International Journal of Emerging Research in Engineering and Technology
Pearl Blue Research Group| Volume 1, Issue 2, 60-68, 2020
ISSN: 3050-922X | https://doi.org/10.63282/3050-922X.IJERET- V1I2P108

.Original Article

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

Bhavitha Guntupalli
Independent researcher, USA.

Abstract - *Although code reviews are more frequently seen as boring or divisive, when done well they might be a very powerful tool for improving these software quality & promoting their team collaboration. This article uses a human-centered approach to clarify the code review process and offers many useful guidance for both submitters and reviewers to improve their output and reduce pain during reviews. The book opens by stressing common frustrations such as unclear comments, too much inspection, or too quick approvals and clarifies how these elements not only stop development but also erode team trust. It then offers sensible recommendations for development: how reviewers may provide polite, helpful feedback that is really more valuable, and how submitters might create more effective pull requests to support more quick reviews? Effective and poor techniques are clarified by concrete examples and a pragmatic case study, which also show how modest changes in these expectations and communication might produce their major results. While underlining that the human component effective communication, empathy, and the mutual objectives truly improves the review experience the study examines the relevance of technology and processes. These suggestions are meant to foster a culture of learning, mutual respect, and continuous growth regardless of your level of experience as a developer tired of negative review cycles or a new hire trying to fit into your team's process. In the end, code reviews could become chances for their development, knowledge sharing, and a notable improvement in team morale and software quality rather than negative ones.*

Keywords: *Code review, software engineering, pull request, developer productivity, technical feedback, review process, GitHub, best practices, clean code, team collaboration.*

1. Introduction

Code reviews have become a necessary habit in the modern fast software development environment, acting not just as a technical gatekeeping device but also as a team-building and quality- assurance routine. Whether you are implementing the latest feature, fixing a bug, or rewriting old code, getting more attention on your changes might greatly affect the difference between durable, manageable software and a possible production failure. Still, despite their importance, code reviews may turn into unpleasant, ineffective, or, at worst, useless endeavors.

1.1. The Importance of Code Reviews

At their best, code reviews ensure consistency, increase code quality, and act as an educational tool for both the latest and veteran engineers. They guarantee that the software remains intelligible and testable over time, enforce architectural decisions, and help them to spot early defects. Moreover, they promote a culture of mutual respect, knowledge sharing, and teamworkqualities absolutely more necessary for any other successful organization. When done well, a code review goes beyond simple code inspection to include good communication. A discriminating critic might point out references to documentation, justify a bad design choice, or praise an original idea. A competent submitter writes clear commit messages, identifies likely problems, and explains their thinking. Taken together, they provide a feedback loop that goes beyond simple bug-fixing; it promotes professional growth, strengthens team trust, and increases empathy.

1.2. Collective Difficulties

Even with their good intentions, many other teams find code reviews challenging. For submitters, their painstaking efforts may appear to slip into an abyss awaiting hours or days for response, only to be greeted with many vague or insulting comments. Reviewers would find it as an undesirable detour from their own objectives, an interruption. Combining time limits, unclear expectations, and interpersonal dynamics results in a recipe for more conflict. Some assessments become painstaking arguments focused on the space, terminology, or artistic preferences. Others are supporting customs free of actual assessment. In both cases, the aim of the review to improve the code and grow the team is compromised.

1.3. The Cost of Inadequate Code Reviews

One cannot overstate the effects of insufficient code reviews. Disappointed submitters could stop seeking comments completely. Reviewers may get cynical or burn out. Technical debt may build up and little mistakes could go unnoticed. Eventually, a strategy meant for cooperation yields delays, hostility, and low morale. Moreover, the long-term effects might even go outside. Teams that overlook the need of stressing efficient code reviews might suffer with poor quality, inconsistent coding standards, and inadequate onboarding of new hires. What happens? a team that communicates less effectively, a more difficult to maintain their codebase, and a slower and more prone to mistakes development cycle.

Fig 1: Effective Code Review Workflow for Reviewers and Submitters

1.4. Effective Results Are Outstanding

The good thing is, however, that properly done code reviews provide both quantifiable and intangible benefits. Improved code quality produces fewer flaws, more satisfied developers, and many stronger teams. While submitters obtain viewpoints they would not have otherwise explored, reviewers improve their ability to clearly explain their opinions. The crew becomes more in line, competent, and confident gradually. It covers coaching, development, and group codebase accountability going beyond simple error detection. Teams that provide strong emphasis on meaningful and polite code reviews usually show higher productivity, better communication, and a closer connection.

1.5 Articles' Scope

The basic elements that either support or undermine the success of a code review will be discussed in this paper. We will look at:

- The attitude of a competent reviewer: how to be thorough without being harsh?
- Guidelines for submitters: enabling a pleasant and easily available code review mechanism.
- Time, tone, instruments, and team customs optimal practices all matter.
- Steer clear of things like passive-aggressive comments and too slow responses.
- Creating a sustainable review system that balances interpersonal connection, quality, and efficiency.

This article seeks to help beginners as well as seasoned experts improve their code review procedures, therefore turning their output from merely acceptable to quite useful.

2. The Psychology behind Code Reviews

Code reviews provide insight into team culture, communication dynamics, and human psychology in addition to a technical quality check-point. Every comment on a pull request reveals an emotional undercurrent that could either boost or lower morale. Understanding the psychology of code reviews might help to build better, more effective engineering teams.

2.1. Cognitive Demand and the Value of Empathy

Code reviews basically need cognitive effort. Reviewers have to understand the code's construction's justification as well as its functioning. Often managing more numerous contexts frameworks, edge circumstances, project requirements they try to give useful, constructive advice. Here is when empathy becomes absolutely more necessary. Reviewers might be tired, under deadline stress, or just have a bad day. The code author has also committed time, effort, and perhaps many cups of coffee to ready this change. The investment is noteworthy as it helps one develop a psychological link to the work. A harsh or frosty assessment not only questions the code but also targets the person behind it. Sensual assessors understand the human effort involved. They ask clarifying questions instead of assuming anything. They resist needless criticism. Giving comments as a recommendation that is, "What is your perspective on..." rather than a demand that is "You ought to" helps to reduce too much conflict and promote actual communication.

2.2. Ego, Identity, and Defense Mechanisms

For many engineers, coding is not just a job; it's a part of their nature. As such, comments on code might sometimes come out as personal criticism. Even kind comments might be seen as attacks if they make someone's ego unstable. This is a normal human response; it is not a lack of morality. Neuroscience suggests that the brain's sensitivity to social hazardssuch as criticismmay be as strong as its response to physical danger. An apparently minor comment might cause defensiveness, annoyance, or even silence. Reviewers should focus on the code instead of the developer in order to help to solve these problems. For instance, say rather than, "You did not address the null case," "It appears the null case may be overlookedshould we address that as well?" Feedback positioned in the framework of group goals instead of their personal mistakes changes the environment from one of accusation to collaboration.

2.3. The Effect of Tone on Literary Works

Written communication has a multifarious tone. Just words shown on a screen no face expressions, voice inflections, or body language. What meant to be help could come out as contemptuous. What was supposed to be objective may come out as austere or distant. This is why it helps to spend some additional time on controlling your tone. Use polite words. Add a "well observed here," or "thank you for your efforts on this." Emojis are emotional markers; they are not unprofessional. Frequent usage of a kind and appreciative tone by reviewers suggests that feedback is not a hierarchy of assessment but rather a necessary component of a supporting culture.

2.4. Encouraging an Engineering Team to Develop a Growth Mindset

Code reviews should mostly serve development for the team, the codebase, and the individuals. Teams with a growth mentality see comments as a chance for development rather than as evidence of failing. Mistakes turn from embarrassing events into teaching moments. This sort of culture calls for conscious effort; it does not happen naturally. Leaders could show vulnerability by accepting more criticisms straight forwardly. Colleagues could come to believe that nobody generates perfect code. This helps people to feel safe enough to explore opportunities, ask questions, and grow as engineers over time. Code reviews that are driven by curiosity instead of judgment and by compassion instead of criticism go beyond simple quality assurance and become a shared learning and connecting experience.

3. For the Submitters: How to Make Your PR Reviewer-Friendly

Pull requests (PRs) include more than simply code turns-in. They engage in a conversation. Making one not only helps you distribute your work but it also invites feedback and promotes teamwork. Good pull requests help reviewers to clearly understand your work, point out mistakes, and allow them to boldly support your changes. Lesser ones? Their duties are heavy. Here's how you make sure yours falls into the first category.

3.1. Getting Ready Before Turn In

3.1.1. Think back on your choices before choosing "Create PR."

Think back on your efforts before turning in your code. Have you cleaned the mess? Is that sensible? Is it really ready for another inspection?

For your review below is a helpful self-assessment checklist:

- Does the code serve the intended use? Should you not have done a test yet, kindly begin here. Not merely "it compiles" testing, but rather than actual, introspective settings.
- Are you quite proud of it? Think again if you feel ashamed of your explanation of a topic.
- Is it legible? Remove pointless comments, clean debug logs, and organize objects methodically.
- Is the extent focused? If your pull request covers five problems at once, think about separating them into many individual requests.
- Have you given useful criticism? Add breadcrumbs to complicated areas of your code. Help your assessor to understand your thinking.
- Have the tests been added to or changed? This is an obvious comment. Notes to reviewers.
- Bonus idea: pretend to be the reviewer. Go over your changes as if you were first seeing them. What's confusing? Why seems pointless?

3.2. Create Models of Pull Requests

3.2.1. More crucial than one might think are the title and description of the PR.

Think of the title like the subject line of a major email; it defines the tone. Not enough is "rectifying bugs" or "modifying code". When the biography runs more than 300 characters, take care of the overflow problem in the user profile component.

- Your opportunity to tell a story comes from the description. Give your reviewer background: what problem are you looking at?
- Why do you think you decided on this strategy?
- Have you given any other choices you eventually turned down?
- What concessions are required?
- Including links to tickets, design guidelines, pictures, or a little Loom video can help to improve the result greatly.

Make sure you also stress the important points. Have you changed an outside dependability? Change a contract. Interferes with a fragile system? Describe it right now. No reviewer likes to come across halfway unanticipated changes.

3.3. Manage Review

3.3.1. Respect the time of your reviewer. Help them with your assistance.

These are some ways to keep reviewer-friendliness:

Keep your pull requests focused and short. Often a delight to review is a 100-line pull request. A 1,000-line pull request causes unhappiness. Break things down logically, if at all feasible. More succinct pull requests speed things and improve their feedback. Create atomic commits. Commits have to also tell a story. Preferred over "wip" is "refactor sidebar rendering". Each should represent a single idea or change. Name the suitable people and show thought. If your change affects front-end performance, involve someone with that knowledge. Steer clear of treating the whole workforce haphazardly. Friday afternoon same-day assessments are not something you should expect. Also be patient. Reviewers may be handling extra duties. If your PR is urgent, explain the reasons behind it instead of only declaring that it is such.

3.4. Get Comments with Grace

3.4.1. Pull request comments are more about the work than they are about the person.

One often becomes defensive when someone questions your approach or suggests changes. Though it is not given in the way you want, feedback is a great gift.

Here is a way to handle it with certainty and poise:

- Take it not personally. Perfect code is the goal, not bruised egos.
- Ask questions now. If anything is vague, do not apply the change without first asking questionsseek clarification. This helps on both sides to improve understanding.
- Accept and respond. A brief thank you like "Good catch fixed!" or "That makes sense" shows your admiration and participation. Don't let reviewers wonder if you responded to their comments.
- Finish the cycle. List the revisions done after the first ones. I have changed the reasoning to stop prop mutationready for further study. It shows your diligence and helps reviewers save their time.

Every pull request presents a chance for improvement not just for one coder but also for a team.

4. For the Reviewers: How to Give Feedback That Doesn’t Suck

Giving good comments during code reviews goes beyond just pointing out many mistakes. It speaks to being polite, helpful, and too cooperative. A good code review improves code quality and helps the team member programmer to feel valuable. The process is as follows.

4.1. Temporal Issues and Goal

4.1.1. Avoid Unjustified Review Delays

Turning in one's code for evaluation shows that one is ready to advance. Extended reaction times might lead to bottlenecks, therefore influencing not just the person but the overall project. It is frustrating and could slow down momentum. Give the reviewing process top priority. Offer a quick note even if an in-depth investigation is not possible: "I acknowledge this and will address it this afternoon." This small effort shows respect for the time of your coworker.

4.1.2. Clearly State Your Goals

Not every code review seeks for ideal code. Your goals need to be mostly:

- Accuracy: Does it serve as intended?
- Are others able to understand the scenario?
- Does it follow the project's guidelines?
- Performance as necessary: Exist appreciable inefficiencies?

To adopt the attitude of "I would not have approached it in this manner," is too easy. Still, it's not the central concern here. Release the code if it is clear, logical, and working.

4.2. Theory of Code Interpretation

- **Begin with the overall viewpoint:** Try to understand the goals of the code before looking at variances in names and indentation. One relevant initial question to ask is: "What issue does this address?" Ask for clarification if the pull request description or code is not clear. It is better to understand the goal first than let the details trap you.
- **Steer clear of premature criticism:** We know highlighting the extra space or the somewhat strange loop is enticing. Still, avoid giving little problems top priority. Little criticism, including stylistic issues or little enhancements, might obscure the conversation and give the appearance that one is only looking for mistakes. First give the general framework, justification, and design top priority. Defer style notes till the end, or ideally employ automatic linters for this.

4.3. Practice Human Communication

4.3.1. Ask Questions Instead of Directives

Nobody enjoys being screamed at. "Change this." "rectify that." Regardless of your intentions, such a tone might come off as icy and hostile.

- Would renaming this role improve clarity?
- "Can we use the current helper instead of copying it?"
- "What drove your choice of this method rather than [X"?

This makes the communication more cooperative than combative.

4.3.2. Courtesy, Constructiveness, and Exhibit Specificity

If anything looks wrong or is unclear, explain the reasoning. "This is confusing" is less helpful than "It is difficult to ascertain the purpose of this function perhaps a more descriptive name or an explanatory comment would be beneficial?" Improving the code is the goal, not causing unpleasant emotions in other people. A little bit of kindness brings big benefits.

4.3.3. Suggestions, not directives

The words "Consider...," or "You might try...," help to open conversation. It shows that you are open to substitutes and that you realize you do not have any special power on what is best. The regularity with which people have legitimate explanations for their approaches may astound you; furthermore, you could even learn something fresh.

4.3.4. Thank you. What is favorable?

Positive comments are powerful as well as nice. When someone produces a refined design, writes a useful utility function, or improves test coverage, thank you. It encourages good conduct, raises morale, and considerably lessens the resemblance of the entire process to a firing squad.

- "I value how clearly you explained this reasoningtruly understandable."
- "It's great to have that test case for edge conditions."
- "I appreciate the thorough materials; they helped with the review process."

4.4. Steer Clear of Blocking for Unfit Motives

- **Preferences Not Based on Principles:** Everybody has their own way of dressing. Maybe early returns appeal to you. Still another person likes to nest. Refrain from blocking code solely because of personal inclination in the lack of a codified style guide: Think about if this is a personal decision or a group standard. If it relates simply to you, ignore it or remark to it casually without calling for change.
- **Perfection Blocks Progress:** To be straight forward, no code is never perfect. Perfect expectation leads to constant uncertainty and slow development. Authorize the code if it satisfies your team requirements, is clear and strong. There may be certain areas that want improvement, perhaps; however, they could be covered in a next reorganization. A review serves to offer functional, maintainable code rather than to provide beauty.

5. Creating a Culture of Productive Code Reviews

Developing a team environment that supports and values helpful comments can help to improve code reviews beyond a simple chore. Good code reviews are not accidental; they thrive in teams that share these norms, encourage honest communication, and regularly seek improvement. One may foster such a culture as follows.

5.1. Create team norms

Begin with shared expectations. Review of codes becomes difficult when team members have somewhat different presumptions. A pull request (PR) should have what size appropriate? Reviewers' expected response times are what? How much more testing is needed before asking for a review?

Without clear guidelines, one person could send a 2,000-line pull request that goes unmet for five days while another submits 10, 20-line pull requests everyday, expecting timely response. Both of them will undoubtedly become unhappy as a result.

- Instead, develop thorough team-wide procedures. Except strictly necessary, limit pull requests to a maximum of 300 lines.
- Reviewers have 24 business hours to respond.
- Before submission for review, combine human and machine testing.

These are merely agreements that help all parties to be in line; they are not set rules. You are free to review them when your process changes or your team grows. Support good behavior with many tools. One thing is asking people to follow certain guidelines; another is automating such needs. Use templates in your pull requests to guide developers on necessary items such as testing techniques, context, and screenshots. Before getting to a reviewer's interface, activate linters and formatters to find stylistic differences. Run constant integration tests to make sure reviewers know basic requirements have been satisfied. These little improvements help reviewers and submitters to reduce their cognitive load. They help users to focus on important elements instead of syntax or style: the logic, organization, and quality of the code.

5.2. Encourage group projects and timely comments.

Never put off communicating until the pull request phase. Usually, optimal code review comes before the code even exists. Talking about a difficult problem in advance is usually more successful than tackling it directly. A quick design review or a short partnership session might save unwarranted work and coordinate efforts during implementation. For significant changes, architectural additions, or other aspects affecting many areas of the codebase, this is very helpful. Use synchronous evaluations for detailed changes. While most assessments happen asynchronouslywhich is usually acceptablethere are times when a call or screen sharing proves much more helpful. Direct discussion is usually more practical when a pull request shows significant back-and-forth comments or looks to remain stationary. Synchronous reviews help to resolve ambiguity quickly and encourage a shared vision. Since less experienced programmers receive instant explanations and reviewers may give more specific criticism than just comments, they provide great opportunities for mentorship.

5.3. Continuous Improvement

See your review process as a naturally occurring element of your workflow that might be improved. Ask: "How are our code reviews progressing?" occasionally, just as you would evaluate the effectiveness of your sprint or the dependability of your installations. Does the participation of only one or two reviewers typically hinder pull requests? Are the assessments thorough or only authorized to help with merging? Does feedback given to submitters help them grow?

This might be handled via specialized retrospectives every few months focused particularly on engineering workflow techniques or during sprint retrospectives. These type of questions might inspire deep reflection:

- Pull requests typically have a turn around time of what?
- Are we lost in minutiae or assessing the relevant components?
- Are every team member confident in turning in and evaluating these pull requests?

Welcomes change. Your review method has to change along with your team and codebase development. Pull request size limitations might call for change. One may be wise to divide review responsibilities more fairly. Review buddy systems could be excellent for onboarding the latest staff members. The goal is not to establish a "perfect" approach but rather one that helps your staff to routinely produce excellent code and promotes mutual growth.

6. Case Study: Improving Code Reviews in a Real-World Team

6.1. The Challenge

Changing tools or procedures would not easily fix a code review problem the development team at a medium-sized SaaS company discovered. This was more deeply significant. Review rounds stretched many days or even weeks. Pull requests (PRs) were avoided by engineers because of worries about negative comments or extended delays. Reviewers' and submitters' misunderstandings led to unnecessary interactions that rendered the whole process more like a chore than a team effort. Morality sank. Developers begin skirting reviews, which let mistakes unintentionally find their way into production. Engineers were

beginning to avoid one another, observed team leaders. Subtle divide was developing: submitters worried about assessment, evaluators under duress. Though code reviews had turned into a toxic atmosphere, it was not expressed.

6.2. The Correction

The turning point came after an extensive investigation that exposed all relevant information. The team decided to fix the system instead of laying individual guilt. This is their attempt:

6.2.1. Evaluation Checklists

Beginning small, the teams used a shared checklist for submitters and reviewers. For submitters, this meant making sure tests were written, edge cases were handled, and commit statements made sense. Reviewers kept a different checklist: understand the background, spot logical errors, evaluate security flaws, and above all offer sympathetic comments. This helped the notion of "done" to be standardized and expectations to be developed.

6.2.2. "Peer Review Partners"

They then carried out studies matching people as "review partners." Each developer chooses a preferred partner for their pull requests. These people understood one other's code and background, so they were collaborators rather than gatekeepers. It promoted reciprocal respect, helped to ease main line congestion, and gave everyone a consistent point of contact.

6.2.3. Feedback and Tone Training

At last, they held an informal lunch-and-learn event with an eye toward offering helpful criticism. The delivery of the message counted just as much as its substance. Many engineers merely lacked knowledge of how their written tone was received; they were not trying to be rude. Saying things like "Why did you do it this way?" has evolved into "I'm intrigued by this methodcould you elaborate on your reasoning?" They assumed good intentions, extended the benefit of the doubt, and framed their argument positively.

6.3. The Results

Changes began a few weeks later. Public Relations Reaction Time: The average merger period dropped by 40%. A practice never seen before, some assessments were finished on the same day. Improved Team: Submitters demonstrate more assurance in code implementation. Reviewers felt less burden and more value. Originally a green checkmark, a Slack emoji has become the modern equivalent of thanks for a careful evaluation and has evolved into a subtle cultural habit spread naturally. Code Quality: Additional defects were early found thanks to additional examination. Production problems dropped and post-deployment rollbacks were almost nonexistent. What was once a horrible habit turned into a group effort dare we not even nearly pleasant?

6.4. Realizations Acquired

This scenario did not call for complex instruments or authoritative commands. Reevaluating the code review method took front stage. Culture Overcomes Instruments. If the surrounding culture is toxic or apathetic, even the most strong review instrument becomes useless. Dealing with personal problems calls for personal involvement. Examining checklists, using buddy systems & developing tone awareness seem simple little adjustments with many major effects. There are others. They are, nonetheless, also sticky. Regular implementation of these penetrates team procedures and improves everyday experiences. Code reviews ultimately have a social component and go beyond simple technical chores. Teams gain when they see them that way as well.

7. Conclusion and Takeaways

Code reviews are more than just a part of the development process; they are a discourse, a group learning tool, and an opportunity to produce more solid and durable software. When done well, code reviews might help to promote these technical expertise and cooperative growth. Along with their importance, this is a definitive overview of the main points of view for both submitters and reviewers.

- **For those who contribute: Clearly and sympathetically convey:** Your duties as a code contributor go beyond just code submission to encompass the review process. See your pull request as a story you are telling. Help your reviewer to see the justification behind your decisions. When context is lacking, provide comments; link more relevant tickets or discussions; keep changes within a small, targeted range. This saves time and promotes meaningful comments as well. Consider criticisms as an opportunity for development rather than a personal critique. The most skilled developers are those who skillfully use criticism for both personal and professional development rather than those who run from it.
- **For those in review: Act deliberately; avoid judgment:** Reviewing a colleague's work reminds you that you are communicating with a peer rather than merely assessing code. Avoid little criticism or ostentation. Try instead to understand the code's fundamental goal. Ask clear questions instead of assuming anything. Sort issues according to

importance; avoid interfering with a merger for little aesthetic reasons. Mostly, provide a safe space fit for learning. Encouragement of interest even about seemingly little details often starts discussions that progress teams.

- **Respect and Inquiry: The Foundation of Effective Reviews:** Any great code review begins with mutual respect. The entire process runs more smoothly and effectively when reviewers and submitters perceive one another as collaborators rather than adversaries. Combine that respect with inquiry; be ready to probe design choices closely, ask "why," and consider many other approaches. From a gatekeeping role, this paradigm turns code reviews into a reciprocal path for knowledge sharing.
- **Better software and group learning opportunities abound from code reviews:** Code reviews have as their main goals not only increasing software quality but also strengthening team relations and beyond simple correctness. We improve not just the codebase but also our technical aptitude, communication, and cooperation by seeing reviews as a shared learning tool. Regular, high-quality evaluations provide long-lasting benefits like fewer flaws, quicker onboarding of new engineers, and a team sense of responsibility.
- **Still changing:** In the end, code review is a skill that develops with introspection and intentionality. After every assessment, give some thought: What worked? Other than what I did? Did I leave the person motivated and supported? These quick notes improve your capacity for both review and teamwork. Code reviews ultimately need not be uncomfortable. By means of empathy, curiosity, and clear communication, they might be among the most powerful tools available to your team in creating high-quality software and supporting a strong engineering culture.

References

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

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

[3] Reagle, Joseph Michael. *Reading the comments: Likers, haters, and manipulators at the bottom of the web*. Mit Press, 2015.

[4] Ha, Elizabeth, and David Wagner. "Do android users write about electric sheep? examining consumer reviews in google play." *2013 IEEE 10th Consumer Communications and Networking Conference (CCNC)*. IEEE, 2013.

[5] Tsay, Jason, Laura Dabbish, and James Herbsleb. "Let's talk about it: evaluating contributions through discussion in GitHub." *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*. 2014.

[6] Sai Prasad Veluru. "Real-Time Fraud Detection in Payment Systems Using Kafka and Machine Learning". *JOURNAL OF RECENT TRENDS IN COMPUTER SCIENCE AND ENGINEERING (JRTCSE)*, vol. 7, no. 2, Dec. 2019, pp. 199-14

[7] Brown, Stephen. "I have seen the future and it sucks: reactionary reflections on reading, writing and research." *European Business Review* 24.1 (2012): 5-19.

[8] Jani, Parth. "UM Decision Automation Using PEGA and Machine Learning for Preauthorization Claims." *The Distributed Learning and Broad Applications in Scientific Research* 6 (2020): 1177-1205.

[9] Williams, Alex C., et al. "Mercury: Empowering Programmers' Mobile Work Practices with Microproductivity." *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology*. 2019.

[10] Khan, Selim M., James Gomes, and Daniel R. Krewski. "Radon interventions around the globe: A systematic review." *Heliyon* 5.5 (2019).

[11] Freeman, Eric, et al. *Head First Design Patterns: A Brain-Friendly Guide*. " O'Reilly Media, Inc.", 2004.

[12] Allam, Hitesh. *Exploring the Algorithms for Automatic Image Retrieval Using Sketches*. Diss. Missouri Western State University, 2017.

[13] Trivedi, Chirag, Michel J. Cervantes, and Ole G. Dahlhaug. "Experimental and numerical studies of a high-head Francis turbine: A review of the Francis-99 test case." *Energies* 9.2 (2016): 74.

[14] Kupunarapu, Sujith Kumar. "AI-Enabled Remote Monitoring and Telemedicine: Redefining Patient Engagement and Care Delivery." *International Journal of Science And Engineering* 2.4 (2016): 41-48.

[15] Kraft, Matthew A., and Allison F. Gilmour. "Can principals promote teacher development as evaluators? A case study of principals' views and experiences." *Educational administration quarterly* 52.5 (2016): 711-753.

[16] 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.

[17] Liu, Ming, Lei Tan, and Shuliang Cao. "A review of prewhirl regulation by inlet guide vanes for compressor and pump." *Proceedings of the Institution of Mechanical Engineers, Part A: Journal of Power and Energy* 233.6 (2019): 803-817.

[18] Sai Prasad Veluru. "Optimizing Large-Scale Payment Analytics With Apache Spark and Kafka". *JOURNAL OF RECENT TRENDS IN COMPUTER SCIENCE AND ENGINEERING (JRTCSE)*, vol. 7, no. 1, Mar. 2019, pp. 146–163

[19] Tiainen, Jonna, et al. "Flow control methods and their applicability in low-Reynolds-number centrifugal compressorsa review." *International Journal of Turbomachinery, Propulsion and Power* 3.1 (2017): 2.

[20] Arugula, Balkishan, and Sudhkar Gade. “Cross-Border Banking Technology Integration: Overcoming Regulatory and Technical Challenges”. *International Journal of Emerging Research in Engineering and Technology*, vol. 1, no. 1, Mar. 2020, pp. 40-48

[21] Fuller, Thomas, et al. "What affects authors’ and editors’ use of reporting guidelines? Findings from an online survey and qualitative interviews." *PLoS One* 10.4 (2015): e0121585.

[22] Papay, John P., and Susan Moore Johnson. "Is PAR a good investment? Understanding the costs and benefits of teacher peer assistance and review programs." *Educational Policy* 26.5 (2012): 696-729.