How Code Review ~~Frequency and Depth~~ Impacts the ~~Number of Bugs in a~~ Software ~~Release: Hyrex Case Study~~

Shivani Gadipe
College of Engineering
Oregon State University
Corvallis, OR
gadipes@oregonstate.edu

Tu Lam
College of Engineering
Oregon State University
Corvallis, OR
lamtu@oregonstate.edu

Alexander Yang Rhoads
College of Engineering
Oregon State University
Corvallis, OR
rhoadsal@oregonstate.edu

***Abstract*—**
Code reviews are a standard practice in software development and are crucial for improving software quality, reducing bugs, and maintaining consistency within projects. However, there is an ongoing debate about the point of diminishing returns, such as when does the frequency and depth of reviews stop leading to significant improvements in software quality? This paper investigates whether increasing the frequency and depth of code reviews correlates with fewer bug reports in open-source programs. ~~The~~ analysis ~~will be~~ focuse~~d~~ on a single open source repository, with the intention of examining the relationship between review quality and bug reports, using statistical methods to identify any potential correlation. Should a positive correlation be found, this study will propose a set of best practices to improve code reviews, ultimately reducing the resources required for post-release bug fixes.

***Index Terms*—**
Code Review Depth~~: Average number of comments or issues raised by a reviewer per code change.~~

Code Review Frequency~~: The number of individual code reviews.~~

True Positive~~: A correct prediction result that indicates something is true when it is indeed actually true.~~

False Positive~~: An incorrect prediction result that indicates something is true when it is actually false.~~

Mining Software Repositories~~: A field of study examining the date in software repositories.~~

Precision~~: A performance metric meant to track accuracy. It corresponds to the fraction of values that actually belong to a positive class out of all of the values which are predicted to belong to that class.~~

RepoDriller~~: A tool that helps extract data from repositories.~~

I. Introduction

Code reviews are a standard practice in software development to improve code quality, reduce bugs, and maintain consistency across projects. However, ~~quantifying the relationship between the frequency and depth of code reviews, with the number of bugs in software releases, remains obscure. The problem lies in identifying~~ whether frequent and in-depth code reviews lead to better quality software, or if there is a diminishing return beyond a certain point.

As a vital part of the software development process, reviews help teams identify and fix errors before they reach production. However, there is often uncertainty about how frequently reviews should be conducted and how detailed they need to be to maximize their benefits.

By understanding the relationship between code review practices and the number of bugs in a software release, this research aims to help development teams to:

- Improve software quality by identifying optimal review practices.
- Save time and resources by avoiding unnecessary or redundant reviews.
- Enhance team efficiency by focusing on effective review strategies.

This study will provide actionable information to guide teams in adopting better review practices, ultimately leading to more reliable and user-friendly software.

Being a major quality assurance technique for development teams, code reviews are meant to prevent bugs and improve code quality. There exists many approaches to reviewing, leading, and many teams already incorporate a set of style guides for code reviews as a means of ensuring consistency, but even within that set, exact specifications differ, and the effectiveness of each is unknown. Evaluating how different approaches to reviews affect the development process is of vital interest to software engineers, as assessing the most effective practices will improve efficiency and avoid pitfalls.

The study will measure how the frequency and depth of a code review impact the number of bugs upon release. An examination of a code review dataset will take place. The scope of the project will be all programs within a dataset, evaluating code reviews against others within a singular program to maintain consistency. ~~Due to the straightforward metrics of examination, the project should be finishable within five weeks as long as the scale of the dataset is controlled. A~~s discussed earlier, the dataset will start as an examination of a single large scale program. ~~If time permits, expansion of the dataset into other open source projects is possible.~~ Evaluations will be made individually, and in comparison to the results of other projects, so as to provide as much data as needed.

When measuring code review frequency and depth, it's important to consider certain assumptions about the process in order to properly research in this area. One key assumption is that developers are familiar with the code review practices (e.g., frequency, depth, and review criteria) and that these practices are consistent within the team or the project being studied. This ensures that we can isolate the impact of frequency and depth on bug counts that will aid with the discovery in correlation. Another assumption is that the quality of code reviews is consistent across all team members. For this, it is assumed that all reviewers have similar expertise and follow the same guidelines during the review process. Additionally, we assume that the bugs detected during code reviews are representative of the bugs that would have otherwise been discovered post-release of a software. With these assumptions in mind, this can determine a more effective exploration on the correlation between code review practices and bug occurrence, helping to identify patterns in how frequency and depth impact the number of bugs in the code.

As a quality assurance mechanism, several aspects of code reviews are open to the question of examination. Concepts such as the depth and frequency of code reviews inspire questions regarding how they influence the number of bugs within the software. Below are the research questions that seek to evaluate these effects:

1) How does the frequency of code reviews correlate with the number of bugs reported?
2) Is there a reduced rate of bugs in tandem with an increased depth of code reviews?

The dataset will focus on evaluating the open source project Hyrax, a Ruby on Rails engine made by the Samvera community. The repository is publicly available on GitHub, as are the issues and PR (Pull Request). The goal is to analyze the PR on GitHub and assess how many reviewers are actively reviewing each PR. We will then apply a code review formula to determine the correlation between the number of reviewers and the quality of the PRs in the repository.

We will use a formula based on the depth of code reviews, which is calculated by dividing the number of comments, or issues, raised by a reviewer by the number of code changes they reviewed. This allows us to analyze both the frequency of reviews and the percentage of depth in the feedback. If both the percentage of depth and the frequency of reviewers are high, and no significant ticketing issues arise, we can infer a strong correlation between the two factors. On the other hand, if these conditions are not met, we conclude there is no meaningful correlation.

The evaluation will be using any of these standard metrics, including: Precision, Recall (True Positive Rate), Sensitivity, False Positive Rate (FPR), Specificity, Confusion Matrix, and the AUC curve. Based on these calculations, we will assess whether the results align with one of our formulas based on code reviews, which will help us determine if our evaluation aligns with established correlation standards.

II. Background and Motivation

A. Background

In the field of software development, code reviews are a crucial quality process where developers review each other's code for errors, improvements, and adhere to the best practices. The goal is to reduce defects and enhance the maintainability of software. However, there remains an ongoing challenge in determining the optimal frequency and depth of these reviews to maximize software quality. Specifically, it's unclear when the frequency and depth of reviews begin to offer minimal returns in terms of reducing bugs in software. For example, when and if the frequency and depth of reviews stops making a significant increase in quality. Many different code reviews could lead to inefficiencies, and overly thorough reviews might not yield enough additional benefit to justify the time spent on code review as one study suggests that around 30.7% of responses to the answers provided by ChatGPT are negative meaning there are still bugs present after code review with LLM. (Miku Watanabe, 2024) [1].

To address this problem, we propose to analyze the relationship between the frequency and depth of code reviews and the number of bugs in software releases. Frequency in this context refers to how often code reviews are conducted during the development lifecycle which could be from the reviewer themselves, while depth is a metric that quantifies how comprehensive each review is, based on the percentage calculated from the formula provided. Our approach is to collect data from development open-source projects and correlate review patterns with the number of bugs found in their final releases. By doing so, we aim to determine whether there is an optimal point where increasing the frequency and depth of reviews results in a significant reduction in defects or whether additional effort beyond a certain threshold yields minimal improvements. This research will provide actionable insights into how software engineering teams can structure their code review processes for maximum efficiency, reduction in bugs, and ultimately contributing to higher-quality software releases.

B. Motivation

One prominent understanding of code review motivation can be found in the 2012 case of the Knight Capital Group, a financial services firm, which lost $440 million in 45 minutes due to a bug in its trading software update that it has released to the public. This could have been prevented with more thorough and frequent usage of code reviews.

~~Imagine a software development team working on a mobile app for a large retailer. In this case, a seemingly small error in the checkout process, unnoticed during initial code reviews, results in thousands of failed transactions on launch day. While this might seem like a minor bug, the financial and reputational damage can be substantial leading to lost revenue, frustrated customers, and, in some cases.~~

~~Optimizing code reviews can prevent such issues from arising. If the right review practices were applied, the issue could have been caught earlier, saving the team countless hours of fixing post release issues and significantly reducing the risk of a high profile failure.~~

~~This study seeks to answer the question: When and if the frequency and depth of reviews stops making a significant increase in quality?~~

~~By identifying the most effective frequency and depth of code reviews, the help of the development teams hope to improve the software quality without wasting valuable time. With the results of this research, development teams could adopt more efficient, targeted review strategies, ultimately leading to faster release cycles, higher quality software, and a smoother user experience. By applying metrics and looking through the overall relations between code review frequency and depth, the team hopes to bring a brighter idea of these practices that can help with the development of minimizing bugs on launch products.~~

III. Related Work

Literature regarding code review quality can be put into several broad categories. There are direct surveys with participant feedback that makes up the dataset, there are developmental studies that track the results of an iterated system, as well as direct data extraction from repositories. Despite the larger differences, they mostly aim to either improve understanding of code reviews or their practices.

Oussama Ben Sghaie and Houari Sahraoui's 2024 study primarily intersects with research in three key areas of code review automation: quality estimation, comment generation, and code refinement [2]. Each of these tasks has been addressed separately in the literature, with recent efforts focusing on leveraging pre-trained language models. However, existing approaches often treat these tasks in isolation, ignoring the connections between them. Quality estimation methods have been used to assess code quality, comment generation techniques focus on generating helpful review comments, and code refinement strategies aim to improve code quality. Our work differentiates itself by proposing the art of looking into the correlation between frequency of code review and depth cause an effect on buggy software being released or not. Even though one is looking into improving the technique of code review, this work gives us insight as hoping the accurate code increase can benefit the less bugginess of software launch.

The 2023 study by Asif Kamal Turzo and Amiangshu Bosu contributes to the field of code review effectiveness, focusing specifically on the factors influencing the usefulness of code review comments (CRCs) in Open Source Software (OSS) development [3]. Previous research has explored aspects of code review automation, defect prediction, and reviewer recommendations. This work intersects with these areas by exploring how both technical and linguistic elements of a CRC influence its perceived usefulness, while also considering contextual and participant-related factors. The findings suggest a need to reconsider reviewer recommendation models, urging a more context-sensitive approach to improving code review effectiveness. This work can help provide a good contrast in the case of the correlation as if the improvement of more frequency reviewer is determined to be there, would this cause less bugs in code when present on release on software?

M. Nejati, M. Alfadel and S. McIntosh's 2023 study addresses the intersection of build system maintenance and code review practices, particularly in relation to build specifications [4]. Build systems play a critical role in integrating source code into executables, and maintaining these systems is known to be a challenge due to the complexities they introduce. Code review has been widely studied in the context of production and test code, but its application to build specifications remains under-explored. This research expands on finding that comments on changes to build specifications are more likely to point out defects than rates reported in the literature for production and test code, and evolvability and dependency-related issues are the most frequently raised patterns of issues. The study indicates to our work that the relevant code review has a significant effect on bugs presented in code and the more reviews there are, the less likelihood of bugs shown in software launch.

A 2021 survey by Wang, Dong, et al. sought to answer three research questions: what the extent of link sharing in review discussion was, if the number of links in review discussion correlate with review time, and what the common intentions of link sharing are [5]. Focusing on reviews in the Openstack and Qt projects, the study limits its dataset to closed reviews with a comment by the reviewer. To answer the research questions, quantitative analysis is applied for each. Qualitative analzysis is also applied to questions involving reviewer intention. In the case of link number, the number of reviews was measured against number links, whether it was the reviewer or author who shared the link, and where the link leads (if it's related to internal project sites or not, and if it links to a GitHub activity, communication channel, a video or photo, a memo, or a review). Review time correlation was analyzed by measuring line adds, deletes, patch sizes (as in quantity of adds and deletes), patch purpose, number of files, number of review iterations, number of prior patches made by the author, number of comments, number of author comments, number if reviewer comments, number of reviewers, number of external links, internal links, and total links. These variables were fitted into an Ordinary Least Squares multiple regression model. With regards to link sharing intention, seven categories were identified, including providing context, elaboration, clarification, explanation of necessity, proposals, expert suggestions, and informing split patches. The three most common of the categories, context, necessity explanations, and elaborations, were directly measured as an overall percentage. Additional developer feedback was also sought out as a part of the research. This included the degree to which developers agreed with the conclusions of the study, as a survey on what of the seven link intention categories do they most commonly have. Similar to our study, the paper seeks to analyze correlation of code review performance with certain practices, though it focusing on the specific topic of link sharing, as opposed to a more broad one like code frequency and depth.

In the 2021 research paper by AlOmar, Eman Abdullah, et al., a case study using the work of 24 developers at Xerox examines refactoring practices in the context of modern code reviews [6]. Five research questions are covered: what motivates developers in the context of modern code reviews, how developers document refactoring for code reviews, what are the challenges that reviewers face with regards to refactoring changes, what mechanisms do developers and reviewers use to ensure correctness after refactoring, and how developers and reviewers assess and perceive the impact of refactoring on code quality. The study's dataset for qualitative analysis used surveys given out to the developers asking about their background, motivation, documentation practices, challenges as a reviewer, verification practices, and the implications of refactoring. The data for qualitative analysis was metadata extracted from review requests, using the Mann-Whitney U test to compare data. Evaluation with regards to direct survey results considers Xerox's own internal guidelines, as well as percentages cataloging response quantities. This paper heavily deviates from evaluation practices of our own, directly asking input from code reviewers. Still, both papers share a similar approach to metadata analysis.

A 2025 study (Oliveira, Delano, et al.) tries to understand how code review comments improve code understandability [7]. The research questions are: How often do reviewers ask for code understandability improvements in code review comments, what are the main issues pertaining to code understandability in code review, how likely are understandability improvement comments to be accepted, what code changes are found in understandability improvements, to what extent are accepted code understandability improvements reverted, and if linters contain rules to detect the identified code understandability smells. Data is collected from the code reviews of open source Java projects, which are checked for references to understandability. The reviews are grouped by their topic, such as if the suggestions pertain to understandability, rate of acceptance for improvements, or what changes to understandability exist. This information is separated into different categories, and tallied into percentages. Further analysis takes the data and puts it into the Kappa coefficient. The direct taking of data from code reviews mirrors our own approach. However, the study requires the team to analyze and categorize their data before it can be quantified, whereas research regarding depth and frequently can be immidietly put into numbers.

The 2022 study by Gonçalves, Pavlína Wurzel, et al. tests whether giving explicit review strategies improves code review performance and reduces cognitive load. There are two review questions, each comprising two smaller subproblems [8]. The first: if guidance in reviews leads to higher review effectiveness (proportion of defects found), as well as higher efficiency (functional defects found over time). The second asks if the effectiveness and efficiency of guidance is mediated by lower cognitive load. The study took the form of an experiment, hiring 70 developers and tracking their background (education, prior experience, and age). Developers were split into three groups and given either nothing, a checklist that lists what to look out for, or a guided checklist that specifies the scope of what to look out for. Cognitive load is measured via a questionnaire. The results of the three groups are measured against the developer's individual backgrounds and the answers from the cognitive load surveys. Both our own paper and theirs seeks to answer the same question of improving code performance, but instead of extracting metadata from repositories and comparing it, this paper receives data via direct behavioral experiments.

The 2024 document by Frommgen et al. is an internal Google study looking into the capability of machine learning to automate and streamline the code review process [9]. The research question is the same as the premise: the extent to which a machine learning model can aid in all aspects of the code review process. Data took the form of the code review suggestions the studied ML model gave across iterations, and how participants responded to them. Participant responses (as in what they did with the suggestions) were categorized and calculated as a percentage.The suggestions by the model were separated into different key categories, such as suggestions to complicated code reviews and open ended reviews, and were analyzed qualitatively. Verbal feedback from participants was also analyzed on a qualitative basis. The Google study seeks to improve the code review process, similar to our own, but the avenue of its research revolves around how specific technology (ML models) can improve review practices, rather than analyzing the performance of review practices directly.

The 2022 paper by Gonçalves, Pavlína Wurzel, et al. investigates how developers perceive code review conflicts and addresses interpersonal conflicts during code reviews as a theoretical construct [10]. The study seeks to answer five research questions: how developers perceive and experience conflicts during code review, what are conflicts during code review like, what the positive and negative consequences of conflicts during code review are, what factors intervene in the conflict dynamics, and what strategies developers use to prevent and manage conflicts. Data from this study revolved around a series of interviews asking participating developers inquiries derived from the research questions themselves. Answers were compiled in broad categories and analyzed for implications on the basis of the developers' individual answers. As a direct feedback survey, this paper heavily deviates in terms of dataset origin from our own. Despite that, it still is trying to examine exact trends in code review behavior.

M. B. Ada and M. U. Majid's 2022 paper on student code peer reviews attempts to develop a system to increase the engagement of programming students during peer code reviews [11]. The research question is the same as the premise, but more specifically it means to develop a system "that aims to provide course instructors with the capability to create and conduct formative and summative peer code review exercises for their students... improve student engagement with the review process and motivation to achieve better learning." This takes the form of an experiment that sees participating students provide feedback to reviewers, as well as giving rewards for both completing reviews and handing out high quality ones. Results from the experiment were taken directly from participants in the shape of feedback forms. Answers to these forms were then quantified into percentages. This is another direct feedback survey, and one aimed at resolving educational needs rather the broadly professional ones. Our own paper analyzes information from repositories, comparing different quantities from that set, to answer research questions regarding code review practices. However, this study is relevant in how it both qualitatively and quantitatively analyzes its data, which showcases avenues of approach for our own study.

IV. Approach

The approach to this research is designed to address the research questions by analyzing bug reports from the open-source software repository Hyrax. Given that Hyrax has been actively developed since 2016, we expect it to have a rich dataset of user-reported issues, particularly related to bug fixes. Our solution will involve two different kinds of approaches: qualitative analysis of bug tickets and quantitative techniques for evaluating code review to see potential correlation.

First, we will focus on analyzing historical ticketing data from the repository, specifically bug-related tickets. Using Mining Software Repositories (MSR) techniques, we will extract relevant information about the nature and frequency of bug reports, identifying patterns and insights that could correlate the frequency and depth of code reviews. Frequency being the number of reviews, while code depth is the average number of comments or issues raised by a reviewer per code change.

To analyze the impact of code reviews on the presence of bugs, we will compute metrics such as code review frequency and depth. These will be cross-referenced with the reported bugs in the ticketing system to assess if bugs are more likely to be associated with certain review patterns and we can compare this based on the information given through MSR. We will also evaluate the precision of code using established evaluation metrics, such as the precision method to determine if higher code review frequency correlates with better code quality and fewer reported bugs.

This multi-technique approach will provide actionable insights into the relationship between code reviews and the prevalence of bugs in open-source software. The expected outcome is to identify patterns that can help developers improve code quality by focusing on code review practices.

V. Evaluation

A. Dataset

This study will examine code reviews from the open-source Hyrax repository. Specifically, it will analyze 3,781 closed pull request reviews and 1,960 issues. Of these, 466 issues are related to bugs, while 185 pull request reviews are labeled as bug fix notes. This dataset has been selected based on its relevance to the research questions at hand, focusing on a well-established repository with a significant volume of issues and pull requests. Prior research has indicated that repositories like Hyrax offer valuable insights into bug-related discussions and fixes. Although there are existing benchmarks for similar studies, the Hyrax project was chosen as it provides enough data to act as a rich source for analyzing the nature of bug fixes.

B. Metrics

To evaluate the effectiveness of the code review process in identifying the correlation between review frequency and depth, we will use the following metrics:

1) **Code Review Frequency:** This metric is defined as the number of active reviewers in a pull request (PR) who contribute suggestions. A higher frequency could indicate a more collaborative review process, potentially leading to better bug detection.
2) **Code Review Depth:** This is calculated as the ratio of the number of changes requested by the reviewer to the number of changes actually made by the PR owner. A higher depth value suggests that reviewers are providing

detailed feedback and that the PR owner is addressing those suggestions.

$$Depth = \frac{No.\ of\ changes\ requested\ by\ the\ reviewer}{No.\ of\ changes\ made\ by\ the\ owner\ of\ PR} \quad (1)$$

3) **Precision:** The precision metric is defined by the formula:

$$Precision = \frac{TP}{TP + FP} \quad (2)$$

- **TP (True Positive):** The number of changes that the reviewer identified as bugs and which were verified as bugs after being addressed by the PR owner.
- **FP (False Positive):** The number of suggestions made by the reviewer that were not actually bugs when reviewed by the PR owner.

These metrics were selected based on their relevance to understanding the impact of code review on bug detection By measuring precision, and code review frequency and depth, we can assess how effectively code reviewers are identifying genuine issues and distinguishing them from non-issues and see the correlation affect the software or not on release.

C. Experiment Procedure

The experiment will begin by analyzing the Hyrax repository, focusing on closed issues and linked pull requests (PRs). We will employ the Mining Software Repositories (MSR) technique using the **RepoDriller** tool to extract key repository data, such as commits, branches, tags, developer information, code modifications, and diffs. These data points will serve as the foundation for calculating the necessary metrics outlined in the **Evaluation Metrics** section.

Once the data is extracted, we will compute the **Code Review Frequency** and **Code Review Depth** based on the formulas provided in the Metrics section. Specifically, we will calculate the frequency of reviewers contributing to each PR and the depth by analyzing the number of requested changes compared to the changes made by the PR owner.

To validate the reliability of these metrics, we will manually inspect the repository to determine whether bugs reported in the PRs persist after the software release. If the issue is addressed in future PRs or releases and is linked to the original PR, we will consider the bug resolved, confirming the accuracy of the calculated metrics. If no link exists, we will interpret this as evidence of a correlation between code review frequency/depth and bug persistence.

In addition, we will apply **Precision** as an evaluation metric. Using the **TP** (True Positive) represents the number of reviewer-identified issues that were verified as actual bugs, and **FP** (False Positive) represents the reviewer-identified issues that were not bugs. This metric will help us assess the validity of our findings and provide additional support for determining the correlation between code review characteristics and bug detection. By following these steps, we aim to rigorously evaluate the effectiveness of code review frequency and depth in identifying and resolving bugs, ultimately answering our research question.

D. Results

[[THIS SECTION SHOULD BE ORGANIZED IN TERMS OF THE RESEARCH QUESTIONS YOU ASK. RIGHT NOW, YOU DON'T HAVE THE ANSWERS, BUT JUST CREATE HIGH-LEVEL TEXT DESCRIBING YOUR RESULTS THAT YOU WILL EDIT LATER, WHEN YOU COMPUTE THE EXACT RESULTS.]]

VI. Discussion and Threats to Validity

VII. Contributions

References

[1] M. Watanabe, Y. Kashiwa, B. Lin, T. Hirao, K. Yamaguchi, and H. Iida, “On the use of chatgpt for code review: Do developers like reviews by chatgpt?” in *EASE ’24: Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, 2024, p. 375–380.

[2] O. Ben Sghaier and H. Sahraoui, “Improving the learning of code review successive tasks with cross-task knowledge distillation,” *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, Jul. 2024. [Online]. Available: https://doi.org/10.1145/3643775

[3] A. K. Turzo and A. Bosu, “What makes a code review useful to opendev developers? an empirical investigation,” *Empirical Software Engineering*, vol. 29, no. 1, p. 6, Nov 2023. [Online]. Available: https://doi.org/10.1007/s10664-023-10411-x

[4] M. Nejati, M. Alfadel, and S. McIntosh, “Code review of build system specifications: Prevalence, purposes, patterns, and perceptions,” in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023, pp. 1213–1224.

[5] D. Wang, T. Xiao, T. Patanamon, R. G. Kula, and K. Matsumoto, “Understanding shared links and their intentions to meet information needs in modern code review:,” *Empirical Software Engineering*, vol. 26, no. 5, Sep Sep 2021, copyright - © The Author(s) 2021. This work is published under http://creativecommons.org/licenses/by/4.0/ (the “License”). Notwithstanding the ProQuest Terms and Conditions, you may use this content in accordance with the terms of the License. [Online]. Available: https://oregonstate.idm.oclc.org/login?url=https://www.proquest.com/scholarly-journals/understanding-shared-links-their-intentions-meet/docview/2549481919/se-2?accountid=13013

[6] E. A. AlOmar, H. AlRubaye, M. W. Mkaouer, A. Ouni, and M. Kessentini, “Refactoring practices in the context of modern code review: An industrial case study at xerox,” in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2021, pp. 348–357.

[7] D. Oliveira, R. Santos, B. de Oliveira, M. Monperrus, F. Castor, and F. Madeiral, “Understanding code understandability improvements in code reviews,” *IEEE Transactions on Software Engineering*, vol. 51, no. 1, pp. 14–37, 2025.

[8] P. W. Gonçalves, E. Fregnan, T. Baum, K. Schneider, and A. Bacchelli, “Do explicit review strategies improve code review performance? towards understanding the role of cognitive load,” *Empirical Software Engineering*, vol. 27, no. 4, Jul Jul 2022, copyright - © The Author(s) 2022. This work is published under http://creativecommons.org/licenses/by/4.0/ (the“License”). Notwithstanding the ProQuest Terms and Conditions, you may use this content in accordance with the terms of the License. [Online]. Available: https://oregonstate.idm.oclc.org/login?url=https://www.proquest.com/scholarly-journals/do-explicit-review-strategies-improve-code/docview/2660495235/se-2?accountid=13013

[9] A. Froemmgen, J. Austin, P. Choy, N. Ghelani, L. Kharatyan, G. Surita, E. Khrapko, P. Lamblin, P.-A. Manzagol, M. Revaj, M. Tabachnyk, D. Tarlow, K. Villela, D. Zheng, S. Chandra, and P. Maniatis, “Resolving code review comments with machine learning,” in *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice*, ser. ICSE-SEIP ’24. New York, NY, USA: Association for Computing Machinery, 2024, p. 204–215. [Online]. Available: https://doi.org/10.1145/3639477.3639746

[10] P. Wurzel Gonçalves, G. Çalikli, and A. Bacchelli, “Interpersonal conflicts during code review: Developers’ experience and practices,” *Proc. ACM Hum.-Comput. Interact.*, vol. 6, no. CSCW1, Apr. 2022. [Online]. Available: https://doi.org/10.1145/3512945

[11] M. B. Ada and M. U. Majid, “Developing a system to increase motivation and engagement in student code peer review,” in *2022 IEEE International Conference on Teaching, Assessment and Learning for Engineering (TALE)*, 2022, pp. 93–98.