purposes mentoring, knowledge exchange, and quality assurance:

- Quality Assurance (QA): The First and most important goal is quality assurance (QA). Reviews assist in locating design faults, mistakes, or unplanned results before codes are utilized. Human reviewers, unlike machines or linters, provide context and intuition to find logical errors, edge cases, or usability issues that they usually overlook.
- Shared Knowledge: The reviewing of the codes makes it possible that the members of the team are exposed to the areas of the codebase that they might be personally involved in. This, in turn, facilitates the reduction of "bus factor" problems and the improvement of cross-functional awareness. This exposure gradually advances a huge shared understanding of architectural patterns, coding standards, and system design.
- Mentorship & Growth: The code reviews are the most effective learning method for brand-new engineers. It gives opportunities for seasoned engineers to confirm that their practices are correct and they can be mentors and guides. A thorough review not only enables submitters to repair their code but also to get to the very root of changes, thus being able to contribute more effectively in the future.

A code review done successfully means it not only finds and solves issues but also establishes good habits and team spirit and reaps benefits through collective action.

2.2. Attributes of High-Quality Reviews

Though the "ideal" review has no clear criterion, outstanding reviews usually show a few basic traits:

- Constructive: Productful: Crucially important are the tone and context of feedback. Instead of saying, "This is incorrect," "This could be simplified to X due to..." One aims to empower rather than to limit. Pay closer attention to the code than the programmer would.
- Respectful: Reviewing something cannot ever seem to be a personal attack. Respect consists in knowing tone, avoiding sarcasm, and assuming good intentions. Even when one is solving issues, empathy and compassion greatly improve the outcomes of conversations.
- Timely: A slow assessment can affect a sprint even in five days. Timely comments help teams to stay free from obstacles and to keep a high growth speed. A simple "I will review more thoroughly by the end of the day" shows awareness.
- Context-Aware: Context-awareness of the overall background of a pull request (PR)including the functionality, the business justification, and the constraints"helps to prevent misaligned comments." Reviewers should closely review the PR description, probe clarifying questions, and make sure their

comments complement the scope and intent of the change.

- Balanced: Just as important for identifying issues is stressing strengths. Encouragement of well-organized projects, suitable titles, or complete test coverage helps to promote good behavior and renders evaluations more positive than aggressive.

2.3 Common Pitfalls

Even codes that operate with good intentions can still go wrong. Examples of anti-patterns that negatively affect team performance are presented here.

- Over-Commenting: Reviewers who are overly focused on pointing out every line of the code, especially minor stylistic details, become unneeded distraction sources. Submitter can be overloaded by this, the iterative process can be disturbed, and the really significant input can be hidden.
- Nitpicking: Getting rid of unclarity, if one point is definitely true, arguing endlessly over whether to use spaces or tabs, the location of the curly braces, as well as the length of the variable name, takes away attention from the main problems. Instead of human dispute, automation with linters and formatters should solve many issues.
- Slow Responses: Deferred assessments lead to lowered output. Usually, they are the source of merging troubles, context switching, or the creation of outdated branches. Choosing "actual work" over code review reduces its value and transforms it into an inconvenient bottleneck for teams.
- Unclear Feedback: Such comments as "requires improvement" or "that is not what I expected" are too vague to be useful. Clearly, they offer neither a direction nor pinpoint exactly that which has to be done. Good comments precisely and practically, thereby clarifying the need for adjustment.
- Reviewing Without Understanding: Reviewers are at times rushing through the review of a PR or they do not accept reasoning fully and therefore criticism is created. In this way, omitted problems or errors in the comments are a consequence of that. Good comments depend on knowing the process, particularly for big pull requests.

3. Tips for Code Submitters

Turning in codes for review could make one sensitive to criticism, exposed, and vulnerable. Code contributing can turn into a fun and helpful hobby with the correct mindset and few reasonable guidelines. This section looks at achievable plans for submitters to enable more efficient review cycles, lower conflict, and raise code quality.

3.1. Write Clean, Self-Explanatory Code

Fewer comments in a code review will come from writing code needing no justification. Clear, concise, understandable codes simplify the review process and reduce the cognitive load on testers.

3.1.1. Clean Code Means

- Clean codes seek unambiguous variable and function names.
- Keeping precise, focused abilities.
- Eliminating duplicate TODAYs or commented-out codes.
- Using the style guide of the project, keep consistent in your writing.

When clarity will work, be innovative. Describe your goal more precisely using more words than with a convoluted one-liner. Think about your future colleagues (or perhaps your future self) responsible for upholding the code six months hence; neat code is a modest act of compassion toward them.

3.2. Use Meaningful Commit Messages

Committed messages are one underappreciated instrument available during the submission process. They enable reviewers to grasp your perspective and the reasoning behind the modifications.

Notes on good commitment:

- Begin with a brief overview. Build a retry mechanism in the API client.
- Retries help to lower sporadic 503 errors coming from upstream systems.
- Sort important modifications into several commits with various purposes.

Avoid using such language as "fix," "changes," or "oops." Reviewers should not have to go through every file to grasp the changes done. A good commit statement acts as a guide; it facilitates appropriate negotiation of your PR by reviewers.

3.3. Keep Pull Requests Small and Focused

With 1,200 lines, the most demanding pull request for a reviewer touches a fair portion of the codebase. Examining more takes more time; comprehensive pull requests are more difficult to grasp; typically, they suffer delays or disregard.

Simplify your pull requests by:

- Dissecting big ideas into little adjustments.
- Staying with one task each pull request helps to reduce scope creep.
- Unless absolutely requiredthat is, adjusting irrelevant formattingavoid doing surface repairs.

If your reviewer needs a pot of coffee and a three-hour interval to review, generally speaking, your pull request is too big. Apart from their simplicity of evaluation, small pull requests are also more under control for debugging or reversal should an issue develop.

3.4. Add Context: PR Descriptions, Diagrams, Links

One of the most untapped resources in a submitter's arsenal is the pull request description. A well-crafted description not only allows reviewers to get a better and more focused understanding of the main issues but also saves them time.

What to include:

- A statement of the purpose of the PR
- The reason it was necessary (the issue it addresses)
- The way it fixes the problem (main design decisions)
- Visuals or test results for UI or performance changes
- References to tickets, discussions, or design docs

This information provides reviewers with a mental picture of the situation before they dig into the code. But even a clever PR can seem like a mystery without these supplementary details.

3.5. Preemptive Comments: Explain Design Choices

Predict questions before they even come. If there is a design decision that could be perceived as questionable, a short explanatory note in the code or PR description would be enough to clarify it. Besides building trust, this also enables the review to be done faster by cutting out unnecessary clarifications. It clearly demonstrates that you have considered possible concerns and that you are ready to discuss them. Moreover, it also allows reviewers to concentrate on the real issues, not on confusions.

This method can be particularly helpful in the following situations:

- Making a choice between competing patterns or libraries
- Applying a workaround due to a known bug
- Non-obvious optimizations writing

3.6. Don't Take Feedback Personally

This issue is actually more about the mindset than the mechanics, but it is very important. Code reviews are not a verdict of your intellect, character, or skills they are a discussion on the code, not the person. Each developer, regardless of seniority, gets feedback. And every piece of feedback, even if it is not perfectly expressed, is a chance to become better.

To keep a positive outlook:

- Take a moment before you react in a defensive way to the comments.
- Give reviewers the benefit of the doubt and assume that they have a good intention.
- Consider reviews as cooperationrather than fighting.

In case you feel that it is unfair or ambiguous, then no problem, you can ask for an explanation or argue with politeness. Feedback is a shared responsibility; however, it is most effective if both parties are not focused on winning but rather on learning.

3.7. Respond Promptly and Professionally

When you get remarks, reply quickly and gently. This involves appreciating comments, asking questions when necessary, and changing the pull request instead of suggesting the unthinking acceptance of every idea.

Best practices:

- Use checkboxes or threading to track feedback resolution.
- Thank reviewers for helpful suggestions.
- Explain you're reasoning if you choose not to make a change.
- Avoid ghosting if you're blocked, say so.

Professional answers sustain momentum and demonstrate respect for the reviewer's time. Good communication fosters respect among people despite probable variations. Remember also that a reviewer could not have the same contextual knowledge as you; hence, make time to clarify as necessary and provide them with the necessary tools to aid in better codes generally.

4. Tips for Code Reviewers

Beyond basic problem identification, a competent code reviewer addresses establishing confidence, peer guidance, and impact on the direction and quality of a codebase. Your remarks might motivate, direct, or discourage. Your communication style is equally as vital as the words you choose. This section presents reasonable recommendations for readers that investigate the essence of good, sympathetic assessment and surpass simple ideas.

4.1. Read with Empathy and Curiosity

Review every pull request knowing that the code you are looking at has been given considerable thought by a real human. They may have been negotiating unexplored code pathways, juggling edge events, or under deadline. See it as a coworker seeking knowledge and support rather than as something to "evaluate."

- Consider moral goals. The code could be faulty even if the person in charge most definitely made a great effort.
- Exude excitement. You criticize, then ask, "What could have motivated their approach?"
- Think of the tone. Even with technical precision, a negative remark or direct criticism can impair morale.

Empathy sharpens teamwork. It implies that you are there not merely to point out flaws but also to be helpful.

4.2. Focus on Functionality First, Style Later

Not all problems are the same. Prioritize your energy accordingly.

- Firstly, make sure the program is working. Has it been checked against the requirements? Are the edge cases covered? Does it introduce any regressions?
- After that, focus on the structure: Is the code organized logically? Is it testable and modular?
- Finally, if your linter or formatter hasn't made these changes, stylistic changes are acceptable.

Don't make it hard for someone to find functional issues amidst a lot of very small ones about indentation or naming. And if those issues are already taken care of by tooling, there isn't much point in manually flagging them. An insightful reviewer supports the submitter in concentrating on the main thingsfirst of all, correctness, then clearness.

4.3. Prioritize High-Risk Areas (Security, Architecture)

Not all lines of code have the same properties. Learn to identify and examine his impact-rich areas, such as:

- Security: Is the system secure or are there potential security breaches that can come from user input that is not safe, missing validations, or hardcoded secrets?
- Architecture: Are there any issues that can be caused by the change in architecture, such as high coupling or hidden dependencies?
- Data handling: Have the data models been utilized in a proper way? Is there any potential for the data to be corrupted or lost?
- Performance: Is there any chance that this implementation can become a reason for latency spikes or the database being loaded unnecessarily?

Concentrate your detective energy on those parts of the codebase that have a large impact in case something goes wrong. For less risky or isolated changes, a more lightweight review may be sufficient. The triage approach makes your reviews not only more efficient but also more valuable.

4.4. Provide Actionable, Concise Feedback

An ideal code review does not leave the submitter confused or overloaded with unnecessary information. Use clear language. Use short sentences. Use polite words.

Explicit feedback means:

- Examples: "Rename this to userEmail" is a more correct statement than "Naming could be improved."
- Giving the reason: "This method could be async to avoid blocking the main thread."
- Suggesting other options: "Could we use a map instead of a loop here for better lookup performance?"

Moreover, cluster similar issues. If the same issue pops up in other files, do not repeat yourself but write a summary instead. Furthermore, if implementing the suggested behavior in the present moment is not the top priority, it is better to give an example such as a TODO or a follow-up ticket. Using each PR for the removal of clutter is not recommendable.

Lastly, don't make vague statements, e.g.,

- "Hmm."
- "Interesting."
- "I don't like this."

They make the submitter uncertain about how to change the code and the reasons. The one who understands the most becomes the one who most clearly expresses the intention.

5. Process and Tooling Best Practices

Even with conscientious workers on both sides, code reviews might fail in the absence of well-defined policies and accompanying instruments. Goodwill cannot maintain quality and efficiency in review systems; they also need a methodical approach. This section describes optimal strategies teams could apply to raise code review consistency, fairness, and efficiency. Code review implemented well becomes a team scalable resource as well as a quality checkpoint.

5.1. Code Review Checklists for Teams

One uniform checklist not only minimizes the risk of errors but also ensures that the code review process is thorough and consistent with the priorities of the team. On the other hand, without a standardized process, reviews can be very different in the amount of depth and focus depending on the reviewer. However, this inconsistency could result in the team being unclear about the standards, a bug going unnoticed or even frustration among the team.

What to include in a code review checklist:

- Correctness: Is the code capable of fulfilling functional requirements? Are all the edge cases taken care of?
- Security: Does it come to your mind that there are some inputs that need validation? Is there any sensitive data that may be leaked?
- Readability: Is the code understandable at a glance? Are variable names and comments informative?
- Testing: Are there any unit/integration tests? Are they working?
- Performance: Can you spot some inefficiencies? Will this be able to work under a heavy load?
- Style: Is the code consistent with the formatting and the way of naming agreed upon?

The checklists should not be too heavyand more of a guide than a strict form. It is desirable that teams initially make them and then continuously develop them together, ensuring they are always up to date as the projects increase in size.

5.2. Using GitHub/GitLab Tools Effectively

Platforms like GitHub and GitLab are filled with numerous review features that sometimes users don't fully utilize. These features can help a reviewer greatly simplify the process and minimize the conflicts with co-workers if they know how to use them.

Best practices for using platform features:

- Draft PRs/MRs: Allow people who submit work to indicate that they do not consider it to be complete yet but they would like to get some early feedback.
- Code owners: Tell specific individuals which parts of the code they should review in order to be sure that their expertise will be used most effectively.
- Suggested changes: Reviewers can do this instead of writing a lot of comments. They make the changes they suggest to the code, and submitters accept those changes with one click.
- Review summaries: Utilize comments or checklists located at the top of PRs to reiterate the outstanding issues or give approvals.
- Required reviews: Have branch protection rules in place to stop merges without at least one review.

Moreover, as labels, review status indicators, and notifications are all interconnected,Wrong usage of them can lead to misunderstandings and hence it is very important to use them in a correct way. Tools are only as good as the team's discipline in using them effectively.

5.3. Automated Linting, Formatting, and Static Analysis

Clearly, even some things do not require human judgmentautomation can take care of repeated and style-related issues; hence, it cannot be a burden to the reviewers as they can focus on logical and architectural problems only. These are some automation tools commonly used:

- Linters (e.g., ESLint, Flake8): Find out syntax errors and style violations.
- Code formatters (e.g., Prettier, Black, gofmt): Keep formatting consistent.
- Static analysis (e.g., SonarQube, CodeQL, Pylint): Spot bugs, complexity and security issues that might appear.
- Pre-commit hooks: Make sure that there is no problematic code in the repo.

The main point is to make these tools work along with the CI/CD pipeline so that code cannot be merged before it passes the automated verification. Doing so will not only reduce the friction between submitters and reviewers but also eliminate unnecessary comment churn over style. It would be great if teams would not only record the list of available tools but also provide the configuration files that are necessary to get everyone on the same page. In this case, automation will play the role of “the bad cop,” and human beings will be those who have more meaningful discussions.

5.4. SLOs for Review Time and Quality

Slow or irregular reviews have been known to damage morale, cause late releases, and also decrease the quality of the product. The use of Service Level Objectives (SLOs) for code reviews has the potential to not only define team-wide expectations but also ensure that the reviews continue to flow smoothly without any backlogs.

For example, a typical set of review SLOs might be

- PRs should get a first review within 24 hours.
- A PR should never be left without an update for more than 48 hours.
- Reviews should give feedback that can be acted upon within one round (if possible).
- A PR should never be merged with unresolved critical comments.

SLOs should be considered as a guide rather than strict rulesi.e., building healthy habits instead of putting pressure. Methods like publicly available dashboards or Slack bots can enable teams to track review statistics (e.g., average time for review, PRs in queue) and recognize bottlenecks early. If the introduction of SLOs is done in a thoughtful manner, it can lead to a situation wherein the team members become responsible and their cognitive load decreases, along with an improvement of their reaction speed.

5.5. Rotations and Review Load Balancing

Reviewer burnout is one of the major reasons for the long delays in review. Such a situation occurs when a few senior developers are reviewing the majority of the PRs and others are not contributing. So, to prevent such a situation, teams ought to set up structured review rotations and load balancing.

Strategies to distribute review workload:

- Rotating “review captain”: Each week one person is in charge of review triage; this makes sure that all PRs get a first pass or are assigned appropriately.
- Peer-review pairings: Change parts of the team members to bring in new eyes and reduce the number of silos of the tribal knowledge.
- Tool-assisted assignment: Employ bots (e.g., Reviewable, Review Roulette) to distribute work randomly or logically among people based on the last activities or the ownership.
- Set limits: Limit the number of concurrent reviews per person so he/she does not get overloaded.

Such intentional distribution not only increases the quality but also prevents burnout and provides equal learning opportunities especially for the new personnel.

6. Psychological Safety and Communication Culture

A human being supports every line of codeconsidered, learned, and sometimes questioned choices. Review of codes is basically a human event, technical as they are. Especially for first-year engineers or team newcomers, one rude comment or harsh phrase might have long-lasting effects. Therefore, a good engineering culture depends heavily on psychological safety, which is not only helpful but also quite necessary. This part explores how teams could create a communication climate that supports learning, welcomes criticism, and improves cooperation by means of which one might support another.

6.1. Building a Safe Environment for Feedback

Psychological safety refers to a situation where team members have sufficient trust and confidence to be able to freely express their thoughts, admit mistakes, and exchange feedback without fearing humiliation or retaliation. With code reviews, this notion of safety is the following:

- Reaffirming the fact that the discussion can be open and disagreement can occur without the need for shame.
- Providing a situation where all voicesespecially quieter onesare important.
- Set an example of being open: it is very effective when senior engineers say, “I don’t know either” or “Thanks for giving me the new angle to look at this.””

6.1.1. In order to make such an atmosphere exist:

- We can expect that reviewers should initiate the process with the idea of helping, not judging.
- Submitters should be allowed to explain the decisions without any fear and even ask for clarification.
- All the people who participate should always adhere to the assumption of positive intent.

After safety becomes a fact, people are more willing to share their good ideas, ask questions, and go against the common belief which is, of course, better code as well as stronger teams.

6.2. Training Juniors without Micromanagement

Mentoring code review from less experienced developers can be a very tricky situation. The question here is how to be both a teacher and a guide without losing the interest of the recipient.

The most effective practices for mentoring through the code reviewing process:

- Initially, focus on what they got right.
- Give concrete examples in your feedback. “Avoid magic numbers” is much more appropriate than “this looks wrong.”
- When you propose changes, do it with examples or short parts of the code.
- Do not force them to agree with your whole point but create a condition where they can come up with different ideas on their own.

Remember, don’t use the review session as a personal checklist of how you would’ve done the task. Instead, be on the lookout for teachable moments that instill confidence in the new ones. Allow them to take full responsibility for their mistakes, correct them, and benefit from the experience. Micromanagement is the mother of dependence. Mentorship is the source of independence.

6.3. The Role of Engineering Managers

Engineering managers lead largely in the creation of review culture. Their work is to establish the tone, model behavior, and build systems enabling development and safety rather than to review every line of code.

Key responsibilities include:

- Setting expectations: Mostly you are responsible for creating guidelines for timely, pleasant, and constructive assessments.
- Intervening early: Quickly handling when they reveal unfavorable dynamics or repeated conflicts
- Balancing feedback: Ensuring that juniors are not only under criticism while seniors are also suitably challenged guarantees fair comments.
- Monitoring metrics: Does reality show in the reviews? Are certain particular people prone to overwhelm? Are any voices absent?

Sometimes managers can provide a model of leadership by showing up for evaluations, praising great work, and leaving comments for others. Encouragement of empathy, curiosity, and equity in the evaluation process helps managers to transmit a clear message: feedback is not a threat; it is a tool for group excellence.

7. Case Study: Transforming Review Culture at DevCraft Inc

The code review process at DevCraft Inc., a mid-sized product engineering business that focuses on cloud-native SaaS apps, had become a constant source of problems. The development team was good at what they did, but there was tension in the engineering teams because the reviews weren't always good, there were communication problems, and the organization relied too much on a few senior engineers. This caused features to be released later than planned, and junior developers were getting burned out and losing morale.

7.1. Initial State: Siloed Teams and Broken Feedback Loops

Before any intervention, DevCraft's code review tool found unrelated processes and poor alignment. Every team has their unwritten rules on the standards for a "good review." While some critics point out thorough, pedantic remarks on little problems, others approve large pull requests (PRs) without closely reviewing the content.

The PR study found no consistent SLA. Often more than a week, developers regularly ran with delays of many days that caused cognitive dissonance and stopped progress for feedback. From this came longer review cycles, merging delays, and growing team discontent.

Moreover, the approach of communication applied during the tests had become somewhat unpleasant. Junior developers hesitating to question concepts or challenge regulations were fearful of exposing themselves or breaking rules. Many times lacking clarity or obvious dismissiveness, statements erode confidence and motivation. Internal engagement polls revealed that many engineers felt the appraisal process added more stress than value.

7.2. Intervention: Process, Empathy, and Tooling

Recognizing the mounting cost of these issues, DevCraft's engineering leadership launched a three-pronged initiative aimed at revitalizing the review culture: standardizing process, fostering psychological safety, and introducing supportive tooling.

7.2.1. Checklists for Consistency

Responding to the growing costs associated with these challengesstandardizing methods, promoting psychological safety, and deploying supporting toolsDevCraft's engineering leadership began a three-pronged approach to rejuvenate the review culture. The first phase consisted in using all-team comprehensive code review checklists in reduced form. These lists contained guidelines for:

- Operational excellence
- Assessment of test completeness
- Problems concerning performance and security
- Openness and environmentally friendly behavior
- Clearly state your pull requests and commit messages.

Today's reviewers follow a uniform code evaluation process, which allows younger engineers to grasp the expectations placed upon them and helps to lower subjectivity.

7.2.2. Empathy and Communication Training

DevCraft collaborated with an external facilitator to conduct "Empathy in Engineering" workshops and thus they fostered emotionally positive interactions in the reviews. These meetings comprised

- Acting out various review situations
- Finding negative communication habits
- Giving feedback in the form of questions instead of commands
- Making praise and positive reinforcement a habit

The leadership and senior staff were quite urged to set the example of being open emotionally by requesting feedback and owning former wrongdoing in the review process.

7.2.3. Tooling and Automation

DevCraft has also improved its tooling:

- Reached out to the GitHub saves feature to make sure conversations with the team are dealt with less.
- Configured pre-commit hooks and linting via CI so that formatting and style are always updated automatically.
- Added a "review load dashboard" that showed the number of PRs still open, average response times, and the workload of reviewers.

Such changes not only made the work of reviewers less boring but also allowed them to concentrate on more meaningful and higher-level feedback.

7.3. Outcomes: Faster Merges, Happier Teams

Six months after rolling out the initiative, DevCraft experienced concrete and measurable improvements in several areas:

- Time-to-merge had dropped by 40%. PRs that were still open days prior to this were now reviewed and merged within 24–48 hours; thus, teams were able to ship features faster and with fewer merge conflicts.
- Developer satisfaction went up a lot. Response surveys indicated a 35% increase in positive feelings about the review process, with juniors identifying "more approachable feedback" and "less fear of asking questions."
- More review participation. The mid-level developers who were reluctant earlier started reviewing PRs regularly as per the checklist guidance and the change in the tone of the conversation.
- Defects in the QA had become fewer. The improvement of review hygiene and the involvement of more people in monitoring high-risk changes helped identify problems at an earlier stage of the cycle, resulting in higher-quality releases.

7.4. Lessons Learned: Culture Change Is a Marathon, Not a Sprint

The transfiguration at DevCraft was not an instance of instant success. The next lessons are described in a list below:

- Balance is the main thing. Automation assisted to implement consistency and at the same time, human judgment was still crucial. The reviewers had to learn where to be strict and where to be flexible.

- Empathy must be a part. If there is no training and continuous encouragement, the old habits can appear again. Managers kept the theme of empathy on their minds when they had retros and 1:1s.
- Culture change is indeed a process. The initial resistance came from both sides: the reviewers and the submitters. It needed repeated reinforcement, visible leadership support, and small wins to build momentum.
- Feedback should be reciprocal. Developers were given not only the opportunity to receive feedback but also to leave comments for reviewers; thus, a two-way learning loop was created.

8. Conclusion and Key Takeaways

Code reviews offer chances for group enhancement, mentoring, and cooperation instead than being milestones in a development cycle. This guide has looked at methods in which submitters and reviewers could help to foster higher relevancy, respectfulness, and output of code reviews. First for those submitting codes, organization and clarity are important. Write simple, self-explanatory code; keep focused and brief pull requests; use major commit messages; and carefully and professionally react to comments utilizing PR descriptions and comments to set the backdrop. See remarks as an opportunity for growth rather than as a jab directed personally.

Reviewers should especially be deliberate and compassionate. View assessments from the eye of research instead of judgment. Give usability first priority; next, give automated tools artistic considerations. Ask inquiries; point out problems like design or security; and regularly offer brief, perceptive remarks instead of instructions. Reward exceptional performance. Not to dominate a discussion, the goal is to enable a colleague to improve in coding. On a team, structure is really important. Lists guarantee consistency. Automaton silence noise: Together, established techniques, fair allocation of the review load, and psychological safety taken build a stronger review culture. Managers are mostly responsible for creating this climate since they ensure that comments in all directions are freely and politely flowing.

A good code review is a conversation instead of a conflict. Under this cooperative atmosphere, developers help each other in professional development, improve product quality, and coordinate around uniform coding standards. Properly done, code reviews become more than just a tool for learning, trust, and continuous developmenta source of concern as well. Remember that every word you say gently shapes culture, not only changes code when you next open or review a pull request.

References

[1] Laurvik, Torgeir Sandnes. *Design process behind an educational review system for student submissions-applying knowledge from professional code reviews to an educational assessment setting*. MS thesis. NTNU, 2021.

[2] Pasupuleti, Vikram, et al. "Impact of AI on architecture: An exploratory thematic analysis." *African Journal of Advances in Science and Technology Research* 16.1 (2024): 117-130.

[3] Lalith Sriram Datla, and Samardh Sai Malay. “Transforming Healthcare Cloud Governance: A Blueprint for Intelligent IAM and Automated Compliance”. *Journal of Artificial Intelligence & Machine Learning Studies*, vol. 9, Jan. 2025, pp. 15-37

[4] Radez, Jerica, et al. "Why do children and adolescents (not) seek and access professional help for their mental health problems? A systematic review of quantitative and qualitative studies." *European child & adolescent psychiatry* 30.2 (2021): 183-211.

[5] Cohen, Jason, Steven Teleki, and Eric Brown. *Best kept secrets of peer code review*. Smart Bear Incorporated, 2006.

[6] Talakola, Swetha. “Transforming BOL Images into Structured Data Using AI”. *International Journal of Artificial Intelligence, Data Science, and Machine Learning*, vol. 6, no. 1, Mar. 2025, pp. 105-14

[7] Platt, David S. *Why software sucks--and what you can do about it*. Addison-Wesley Professional, 2007.

[8] Syed, Ali Asghar Mehdi. "Zero Trust Security in Hybrid Cloud Environments: Implementing and Evaluating Zero Trust Architectures in AWS and On-Premise Data Centers." *International Journal of Emerging Trends in Computer Science and Information Technology* 5.2 (2024): 42-52.

[9] Zanjani, Motahareh Bahrami, Huzefa Kagdi, and Christian Bird. "Automatically recommending peer reviewers in modern code review." *IEEE Transactions on Software Engineering* 42.6 (2015): 530-543.

[10] Allam, Hitesh. “Intent-Based Infrastructure: Moving BeyondIaC to Self-Describing Systems”. *International Journal of Artificial Intelligence, Data Science, and Machine Learning*, vol. 6, no. 1, Jan. 2025, pp. 124-36

[11] Jabbar Mohammad, Abdul. “Integrating Timekeeping and Payroll Systems During Organizational TransitionsMergers, Layoffs, Spinoffs, and Relocations”. *Los Angeles Journal of Intelligent Systems and Pattern Recognition*, vol. 5, Feb. 2025, pp. 25-53

[12] MacLeod, Laura, et al. "Code reviewing in the trenches: Challenges and best practices." *IEEE Software* 35.4 (2017): 34-42.

[13] Jani, Parth. "Modernizing Claims Adjudication Systems with NoSQL and Apache Hive in Medicaid Expansion Programs." *JOURNAL OF RECENT TRENDS IN COMPUTER SCIENCE AND ENGINEERING (JRTCSE)* 7.1 (2019): 105-121.

[14] Remillard, Jason. "Source code review systems." *IEEE software* 22.1 (2005): 74-77.

[15] Veluru, Sai Prasad. "Reversible Neural Networks for Continual Learning with No Memory Footprint."

International Journal of AI, BigData, Computational and Management Studies 5.4 (2024): 61-70.

[16] Boland, Angela, Gemma Cherry, and Rumona Dickson, eds. "Doing a systematic review: a student′ s guide." (2017).

[17] Chaganti, Krishna Chiatanya. "Securing Enterprise Java Applications: A Comprehensive Approach." *International Journal of Science And Engineering* 10.2 (2024): 18-27.

[18] Hamasaki, Kazuki, et al. "Who does what during a code review? datasets of oss peer review repositories." *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE, 2013.

[19] Sriram Datla, Lalith, and Samardh Sai Malay. "Zero-Touch Decommissioning in Healthcare Clouds: An Automation Playbook With AWS Nuke and GuardRails". *Los Angeles Journal of Intelligent Systems and Pattern Recognition*, vol. 5, Mar. 2025, pp. 1-24.

[20] Chaganti, Krishna Chaitanya. "AI-Powered Threat Detection: Enhancing Cybersecurity with Machine Learning." *International Journal of Science And Engineering* 9.4 (2023): 10-18.

[21] Okoli, Chitu. "A guide to conducting a standalone systematic literature review." *Communications of the Association for Information Systems* 37 (2015).

[22] Talakola, Swetha. "The Optimization of Software Testing Efficiency and Effectiveness Using AI Techniques". *International Journal of Artificial Intelligence, Data Science, and Machine Learning*, vol. 5, no. 3, Oct. 2024, pp. 23-34

[23] Tawfik, Gehad Mohamed, et al. "A step by step guide for conducting a systematic review and meta-analysis with simulation data." *Tropical medicine and health* 47 (2019): 1-9.

[24] Balkishan Arugula, and Suni Karimilla. "Modernizing Core Banking Systems: Leveraging AI and Microservices for Legacy Transformation". *Artificial Intelligence, Machine Learning, and Autonomous Systems*, vol. 9, Feb. 2025, pp. 36-67

[25] Ridley, Diana. "The literature review: A step-by-step guide for students." (2012): 1-232.

[26] Allam, Hitesh. "Policy-Driven Engineering: Automating Compliance Across DevOps Pipelines." *International Journal of Emerging Trends in Computer Science and Information Technology* 6.1 (2025): 89-100.   -mar

[27] Veluru, Sai Prasad. "Threat Modeling in Large-Scale Distributed Systems." *International Journal of Emerging Research in Engineering and Technology* 1.4 (2020): 28-37.

[28] Kraus, Sascha, Matthias Breier, and Sonia Dasí-Rodríguez. "The art of crafting a systematic literature review in entrepreneurship research." *International Entrepreneurship and Management Journal* 16 (2020): 1023-1042.

[29] Abdul Jabbar Mohammad, and Guru Modugu. "Behavioral TimekeepingUsing Behavioral Analytics to Predict Time Fraud and Attendance Irregularities". *Artificial Intelligence, Machine Learning, and Autonomous Systems*, vol. 9, Jan. 2025, pp. 68-95.

[30] Wolfswinkel, Joost F., Elfi Furtmueller, and Celeste PM Wilderom. "Using grounded theory as a method for rigorously reviewing literature." *European journal of information systems* 22.1 (2013): 45-55.

[31] Jani, Parth. "Document-Level AI Validation for Prior Authorization Using Iceberg+ Vision Models." *International Journal of AI, BigData, Computational and Management Studies* 5.4 (2024): 41-50.

[32] Arugula, Balkishan. "Prompt Engineering for LLMs: Real-World Applications in Banking and Ecommerce". *International Journal of Artificial Intelligence, Data Science, and Machine Learning*, vol. 6, no. 1, Jan. 2025, pp. 115-23

[33] Kupanarapu, Sujith Kumar. "AI-POWERED SMART GRIDS: REVOLUTIONIZING ENERGY EFFICIENCY IN RAILROAD OPERATIONS." *INTERNATIONAL JOURNAL OF COMPUTER ENGINEERING AND TECHNOLOGY (IJCET)* 15 (2024): 981-991.

[34] Vasanta Kumar Tarra. "Ethical Considerations of AI in Salesforce CRM: Addressing Bias, Privacy Concerns, and Transparency in AI-Driven CRM Tools". *American Journal of Autonomous Systems and Robotics Engineering*, vol. 4, Nov. 2024, pp. 120-44

[35] Kodete, Chandra Shikhi, et al. "Robust Heart Disease Prediction: A Hybrid Approach to Feature Selection and Model Building." *2024 4th International Conference on Ubiquitous Computing and Intelligent Information Systems (ICUIS)*. IEEE, 2024.

[36] Bacchelli, Alberto, and Christian Bird. "Expectations, outcomes, and challenges of modern code review." *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013.

[37] Sreekandan Nair, S., & Lakshmikanthan, G. . (2021). Open Source Security: Managing Risk in the Wake of Log4j Vulnerability. International Journal of Emerging Trends in Computer Science and Information Technology, 2(4), 33-45. https://doi.org/10.63282/d0n0bc24