Studying Practical Challenges of Automated Code Review Suggestions

by

Farshad Kazemi

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2024

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner:	Dr. Christian Bird Principal Researcher Microsoft Research
Supervisor:	Dr. Shane McIntosh Associate Professor David R. Cheriton School of Computer Science University of Waterloo
Internal Member:	Dr. Michael W. Godfrey Professor David R. Cheriton School of Computer Science University of Waterloo
	Dr. Meiyappan Nagappan Associate Professor David R. Cheriton School of Computer Science University of Waterloo
Internal-External Member:	Dr. Weiyi Shang Associate Professor Electrical and Computer Engineering University of Waterloo

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Code review is a critical step in software development, focusing on systematic source code inspection. It identifies potential defects and enhances code quality, maintainability, and knowledge sharing among developers. Despite its benefits, it is time-consuming and error-prone. Therefore, approaches such as Code Reviewer Recommendation (CRR) have been proposed to streamline the process. However, when deployed in real-world scenarios, they often fail to account for various complexities, making them impractical or even harmful. This thesis aims to identify and address challenges at various stages of the code review process: validity of recommendations, quality of the recommended reviewers, and the necessity and usefulness of CRR approaches considering emerging alternative automation. We approach these challenges in three empirical studies presented in three chapters of this thesis.

First, we empirically explore the validity of the recommended reviewers by measuring the *rate of stale reviewers*, i.e., those who no longer contribute to the project. We observe that stale recommendations account for a considerable portion of the suggestions provided by CRR approaches, accounting for up to 33.33% of the recommendations with a median share of 8.30% of all the recommendations. Based on our analysis, we suggest separating the reviewer contribution recency from the other factors used by the CRR objective function. The proposed filter reduces the staleness of recommendations, i.e., the Staleness Reduction Ratio (SRR) improves between 21.44%–92.39%.

While the first study assesses the validity of the recommendations, it does not measure their quality or potential unintended impacts. Therefore, we next probe *the potential unintended consequences of assigning recommended reviewers*. To this end, we study the impact of assigning recommended reviewers without considering the safety of the submitted changeset. We observe existing approaches tend to improve one or two quantities of interest while degrading others. We devise an enhanced approach, Risk Aware Recommender (RAR), which increases the project safety by predicting changeset bug proneness.

Given the evolving landscape of automation in code review, our final study examines whether human reviewers and, hence, recommendation tools are still beneficial to the review process. To this end, we focus on the behaviour of Review Comment Generators (RCGs), models trained to automate code review tasks, as a potential way to replace humans in the code review process. Our quantitative and qualitative study of the ACR-generated interrogative comments shows that ACR-generated and human-submitted comments differ in mood, i.e., whether the comment is declarative or interrogative. Our qualitative analysis of sampled comments demonstrates that ACR-generated interrogative

comments suffer from limitations in the ACR capacity to communicate. Our observations show that neither task-specific ACRs nor LLM-based ones can fully replace humans in asking questions. Therefore, practitioners can still benefit from using code review tools.

In conclusion, our findings highlight the need for further support of human participants in the code review process. Thus, we advocate for the improvement of code review tools and approaches, particularly code review recommendation approaches. Furthermore, tool builders can use our observations and proposed methods to address two critical aspects of existing CRR approaches.

Related Publications

Earlier versions of the work in this thesis have been published as follows:

- **Characterizing the Prevalence, Distribution, and Duration of Stale Reviewer Recommendations** (Chapter 4). Farshad Kazemi, Maxime Lamothe, Shane McIntosh. IEEE Transactions on Software Engineering (TSE), In Print, 14 pages.
- **Exploring the Notion of Risk in Code Reviewer Recommendation** (Chapter 5). Farshad Kazemi, Maxime Lamothe, Shane McIntosh. In Proceedings of the International Conference on Software Maintenance and Evolution (ICSME 2022), pp. 139–150.
- **Interrogative Comments Posed by Automatic Code Reviewers** (Chapter 6). Farshad Kazemi, Maxime Lamothe, Shane McIntosh. IEEE Transactions on Software Engineering (TSE), Under Submission, 13 pages.

The following publication, while not directly related to the material in this thesis, was produced concurrently with the research conducted for this thesis.

- **Reevaluating the Defect Proneness of Atoms of Confusion in Java Systems** Guoshuai Shi, Farshad Kazemi, Michael W. Godfrey, Shane McIntosh. To appear in the Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM 2024), 12 pages.

Acknowledgements

It has been four years since I began my PhD journey at the University of Waterloo, and looking back, I am filled with gratitude, happiness, and joy for the amazing people who have been part of it or crossed paths with me along the way.

First and foremost, my deepest thanks go to my supervisor, Dr. Shane McIntosh. Your constant guidance, support, and belief in me have meant the world. I have learned so much under your supervision, and I truly appreciate everything you have done to help me succeed on this journey.

Then, I would like to thank Dr. Maxime Lamothe, who was like a co-supervisor to me; I cannot thank you enough for your insights and collaboration. Even though we were not always in the same place, you were incredibly generous with your time, and your support was invaluable throughout my PhD. Working with you has been an incredible learning experience, and getting to know you as an amazing person has been a bonus.

A special thank you to Albert Shi for trusting me in your academic journey as we co-authored a paper together. It has been a privilege working alongside you. Collaborating with Dr. Mike Godfrey, who also graciously joined my committee, was equally priceless. Mike, your insightful advice and our conversations have guided me through the complexities of this journey, and I am truly grateful for your support.

I also want to extend my heartfelt thanks to my committee members: Dr. Christian Bird, who kindly accepted my invitation to join the committee and visited us at the University of Waterloo, and Dr. Meiyappan Nagappan and Dr. Weiyi Shang, who took the time to review this thesis and provide invaluable feedback. Your support has been instrumental in shaping this work.

Next, I want to thank all the members of the Software Repository Excavation and Build Engineering Labs (REBELs) and the University of Waterloo Software Engineering (SWAG) Lab, with whom I have shared many conversations over the past years. Some names that come to mind are Akinbowale Akin-Taylor, Anubhav Gupta, Faizan Khan, Gareema Ranjan, Gengyi Sun, Keheliya Gallaba, Mahtab Nejati, Dr. Mahmoud Alfadel, Mehran Meidani, Mingyang Yin, Nimmi Rashinika Weeraddana, Raymond Chang, Dr. Rungroj Maipradit, Shaquille Pearson, Xiaoyan Xu, Xueyao (Eve) Yu, Yaxin Cheng, Yelizaveta Brus, Zhenyang Xu, Zhili Zeng. I hope you will forgive me if I have missed anyone's name—this journey has been long, and I have been fortunate to meet so many wonderful people along the way. Working with all of you has made this journey both enjoyable and fulfilling.

I extend my heartfelt gratitude to Dr. Mahmoud Alfadel for all the help and guidance he provided throughout my journey as our postdoc. While I was not lucky enough to work with him during my journey, I still benefited from his advice and experience.

I would also like to give a shoutout to my office roommates during my PhD: Partha Chakraborty, Noble Saji Mathews, and Aniruddhan Murali, with a special mention to Yiwen Dong from the room next to us. No one could ask for better roommates—I guess there is a reason our room was famous for the sound of laughter that constantly echoed from it.

My deepest thanks go to my parents and older brother. Your unwavering support and encouragement have been my foundation. I could not have done this without you.

And finally, to my wife—words cannot express how grateful I am to have you by my side. You have been my rock through the ups and downs, cheering me on through rejections and celebrating the successes. I am truly blessed to have you in my life.

Thank you all for being part of this journey.

Dedication

This is dedicated to my soon-to-be-born son, who has already brought immense joy into my life; to the love of my life, whose unwavering trust and support have been my strength; to my parents, whose encouragement made this thesis possible; and to my older brother, who has been my guide and mentor since childhood.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

ACR Automatic Code Review 2, 3, 5, 6, 14, 15, 17, 21, 79, 83, 108, 110, 111, 113

API *Application Programming Interface* 24, 98, 104

AST Abstract Syntax Tree 20

CRR Code Reviewer Recommendation xvi, 1–7, 9, 11, 13, 14, 17–20, 24, 26–29, 32–42, 44–54, 78, 79, 81, 108, 110–114

CSR Changeset Safety Ratio xvii, 55–57, 61, 65–72, 74–79

DF *Data Filter* 85

DWLR Developers' Work Load Ratio 49

FaR Files at Risk of turnover xvii, 56, 65, 67–72, 76, 78

JIT Just-In-Time 19–21

LLM Large Language Model xvii, 5, 6, 14, 23, 81–83, 85–92, 96, 97, 100–108, 112, 113

LTC *Long Term Contributors* 33

MCR Modern Code Review xvi, 10, 11, 13

MR Merge Request 9, 12, 82, 84, 85, 88, 89, 96

PR Pull Request xix, 9, 12–14, 17–19, 21, 31, 35, 46, 49, 54, 56–59, 61–66, 68–72, 74–78, 112

PRE Present Reviewers Expertise 49, 50

RAS *Recommender Adaptability Score* xix, 28, 39, 41

RCG Review Comment Generator xvii, xix, 3, 5–7, 21–24, 81–83, 85–108, 112, 114

RQ *Research Question* 27, 28, 87

SRR Staleness Reduction Ratio xx, 48, 49, 143

SVM Support Vector Machine 17

VCS Version Control System 12, 13, 17, 18, 110

Chapter 1

Introduction

Code review is a well-accepted practice and a crucial part of software quality assurance [6, 105], even though it is time-consuming and expensive [77]. Unfortunately, it is usually the reviewers who are pressured into delivering quick reviews to keep up with business needs [13]. To address this demand, approaches have been proposed to automate parts of this process [143, 162, 183]. These approaches aim to streamline the review workflow [22], reduce the time required for each review [90] and maintain high software quality [3].

Despite all the efforts to automate the review process, a human touch is still necessary to ensure the reliability of reviews [160]. While automation can handle routine checks, the complexity of code changes often requires human judgment since the review performance of these models is still sub-optimal [99]. This blend of automated assistance and human oversight helps to maintain a balance between speed and thoroughness in code reviews [61, 160, 168].

Automation is a pillar of the review process that aims to minimize the submit-to-review time and efforts of reviewers by reducing the reviewer responsibility [22, 160]. Nowadays, automation can recommend who should review the changeset [183], suggest where code changes may need to be revised [90] , and generate review comments [61], thereby allowing reviewers to focus on other issues. This targeted assistance can lead to more efficient and effective code reviews [22].

An effective code review automation approach, such as Code Reviewer Recommendation (CRR) or review comment generation, should consider aspects of the changeset under review and recommend accordingly [69]. Recommending inappropriate reviewers or generating irrelevant comments could potentially slow down the review process, leading

to practitioners abandoning the recommendation tools in the long run [185]. Hence, it is crucial for these tools to accurately understand and address the context of the changes [80].

1.1 Problem Statement

While code review automation can boost the code review process, inadvertently, it imposes additional challenges on software development. We define challenges as the issues and problems encountered when interacting with automated review suggestion tools, which have been overlooked or ignored to either simplify the problem or avoid adding complexity during development. We describe these challenges as practical because they are not typically considered in standard evaluations of review suggestions, yet they exist in real-world scenarios and may undermine the effectiveness of these tools:

Thesis Statement

Practical challenges in code review processes diminish the usefulness of code review automation. A multi-faceted approach to address issues like reviewer staleness and the bug-proneness of changesets will improve the interplay between human and automation, thereby enhancing automation outcomes.

In this statement, the enhancements refer to improvements in both the quality aspects of code review, such as more effective detection of bugs and code smells and a faster review process, as well as the collaborative aspects, including more constructive discussion threads with reduced toxicity. Indeed, the extent to which different code review automation approaches impact projects varies, i.e., some automation approaches are more sensitive to these challenges. Therefore, in this thesis, we aim to empirically study the influence of some of these challenges on existing CRR systems as one type of automation approach and their relevance considering the current state of Automatic Code Review (ACR) process. Our goal is to enable practitioners to make more data-grounded choices when they decide to use these automation approaches and help them adjust these tools to minimize the unintended negative impact depending on the attributes of their projects while giving researchers and tool developers directions to focus their efforts.

1.2 Thesis Overview

We now provide the scope of this thesis. Figure 1.1 highlights the overview of this thesis. First, we start by providing the necessary background information:

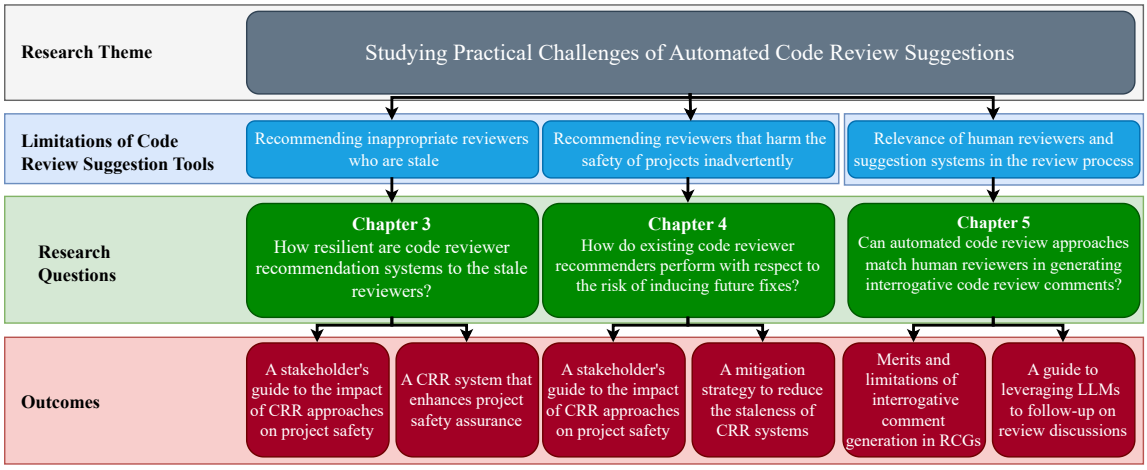


Figure 1.1: Scope and overview of this thesis

Chapter 2: *Background and Definitions*
In this section, we provide the reader with some background information.

Chapter 3: *Related Work*
We review the prior studies related to code review automation approaches, especially the existing research on CRRs systems, ACR, and Review Comment Generator (RCG)s to position this thesis with respect to prior research.

Then, we focus on the main body of the thesis. Our attention centers around three potential limitations of code review automation approaches (blue boxes). For each of these challenges, we seek to answer research questions (green boxes) that lead to promising prospective outcomes (red boxes). We format our studies in the chapters described below.

1.2.1 Limitations of Code Reviewer Recommendation Systems

Researchers and tool builders often simplify the complex scenarios of the real world to find a minimum viable solution. While these assumptions are necessary in early steps, when

automation tools are developed based on these assumptions, their misalignment could hinder the usefulness of these systems and render them ineffective [32, 80, 185].

These assumptions can happen in various stages of the process, ranging from as early as considering the available options to long after the process is finished when practitioners face the negative unintended impact of accepting these suggestions of these automation approaches. While most research on CRR systems are focused on positive aspects of these approaches [9, 22, 183], recent empirical studies brought up some of these misalignments act as the barrier for developer adaption [80, 185]. Therefore, it is crucial to identify and mitigate these challenges in automation systems.

In this thesis, we empirically study the extent of two of such challenges in CRR systems as an examplar of the code review automation approach:

Chapter 4: *Recommending inappropriate reviewers who are stale*
Recommending reviewers who are no longer contributing to the project, i.e. stale reviewers, can defeat the purpose of using CRR systems and increase the review process. Moreover, frequent occurrences and suggesting those who are long gone could even erode the developers' trust in the recommendation tool, rendering them useless [185]. In this chapter, we visit the possible impact of such recommendations and, using an empirical study, their potential severity on the review process by simulating the review process of three projects with more than 5.8K contributors.

Chapter 5: *Recommending reviewers that harm project's safety inadvertently*
In this thesis, we refer to safety from the perspective of the users of software. Thus, safety is threatened by the presence of defects and is improved by preventing defects. Proposed code changesets vary drastically concerning attributes such as complexity and bug-proneness [59]. Recommending appropriate reviewers to review code changes without considering this variation can adversely affect the project in different aspects. For instance, previous studies have shown that constantly recommending experts could worsen the bus factor [131] by preventing concentrating the knowledge distribution among these reviewers [108]. In this chapter, we explore how CRR systems can harm project safety by empirically studying the performance of seven CRR systems concerning bug-proneness of the studied projects if they accept CRRs from these systems.

1.2.2 Relevance of Code Reviewer Recommendation Systems

While CRRs aim to help reviewers boost the review process, a recent field of research has emerged to automate the review process [162]. Indeed, achieving a fully automated code review process could render the recommendations useless if these models can fully replace the human reviewers [160].

Researchers proposed that ACR process is comprised of four major tasks which should be performed to achieve full autonomy in code review: (1) code quality estimation [90], which would indicate which parts of the code need to be revised, (2) revising the code before review [94, 154], or (3) after review with review comments [21, 161, 162, 167, 178], to align it with the code with the expected quality, and (4) code comment generation [89, 90, 162] which generates comments given the code change.

In this part of this thesis, we focus our effort on investigating whether CRR systems are still applicable, considering the state-of-the-art RCGs, which could be a crucial part of the transitioning to ACR process:

Chapter 6: *Relevance of human reviewers and suggestion systems in the review process*
More often than not, interrogative code review comments lead to review discussion threads, which not only act as the documentation for the related information to the change [7] but can also change the direction of a project or define future tasks [169, 182]. Furthermore, the proposed set of ACR approaches does not include the interaction of RCGs with code change authors while they generate interrogative comments. This behaviour potentially stifles such favourable conversations during code review. Thus, in this chapter, we investigate the interrogative comments generated by six RCGs; three of them were trained to generate code comments given a change (task-specific models), and the other three are Large Language Model (LLM)-based models prompted to review the code (LLM-based). We explore these review comments concerning their comment mood, i.e. whether they are interrogative or declarative. We mine 172,919 comment instances along with their corresponding code changes from Gerrit project[1] and generated ∼1.2M comment instances using these RCGs. We study these comments quantitatively and qualitatively to determine whether RCGs pose questions similar to those asked by humans and to understand the nature of these questions. Furthermore, leveraging the capabilities of LLMs, we propose a new task for ACR: determining whether a code review comment has been resolved and following

[1] https://gerrit-review.googlesource.com

up if it has not. This task closes the loop of ACR process and helps improve the current flow of existing tools.

1.3 Thesis Contributions

This thesis demonstrates that:

1. Although the recency of contribution can diminish the risk of recommending stale reviewers, the multi-faceted behaviour of CRR systems can overshadow their impact if it is not separately considered (Chapter 4).

2. CRR systems may risk project safety by recommending less experienced reviewers due to their multi-faceted behaviour. While this tactic can benefit projects when code changes are not risky, when dealing with bug-prone changesets, it would reduce the project safety (Chapter 5).

3. While there is no silver bullet to resolve these practical challenges, enabling practitioners to adjust the provided CRRs with adaptable and yet simple configurations can help practitioners adopt these tools. Therefore, our mitigation strategies include a setting to adjust depending on the project state, as well as guidance on the potential side-effects of changing these configurations (Chapter 4 and Chapter 5).

4. Our proposed techniques for mitigating the issue with stale reviewers and project safety can combine with existing CRR systems to improve the staleness (Chapter 4) and safety (Chapter 5) of their recommendations.

5. Neither task-specific nor LLM-based RCGs can replace human reviewers in asking questions at the current stage of development (Chapter 6).

6. RCGs differ from humans when reviewing a code change, especially when the code change is discussion-inducing. This difference is in terms of both the quantity and quality of the questions. For instance, while human reviewers use interrogative questions to suggest a solution when pointing out a problem, the RCGs do it noticeably less frequently in the sample dataset. Instead, they asked more often than humans about the purpose of the code change (Chapter 6).

7. While LLMs' performance on discussion thread resolution shows promising results when they are required to follow up on the thread and respond or answer to the author, they show poor performance on most of the comment types (Chapter 6).

1.4 Thesis Organization

The remainder of this thesis is organized as follows. First, in Chapter 2, we will cover the necessary definition and lay out the ground for the body of this thesis by defining the key terms. Then, Chapter 3 presents the related research and positions this thesis relative to them. Chapter 4 is the first part of the main body of this thesis, where we explore the staleness of the CRRs systems and how to improve it. We then explore another practical challenge in Chapter 5 by reviewing the negative impact of the reviewer recommendation on the project safety. Then, we investigate the relevance of using code review suggestion tools by exploring the comment moods generated by the state-of-the-art RCGs in Chapter 6. Finally, we conclude in Chapter 7 and outline potential future research directions.

Part I

Preliminaries

Chapter 2

Background and Definitions

In this chapter, we first provide an overview of the code review process in its modern form. Then, we describe the position of CRRs systems as one form of code review suggestion tools that we focus on in this thesis.

2.1 Code Review Terms

To lay the ground for the rest of this thesis, we define key terms related to the code review process in this section.

2.1.1 Code Changeset

Nowadays, one part of the software development process is to develop new features and fix bugs by modifying project files to meet business demands [133, 158]. We define a *code changeset*, a.k.a *code change*, as a set of project files, such as media and code, changed to fulfil an objective like a new feature or a bug fix.

2.1.2 Pull Request

Once a code changeset is completed to serve its purpose (such as fixing a bug), one of the change contributors, known as the *submitter*, sends a request for the change to be merged into the target branch [186]. This request is called a Pull Request (PR) or Merge Request (MR) and should be approved by maintainers with the merge privilege, a.k.a. *merger* [186].

2.2 Modern Code Review Process

The Code Review process has been an integral part of the software development life cycle for a long time [40]. The main goal of this process is to ensure the quality of the proposed changesets before they are merged. Despite existing for a long time, the code review process has gone through considerable changes, transforming from a formal code inspection practice [40] to a light-weight tool-based process that is informal [6, 7]. The latter form of code review process is called Modern Code Review (MCR), which we refer to when discussing code review in this thesis. Normally, the submitter of the code changeset asks other contributors to check the health of the change before merging it when the proposed code changeset is submitted. The review process is mainly to assure the mergers that the code changeset is safe concerning aspects such as quality and following the code styles by acquiring confirmation from other contributors [51]. This practice of reviewing code changes, which is asynchronous and happens in an informal setting via tools such as Gerrit[1], GitHub[2], and Phabricator[3], is known as *Code Review* [7, 138]. In short, these tools enable contributors to submit their proposed changes to be reviewed by the project maintainers and, once approved, become a part of the repository. The developers who check the health and safety of a changeset are referred to as *code reviewers* and usually could be authors and submitters of other code changes in the same project [42].

2.2.1 Overview of Modern Code Review

Figure 2.1 provides an overview of MCR in the available platforms. Below, we describe each step of this process:

[1] https://www.gerritcodereview.com/
[2] http://github.com/
[3] https://www.phacility.com/

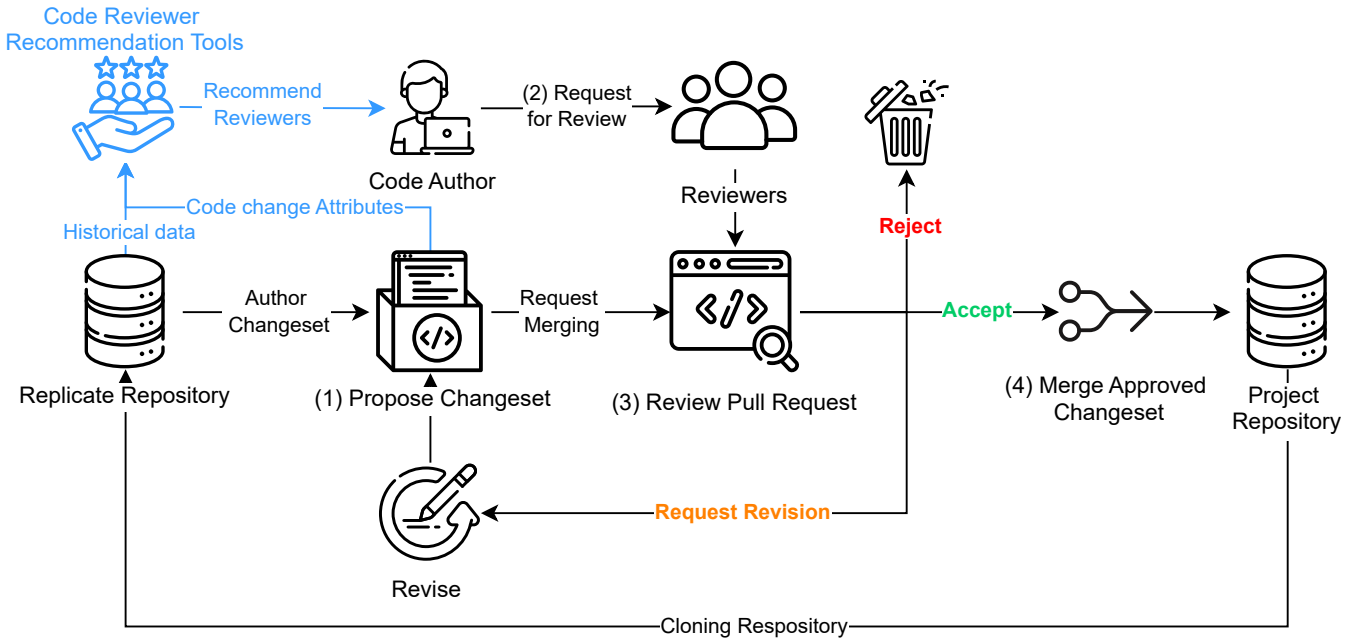


Figure 2.1: The overview of the Modern Code Review (MCR) process (in black) and the role of CRR tools (in blue) in this process.

1. **Code Changeset Preparation:** When a contributor decides to propose a change to the code to fix a bug or implement a feature, they usually start by replicating the project using a form of Version Control System (VCS). Then, they make a series of changes to existing code to accomplish the desired task, i.e. code changeset. Once their changeset is prepared, they propose this changeset with some explanation in the form of a request, which is usually called with names such as Pull Request (PR) or Merge Request (MR) in code review tools.

2. **Requesting Reviews:** In order for the code changeset to be approved, the Author should find a suitable candidate who has enough time to review the changeset and is willing to carefully inspect the code for possible issues such as defects [103], inconsistencies [75], or even the alignment of the proposed feature with the project goal [169, 182]. Finding an available appropriate reviewer has shown to be effective in the project quality [6, 106] and due to factors such as having high workload of the requested reviewers, it is a non-trivial task [78].

3. **Reviewing Pull Request:** Once the invited developers accept the review invitation, they examine the code for possible issues [7]. While different projects have different review processes and tools to perform them, they generally consist of a web-based UI that reviewers can use to scan the changes. The review platform categorizes the changes based on their location and groups them into smaller sets called code hunks for easier inspection. At this stage, reviewers communicate their concerns to the authors and ask for a revision of the code if necessary [6]. This iterative process is normally done asynchronously within the review platforms, such as GitHub. Not only does this process improve the quality of the code change and prevent bugs from entering the production, but some of the discussion threads that happen in this stage would act as the documentation for the project in future and may even lead to the creation of more task in future [169, 182].

4. **Review Resolution:** Each PR may be subject to multiple iterations of discussions and will be either rejected or accepted, which means the change will be discarded or become a part of the future version, respectively. While this decision is based on review discussions and code revisions, the reviewers are the ones who vote on what should be the destiny of the submitted PR. Sometimes, contributions are stalled in the review resolution step for reasons such as having multiple PRs sent simultaneously. Researchers previously investigated the effect of this delay [49, 50, 179], the approaches to prioritize the concurrent PRs [5, 165], as well as factors that contribute to time and outcome of PRs both in open source [72, 86] and industry [79].

2.3 Code Review Suggestion tools

Despite the optimization and heavily using tools to ease the review process [7], code review is still time-consuming [106], which makes it a great candidate for optimization. Code review suggestion tools, such as review comment recommendation [183] or file review ordering [8, 45], have been proposed to boost to help authors and reviewers with this process. Below, we elaborate more on CRR tools, which are the main focus of this thesis.

2.3.1 Code Reviewer Recommendation Systems

In large projects with hundreds of developers, finding the right reviewer for a PR is hard [152]. The issue can be exacerbated as some potential reviewers may be busy while the submitter is oblivious to their schedule. Thus, their review request may get rejected and elongate the review process of the PR [137]. To alleviate this problem, *Code Reviewer Recommendation*, or CRR for short, has been introduced. The main objective of this set of tools is to suggest the most appropriate reviewers for a PR. The CRR's merit criteria vastly vary, including measures such as reviewers' expertise [152], increase team awareness [6], balance contributors' workload [2], and familiarity with the change [149].

2.3.2 Overview of the Role of Code Reviewer Recommendation Systems in the Modern Code Review Process

Figure 2.1 demonstrates how code reviewer recommendation tools are situated concerning their role in MCR in blue. These tools use the historical data from VCS, the information about the change, and suggest a list of recommended reviewers for the change. The authors either accept these changes and invite the suggested reviewers to review them or discard these recommendations.

2.3.3 Evaluation Measures of Code Reviewer Recommendation Systems

Conventionally, past reviews and their actual reviewers were considered the benchmark. Still, researchers argued this evaluation method is potentially biased and problematic and brings little value [32, 47, 80]. A simulation evaluation technique has been proposed to

determine the merit of the CRR suggestions concerning the measures of interest like reviewer expertise and workload. Mirsaeedi and Rigby [109] set forth the following evaluation measures in their research:

- Core Developers' Workload: Number of code reviews assigned to top ten contributors with the highest review.
- Expertise: The expertise and familiarity of a reviewer with the files changed in a PR.
- Files at Risk: Number of files with less than two knowledgeable active contributors. A contributor is considered knowledgeable if they committed or reviewed the file.

These metrics are measured throughout the time and indicate how one CRR performs. This evaluation approach enables the CRRs to excel the assigned reviewers in reality and provide the researchers with a more fair process for assessing the performance of their tools.

2.4 Automatic Code Review Process

Automatic Code Review (ACR) refers to the efforts of tool builders to automate the review process by breaking down the review into smaller tasks and automating them. Figure 2.2 illustrated the order of these sub-tasks from previous studies.

While fully automating the code review process is the ultimate goal of this decomposition, unfortunately, we are still far from achieving it. Using new technologies such as LLMs helps tool builders to get closer to this goal, and as researchers, we hope that the result of this thesis and the new proposed task in Chapter 6 could pave the way toward this goal. Below, we briefly explain each of these tasks.

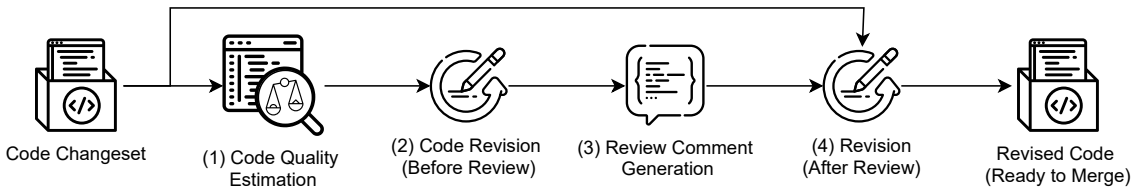


Figure 2.2: The breakdown of the automatic code review process tasks demonstrated in the expected order in an ideal flow.

2.4.1 Code Quality Estimation

Once a code review is submitted, this step would assess the quality of the code changeset by identifying potential issues and areas needing improvement in the code piece, a.k.a. code hunk, or code lines. In this step, for each piece of code changeset, the model should decide whether the code quality is up to the standard [90].

2.4.2 Code Revision (Before Review)

Once the parts of the code with quality issues are marked, this step modifies those parts of the code to address quality issues identified during the estimation phase. This task ensures the code meets the required standards before being reviewed and reduces the workload on reviewers by preemptively fixing common issues and enhancing code quality [21, 162].

2.4.3 Review Comment Generation

Once the new revision of the code is ready for review, this step is responsible for automatically generating comments on code changes, highlighting potential issues, and suggesting improvements. Usually, these comments are generated for code pieces that have been marked as low-quality in the first step [90]. This task uses natural language processing and machine learning to provide relevant, context-aware feedback that assists reviewers in their evaluation [89, 99].

2.4.4 Code Revision (After Review)

The last task of ACR is revising the code given both the commented code and the related code change based on the generated review comments to finalize the code for integration. This revision before the review would lower the cognitive load of the authors and prevent the tax of switching contexts to fix the comments issues raised by the reviewers. This task involves refining the code according to the feedback received, ensuring all identified issues are addressed, and preparing the code for deployment [94, 154].

2.5 Chapter Summary

In this chapter, we provided an overview of the modern code review and how code reviewer recommendations help with the process. We also defined some of the key terms we use in this thesis. Knowing this background information about the key concepts and terms used in this thesis, we move on to the next chapter, explaining the related studies to demonstrate the gap that we are trying to bridge in this thesis.

Chapter 3

Related Work

In this thesis, our focus is to study the automatic code suggestion approaches concerning their staleness and safety of the recommendation, as well as their relevance considering the emergence of Automatic Code Review (ACR). Therefore, there exist four groups of related studies that we should review in this chapter: (1) Reviewer Recommendations, (2) Defect Prediction, (3) Developer Turnover, and (4) Automatic Code Review. We review the related studies about Code Reviewer Recommendation (CRR) systems to understand related works better. Then, we review the studies related to these challenges: defect prediction, developer turnover, and diversity and inclusion. Finally, we conclude the chapter by exploring studies related to automatic code review.

3.1 Reviewer Recommendation

With the spread of Version Control System (VCS), and especially Git, CRR systems started to become popular [9, 30, 177, 180]. The main duty of a reviewer recommendation is to suggest the most suitable reviewers for reviewing PR. Reviewer recommenders often leverage historical data to make recommendations [177, 183]. Other approaches aim to optimize characteristics, such as workload balance [2, 130] or distributing knowledge [109].

For instance, the *CoreDevRec*, a CRR, developed by Jiang *et al.*, uses file paths as the input of the mode and recommends the most familiar reviewers with the Pull Request (PR) [65]. It uses Support Vector Machine (SVM) at its core to make the optimal suggestions and mostly suggests *core developers* as they have the most familiarity with the source code. These suggestions lead to an unwanted higher workload for these contributors.

Similarly, Thongtanunam *et al.* proposed REVFINDER that leverages prior reviews and the similarity of file paths to recommend code reviewers [155].

Zanjani *et al.* proposed *cHRev* recommender [183], which relies on the VCS data to make recommendations. When *cHRev* ranks developers as potential candidates for a code change, it considers the developer's expertise and the recency of the contribution from previous reviews as well as their portion of the contribution.

Chouchen *et al.* [25] devised a multi-objective search-based recommender named *WhoReview*. Similar to previous works, the candidates' expertise and their workload are considered when recommending a suitable reviewer for a code change. Another multi-objective CRR system that considers reviewer-author connection and reviewers' workload is *WLRREC* [2].

Machine learning techniques have also been tested in this area. Strand *et al.* [148] implemented a context-aware machine-learning based CRR, called *Carrot*, based on the LightFM[1] algorithm [83]. They used Gerrit as the training source and tested the CRR in an industrial setting. Then, surveyed the reviewers about its performance. The survey results indicate that more than 50 percent of participants stated that although it helps assign non-obvious reviewers to the PR, it does not affect the lead time for code review.

Regardless of the optimization method, when a new PR is created, the recommender ranks potential candidates or a set of candidates based on the score that its objective function has calculated. Recently, some studies have explored a change in perspective of the goal of the reviewer recommendation process. Kovalenko *et al.* [80] observed that developers are often aware of the top recommendations of CRR approaches, suggesting that other goals, such as workload balancing, might be more appropriate. Gauthier *et al.* [47] found that history-based evaluations of reviewer recommenders are often more pessimistic than optimistic since the proposed reviewers who did not perform a review (i.e., incorrect recommendations) often reported high comfort levels with those review tasks. Mirsaeedi and Rigby [109] proposed Sofia, a multi-objective recommendation system that tries to maximize reviewer expertise and minimize the risk of turnover-based knowledge loss, as well as the workload on the core development team.

Researchers also studied different aspects of the code review such as improving datasets accuracy [151], the notion of security in code review [19, 20], and code review smells [31]. In this thesis, our focus is the evaluation of the performance CRRs and explore approaches to mitigate their misalignments with real-world expectations. Despite the existence of various systems, the mutual impact of their recommendation on real-world phenomena, such as safety or knowledge turnover, still needs to be studied.

[1] https://github.com/lyst/lightfm

3.2 Developer Turnover

Prior research on developer turnover[12, 44, 63] has investigated its impact on software projects. For example, Ton and Huckman [156] investigated the impact of knowledge-intensive employee turnover on operational performance. Mockus [111] studied the effect of developer turnover on software defects. They observed that the departure of developers was associated with an increase in software defect rate. Lin *et al.* [92] explored turnover from an individual-centric perspective, studying contributors of five projects and their correlation with activities and retention. Rigby *et al.* [132] used a Knowledge-at-Risk (KaR) measure to quantify how susceptible industrial and open-source systems are to turnover-induced knowledge loss. Nassif and Robillard [115] replicated and extended the concept to seven other projects and noticed a similar knowledge loss probability distribution among all projects.

In addition, research has explored the impact of developer turnover, proposing identification methods based on behaviour. Avelino *et al.* [4] examined core developer turnover's effect on open-source projects, Miller *et al.* [107] studied reasons for open-source contributor disengagement, and Bao *et al.* [11] created a model to predict long-term GitHub contributors. Qui*et al.* [122] looked into projects' ability to attract new contributors, while Robillard [134] analyzed how employee leaves impact software companies, highlighting the significant disruptions caused by sudden and temporary departures.

In this thesis, we aim to explore how developer turnover affects the CRR recommendations quality concerning the stale reviewers and whether we can mitigate this challenge. Suggesting a contributor who has already left the project may stall the review process, lead to confusion in large projects, and delay merging the new PR.

3.3 Defect Prediction

Defect prediction models help the stakeholders of a project focus their limited resources on bug-prone modules [114]. Practitioners have used defect prediction systems to find bugs in their early stages, reducing technical debt [110, 174] and the effort required to fix them. These Just-In-Time (JIT) models can also help teams identify buggy changes before merging into the repository [76].

Researchers proposed various approaches for identifying the defective code changesets; for instance, Hassan [57] devised a measure for the complexity of a code change called entropy. Analyzing six well-established projects, he observed a correlation between code

change entropy and its faults. In another study, Zimmermann *et al.* [190] proposed using a dependency graph to find the most influential units in the code and argued that these units are more likely to get defective.

Pornprasit and Tantithamthavorn [121] proposed *JITLine*, which identifies the risky commits and the lines causing the defect. Their evaluation of OpenStack and QT projects shows a better performance in terms of accuracy and time compared to previous CRRs.

Recently, with the rise of machine learning approaches, there is a tendency toward using them for bug prediction. Li *et al.* [88] proposed a Convolutional Neural Network-based framework to predict defects using Abstract Syntax Tree (AST) as its input. They tested their approach in seven projects and observed a 12 percentage point improvement compared to previous models.

Seml is another defect prediction model proposed by Liang *et al.* [91]. The authors argued that system features that were previously used to predict bugs are flawed. Therefore, they used word embedding and LSTM deep neural network to extract the features and predict the probability of the bug from a code change. In their evaluation of eight open-source projects, their framework outperforms three state-of-the-art defect prediction.

Le *et al.* [85] proposed a deep multi-task learning model, *DeepCVA*, that identifies the various code vulnerabilities in the code changeset simultaneously. The evaluation of the approach showed 38% to 59.8% higher Matthews Correlation Coefficient and 6.3 times improvement compared to other methods.

Zhu *et al.* [189] implemented an ensemble approach that leveraged the whale optimization algorithm (WOA) and another simulated annealing (SA) algorithm for feature selection, and a deep neural network model is trained to predict bugs in the code changes. They tested their methods over 20 software projects and observed that the feature selection improves the performance of their approach, aligned with similar studies [127].

These defect prediction models are often trained using historical data, and the model is then used to assess new code changes by estimating the likelihood that a given code changeset will induce a future fix (i.e., estimating the fix-inducing likelihood). As some of the studies have been mentioned above, there have been a plethora of contributions to defect prediction studies. However, we focus below on two lines of work that are most relevant, i.e., (1) approaches to more accurately identify fix-inducing changes and (2) proposed indicators of fix-inducing commits. JIT defect prediction models — like any prediction model — will only be as good as their training data. Since the true set of fix-inducing changes is not clearly labelled in historical software data, heuristic approaches are used to recover that signal.

One heuristic that is used to label buggy commits is the SZZ algorithm, which was proposed by Śliwerski*et al.* [145]. The SZZ algorithm first identifies bug-fixing commits by mining for keywords such as "fix" or "bug" in commit messages. Next, potential fix-inducing commits are associated with these fixes by tracing removed or modified lines to the commit(s) that introduced them. Finally, filters are applied to remove potential fix-inducing changes that are unlikely to have caused the bug (e.g., potentially fix-inducing commits that were recorded after the bug was created in the issue tracker). The SZZ algorithm has seen several revisions in the literature [29, 116]. Since improving SZZ is beyond the scope of this thesis, we use the off-the-shelf implementation of SZZ available in the Commit Guru tool [136].

The SZZ algorithm is being used to label the buggy changes, but the JIT models need various measures from code to get trained on a project. The set of indicators that are used to predict fix-inducing changes are derived from the change itself, historical tendencies of the modified areas of code, and characterization of the personnel involved with the change [114]. For example, Kamei *et al.* [68] used measures of the size, purpose, and diffusion of a change, as well as the historical tendencies of the modified modules and the experience of change authors to estimate the likelihood of a change to induce future fixes. Hoang *et al.* [60] and McIntosh *et al.* [104] expanded the set of measures to include review metrics such as iterations, number of reviewers, and comments. Pascarella *et al.* [118] added more detailed measures such as owner's contribution lines and change code scattering. In this thesis, we use the set of measures provided by Commit Guru to calculate the 13 metrics similar to Kamei *et al.* 's set of measures for various PR based on PR's Commits.

3.4 Automatic Code Review

Previous studies [90, 161] defined four primary tasks for Automatic Code Review (ACR): (1) code quality estimation [58, 87, 102]; (2) revising code with reviewer comment [94, 154], and (3) without the comments [21, 161, 162, 167, 178] the review; and (4) code review comment generation. Researchers have developed machine learning models tailored to each of these tasks, including code review comment generation. Indeed, Zhou *et al.* [188] compared three instances of Review Comment Generator (RCG) and three general models adapted for code review. They found that the CodeT5 [125] general model surpassed the best RCG in code revision by 13.4%–38.9% and the T5-review [162] RCG model was the best for code review comment generation. They noted the presence of interrogative comments among the outputs but did not study them in depth.

With the advent of LLMs, Sridhara *et al.* [146] explored using ChatGPT for vari-

ous software engineering tasks, including code review, finding that ChatGPT's responses aligned with human reviewers in only 4 out of 10 tested instances. They observed a divergence in ChatGPT responses from human reviewers due to a lack of context and code comprehension issues. Further, Guo *et al.* [54] examined ChatGPT's revision effectiveness and its strengths and weaknesses in post-review code refinement. Their findings indicate that despite higher operational costs, ChatGPT underperformed with respect to CodeReviewer [90] in one of two studied datasets; however, ChatGPT excelled in code refactoring. Rasheed *et al.* [128] studied the automation of code review using multi-agent LLM-based RCGs, focusing on code smell detection and optimization of code changes. Due to these promising observations, we include LLMs in our study for a comprehensive comparison.

3.5 Review Comment Generators

In this thesis, we investigate how RCGs generate interrogative comments, focusing on three state-of-the-art models (AUGER [89], CodeBert [43], and CodeReviewer [90]), as well as three llm-based RCGs (DBRX,[2] GPT-4,[3] and LLaMA2 [159]). Below, we briefly review the literature on comment generation and the RCGs under study.

Code review comment generation aims to emulate human reviewers and automatically generate review comments for a given code change. The goal is to minimize the workload for reviewers and the delay in the authors' receiving feedback. Review Bot [9], which produces code review comments derived from the outputs of various static analyzers, received approval on 93% of its generated comments. DeepCodeReviewer [55] leverages deep learning to recommend reviews, and CORE [144] uses an attentional Long Short-Term Memory (LSTM) model for automated reviews. CommentFinder [61] addresses the latency in deep learning methods using an information retrieval approach.

While traditional comment generation models largely depended on rule- or retrieval-based approaches, recent advances have shifted focus to task-specific RCG models [167]. This trend typically involves pre-trained models [89, 90, 93, 162] that combine natural and programming language processes to enhance the code review process. In recent work, Vijayvergiya *et al.* [168] developed and evaluated the AI-assisted code review tool AutoCommenter to help with code changes written in C++, Java, Python, and Go. AutoCommenter, built using the T5X model and fine-tuned, detects best practice violations and provides URL references for these violations. Their approach generates comments

[2] https://github.com/databricks/dbrx
[3] https://bit.ly/open-ai-gpt-4

for 68% of best practices commonly referenced by human reviewers, showing promise for AI-assisted code review tools.

With the emergence of Large Language Model (LLM), newer studies have also examined their utility in automating code review tasks. Lu *et al.* [99] fine-tuned LLaMA 6.7B for code review tasks. Despite LLaMA-Reviewer's higher resource demand for inference, it did not consistently outperform state-of-the-art task-specific RCGs like CodeReviewer [90]. Furthermore, Pornprasit and Tantithamthavorn [120] assessed GPT-3.5's capabilities for code review. Their results showed that state-of-the-art task-specific models still tended to outperform GPT-3.5. Given these findings and the high operational costs of LLMs, we choose to consider three high-performing [188] task-specific RCG and two types of LLM-based RCGs for each study.

RCG Models

Researchers have developed various RCGs. In this thesis, we study select six of these models for our experiments: AUGER, CodeBert, and CodeReviewer have been selected as task-specific RCGs based on their prior use [188] and strong performance as detailed above. GPT-4 Turbo[3] is chosen as an exemplar of enterprise models, known for its impressive performance across various tasks, albeit at higher costs. Due to this cost, we select DBRX-instruct[2] and LLaMA2-7B [159] as freely available alternative LLMs for our quantitative analysis. We describe these models below.

- **AUGER** is a comment generator that uses the pre-trained CodeTrans T5 model [38, 125], which was further fine-tuned on ∼10K code review instances from 11 Java projects [89]. AUGER thus can leverage its training data to provide review context. AUGER was assessed with a survey that revealed that 29% of developers found its generated comments useful [89].
- **CodeBert** has a transformer-based architecture [167] and is trained with Masked Language Modeling and replaced token detection using NL-PL pairs and unimodal code data [43]. This approach allows CodeBert to excel in tasks like code search and documentation generation. Like Zhou *et al.* [188], we use the pre-trained CodeBert model, fine-tuning it with ∼50K review records. Fine-tuning this model helps with the generation of the comments instead of code, leading to more understandable generated comments.
- **CodeReviewer** employs CodeT5 [167, 170] and further fine-tunes it on data in the form of `<comment, code hunk>` to process code diffs as input [90]. Compared to

AUGER, CodeReviewer focuses on understanding code changes and their relationship to review comments because its output explicitly highlights line additions and deletions. Indeed, CodeReviewer has been shown to perform comparatively well in terms of comment generation among RCGs [99].

- **DBRX** was introduced in March 2024 and outperforms established models like GPT-3.5, particularly in code-related tasks.[2] It is available in a basic version and an instruct version. The basic version offers robust performance across a broad range of applications, while the instruct version is optimized for scenarios requiring precise control, such as coding assistance. We use the instruct version of the model to achieve the best possible performance with code.

- **GPT-4 Turbo** introduced upgraded capabilities, including support for more diverse inputs such as images.[3] As a proprietary model, its architecture details are not publicly disclosed, and to interact with the model, developers should use OpenAI's *Application Programming Interface* (API).

- **LLaMA2** This model comes in 7, 13, and 70 billion parameters [159]. These models, trained on a dataset spanning 2 trillion tokens from January to July 2023, follow the standard transformer architecture with an auto-regressive model design.

3.6 Chapter Summary

This chapter reviewed studies on code review recommendation, developer turnover, defect prediction, and automated code review tasks, which are connected to the body chapters of this thesis. It also positioned our work within the broader research landscape. In the next chapter, we begin the main body of the thesis, presenting two empirical studies on the challenges of CRR systems and their potential impact.

Part II

Limitations of Code Reviewer Recommendation System

Chapter 4

Studying the Staleness of Code Reviewer Recommendation Systems

> **Note.** An earlier version of the work in this chapter appears in the EEE Transactions on Software Engineering (TSE) Journal [70].

4.1 Introduction

Code Reviewer Recommendation (CRR) approaches have been developed to suggest suitable reviewers for a changeset by ranking potential candidates. They evaluate contributors using their objective functions, taking into account factors such as experience [148], ownership [112], and developer interactions [126] and suggest the candidates with the highest scores to review the changeset. However, they also have unintended impacts on various aspects of the projects, such as the knowledge distribution[108]. Traditionally, CRR approaches are evaluated by applying them to historical data, and comparing recommended reviewer lists to those who performed the review; however, recent studies call this practice into question. For example, Kovalenko *et al.* [80] found that the top recommendations are often known to developers. Thus, when recommendation approaches are *considered correct* (i.e., they recommend the reviewer who performed the review), their recommendations are often obvious choices and of limited value. Moreover, prior work [48] found that recommended reviewers who did not perform the review would often have been appropriate assignees. Thus, when CRR approaches are *considered incorrect*, the implications are often unclear. This raises a question: when can researchers and tool builders be certain that

recommended reviewers are truly incorrect?

In this chapter, we study stale recommendations—a class of recommended reviewers that are certainly incorrect. We define a stale recommendation as a recommended reviewer who has stopped contributing to the project under analysis. Since these contributors cannot perform the review, they add no value to review recommendation lists. Indeed, the interviewees of Kovalenko *et al.* [80] point out that it is not uncommon for recommendation lists to include stale reviewers. Furthermore, Zhang *et al.* [185] found that 91.03% of the negative feedback they received from practitioners about the performance of a proprietary CRR system was about "irrelevant recommendations," where stale recommendations comprise 23.83% of this category. In contrast, other factors, such as a lack of prior participation in code review, only accounted for 8.97% of all the negative feedback. Aligned with prior work [80], the study further revealed that contributors frequently change their focus area or switch teams, potentially making them stale for their prior focus areas and development teams. These observations, coupled with the incontrovertible effect that stale recommendations have on the performance of CRR approaches (unlike other types of incorrect recommendations), highlight the considerable risks that stale recommendations pose to the quality of CRRs and provide an opportunity to better understand stale reviewer recommendations to mitigate the issue. Prior studies have explored the effect of stale reviewers through the lens of turnover-induced knowledge loss [39, 115, 129]; however, to the best of our knowledge, their characteristics and other potential effects on reviewer recommendation are yet to have been explored.

Using data from the Kubernetes, Rust, and Roslyn open-source projects, we study the prevalence of the stale recommendations that are produced by five reviewer recommendation approaches (LearnRec [109], RetentionRec [109], cHRev [183], Sofia [109], and WLRRec [2]). We find that on average, stale recommendations account for 12.02%, 8.33%, and 16.44% of incorrect recommendations that are produced by the cHRev [183], Sofia [109], and WLRRec [2] approaches per quarterly period, respectively, with medians of 10.28%, 6.63%, and 14.72%. Furthermore, when the number of recommendations to be produced is set to one, two, and three, CRR approaches produce at least one stale recommendation for up to 33.52%, 55.47%, and 69.18% of changesets, respectively, with medians of 6.34%, 15.01%, and 23.36%.

Since stale recommendations (1) represent a notable portion of incorrect recommendations, (2) can influence a considerable proportion of changesets, and (3) have clear implications (unlike other incorrect recommendations), we aim to characterize them and propose mitigation strategies. To do so, we address three *Research Question* (RQ):

RQ1 **Are code reviewer recommendation approaches resilient to stale recommendations?**
Motivation: For a CRR approach, it is desirable to be resilient to developer turnover, while also suggesting suitable reviewers. However, in reality, CRR approaches are susceptible to changes in the set of active reviewers. This RQ aims to explore the extent to which stale reviewers are prevalent in the recommendations produced by the studied CRR approaches.
Results: CRR approaches that consider the recency of contributions tend to be more robust to stale recommendations. For instance, RetentionRec suggests no stale recommendations in the studied periods, while all LearnRec [109] recommendations are stale in 80.39% of all the studied quarters. We establish the worst and best performers as benchmarks and assess the effectiveness of each approach relative to these benchmarks. We introduce the *Recommender Adaptability Score* (RAS) to estimate an approach's capacity to handle the volatility of active contributors (larger RAS values indicate greater resilience). We observe that cHRev, Sofia, and WLRRec have median RAS values of 91.87%, 94.27%, and 84.66%, respectively, indicating that WLRRec is the least resilient, whereas Sofia is the most resilient.

RQ2 **Distribution: How is staleness distributed among recommendations?**
Motivation: If staleness is concentrated among a few reviewers, targeting these specific individuals would be more effective than strategies identifying many departed reviewers. Thus, to guide future work, we study how staleness is distributed across personnel.
Results: Staleness is highly concentrated for all of the studied recommendation approaches across the studied projects. Indeed, in 15.31% of periods over various evaluation settings, the top-3 reviewers account for at least half of the staleness.

RQ3 **Duration: How long do stale recommendations linger on suggestion lists?**
Motivation: If most stale recommendations linger for a short period, recommendation approaches need to adapt to a dynamic list of reviewers who left the project. If, on the other hand, reviewers linger in stale recommendation lists for a long period, identifying a stable set of impactful candidates will have a larger effect. Thus, to guide future work, we study the extent to which stale recommendations linger.
Results: Stale recommendations can still be suggested up to 7.7 years after their departure, with a median time of 7 months and 21 days. While the lingering duration of top-3 reviewers increases over time, their proportion in stale recommendations reduces. Approaches that consider the recency of contributions (e.g., cHRev) reduce the number of stale recommendations, but are ineffective when other factors like

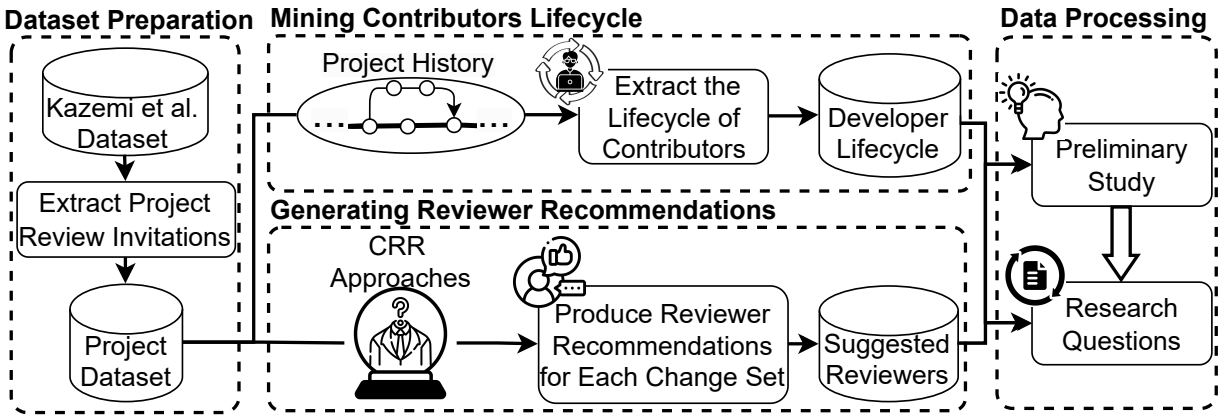


Figure 4.1: The simplified overall architecture of study data analysis.

experience dominate.

Our results suggest that a periodic pruning of the top stale reviewers may help existing approaches with staleness. In industrial settings, such a CRR system could integrate with personnel management systems to identify who left the company; however, team reorganization and internal movement of personnel may still present challenges [185]. To address those challenges, we propose a mitigation strategy to enhance CRR approaches with a separate time-based contribution recency filter to remove stale reviewers from the available developer pool, especially the top ones that have been lingering for a long time. This strategy diminishes stale recommendations by up to 92.16%, 92.39%, and 89.45% for cHRev, Sofia, and WLRRec, respectively. Future work could further improve this by improving the identification of the top reviewers and forecasting stale reviewers.

4.2 Study Design

Figure 4.1 gives an overview of our experiment design for this chapter. This section outlines the dataset preparation (Section 4.2.1), the studied CRR approaches (Section 4.2.4), and the data processing procedures (Section 4.2.5).

4.2.1 Dataset Preparation

We conduct our research using a dataset derived from active open-source projects. The dataset is sourced from prior work [69]. While the dataset includes the information required for recommending reviewers, it lacks the reviewers' invitations for the changesets, which we require for our study. Therefore, we augment the dataset by extracting the required reviewer data using the GitHub API.[1]

Studied Dataset: The dataset includes data from the Roslyn, Rust, and Kubernetes open-source projects. Rust is a popular high-level programming language with over 4.1K contributors. Roslyn provides tools for the analysis of C# and Visual Basic with 548 contributors and is backed by Microsoft.[2] Initially developed by Google, Kubernetes is now managed by over 3.5K contributors and aims to automate operational tasks for container management. The selection criteria for these project were to be active for over 4 years, have more than 10K changesets with review rate of more than 25 percent overall, and have more than 10K files. Further details on these projects can be found in Table 4.1.

4.2.2 Mining Contributors Lifecycle

This component is responsible for determining when a contributor joined and left a project. To this end, we query the contribution logs in the extracted dataset to identify each contributor's last contribution to the project, signifying when they ceased their involvement. To ensure accurate developer identification in the logs, after mining the data from GitHub we implement a cleaning and matching stage. This stage involves preprocessing the extracted user records, considering their names and emails, and removing special characters and diacritics. Moreover, we calculate the distance using the Damerau-Levenshtein algorithm [27] for string matching with a tolerance of 1 to accommodate minor user name and email variations. This helps with user identification and creates comprehensive profiles for developers. We retain this information for our analysis (see Section 4.2.5).

[1] https://docs.github.com/en/rest
[2] https://devblogs.microsoft.com/visualstudio/introducing-the-microsoft-roslyn-ctp/

Table 4.1: The details of the dataset used.

Name	Files	Reviewed Changesets	Developers	Review Invitations
Roslyn	12,313	8,646	469	1,546
Rust	12,472	17,499	2,720	128
Kubernetes	12,792	32,400	2,617	26,164

4.2.3 Key Terms

To enhance the clarity of our methodology, we explicitly define the key terms employed throughout this report. These definitions are used for identifying contributors and establishing the criteria for determining when a contributor is considered to have departed from a project.

Contribution: To identify potential reviewers from a project's developer pool, we consider those who had previous contributions to the project. Thus, we need to provide a crisp definition of contribution in this thesis. There are multiple ways in which individuals can contribute to the advancement of a project, including activities such as reviewing code and reporting bugs. Since end users can submit bug reports, we elect to concentrate on the contributions of code reviewers and changeset authors in this chapter. Furthermore, since the recommendation approaches only consider developers for reviewing changesets, we only consider those who reviewed or authored a changeset in the past.

Developer: Within the context of this chapter, a developer refers to individuals who have authored a portion of the code or participated in reviewing a Pull Request (PR). It is crucial to acknowledge that while they are qualified to review subsequent changes to their own code, not all are actively involved in the review process. The term *contributor* is thus synonymous with *developer*, as it aligns with the definition of contribution.

Reviewer: This term refers to those reviewing a changeset to ensure that new changes do not introduce bugs, fulfill the authors' intent, and adhere to the standards of the repository. Previous studies have shown that choosing the optimal reviewer impacts the quality of the review process [101, 138].

Stale Reviewer: The identification of contributors who have left a project has been explored in various studies using different thresholds from a contributor's latest contribution in periods of 30, 60, 180 days, and even a year [64, 92, 141]. A contributor is deemed stale at a given point in the project's history one day after they cease authoring or reviewing PRs and do not make any subsequent contributions in the project contribution history. We identify when recommendations become stale by first compiling the contributions of

developers from the project history and then determining their first and last contribution. Similar to prior works [44, 92], we classify those who have contributed within the last six months of the project's available history as available developers to ensure all stale reviewers have been inactive for a minimum of six months. Subsequently, we compare the generated CRRs for the studied approaches against the activity lifecycle of developers to determine which recommendations are stale.

In summary, we consider all the developers who have made contributions before the changeset submission as potential reviewers and then apply the CRR approach, using its objective function to prepare a set of reviewers and recommend that they review the changeset. Among these recommended reviewers, some of them may not have been actively engaged in the project development, i.e., stale reviewers.

4.2.4 Generating Reviewer Recommendations

To generate CRRs that align with the state of the project at the time of proposing each changeset, we conduct a simulation of the project's development over time. This simulation involved exclusively considering the data points available before the changeset was proposed. For every changeset, we identify the previous contributors and treat them as the pool of potential reviewers. Subsequently, we feed this data as the input of the CRR approach to reproduce the CRRs for the changeset, which are stored for further analysis.

Studied CRR Approaches: To investigate the quality of reviewer recommendations, we choose five CRR approaches with various recommendation styles. Below, we briefly describe each approach and its selection criteria. Since it is not feasible for one study to implement and evaluate all the available CRRs, we chose five approaches that cover different recommendation styles which are popular among the CRRs approaches [66, 148].

LearnRec [109] solely focuses on mitigating the risk of turnover-induced knowledge loss by promoting knowledge sharing among team members. It recommends contributors who are likely to learn the most from participating in the review of a changeset by estimating the familiarity of the candidate with the modified files. *LearnRec* ranks candidates in ascending order based on the complement of a heuristic, which estimates how much the candidates know about the modified files (i.e., $1 - \mathit{ReviewerKnows}$). Even though *LearnRec* is singular in its optimization focus and would not reasonably be deployed in production, we include it as a benchmark to which other CRR approaches can be compared. Our hypothesis is that this approach will exhibit the lowest resilience to stale recommendations because individuals who are making a one-time contribution to a project are typically ranked as those who stand to learn the most from a review [69, 109].

RetentionRec [109] suggests only *Long Term Contributors* (LTC). While the former approach, *LearnRec*, is an extreme to mitigate the risk of turnover knowledge-loss, the developers who benefit the most from reviewing code may have little to offer in terms of feedback to benefit the authors of changesets. Moreover, they are highly likely to be one-time contributors [69, 109]. As an extreme countermeasure, the RetentionRec approach ranks candidates in descending order according to their frequency and consistency of contribution. The *contribution ratio* measures the proportion of contributions made by a developer during a period, while the *consistency ratio* measures the proportion of sub-periods in which the developer was actively contributing to the project. As developers become more consistent or active, the *RetentionRec* approach is more likely to suggest them as reviewers. Due to the characteristics of the objective function, we expect this approach to exhibit the highest resilience to stale recommendations; however, this tends to overburden the core team since they have the highest frequency and consistency of contributions.

cHRev [183] ranks potential reviewers for a changeset based on their previous reviews and the recency of their contributions. To evaluate the suitability of developer D for reviewing file F, cHRev uses the *xFactor* measure, which is calculated as the sum of three terms: (1) the ratio of the number of review comments made by D on file F to the total number of review comments on F, (2) the ratio of the number of workdays that D commented on reviews of F to the total number of workdays for all reviewers of F, and (3) the inverse of the difference in days between the most recent day that D worked on F and the last date that F has changed, plus one. *CHRev* calculates the *xFactor* for files in the changeset for potential reviewers and recommend those with the highest *xFactor*. In this chapter, we consider *cHRev* as an example of a traditional CRR algorithm, which are approaches that seek to match review suggestions with historical review data [69].

Sofia [109] aims to balance multiple objectives, i.e., the knowledge distribution among active team members and the expertise of reviewers assigned to tasks. Suppose the number of knowledgeable developers in the project for any file in the changeset R is N. When N is greater than a risk tolerance threshold ($N = 2$ in the original paper [109]), *Sofia* uses *cHRev* to rank recommendations by their expertise. Conversely, when N is below the risk tolerance threshold (i.e., there are fewer than N developers in the project that have knowledge of the file) *Sofia* uses the combination of *RetentionRec* and *LearnRec* to rank LTC candidates who can learn the most by reviewing R.

WLRRec (WorkLoad-aware Reviewer Recommendation) [2] takes into account the workload of potential reviewers when ranking candidates for a changeset as well their social interactions. The idea is that if a reviewer is already very busy, they are less likely to agree to take on another task. When ranking candidates, *WLRRec* considers their past

rate of accepted review invitations (review participation rate), the assigned reviews that candidates still have pending (remaining reviews), and the expertise and experience that candidates have with respect to the code under review (ownership and experience). We study *WLRRec* because it is a state-of-the-art approach that does not place importance on the recency of the candidate reviewer contributions. This attribute is desirable to help us comprehend a broad range of CRR characteristics in this chapter.

4.2.5 Data Processing

In this component, we address the research questions outlined in Section 4.1. Our investigation begins with a preliminary assessment of the prevalence of stale recommendations in CRR systems. If noticeable prevalence is observed, we will proceed to conduct a more in-depth analysis of the data collected in the previous stage.

We rely on historical data from Git repositories to produce CRRs. Further information on the history of each project can be found in Table 4.1. The historical data from each studied project is stratified into quarterly (three-month) intervals and CRR performance is evaluated for each interval. This approach aligns with previous studies on knowledge turnover [109, 115, 132], which also chose quarterly intervals. The rationale behind this choice is that it provides a balance: quarterly intervals are long enough to capture trends and patterns effectively, yet not so long that crucial details are obscured. Additionally, smaller time-frames have shown to be more susceptible to extreme events when compared to their respective means [115].

To confirm that code review was consistently carried out, we focus on contiguous periods where more than 80% of integrated changesets were reviewed. Figure 4.2 shows the quarterly review rates for each project.

4.3 Preliminary Study

Prior studies have shown that developers complain about stale recommendations [80, 185]; however, the prevalence of and reasons for stale recommendations remain unexplored. Therefore, we conduct a preliminary study of stale recommendations in CRR systems.

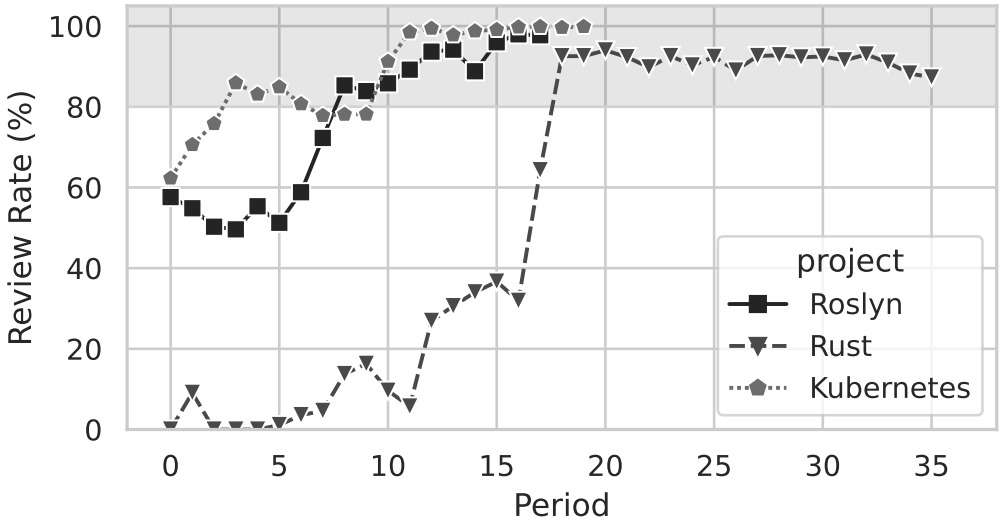


Figure 4.2: Quarterly review rates of Rust, Roslyn and Kubernetes projects.

4.3.1 Approach

To gauge the potential impact of stale recommendations, we study the rate at which incorrect recommendations are stale. We apply the studied CRR approaches to produce reviewer recommendations at several points in time. Then, we identify incorrect recommendations, i.e., recommended reviewers who did not review the code. Next, we measure the prevalence of stale recommendations in incorrect ones, and the ratio of all changesets (i.e., PRs) that have at least one stale recommendation since they can potentially be impacted by stale recommendations.

4.3.2 Results

Stale recommendations frequently account for a considerable proportion of incorrect recommendations with an average of 12.59% of incorrect recommendations for non-naïve approaches [69], i.e., CRR approaches that optimize the recommendation for multiple objectives such as cHRev, Sofia, and WLRRec. Specifically, the average proportion of stale recommendations to the incorrect ones over reviewer set sizes of one to three for LearnRec, RetentionRec, cHRev, Sofia, and WLRRec is 97.13%, 0%, 12.03%, 8.33%, and 16.44%,

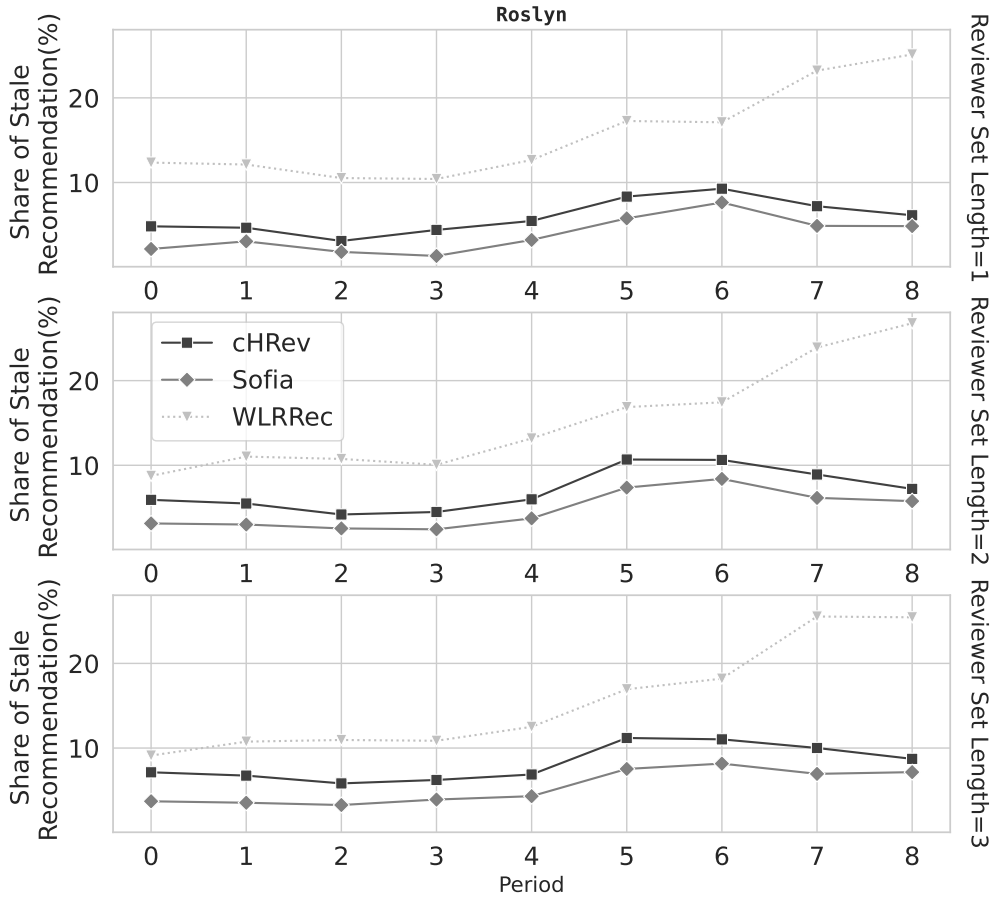


Figure 4.3: Share of stale recommendations over time for studied projects. Rows indicate variations for reviewer set sizes ranging from 1 to 3.

respectively, with the median share of 100%, 0%, 10.28%, 6.63%, and 14.73% across all the studied periods. These proportions indicate that for all studied approaches, except for RetentionRec, stale recommendations account for a non-negligible proportion of incorrect recommendations. By exclusively suggesting contributors who exhibit a higher likelihood of remaining engaged in the project, RetentionRec surpasses other existing CRR methods in terms of mitigating the issue of stale recommendations. However, RetentionRec's superiority in this aspect comes at the cost of imposing a significant workload on core developers, rendering it impractical [69, 109].

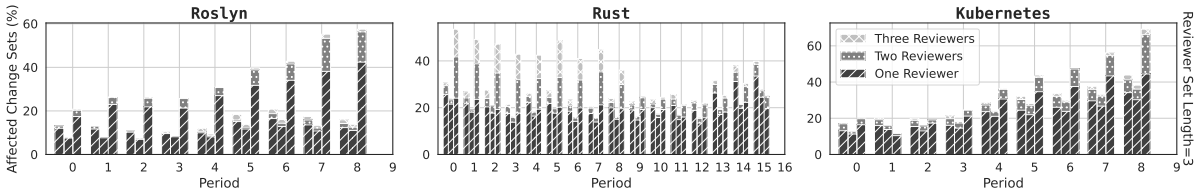


Figure 4.4: Prevalence of potentially impacted changesets by stale recommendations for cHRev (left), Sofia (middle), and WLRRec (right) for each period (percentage).

The clarity of the impact of stale recommendations compared to other types of incorrect recommendations warrants a closer inspection. Prior research indicates that the reviewers of a changeset are not necessarily the optimal choices [95], whereas those who are recommended but have not conducted the review often possess the necessary qualifications to conduct the review [48]. Hence, it is not straightforward to identify the truly incorrect recommendations among the produced reviewer recommendations. Stale recommendations, however, are unequivocally incorrect—they are unavailable to conduct the review. Furthermore, in specific periods (e.g. last periods of Kubernetes project), the proportion of stale recommendations becomes considerable, likely a factor that contributes to the negative feedback that was observed in previous studies [80, 185]. Therefore, mitigating stale recommendations directly enhances the performance of CRR approaches and improves the experience of teams that use them.

The size of the recommendation set has little influence on the proportion of stale recommendations, whereas the proportions vary substantially from one project to another. Figure 4.3 shows the proportion of incorrect recommendations that are stale over time for the three studied projects for reviewer set sizes from one to three. Each line shows the proportion of stale recommendations that are not influenced by the recommendation set size, whereas the project and CRR approach affect their proportion. While initial observations suggest reviewer set size does not influence the prevalence of stale reviewers, further analysis of potentially affected changesets indicates otherwise. Reviewer set size impacts the extent of affected changesets–a more important measure of stale reviewers' potential effect. Additionally, the figure indicates that project-specific factors, such as knowledge turnover rates, have a substantial impact on stale reviewer rates.

A considerable proportion of changesets has at least one stale recommendation. Our analysis reveals that up to 33.52%, 55.47%, and 69.18% of changesets include at least one stale recommendation when recommendation sets are of length 1, 2, and 3, respectively, with corresponding medians of 6.34%, 15.01%, and 23.36%. Figure 4.4 shows the

proportion of changesets that include stale recommendations (Y-axis) over time (X-axis) across the studied projects (horizontal grid) with recommendation lists of length 3. This figure shows that the share of influenced changesets by stale recommendations tends to grow over time in all settings, except in the Rust project where WLRRec is applied. This difference is due to Rust's lack of review invitation records. While this might occur in real-world projects for various reasons, such as using alternative communication channels for review requests or allowing reviewers to self-select changesets to review, we consider it an interesting application of WLRRec to such projects rather than a threat to the validity of WLRRec results for Rust. Nevertheless, this lack of information causes a divergence between Rust and other projects when using WLRRec, which takes into account the accepted review invitation rate. Since the accepted invitation data is not available on Rust's GitHub, WLRRec does not perform as well. However, over time, other factors in the WLRRec algorithm, such as Reviewing Experience and Familiarity, compensate for this limitation. The results for reviewer set lengths of one to three are shown in Figure A.2.

Stale recommendations represent a considerable and persistent issue within CRR systems, as evidenced by the median percentage of changesets containing at least one stale recommendation, which ranges from 6.34%, 15.01%, to 23.36% for reviewer set lengths of one, two, and three, respectively. This trend not only underscores their prevalence but also suggests an increasing tendency over time, highlighting the shortcomings of widely-used CRR systems with this regard. A deeper investigation is warranted to better characterize their occurrences to guide the development of mitigation approaches.

4.4 RQ1: The Prevalence of Stale Reviewers in Code Reviewer Recommendations

In this section, we study the prevalence of stale recommendations that are produced by our studied approaches.

4.4.1 Approach

We evaluate the recommendations for studied approaches over the quarterly periods with recommendation lists of lengths one, two, and three. For each period, we compute the share of stale recommendations over all the incorrect recommendations. To assess the quality

of studied approaches, we propose the *Recommender Adaptability Score* (RAS) measure, which gauges the approach's ability to respond to developer turnover and consider only active contributors:

$$RAS(CRR) = \frac{AUC(CRR_{LearnRec}) - AUC(CRR)}{AUC(CRR_{LearnRec}) - AUC(CRR_{RetentionRec})} \tag{4.1}$$

AUC refers to the Area Under the Curve that plots the proportion of stale recommendations against time (i.e., studied periods). We expect LearnRec and RetentionRec to produce the worst and best performance (i.e., the largest and smallest possible AUC), respectively. Therefore, the RAS calculates how much closer the CRR is to the optimal (best) performance than to the worst performance. The RAS ranges between 0–1; higher values indicate better performance.

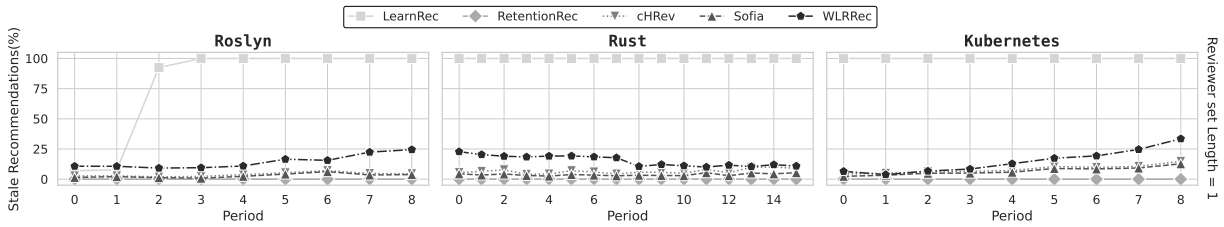


Figure 4.5: The proportions of stale to all recommendations (y-axis). The period numbers are normalized, with zero representing the oldest period.

4.4.2 Results

Figure 4.5 shows the proportion of stale recommendations (Y axis) over time (X axis) across the studied projects (Horizontal grid) for reviewer set length of one. The results

Table 4.2: Measured *Recommender Adaptability Score* (RAS) values for each setting. A higher RAS indicates better adaptability to developer turnover.

Project	Roslyn			Rust			Kubernetes		
Approach \ Reviewer set length	1	2	3	1	2	3	1	2	3
cHRev	0.9521	0.9448	0.9349	0.9356	0.9065	0.8902	0.9188	0.8945	0.8783
Sofia	0.9647	0.9595	0.9540	0.9632	0.9428	0.9313	0.9336	0.9135	0.8996
WLRRec	0.8280	0.8442	0.8466	0.8220	0.8041	0.8588	0.8757	0.8560	0.8676

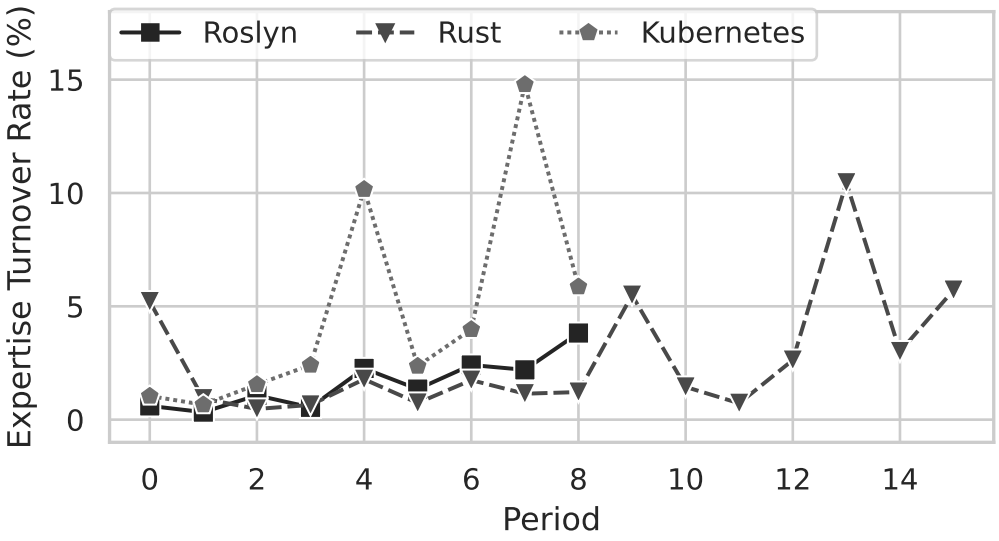


Figure 4.6: Developer expertise turnover rate for the studied periods over time. We consider the first studied period to be zero in all projects.

when the length of the recommendation list is set to two and three are shown in Figure A.1 in Appendix A.

Stale recommendations account for up to 33.33% of all suggested reviewers with a median share of 8.3% of all of the recommendations. Figure 4.5 largely confirms previously reported developer complaints [80], i.e., that CRR approaches often suggest stale reviewers. CRR approaches, configuration settings, and periods have a considerable effect since Figure 4.5 also shows that the proportion of stale recommendations can drop to as low as 0.33%; however, even a minimal presence of stale recommendations—especially those involving stale reviewers who have long since departed from the project—can erode developers' trust in CRR systems, thereby affecting their usability. Additionally, reducing the incidences of stale recommendations, even if they constitute a small proportion of the recommendations at times, would certainly enhance the rate of correct recommendations, unlike other types of recommendations which may have ambiguous implications [32, 48].

Considering the recency of candidate contributions enhances the quality of CRRs. We find approaches that consider the recency of contributions outperform WLRRec, a CRR approach that does not consider recency. Figure 4.5 indicates that RetentionRec is the

best CRR approach with no stale recommendations, while LearnRec is the worst, providing stale recommendations in 80.39% of all the studied periods. The poor performance of LearnRec in recommending active contributors can be attributed to its prioritization of contributors with the greatest learning opportunity. This naïve prioritization can lead to sizable knowledge loss [69, 109]. Meanwhile, RetentionRec prioritizes the most active contributors and is thus least prone to making stale recommendations; however, RetentionRec tends to overburden core developers (by design).

The performance of Sofia and cHRev—CRR approaches intended for actual deployment—falls in between these two extremes, with Sofia exhibiting a slight advantage over cHRev. We suspect that this is due to Sofia's use of RetentionRec to mitigate the risk of knowledge loss. Both Sofia and cHRev consider recent contributions of candidates. In contrast, WLRRec employs a combination of candidates' prior interactions, prior accepted review rate, experience, and workload to rank reviewer candidates without considering the recency of their contributions.

Table 4.2 shows the RAS scores of the studied approaches. LearnRec and RetentionRec are not included in the table, as they are the benchmarks used to compare other CRR approaches (worst and best performers, respectively). The median RAS scores of 0.9187, 0.9427, and 0.8466 for cHRev, Sofia, and WLRRec, respectively, suggest that they perform more similarly to RetentionRec (optimal) than LearnRec (worst). Moreover, the RAS obtained from Table 4.2 for cHRev, Sofia, and WLRRec exhibit a standard deviation of 0.0262, 0.0228, and 0.0181, respectively. This suggests that Sofia and cHRev are relatively more susceptible to variations in the reviewer set size and project, as compared to WLRRec.

Abrupt changes in developer expertise turnover led to a delayed impact on the staleness rate of CRRs, a trend observed across all projects analyzed. To explore further, Figure 4.6 plots the expertise turnover rate of the studied projects over quarterly periods. The figure plots the ratio of previous contributions from developers who stopped contributing to the project during each period against the total prior contributions from all developers who were active by the end of the period. Period numbers are normalized to begin from zero to simplify comparisons across projects. For example, in the Roslyn project, there is a decline in the turnover rate between periods 2 and 3, followed by a steady increase until the end of the timeframe with small peaks at periods 4 and 6, as depicted in Figure 4.6. In this case, we observe a comparable trend in the proportion of stale recommendations, with a gentler slope for both segments in Figure 4.5. The figure also shows two small peaks at periods 5 and 8 with a delay from the expertise turnover peaks. The fluctuation of the RAS scores for one CRR approach over different projects also confirms the resiliency of the CRR approaches against developer expertise turnover. For Kubernetes and Roslyn, the developer expertise turnover rate in Figure 4.6 shows an upward trend with a similar

pattern of peaks occurring with 1 or 2 periods of delay in Figure 4.5. For the Rust project, however, while the upward slope for the share of stale recommendations is not as steep as the expertise turnover rate, we can still observe the impact of periods that have peak turnover ratio, such as periods 9 and 13, in Figure 4.5 with 1–2 periods of delay in periods 11–12 and 14–15. The WLRRec approach exhibits weaker adherence to the trend. We suspect this is because the Rust project does not keep any record of review invitations (i.e., developers invited to review changesets). This lack of data profoundly impacts the quality of recommendations generated by WLRRec since review invitations are part of its objective function.

Stale recommendations account for a considerable portion of the suggestions provided by CRR approaches, accounting for up to 33.33% of the recommendations with a median share of 8.3% of all of the recommendations that were produced. Although considering the recency of the candidate's contributions can partially mitigate the negative impact of stale recommendations, the performance of cHRev concerning stale recommendations suggests that solely considering this metric cannot eliminate this type of incorrect recommendation.

4.5 RQ2: The Distribution of Stale Recommendations Across Reviewers

In this section, we study the distribution of stale recommendations across the reviewers of the studied projects.

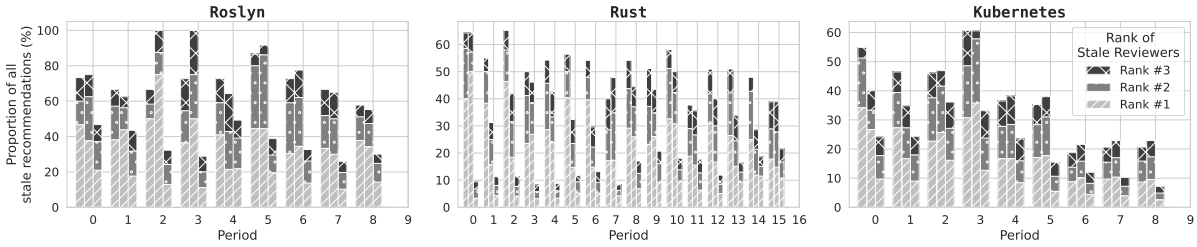


Figure 4.7: The share of top-3 reviewers' recommendations of all stale recommendations for cHRev (leftmost bar), Sofia (middle bar), and WLRRec (right bar) for studied quarterly periods with reviewer set length of one.

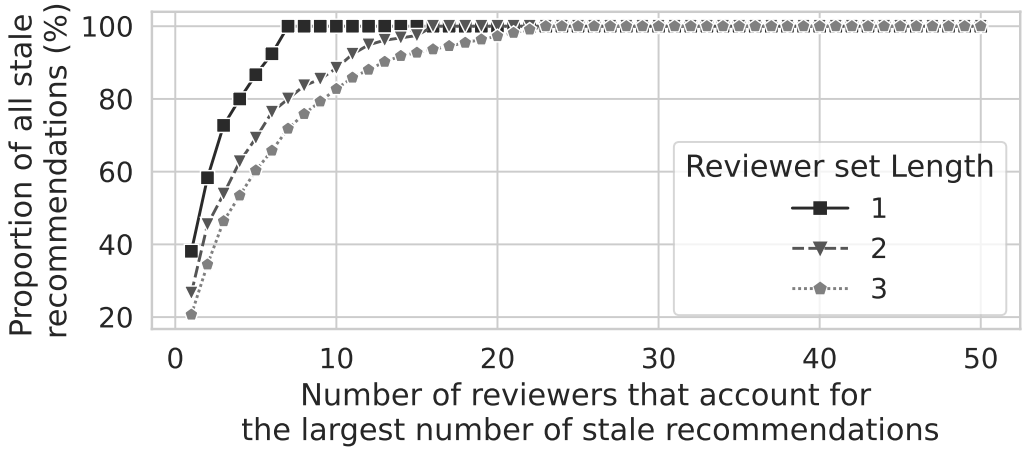


Figure 4.8: Change of top-N reviewers' share in stale recommendations with value of N when Sofia is applied to Roslyn (reviewer set lengths 1-3).

4.5.1 Approach

To study the distribution of stale recommendations, we calculate the proportion of stale recommendations accumulated by each reviewer quarterly. We aim to analyze and justify our observations to identify the most influential factors contributing to stale recommendations.

4.5.2 Results

Figure 4.7 shows the proportion of stale recommendations accumulated by the top-3 most recommended reviewers to all stale recommendations for each quarterly period when the recommendation list length is set to one. We normalize study period numbers to begin from zero for easier comparisons. Results for recommendation list lengths two which are shown in Figure A.3 in Appendix A.

A small number of reviewers account for a substantial proportion of the stale recommendations. Figure 4.7 shows that in periods 2 and 3 of the Roslyn project, 100% of the stale recommendations are in reference to three reviewers. On the other hand, Figure 4.7 also shows that the proportion of stale recommendations accumulated by the top-3 reviewers can drop as low as 0.072 (i.e., in normalized period eight of the Kubernetes project). Moreover, in 15.31% of evaluated quarterly periods, over half of the stale recommendations are accumulated by the top-3 reviewers.

Figure 4.8 shows that a few reviewers constitute most of the stale recommendations in the periods of the Roslyn project when Sofia is applied. To enhance the quality of CRRs, these reviewers can be excluded from the candidate list. Similar trends were observed in other projects. These results are shown in Figure A.4 in Appendix A.

The proportion of stale recommendations that the top reviewers accumulate tends to decrease as projects age. Figure 4.7 shows that the proportion of stale recommendations for the top-3 reviewers diminishes over time. For instance, when Sofia is applied, this proportion decreases from 75.00%, 64.29%, and 40.00% in period 0 to 55.26%, 38.89%, and 22.91% in the final studied periods of Roslyn, Rust, and Kubernetes, respectively. Moreover, the periods in between show a decreasing trend.

CRR approaches frequently recommend a small number of reviewers who stopped contributing to the project based on their prior contributions. Although the proportion of such reviewers decreases over time as experienced contributors leave the project, removing them has the potential to considerably enhance the perceived quality of the CRRs.

4.6 RQ3: The Lingering effect of stale reviewer recommendations

In this section, we study how long contributors who left a project linger in suggestion lists.

4.6.1 Approach

To calculate the duration of a lingering stale recommendation, we measure the time between the last contribution and subsequent recommendations of the reviewer. We conduct experiments for studied projects, and assess the consistency and impact of each variable on the duration of lingering stale recommendations. Our findings exclude the LearnRec and RetentionRec approaches since they are considered baseline approaches and are unlikely to be adopted in practice.

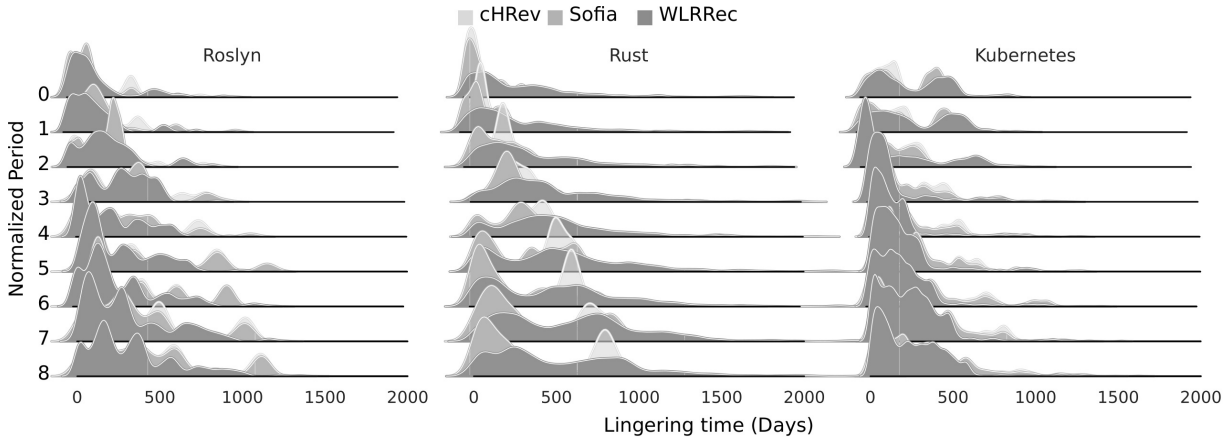


Figure 4.9: The distribution of the duration of stale recommendations (in days) over quarterly periods for the studied projects. Only the first nine periods are drawn.

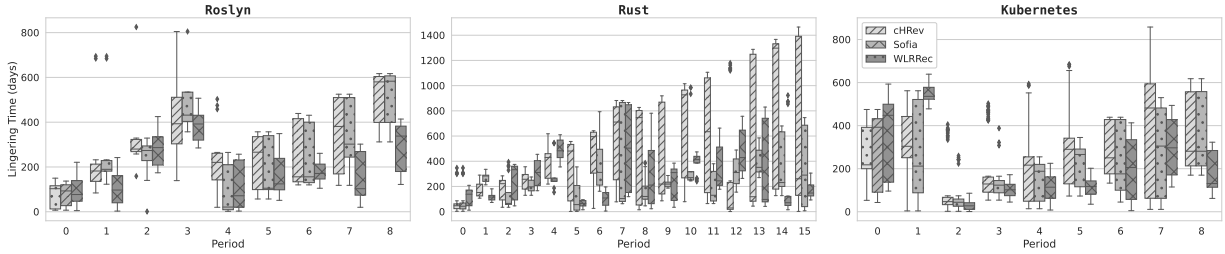


Figure 4.10: The distribution of lingering duration for the top-3 reviewers over quarterly periods.

4.6.2 Results

There exist reviewers who persist in the recommendation list of CRR approaches for up to 7.7 years, with a median time of 7 months and 21 days. Figure 4.9 shows the distribution of the elapsed time (in days) between the departure of reviewers and their subsequent stale recommendations for all the recommendations over specific quarterly periods. Among the studied projects, the upper bounds of the distributions tend to increase. This suggests that the evaluated CRR approaches do not effectively prune their candidate pool over time to eliminate stale recommendations, even though some consider the recency of their contributions. While some may argue that the responsibility of identifying potential reviewers lies with those who are familiar with the current team, this task becomes increasingly challenging as teams grow, particularly in the context of open-source projects or when developers

switch teams internally. For instance, we encountered cases in the Roslyn project where developers moved from Roslyn to Office 365 and other teams within Microsoft. These complexities and barriers indicate that CRR approaches could be used to effectively prune the pool of potential reviewers.

Indeed, contributors who have left a project may be recommended by CRR approaches long after they have left the project. For example, PR #65216 of the Kubernetes project, both cHRev and Sofia recommend developer M when the reviewing set length is set to three. Developer M both joined and left the project in 2014. Nevertheless, upon submission of PR #65216 (more than 3.5 years later), M was recommended as a reviewer due to the extensive contributions to the file "pkg/util/iptables/iptables.go".

In another example (Roslyn PR#33501) cHRev recommends three reviewers, including developer H, who participated in Roslyn's code development from June 2014 to September 2018. In determining H's score, contributions and workdays each equally account for 30% of cHRev's score. The remaining 40% of the score weight is determined by the recency of the contribution. Meanwhile, Sofia deems this changeset at a high risk of turnover-induced knowledge loss and recommends candidates who are actively involved in the project's development. Thus, Sofia does not make any stale recommendations for this changeset. The WLRRec's recommendation for this PR is primarily influenced by previous interactions with the code's author and, as such, it only makes one stale recommendation when the length of the recommendation list is set to three.

The lingering attribute of stale recommendations differs among CRR approaches, influenced by their objective functions. CHRev shows the highest staleness (median staleness of 245 to 279 days) across the projects under scrutiny, with Sofia performing slightly better (median staleness of 201 to 258 days). WLRRec presents the lowest median staleness in Roslyn and Kubernetes (160 and 203 days), but the highest in Rust (371 days). We suspect that these differences are explained by the focus of cHRev on maximizing expertise, which can overshadow the recency, leading to stale recommendations. Sofia is similar to cHRev, but includes an adjustment for knowledge turnover risks, and uses RetentionRec to recommend active reviewers in high-risk changes. By considering social dynamics and reviewer request responsiveness, WLRRec provides a more balanced recommendation distribution in the Roslyn and Rust project. Rust's poor performance (371 days median staleness) likely stems from missing review request responsiveness records, underscoring this feature's importance in countering lingering stale recommendations.

As projects age, CRR approaches tend to accrue a larger candidate list without pruning those who have left the project. This exacerbates the impact of top reviewers who left a project many periods ago. CRR approaches with **high reliance on the expertise** of

the candidates (e.g., cHRev) are especially prone to this problem. As Figure 4.9 suggests, some of the reviewers may remain on the candidate list for a long time. These developers are most likely stuck in the candidate list due to their experience and prior contributions to important modules, which may degrade the performance of CRR approaches over time. Figure 4.10 shows this distribution for the top-3 reviewers over quarterly periods when the reviewer set length is set to one and confirms the increasing tendency of the longevity of the lingering duration over time. The results for reviewer set lengths two and three also follow the same trend which can be seen in Figure A.5 in Appendix A.

Sudden declines in the staleness of the top-3 reviewers are linked to increased release frequencies paired with a surge in new file additions, followed by high developer expertise turnover. This pattern is evident in periods 15 and 16 for Roslyn and 12 and 13 for Kubernetes, as shown in Figure 7. Our analysis indicates that these declines in the lingering days of the top-3 stale reviewers occur after periods marked by heightened release activities, new file additions, and developer turnover. New releases are often accompanied by a spike in bugs or feature requests that are related to newly added files, necessitating the involvement of developers who are familiar with those files. Suppose these developers have recently left the project. In that case, they are likely to be recommended, thereby reducing the lingering days of the top-3 stale reviewers by suggesting those who worked on those releases. For Roslyn, this effect can be observed with an increase in new files in PRs and a peak in release rates between periods 12 to 15. This is coupled with an increase in developer turnover that peaks during periods 14 and 16. Consequently, a noticeable drop for top-3 developer lingering days occurs between periods 15 to 16. In Kubernetes, the drop occurs between periods 12 and 13, triggered by a spike in developer turnover in period 11 and a high release rate compared to the median rate of all releases from periods 9 to 12, with an increase in new file additions in PRs from periods 8 to 11. The Rust project does not exhibit such concurrent events; hence, there is no similar decline in lingering days for its top-3 stale reviewers. Further detail of these rates and analyses are available in Figures figs. A.6 to A.9 in Appendix A.

Comparing our latest findings with our previous research question shows that while

the proportion of top-3 reviewers may decrease over time, their residual effect tends to increase. This highlights the need to remove the top reviewers who have left the project from candidate pools to address the lingering effect present in CRR approaches.

CRR approaches make stale recommendations frequently, even years after contributors have left a project. While the percentage of the top reviewers of all the stale recommendations may decrease over time, their residual effect tends to increase. Regular pruning of the candidate pool could provide a reliable way to improve the perceived quality of reviewer recommendations.

4.7 Mitigation Plan

Based on our findings, we propose to incorporate the recency of developer contributions and filter out stale reviewers as a new stage for CRR approaches. This filter can ensure that the recency factor is not overshadowed by other variables, and can integrate with existing CRR approaches.

4.7.1 Approach

We propose a time-based filtering stage that excludes developers from the recommendation list if they have not contributed within a specified timeframe ($P_{ContributionGap}$). We assess the effect of different $P_{ContributionGap}$ durations—one year, six months, three months, and one month—on the performance of each approach to reveal the potential impact of applying the proposed factor and its effectiveness on studied approaches and projects. To this end, we apply the filter across these intervals and measure the metrics below:

Staleness Reduction Ratio (SRR) quantifies the improvement in recommendation staleness of a CRR approach upon integrating our time-based filter. SRR is calculated as the relative increase in the proportion of all the recommendations:

$$SRR = \frac{SSR_{\text{No Filter}} - SSR_{\text{Time-based Filter}}}{SSR_{\text{No Filter}}} \tag{4.2}$$

$SSR_{\text{No Filter}}$ and $SSR_{\text{Time-based Filter}}$ represent the original staleness without, and with the filter applied, respectively.

Developers' Work Load Ratio (DWLR), inspired by prior work [109], evaluates the potential workload on non-stale recommended reviewers should they accept all recommendations. Workload is estimated using the reviewer's share of total review tasks.

F1-Score assesses the performance of the proposed filter in predicting stale recommendations. In this context, true positives are correctly replaced stale recommendations, false positives are non-stale recommendations mistakenly replaced, true negatives are non-stale recommendations accurately left unchanged, and false negatives are stale recommendations that were not identified and thus remained.

Present Reviewers Expertise (PRE) assesses the impact of the time-based filter on the expertise level of non-stale recommended reviewers. For each PR, after excluding stale reviewers, we measure the expertise of remaining recommended reviewers based on their prior contributions to the files involved in the PR. Expertise is defined following the formulation of Mirsaeedi and Rigby [109], i.e., focusing on the proportion of modified files which previously contributed to by the reviewer before the submission of the PR.

4.7.2 Results

SRR decreased substantially, with reductions ranging from 21.44% to 92.16% for cHRev, 22.42% to 92.39% for Sofia, and 21.62% to 89.45% for WLRRec. Remarkably, this filter also enabled LearnRec, a naïve baseline approach, to reduce stale recommendation rates by 19.93% to 92.48%. Thus, the time-based filter successfully achieves its primary goal of substantially lowering stale recommendations.

Reducing $P_{ContributionGap}$ enhances SRR but restricts the pool of available reviewers, thereby increasing the workload for active contributors. As anticipated, narrowing the interval for recent contributions can exclude active contributors, potentially increasing the workload for other developers. The median Developers' Work Load Ratio (DWLR) for the top-3 most recommended non-stale reviewers across studied periods increases with shorter $P_{ContributionGap}$ intervals—8.33%–13.33% for 1 year, 9.69%–15% for 6 months, 11.11%–16.67% for 3 months, and 13.41%–19.14% for 1 month, compared to 6.59%–12.29% without the filter. This trend highlights the trade-off in setting $P_{ContributionGap}$ values for the time-based filter.

The time-based filter considerably improves CRR approaches that overlook contribution recency or seek recommendation diversity, with F1-Scores ranging from 0.0254-0.1894 for cHRev, 0.0196-0.1560 for Sofia, and 0.2611-0.3587 for WLRRec. The time-based filter excels in contexts with higher stale recommendation rates. For cHRev and Sofia, the high precision shows that the time-based filter effectively identifies long standing stale

reviewers, but struggles with recent stale ones, hindering their recall due to low rates of stale recommendations. In contrast, WLRRec suffers from a lower precision, but enjoys a higher recall, due to its higher rate of stale recommendations. LearnRec benefits across both metrics, indicating the filter's broad applicability for this baseline approach.

Applying the proposed filter to CRR approaches maintains or improves the expertise of non-stale recommended reviewers in our studied cases. The median Present Reviewers Expertise (PRE) by cHRev and Sofia remains unchanged, while their mean expertise increases slightly, ranging from 0.73%–2.44% for cHRev and 0.70%–2.30% for Sofia. Conversely, WLRRec shows a notable median PRE improvement of 16.66%–40%, with mean expertise rising by 3.32%–10.64%. This enhancement results from the filter's exclusion of stale reviewers, thus increasing the likelihood of selecting reviewers with relevant expertise. Further analysis with the Wilcoxon signed-rank test shows significant changes in the distribution of PRE across settings, except for WLRRec over a 1-year interval in Roslyn for reviewer sets one and two, and Kubernetes for set three, highlighting significant impact of the time-based filter on reviewer selection. WLRRec's behavior over a 1-year interval is likely linked to stale recommendations from reviewers unfamiliar with modified files but who have recent interactions with the author and a high review invitation responsiveness. The recency of these interactions means the 1-year filter has little effect on these recommendations.

Detailed measurements for each experiment are available in Appendix A.

Filtering out inactive developers does not compromise recommendation quality; it actually enhances the expertise of the available recommended reviewers, with median improvements of PRE ranging from 0.73% to 10.64% across studied approaches. However, it may impose an additional workload on active reviewers. Thus, selecting an optimal cut-off interval is essential to minimize stale recommendations without overburdening developers by excluding too many potential reviewers.

4.8 Threats to Validity

Construct Validity. In this chapter, we explore stale recommendations in CRR approaches, defining staleness as the interval between a developer's last contribution and subsequent recommendations. Because exact departure times are not usually documented, we should estimate the developer departure. We approximate the developer departure as one day after their last contribution if there has been no activity for at least 180 days.

While this method may threaten the construct validity of our study, it is a widely accepted approximation in prior research [64, 92, 141].

Internal Validity. Contributors labelled as stale reviewers may be only taking a temporary hiatus. The last two quarterly periods are removed from the analysis to mitigate this threat, i.e., the shortest period of absence that will cause us to label a reviewer as stale is six months. We may also mislabel a recommendation as stale if it occurs shortly after the reviewer's last contribution, even though they remain active. However, we find that such instances are rare. In the Roslyn project, stale recommendations for reviewers who departed within one day, one week, and one month of their last contribution comprised only 0.38%, 2.26%, and 7.94% of all stale reviewers, respectively. In Rust, the corresponding percentages were 0.32%, 2.17%, and 7.7%, and in Kubernetes, 0.37%, 3.13%, and 10.89%, indicating that these cases represent a small proportion of all stale recommendations.

External Validity. Although we study five different CRR approaches, the outcome of our analysis may not generalize to all settings. However, our findings highlight the degree to which current approaches are susceptible to stale recommendations and their characteristics for three studied projects of varying scales and domains. Including more projects could enhance the external validity of our results; however, running our simulations takes several days per project which means extending the dataset would require a large quantity of computational resources. Nevertheless, replication of the study in other contexts may prove fruitful.

4.9 Conclusions and Lessons Learned

CRR approaches have been criticized for producing unactionable recommendations [80, 185]. Stale recommendations (i.e., recommended reviewers who no longer contribute to the project) are a concrete type of incorrect recommendations that hinder CRR performance and erode developer trust, especially when the recommended reviewer has long abandoned the project. Since stale reviewers can no longer effectively contribute to the project, they are truly incorrect recommendations, providing a unique opportunity for improvement. Therefore, in this chapter, we examine three projects using five different CRR approaches to understand their nature. Our investigation focuses on the prevalence of these recommendations (Section 4.4), the distribution of stale reviewers (Section 4.5), and the duration for which stale reviewers continue to appear in recommendations (Section 4.6). From our findings, we derive the following actionable insights for both practitioners and developers of CRR tools:

- **We recommend that practitioners configure their CRR systems to generate a larger number of recommendations than they may need.** This method acknowledges the inherent tendency of CRR systems to suggest stale reviewers to varying extents (as discussed in RQ1) and mitigates the issue by providing additional reviewer options, thereby reducing the likelihood of being affected by stale recommendations. This strategy, however, does not counteract the erosion of trust that is associated with recommending stale reviewers who have left the project.

- **The effect of stale recommendations is particularly substantial when experienced developers leave, as the proportion of stale recommendations among all CRRs is directly influenced by the rate of expertise turnover (as shown in RQ1)**. Consequently, for projects facing an exodus of experienced developers, we advise (1) employing CRR approaches that emphasize the recent contributions of reviewers, such as Sofia; and (2) whenever feasible, tuning the hyperparameters of CRR systems to weigh the contribution recency more prominently.

- **Our findings suggest that the recency of developer activities should be incorporated as a stage before recommendations are generated.** Doing so can mitigate the predominance of other factors, such as developer expertise, that may overshadow the recency of developer contributions. We observe a positive effect in approaches like cHRev and Sofia, where considering developer recency as a separate preparatory stage reduces the likelihood of stale recommendations by 19.93%-92.48%, depending on the cut-off parameter of the filter, and the project and CRR approach under scrutiny.

- **Identifying and replacing frequently recommended stale reviewers are crucial.** Throughout our study, we noted instances where pinpointing the top-3 stale reviewers and substituting them with active developers reduces incidences of stale recommendations by up to 22.10% overall, with medians over the studied periods of 37.16%, 32.62%, 17.01% for the cHRev, Sofia, and WLRRec approaches, respectively. Therefore, identifying frequently recommended stale reviewers and removing them from the pool of available developers can alleviate the general staleness issue.

- **Implementing a threshold for the latest reviewer contribution can help mitigate the issue.** Our analysis reveals that CRR approaches like cHRev, Sofia, WLRRec, and LearnRec allow stale reviewers to persist over extended periods of time. This trend worsens as projects mature, as evidenced by the increasing distribution of lingering time among recommendations (Figure 4.9). To counteract this, we propose implementing a maximum threshold for the latest reviewer contribution.

Doing so will reduce the effect of recommendations drifting away from the active pool of developers. Our evaluation of various intervals indicates a trade-off between the developers' workload and the staleness of recommendation while having either a positive or negligible impact on the knowledge of non-stale reviewers. Unfortunately, it is not possible to recommend one best cut-off interval to mitigate staleness due to the contribution of various factors, such as knowledge turnover rate. However, our findings can help orient practitioners to obtain more useful recommendations in their specific circumstances. Using this mitigation strategy, one can strike a balance between the potential for sharing task knowledge between stale reviewers and active developers and their decreasing likelihood of responding as the time since their latest contribution grows.

4.10 Chapter Summary

In this chapter, we explore the challenge of using CRR systems with stale reviewer recommendations. We demonstrate that CRR systems are negatively impacted by this issue and require a pre-filtering stage for mitigation. While staleness can immediately affect code review velocity, not all challenges faced by CRR systems have such an immediate impact. In the next chapter, we address a different challenge that gradually degrade the defect-proneness of project.

Chapter 5

Exploring the Notion of Risk in Code Reviewer Recommendation

> **Note.** An earlier version of the work in this chapter appears in the Proceedings of the International Conference on Software Maintenance and Evolution (ICSME 2022), [69].

5.1 Introduction

Finding reviewers with the time to review a code change and familiarity with the modified subsystems has been a challenge in organizations who adopt code review [138, 152]. This is especially the case for large organizations with hundreds of developers. In such organizations, the authors of a changeset may not yet have a professional relationship with the team responsible for overseeing the development of all of the components that they have changed. Therefore, they may request someone to review the changeset who are either too busy, or lack the necessary familiarity to review their proposed change. Unfortunately, this mean that the time-to-merge would increase and the Pull Request (PR) may be abandoned [73]. While Code Reviewer Recommendation (CRR) approaches aim to help stakeholders to find suitable reviewers [171], it would be hard to assess the performance of the recommendation concerning their impact on aspects like project safety, i.e. have a lower number of bugs in the project.. As a result, CRR systems often suggest obvious choices and add little value to the projects [80].

Conventional reviewer recommendation studies evaluated their proposed approaches against historical records, i.e., who performed each task in the past [175]. However, more

recent work explores how recommendation approaches can be used to balance quantities of interest [109, 130]. These approaches consider previous interactions of the candidates with the modified files, the workload of the candidates at the time of the code review, and previous interactions between the developers in the project. Candidate reviewers are then ranked based on these metrics, and top-ranked candidates are suggested to decision-makers.

The results from previous studies suggest reviewers who share properties with those who performed similar reviews in the past and improve evaluation metrics such as files at risk. While the measures that have been proposed by previous studies align with important dimensions, the risk of defect proneness has not been explored. The risk of defect proneness of a code change indicates how probable it is for the change to induce fixes in the future. As an intervention, changes with a high risk of inducing future fixes may be assigned to subject matter experts for review. Prior work suggests that subject matter experts may be more adept at identifying problems during the review process [77, 181]. However, this intervention is likely to impose a greater burden on key team members.

In this thesis, we take the position that an ideal recommendation approach should balance the trade-off between the burden on expert reviewers and the risk of defect proneness. Therefore, we set out to incorporate defect proneness in the reviewer recommendation process. More specifically, we set out to address the following research questions:

RQ1 How do existing code reviewer recommenders perform with respect to the risk of inducing future fixes?

Motivation: Every code change induces some degree of risk. The degree of risk varies based on the change and its domain [157]. A key goal of the code review process is assessing and mitigating the risk of introducing defects during or shortly after the code integration process [36]. It is crucial to involve subject matter experts in the review process to achieve that goal. Otherwise, if non-experts review high-risk tasks, defects may slip through the integration process. Thus, we first set out to understand how well existing reviewing assignments and CRR-based reassignments perform in terms of risk mitigation.

Results: We observe an inherent trade-off between our studied quantities of interest. For instance, the RetentionRec recommender – a reviewer recommendation approach proposed to minimize the risk of developer turnover-based knowledge loss while ignoring other quantities of interest – reduces files at risk by up to 23.89% with respect to the reviewers who have already performed the review. On the other hand, RetentionRec underperforms in terms of the Changeset Safety Ratio (CSR) – a measure that we propose to indicate the performance of a recommendation approach concerning the safety of the code change process – by 4.56% to 37.07%.

RQ2 How can the risk of fix-inducing code changes be effectively balanced with other quantities of interest?

Motivation: Optimizing for other quantities of interests, such as Files at Risk of turnover (FaR), without considering defect proneness is unlikely to perform well due to the inherent trade-offs discovered in RQ1. Therefore, an approach is needed to incorporate defect proneness in recommendation decisions without overly disrupting other quantities of interest. To that end, we propose CSR – a reviewer recommendation approach that aims to incorporate defect risk into recommendations – and set out to evaluate how well it performs.

Results: Our experiments indicate that CSR increases the expertise of reviewers assigned to reviews by 12.48% and the CSR by 80.00% while reducing FaR by -19.39% and only increasing the core development team workload by 0.93%. Moreover, we find that project or team-specific tolerance of risk can be incorporated by adjusting the threshold P_D, which is the threshold of the likelihood of fix-inducing PRs at which changes are deemed risky enough to require intervention. The effective P_D interval is defined as the change interval for which the performance of the CSR is impacted. For instance, in Roslyn, the effective interval of P_D is 0 - 1; however, the effective interval of P_D is 0 - 0.3 and 0 - 0.1 for the Kubernetes and Rust projects, respectively. Thus, P_D must be calibrated to its effective range for CSR to achieve optimal results.

RQ3 How can we identify an effective fix-inducing likelihood threshold (P_D) interval for a given project?

Motivation: The performance of CSR depends on the P_D setting. P_D itself is dependent on a project's past defect proneness. Moreover, different projects may assign different weights to the importance of defect proneness. Therefore, we set out to propose approaches to support stakeholders in tuning P_D to an appropriate value for their development context.

Results: We propose static, normalization, and dynamic approaches to tune the value of P_D. Results that explore P_D settings in risk-averse, risk-tolerant, and balanced contexts indicate that the proposed methods affect the performance of CSR significantly. Moreover, the dynamic method outperforms the others in risk-averse and balanced contexts to a statistically significant (Conover's Test, $\alpha < 0.05$) and practically significant degree (Kendall's W = 0.0727 - 0.543, small - large).

Table 5.1: The detail of dataset used to evaluate the proposed method (based on the prior work of Mirsaeed *et al.* [109]).

Name	*Files*	*Review PRs*	*Years*	*Developers*
Roslyn	12,313	8,646	5	469
Rust	12,472	17,499	9	2,720
Kubernetes	12,792	32,400	5	2,617

5.2 Studied Datasets

In this section, we present the sources of data and the projects used to conduct our study and the rationale for their selection.

Data Source. To evaluate CSR, we seek to ground our analysis in a comparison to previous multi-objective reviewer recommenders [109]. Therefore, to obtain a fair comparison, we begin with the same subject systems that Mirsaeedi and Rigby studied [109]. However, two of these projects, CoreFx and CoreCLR have been since merged with the *.Net Runtime* project. Due to this migration, Commit Guru was unable to obtain the necessary information for the prediction model and rendered us unable to process the master branch for possible fix-inducing commits. As a result, we omit CoreFx and CoreCLR, focusing our analysis on Rust, Kubernetes, and Roslyn. Rust and Kubernetes are community-driven projects, and Roslyn is an industry project developed openly on GitHub. These projects are well-established (more than four years old) with more than 10K PRs. Kubernetes has had a significant impact on cloud computing platforms with more than 3.1K contributors. Roslyn, with 524 contributors, is an open source .NET compiler platform for languages such as C# and VB. Finally, Rust, with 3.8K contributors, is a multi-paradigm, general-purpose programming language. Further details of these projects are listed in Table 5.1.

Data Collection. We begin our data collection process by downloading the relevant details from the replication package provided by Mirsaeedi and Rigby [108]. The shared data includes commits, files that have been modified in each commit, developers involved in a PR, a list of developers and reviewers of each PR, and developers' interaction with the PR. To perform defect analysis, our approach requires a list of the commits that comprise each of the PRs. Moreover, we need to compute the measures listed in Table 5.2 to train our defect prediction model. We use the GitHub API to gather the additional data for each PR in the data set. We did not use the commits of a PR to calculate additions and deletions since they might have cancelled each other out (e.g., one line added in one commit could be removed in the next commit of the same PR). Instead, we calculate the

net number of additions and deletions extracted directly for each of the PRs.

Table 5.2: The risk measures produced by Commit Guru which is used in this chapter to predict fix-inducing PRs (from Kamei *et al.* study [68]).

Dim.	*Name*	*Definition*	*PR Calculation Approach*
Diffusion	NS	Number of modified subsystems	Calculated from the changed files' paths
	ND	Number of modified directories	Calculated from the changed files' paths
	NF	Number of modified files	Calculated from the changed files' paths
	Entropy	Distribution of modified code across each file	Averaged over commits' entropy
Size	LA	Lines of code added	Extracted from the GitHub API
	LD	Lines of code deleted	Extracted from the GitHub API
	LT	Lines of code in a file before the change	Averaged over commits' LT
History	NDEV	Number of developers that changed the modified files	Averaged over commits' NDEV
	AGE	Average elapsed time since the last change of files	Averaged over commits' AGE
	NUC	Number of unique changes to the modified files	Averaged over commits' NUC
Experience	EXP	Developer experience	Averaged over commits' EXP
	REXP	Recent developer experience	Averaged over commits' REXP
	SEXP	Developer experience on a subsystem	Averaged over commits' SEXP

5.3 Study Design

This chapter is comprised of two parts: (1) identifying fix-inducing PRs and (2) evaluating reviewer recommendation approaches. This section describes each part of our study and explains the rationale behind our design decisions.

5.3.1 Identifying and Predicting Fix-Inducing Pull Requests

Because our approach aims to incorporate the notion of risk in the recommendation process, identifying fix-inducing PRs with which to evaluate our approach is an important part of the study. In this chapter, we operationalize risk by mining the repositories of the studied

projects for defect-fixing, and fix-inducing commits using Commit Guru [136]. Figure 5.1 provides an overview of our discovery process for risky PRs.

Step1: Extract defect prediction data

We first apply Commit Guru [136] to the studied repositories in order to produce data sets of fix-inducing commits, as well as a popular set of measures for their prediction. Commit Guru clones each repository, computes commit-level measures that share a relationship with risk (e.g. patch size, diffusion), and applies the SZZ algorithm [145] to identify which historical commits have induced future fixes. Finally, a logistic regression model is fit to estimate the riskiness of code changes. Table 5.2 shows the set of used risk measures.

Although studies by Quach *et al.* showed some of the limitations of SZZ ([123, 124]), its output is still an indicator of bug-inducing probability. Moreover, we decided not to use manually verified bug datasets such as the one by Rodriguez-Perez *et al.* [135] as we wanted to view the effects of the recommendation approaches in their natural habitat, which would normally be automated and include tools such as SZZ.

Step2: Train and test PR-level risk model

We use the risk measures extracted by Commit Guru to train defect prediction models. A logistic regression method is used to train the model for each quarter (three months). The three-month time interval is based on similar studies, like Mirsaeedi and Rigby [109], and retraining this period length setting allows us to extend reviewer recommendation approaches to incorporate risk and more directly compare results. Moreover, updating the prediction model in short (three months) intervals has been recommended to counteract concept drift [37]. This step is decomposed into the following tasks:

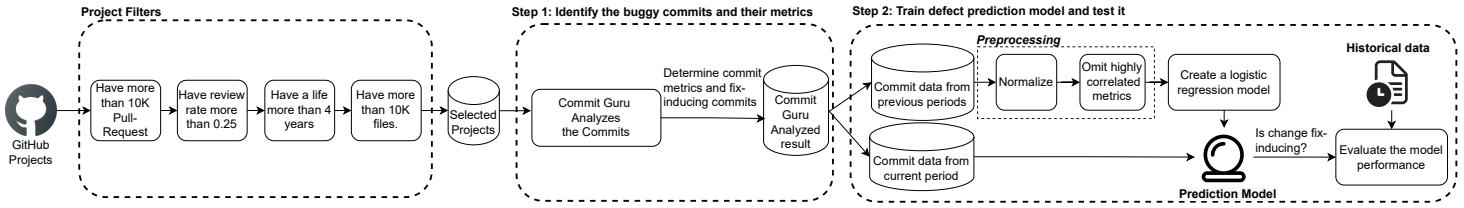


Figure 5.1: The simplified overall architecture of the project selection filters and the defect prediction process.

1. ***Data preprocessing.*** Before training the models, data must be preprocessed to counteract biases. First, we standardize the risk measures since their magnitudes vary broadly. We use Scikit StandardScaler[1] to transpose all risk measures' values to have zero mean and unit variance. Then, we identify highly correlated measures, as they affect the model's performance. To this end, we calculate pairwise Pearson correlation (ρ) between each pair of risk measures. As suggested by Tay [150], any pair of risk measures with $|\rho| > 0.6$ is considered to have too much similarity to include in the same model fit. In such cases, we remove all the measures but one (Based on their order of appearance as listed in Table 5.2).

2. ***Fit defect prediction model.*** Once the data has been preprocessed, we use the data to fit a logistic regression model for every quarter using the previous quarters' data. The model then estimates the likelihood that each code change will be fix-inducing in the following quarters.

3. ***Aggregate risk estimates to the level of PRs.*** Using the trained models, we estimate the riskiness of each PR by aggregating the risk measures across all of the PR changes. We use the PR's commits risk measures to calculate the risk measures for a PR. Table 5.2 has a brief explanation of how each of these risk measures is calculated from the set of commits belonging to a PR. Using the PR risk measures, the model estimates the PR's likelihood of inducing a future fix. We use the balanced accuracy performance measure to evaluate the performance of our models since our datasets are inherently imbalanced, i.e. there are more non-fix-inducing PRs than fix-inducing PRs. The median balanced accuracy over different periods for Roslyn, Rust, and Kubernetes projects are 75.9%, 50%, and 97.5%, respectively.

5.3.2 Ranking Potential Reviewers of a Pull Request

As the next step, we use the fix-inducing likelihood of the PR and its risk measures to suggest reviewers for each PR. We evaluate seven baseline approaches (RQ1) as well as our proposed method, CSR (RQ2). We describe the baseline approaches below, and describe CSR in Section 5.5.2.

[1]https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html

AuthorshipRec

Suggested by Mockus and Herbsleb [112], the authorship of a file is an important factor when assigning software experts to (reviewing) tasks. Bird *et al.* [16] formulated the AuthorshipRec in their paper based on the proportion of the files that a developer modified prior to the PR.

RevOwnRec

Thongtanunam *et al.* [153] suggested a new reviewer recommender based on the developers' previous review history. The rationale was that the project code reviewers for each project subsystem are constant most of the time. Similarly to AuthorshipRec, RevOwnRec considers the proportion of a developer's reviews or modifications relative to all of the reviews and modifications in a PR.

cHRev Recommender

The cHRev recommender [183] is a popular conventional reviewer recommender. When ranking developers as potential candidates of a code change, cHRev considers the developer's expertise from previous reviews as well as the recency of the contributions. To rate the fit of a developer D for reviewing a file F, the *xFactor* was used:

$$xFactor(D, F) = \frac{C_f}{C'_f} + \frac{W_f}{W'_f} + \frac{1}{|T_f - T'_f| + 1} \tag{5.1}$$

Where C_f, W_f, and T_f represent the number of review comments, the number of workdays that D commented on the file's reviews, and the most recent day that D worked on F, respectively. The prime versions of the variables in the denominator represent the total number used to normalize the output. Then, the fit for each developer is estimated using the summation of the *xFactor* for all the files in the code change.

LearnRec

The LearnRec recommender is designed to distribute knowledge among team members. *LearnRec* suggests developers who are poised to learn the most from reviewing a PR. *ReviewerKnows* has been suggested as a way to measure how knowledgeable a potential reviewer is about a review request [109]. The *ReviewerKnows* estimates how familiar a

developer would be with the modified files of a review request. It is usually favourable to distribute the knowledge among developers in repositories to mitigate any loss of knowledge if any developer leaves the project. To this end, *LearnRec* is formulated by subtracting *ReviewerKnows* from one, which estimates how much a developer can learn by reviewing a PR. This metric can be used to create a reviewer recommender that distributes the knowledge among the project developers by assigning the review to the developer with the largest *LearnRec*.

RetentionRec

Although LearnRec seems like a reasonable choice to prevent knowledge loss, in reality, many developers do not contribute to a project over a long time [187]. Those who stand to learn the most may leave the project before that knowledge can be put to use. To mitigate this issue, *contribution ratio* and *consistency ratio* have been proposed. The *contribution ratio* for a developer is the proportion of contributions during the previous particular period of time (e.g., one year) for which the contributor is responsible. The *consistency ratio* is the proportion of sub-periods (e.g., months) that the developer was actively contributing to the project throughout a study period (e.g., year). As developers become more consistent or more (proportionally) active, the *RetentionRec* increases, suggesting that it is less likely that they will leave the project.

TurnoverRec

Mirsaeedi and Rigby [109] multiplied *RetentionRec* and *LearnRec* and created *TurnoverRec*. This recommender helps with distributing knowledge among the more active members of the development team. Recommending reviewers based on this measure minimizes the risk of turnover-induced knowledge loss caused by developers leaving the company by distributing knowledge among active members.

Sofia

Sofia [109] is a combination of *TurnoverRec* and *cHRev* whose objective is to distribute knowledge among the more active team members whenever files with a large risk of knowledge loss are present in a PR. The scoring function used for the developer (D) and the

code change R is:

$$\begin{cases} cHRev(D,R), & if\ |knowledgeable(f)| \leq d, any f | f \in R \\ TurnoverRec(D,R), & otherwise \end{cases} \tag{5.2}$$

We consider d=2 in this equation, similar to the original work by Mirsaeedi and Rigby [109], to prevent any knowledge loss by leaving one developer from the team.

5.3.3 Recommendation Component

We apply these reviewer recommender to our datasets and calculate the recommenders' scores for all the candidate reviewers. We then rank potential candidates based on the scores. Configurable parameters include the number of reviewers per PR and the maximum number of files per PR for the reviewer's knowledge. For the purposes of this study, we choose only the top suggested candidate per PR and randomly replace it with one of the actual reviewers (to match prior work [109]). We only consider PRs with less than 100 files and do not associate the PR with developers' knowledge otherwise regarding maximum files per PR. It is because one developer cannot perceive large code changes as argued by Bird *et al.* [132].

5.4 Evaluation Setup

In this section, we describe the evaluation metrics used to assess the performance of reviewer recommenders and our rationale for selecting those metrics. As explained in Section 2, conventional recommendation approaches aim to recommend the reviewers who performed the task [9, 56, 126, 183]. However, Kovalenko *et al.* [80], suggest that recommending the reviewer who reviewed a PR provides little value to the project. Furthermore, there exist many qualified developers who may not have reviewed PR but would have been comfortable doing so [47]. Conventional evaluation methods consider these recommendations incorrect and penalize the recommenders for making such suggestions.

To assess the effect of a recommendation approach on the mitigation of the risk of fix-inducing PRs, we leverage the simulation approach presented by Mirsaeedi and Rigby [109]. These measures quantify previously discussed aspects of the reviewer recommendation process and estimate the performance of a reviewer recommender through history-based simulation. We run simulations for the selected projects and compare the outcome of the

recommenders with one another with respect to the evaluation measures. We expand the set of evaluation measures proposed by Mirsaeedi and Rigby [109] to incorporate the CSR — a cumulative measure of the risk of fix-inducing changes in a given period of time. These measures originated from the challenges and expectations of the researchers who studied the code review process and recommendation approaches prior to this study [6].

In the remainder of this section, we explain each of the recommendation evaluation measures we employ in this chapter.

Expertise. *Expertise* of the reviewers assesses the recommended reviewers by the expertise that they have in the PRs they have been tasked to review. It is the primary evaluation criterion used in past studies [25, 74]. Past work has indicated the important role that involving subject matter experts has on the review process [18, 77]. To quantify this measure, Mirsaeedi and Rigby [109] proposed the following measure:

$$Expertise(Q) = \sum_{R}^{Reviews(Q)} \frac{FileReviewersKnow(R)}{FileUnderReview(R)} \tag{5.3}$$

Where Q is the quarter in which this metric is calculated. A developer is assumed to know a file if they have modified or reviewed the file prior to the PR reviewing task.

CoreWorkload. Having all PRs reviewed by experts is ideal, but there is an inherent trade-off between the time that experts invest in reviewing PRs and the amount of time they have for other development tasks [77]. The problem amplifies as projects grow if the core developer teams do not grow as well. Mirsaeedi and Rigby [109] proposed a static core team size of the top 10 reviewers and using the following equation, estimate the reviewing workload that the core team is coping with:

$$CoreWorkload(Q) = \sum_{D}^{Top10Reviewers(Q)} NumReviews(D) \tag{5.4}$$

Files at Risk of turnover (FaR). The loss of knowledge caused by knowledgeable developers leaving a project may consume resources and even stall its progress. The *FaR* measures the number of files known by zero or one developer in a period of one quarter. The formula [132] to calculate this measure is:

$$FaR(Q) = \{f | f \in Files, |ActiveDevs(Q, F)| \leq 1\} \tag{5.5}$$

Where *ActiveDevs* represent the developers who are familiar with the set of files F and are

still actively contributing to the project by the end of quarter Q.

Changeset Safety Ratio (CSR). The replacement of reviewers does not affect the incidences of bugs in our simulation. Instead, to assess the impact of replacing reviewers on risk, we assume that having an expert, preferably one who has recently interacted with files in the code change, will reduce the likelihood of merging fix-inducing code changes [16]. To this end, we formulate the Changeset Safety Ratio (CSR) as a measure of how well the review assignments have mitigated the fix-inducing likelihood of a set of PRs:

$$CSR(Q) = \sum_{R}^{Reviews(Q)} (1 - DefectProb(R)) \times MaxXFactor(R) \tag{5.6}$$

The *DefectProb* is the risk estimate of a PR being fix-inducing, and the *MaxXFactor* is the maximum score of the *xFactor* (equation 5.1) among all the suggested reviewers of a PR. The xFactor incorporates both the recency and quantity of contributions in assessing reviewer expertise and is at the core of the cHRev recommender [183]. If the risk of inducing a future fix that a PR presents is small, we may assign developers with less expertise to that code change without impacting the CSR disproportionately. Increases in CSR indicate that the code change is less likely to be fix-inducing or that the developer's maximum expertise has increased. In either case, increases to CSR suggest that the review process is performing well in terms of risk mitigation.

5.5 Experimental Results

In this section, we describe our experiments, the results and the analysis of the results. We use the percentage of change to evaluate different reviewer recommenders' performance:

$$\Delta MeasureChange(Q) = (\frac{SimulatedMeasure(Q)}{ActualMeasure(Q)} - 1) \times 100 \tag{5.7}$$

The $ActualMeasure$ and $SimulatedMeasure$ refer to the calculated evaluation metric for the historical data and a simulation run, respectively.

Table 5.3: Recommender performance vs. reality. Up and down arrows indicate improvement and degradation, respectively.

CRR	Project	Expertise	Workload	FaR	CSR
AuthorshipRec	Roslyn	15.52% ↑	-7.045% ↑	34.91% ↓	17.50% ↑
	Rust	10.64% ↑	4.09% ↓	42.58% ↓	16.66% ↑
	Kubernetes	12.87% ↑	-2.07% ↑	18.60% ↓	18.36% ↑
RevOwnRec	Roslyn	21.82% ↑	1.83% ↓	17.5% ↓	2.76% ↑
	Rust	12.72% ↑	8.16% ↓	98.62% ↓	-9.57% ↓
	Kubernetes	18.56% ↑	3.89% ↓	-4.05% ↑	1.08% ↑
cHRev	Roslyn	12.35% ↑	-1.52% ↑	0% –	75.06% ↑
	Rust	7.72% ↑	-2.11% ↑	11.84% ↓	92.09% ↑
	Kubernetes	13.97% ↑	-3.06% ↑	-11.27% ↑	104.31% ↑
LearnRec	Roslyn	-23.85% ↓	-34.77% ↑	138.84% ↓	-36.20% ↓
	Rust	-50.27% ↓	-50.26% ↑	122.63% ↓	-61.44% ↓
	Kubernetes	-34.98% ↓	-34.55% ↑	49.1% ↓	-46.38% ↓
RetentionRec	Roslyn	22.92% ↑	20.36% ↓	-23.89% ↑	-27.22% ↓
	Rust	13.38% ↑	15.70% ↓	-16.86% ↑	-4.56% ↓
	Kubernetes	19.75% ↑	47.78% ↓	-20.94% ↑	-37.07% ↓
TurnoverRec	Roslyn	-14.66% ↓	0.67% ↓	-38.33% ↑	-33.51% ↓
	Rust	-34.21% ↓	-4.38% ↑	-23.66% ↑	-53.43% ↓
	Kubernetes	-25.72% ↓	-0.09% ↑	-30.32% ↑	-44.49% ↓
Sofia	Roslyn	7.38% ↑	4.03% ↓	-34.9% ↑	55.22% ↑
	Rust	4.97% ↑	0% -	-25.42% ↑	73.09% ↑
	Kubernetes	9.42% ↑	1.70% ↓	-28.67% ↑	96.74% ↑

5.5.1 RQ1: How do existing code reviewer recommenders perform with respect to the risk of inducing future fixes?

In this experiment, we seek to determine whether reviewer recommenders mitigate the risk of inducing future fixes by introducing an evaluation measure (*CSR*).

Approach

For each studied system, we analyze the historical data and fit one model per quarter to estimate the likelihood that a PR is fix-inducing. Then, starting from the second quarter, we use a model fit of the previous quarter to estimate the fix-inducing likelihood of each PR. We use PR metrics listed in Table 5.2 as the model's input. We then rank potential reviewers for each PR using the seven baseline recommendation approaches. For every PR in each studied system, we swap one of the actual reviewers with our top candidate and evaluate the performance of this change by calculating the *MeasureChange* according to Equation 5.7.

Results

Table 5.3 presents the results of this experiment. The up and down arrows next to the numbers indicate performance improvement and degradation, respectively.

Analysis

For *AuthorshipRec*, code owners are predominantly assigned to reviews. Thus, increases to CSR are not unexpected, since coders owners are among the most knowledgeable contributors to whom reviewing tasks may be assigned. However, this assignment prevents others from learning about files they have not developed, which causes the *FaR* measure to degrade. For *RevOwnRec*, each studied system has a trusted developer circle for the reviews; hence this recommender fails to optimally distribute knowledge and improve *FaR*. Since these reviewers may not be the file owners, the *CSR* also tends to decrease or not to change considerably.

For *cHRev*, the score function is based on *xFactor*. Hence, the *CSR* is consistently improved, notably at the cost of limiting the improvement of *workload* for the core development team in comparison to other recommendation approaches.

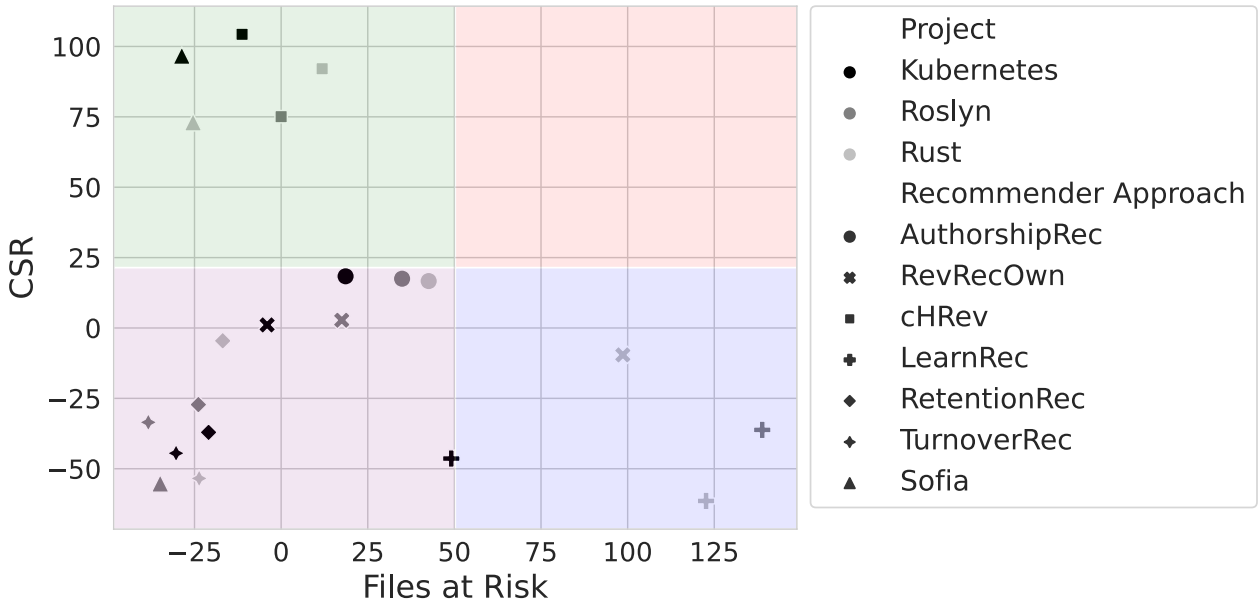
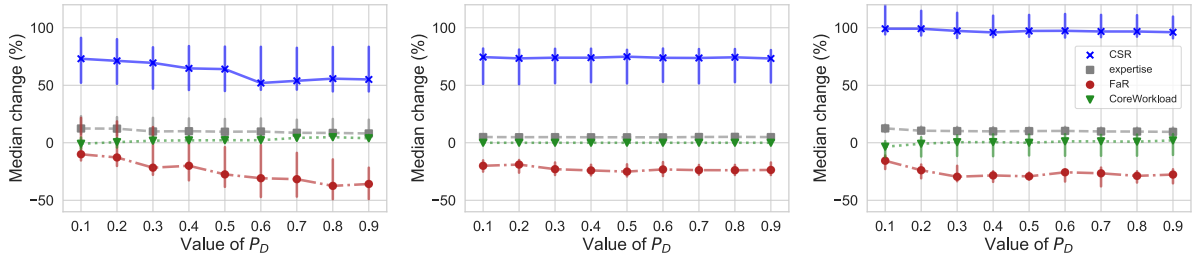


Figure 5.2: Relation analysis of CSR and *FaR*.

For *LearnRec*, there is no consideration for the retention of recommended candidates, so the *FaR* measure tends to increase because many reviewers leave the project. The suggested reviewers by this recommendation system are not experts, but seek to learn by reviewing the PR, so the *CSR* measure tends to decrease.

For *RetentionRec*, the recommender suggests candidates with the most knowledge about the project, not a specific PR. As a result, undesirably, the *core developers' workload* increases because they are mostly permanent developers of a project. However, their knowledge causes *CSR* and *expertise* to improve.



(a) Roslyn (b) Rust (c) Kubernetes

Figure 5.3: The effect of P_D on the performance of CSR for each evaluation metric, on different projects over different quarters.

For *TurnoverRec*, the recommender favours the most permanent candidates, regardless of the degree of knowledge that they have about the code being modified by a PR. This bias leads to knowledge retention, thus improving *FaR*. However, since distributing knowledge among developers is an important risk mitigation measure, the choice of less knowledgeable candidates causes the *CSR* to decrease.

For *Sofia*, when none of the changed files are at risk of turnover, *cHRev* is used. This compensates for *expertise* and *CSR* measures that are lost due to knowledge distribution caused by *TurnoverRec*. However, most of the time, this is at the cost of increasing the *workload* for the core development team. *Sofia* uses *TurnoverRec* for changesets with files at risk of knowledge loss, which has a favourable effect and causes the *FaR* measure to improve.

Figure 5.2 shows the relation between CSR and *FaR* in our experiments. The bottom-left quadrant shows evidence of a trade-off between CSR and *Files at Risk* for approaches that optimize only one characteristic. Meanwhile, cHRev and Sofia mostly present results in the top-left quadrant. This indicates that they are generally robust to the trade-off between CSR and *FaR*, and can broadly optimize both the risk of knowledge turnover and CSR. Finally, the bottom-right quadrant shows that optimizing for learning opportunities (e.g., using LearnRec) negatively impacts both *FaR* and CSR.

These observations indicate that if there is no deliberate effort to distribute knowledge, as the *FaR* improves, unless the necessary restrictions are put in place, such as a limitation on the most knowledgeable reviewers, the *CSR* degrade. This decrease, in turn, increases the chance of merging a fix-inducing PR into the project. This suggests that there is an inherent trade-off between the *FaR* and *CSR* evaluation measures. This does not hold in all cases. For example, in *LearnRec*, both *FaR* and *CSR* decreases which is likely because the recommended candidates leave the project as retention is not considered in the score function. Since leavers may leave a gap in the team understanding of an area of the codebase, the *FaR* and *CSR* measures tend to degrade. For *Sofia*, the recommender's candidate scoring function maximizes the expertise of the reviewers unless there is a file with few knowledgeable developers in the changeset. In these cases *Sofia* tries to distribute knowledge which lessens the *core workload* and improves the *FaR*. This active effort cancels out the native trade-off and improves both *FaR* and *CSR*.Sofia works better in terms of fix-inducing code changes, but like other approaches, it does not have any parameter to control the sensitivity to these changes. The inflexibility may become a barrier to adoption for this recommender as it cannot be tuned to suit the needs of users.

The evaluation results indicate that unless active effort is put into knowledge distribution while keeping the expertise high, the CSR and FaR have an innate trade-off. In cases where both CSR and FaR are maximized, other measures such as core developer workload suffer. Hence, one cannot simultaneously optimize suggested reviewers with respect to the risks of knowledge loss and fix-inducing changes.

5.5.2 RQ2: How can the risk of fix-inducing code changes be effectively balanced with other quantities of interest?

To balance the innate trade-off between *FaR* and *CSR*, we suggest using a hybrid reviewer recommendation approach to optimize the recommendation process based on the PR fix-inducing likelihood. We propose a recommender to improve the *CSR* when a PR has a high risk of being fix-inducing. The objective function for the proposed Changeset Safety Ratio (CSR) is formulated as:

$$RAR(D,R) = \begin{cases} Sofia(D,R), & DefectProb(R) \leq P_D \\ cHRev(D,R), & otherwise \end{cases} \tag{5.8}$$

In this formula, the P_D represents the threshold for the likelihood of PRs to be fix-inducing. If the P_D threshold is exceeded, *cHRev* is used to suggest experts. Otherwise, *Sofia* will suggest reviewers for the PR. The *cHRev* ranks candidate reviewers based on their familiarity with the changed files while *Sofia* opportunistically distributes knowledge when the modified files are not at risk of turnover.

Approach

We study the performance of *CSR* in terms of the *coreWorkload*, *FaR*, *expertise*, and *CSR* measures. We also study the impact that varying the P_D threshold from 0.1 to 0.9 has on CSR performance.

Results

Figure 5.3 shows the evaluation measures as the P_D changes for the studied systems.

Analysis

Figure 5.3 shows that as the value of P_D increases, the tolerance of *CSR* for fix-inducing PR grows. As a result, we expect more knowledge distribution leading to a decrease in *CSR*. As fewer experts are assigned to the tasks, the overall *expertise* also diminishes, which is not an unexpected outcome.

However, there are project-specific trends that are worth noting. For example, Figure 5.3a shows that the evaluation measures for the Roslyn project are steadily declining as P_D increases, whereas Figure 5.3c shows that the majority of the impact of varying P_D in the Kubernetes project takes place between $P_D = 0.1$ and $P_D = 0.3$. Moreover, Figure 5.3b shows that for Rust, the impact of varying P_D is relatively small. Overall, the Changeset Safety Ratio (CSR) yields an average change of 12.48%, 0.93%, -19.39% and 80.00% over different quarters for evaluation measures of Expertise, Core workload, *FaR* and CSR, respectively.

A closer look at the model estimates of the likelihood of fix-inducing changes helps to explain these project-specific trends. Figure 5.4 shows the distributions of the estimated likelihood of changes being fix inducing stratified by project and quarter for four quarters. The complete distribution can be found in Figure B.1 in Appendix B. We observe that, unsurprisingly, larger performance fluctuations in Figure 5.3 are associated with the P_D values where the majority of the estimated likelihoods lie in Figure 5.4. Moreover, despite an overall decreasing trend in terms of the likelihood of fix-inducing changes over time, the trend of each quarter is similar to its adjacent quarters. This local similarity may help stakeholders to effectively tune P_D values (see RQ3 for a more detailed analysis).

The CSR settings can be tuned to balance the risks of knowledge loss and fix-inducing changes. Indeed, as the threshold for indicating tolerance of the risk of fix-inducing changes increases, the risk of knowledge loss impacts fewer files. However, identifying the optimal threshold setting requires an awareness of project-specific trends in the model estimates of the likelihood of fix-inducing changes.

5.5.3 RQ3: How can we identify an effective fix-inducing likelihood threshold (P_D) interval for a given project?

Our analysis from RQ2 indicates that the performance of the *CSR* is sensitive to the P_D setting. The effective range of P_D is dependent on the past likelihood of fix-inducing

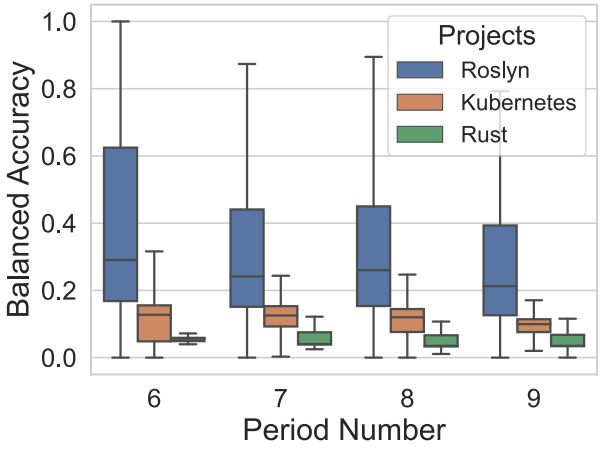


Figure 5.4: Distributions of predicted defect probabilities.

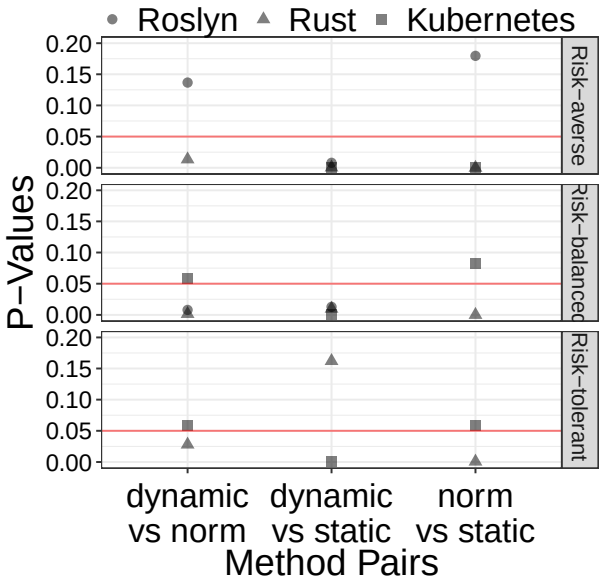


Figure 5.5: Conover Test results.

changes. In this question, we seek to propose an approach to help project stakeholders in the selection of effective P_D settings based on their tolerance for the risk of fix-inducing changes.

Approach

We explore the following three approaches to identify effective periods:

- *Static method*: This baseline considers the effective period spans the entire range between 0 and 1.
- *Normalization method*: The effective range spans between upper and lower extremes of the distribution of likelihoods from the prior periods. To match common outlier definitions, we set out lower and upper extremes to $Q_1 - 1.5 \times IQR$ and $Q_3 + 1.5 \times IQR$, respectively, where Q_i is the ith Quartile, and IQR is the Interquartile range between Q_1 and Q_3. All examples within the range are normalized by the maximum value.
- *Dynamic method*: A selective variant of the normalization method. Instead of considering all previous periods, we only consider the last six months. This allows the model to focus on the current part of the project life cycle.

For each of these three methods, we simulate three different thresholds: 25% (risk-averse recommendation), 50% (balanced recommendation), and 75% (risk-tolerant recommendation) of the effective period for our three projects in the dataset.

Results

Figure 5.6 shows distributions of relative improvement in *CSR* that are achieved for different time periods (points) of the studied systems (plot columns) of our approaches (y-axes) in different configurations (plot rows).

Analysis

We use the Friedman test (two-tailed, paired, $\alpha = 0.05$) [46] and apply it to the *CSR* performance data (Table 5.4). We observe significant differences between the investigated methods in all configurations except for Roslyn in the risk tolerant setting. Next, we use Kendall's W to determine the magnitude of this effect [71] (Table 5.5). Large and small effects are observed in 55% and 22% of the cases, respectively.

We apply the Conover test to discern which pairs cause this significant difference [26]. Figure 5.5 shows p-values for different thresholds with red lines indicating the 0.05 confidence interval. The results imply that the dynamic method significantly affects risk-averse ($P_D = 25\%$) and risk-balanced ($P_D = 50\%$) recommendations in all studied systems. For the normalization method, the effect on the results is inconsistent. The dynamic method considers the pivot of the project in various periods, which affects the *CSR*. In contrast, the normalization method considers the entire history and may not be sensitive enough to react to risk fluctuations as projects age [104].

On the other hand, for risk-tolerant recommendations ($P_D = 75\%$), none of the methods have a consistent effect on the results due to the difference in the distribution of defect proneness for various periods. Roslyn has a high rate of fix-inducing PRs ($P_D > 0.5$) in all the periods, so the approach does not affect the results. However, Kubernetes, which has more fix-inducing PRs in the earlier periods than more recent ones, is affected mainly through a dynamic method.

For risk-averse ($P_D = 25\%$) and risk-balanced ($P_D = 50\%$) recommendations, the dynamic method tends to provide the most value by recommending an effective period while for risk-tolerant recommendation ($P_D = 75\%$), none of the methods outperform others significantly.

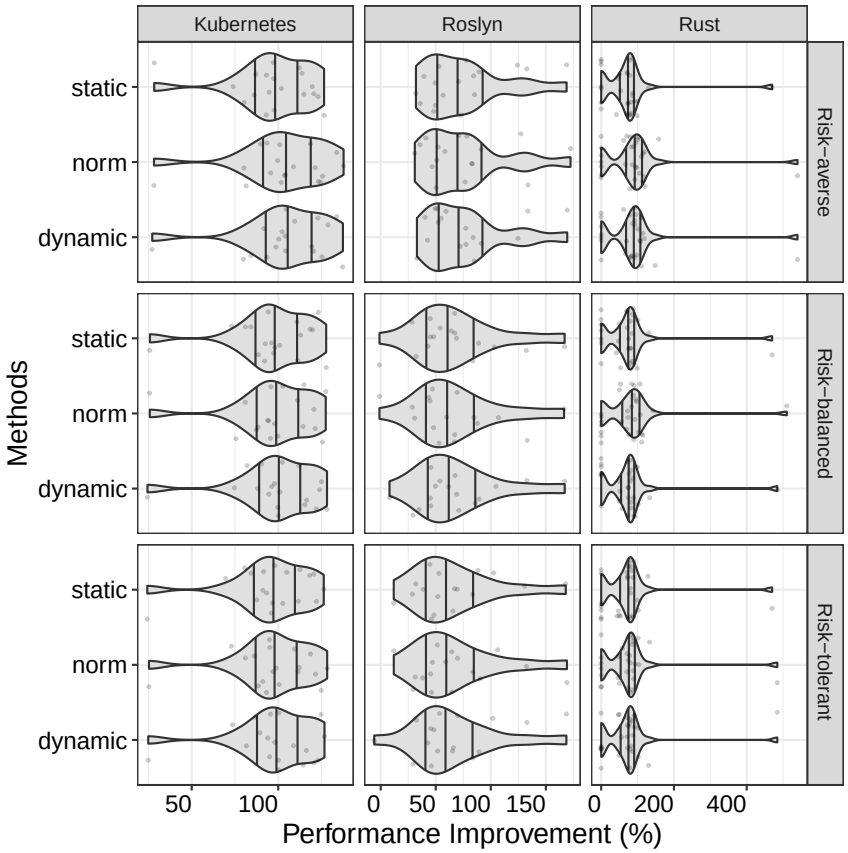


Figure 5.6: The distribution of performance improvement for CSR for different project over time.

5.6 Practical Implications

Below, we summarize what we believe are the practical implications of greatest value for practitioners and researchers.

RQ1) Practitioners can use code review to balance files at risk of abandonment with the risk of fix-inducing changes. Our observations in RQ1 show that if the likelihood of a PR inducing a fix is not considered explicitly as a parameter in the recommenders' objective function, the recommended reviewers may lack the subject matter expertise to prevent future fixes, and in turn, increase the risk of merging fix-inducing PRs. The results also show an inherent trade-off between some of the evaluation measures,

Table 5.4: The χ^2 and p-value results (two degrees of freedom) of the Friedman test applied to the RQ3 values.

Project \ Threshold	25%		50%		75%	
	Chi-Square	p-value	Chi-Square	p-value	Chi-Square	p-value
Roslyn	10.7	0.00473	12.8	0.00164	2.47	0.291
Rust	41.7	< 0.001	38.6	< 0.001	15.8	< 0.001
Kubernetes	23.1	< 0.001	16.5	< 0.001	20.6	< 0.001

Table 5.5: Effect size and magnitude for Kendall's W (RQ3).

Project \ Threshold	25%		50%		75%	
	Effsize	magnitude	Effsize	magnitude	Effsize	magnitude
Roslyn	0.315	moderate	0.377	moderate	0.0727	small
Rust	0.695	large	0.643	large	0.264	small
Kubernetes	0.607	large	0.435	moderate	0.543	large

such as *FaR*, and the risk of merging fix-inducing PRs. We propose *CSR* as a heuristic to assess the degree to which a (recommended) reviewer assignment mitigates the risk of fix-inducing changes.

RQ2,RQ3) *CSR* can be tuned according to the tolerance of the risk of fix-inducing changes without drastically impacting other properties of interest of the recommended reviewing assignment. Our observations in the first research question indicate that active effort should be made to mitigate the inherent trade-off between *CSR* and *FaR*. To this end, *CSR* is proposed, which uses the P_D setting, as the threshold for the likelihood of a PR being fix inducing, to influence the suggested set of reviewers. The results of the second research question illustrate that *CSR* prevents other evaluation measures from being drastically impacted. The P_D setting can be tuned using a combination of our proposed dynamic method (see RQ3) and input from stakeholders about their tolerance of risk for fix-inducing changes. While project-specific characteristics (e.g., the incidence rate of fix-inducing changes) impact the sensitivity of the approach to the P_D setting, our dynamic approach can be scaled to apply well in different risk tolerance settings.

5.7 Threats to Validity

Below, we discuss the threats to the validity of our study.

Construct Validity. Our implementations may contain errors. To mitigate this risk, we augment an existing data set and vetted code from prior work [109] rather than producing our own from scratch. We share our implementation openly to enable the community to audit and build upon our code [2].

It is also possible that *CSR* does not truly reflect how well fix-inducing code changes are mitigated when assigning reviewers in reality. Because we cannot go back in time and change existing assignments to observe how well *CSR* truly performs, we evaluate its performance using historical data. We mitigate the chances of *CSR* being a poor refection of reality by basing it on proven measurements such as the fix-inducing likelihood and the xFactor [183]. Furthermore, the main idea behind *CSR*, that experts that have recently interacted with files in a code change reduce the likelihood of merging fix-inducing code changes, has been shown to reflect reality in prior studies [16].

To obtain data at a scale required for this study we must use automated tools. However, such approaches are not perfect and may induce errors in our results. To prevent any implementation errors, we use an existing tool (Commit Guru). We sampled the tool's output and manually verified the results. The resulting precision (i.e., 43.9% with confidence=95% and margin=±5%), aligns with prior works [123, 124]. While SZZ may introduce errors into our dataset, our results show that reviewer recommendations can still suggest the most relevant reviewer to reduce fix inducing changes, even when trained on noisy data. Future tools could be used to improve the performance of the approach.

Internal Validity. In this chapter, we consider the effect of assigning experts to review PRs that are potentially fix-inducing using measures, such as *CSR*. While assigning experts rather than novices to review PRs may change such measures, it does not guarantee that they will actually spot more defects. It is possible that other factors, that do not reflect a reduction in defects, are influencing the changes in *CSR*. However, prior studies have shown that experts increase the possibility of detecting fix-inducing PRs before merging, we therefore believe that similar outcomes should hold for our study. Further studies might help to clearly identify the impact of reviewers' experience and *CSR* on catching bugs during the PR process.

The defect prediction in Rust presents a low balanced accuracy. However, the other two projects yield similar results in different experiments, which we believe voids the possibility of the effect of this low accuracy in our experiments.

[2] https://github.com/software-rebels/RAR_Recommender

External Validity. While we apply eight different approaches to three systems, it is possible that our results might not generalize to other approaches or systems. We mitigate this threat by using a large number of approaches and systems with many files and a high volume of PRs. We target such systems because reviewer recommenders are most beneficial in big repositories with many developers. Through this selection we aim to make our findings applicable to the most pertinent systems.

5.8 Conclusions

In this chapter, we set out to explore how using a code reviewer recommender to suggest reviewers can affect the risk of defect proneness. To this end, we introduce a new evaluation measure, *CSR*, and assess seven existing reviewer recommenders against this new measure. Three other measures previously used in the literature are also compared. The results show an inherent trade-off between *FaR* and *CSR* – improvements to one measure often degrade the performance of the other. To balance this trade-off, an adjustable multi-objective code reviewer recommender, *CSR* is proposed. We analyze how *CSR* can be used to tune the recommendations with respect to the tolerance of the risks of fix-inducing PRs and files at risk of knowledge loss. While assigning reviews to project experts can maintain a high CSR, this approach can lead to an increased concentration of knowledge, as indicated by FaR, and elevate the workload on the core team. This represents a potential pitfall that practitioners should be aware of. Although our adjustable CRR approach assists project maintainers in finding an optimal balance based on their risk tolerance, completely mitigating this issue is challenging, as improving one aspect will adversely affect the other. Our findings suggest that:

- There is a trade-off between knowledge distribution and the likelihood of merged PRs being fix-inducing. However, this trade-off may be resolved by simultaneously optimizing recommendation strategies for both measures. This optimization, in turn, may lead to a decrease of other evaluation measures like core developers' workload.
- *CSR* can be tuned to balance the risk of knowledge loss and fix-inducing changes by tuning the P_D setting. However, identifying the optimal threshold setting requires an awareness of project-specific trends in the model estimates of the likelihood of fix-inducing changes. The results yield the average change of 12.48%, 0.93%, -19.39% and 80.00% over different quarters for evaluation measures *Expertise*, *Core workload*, FaR and *CSR*, respectively. For the proposed measure, *CSR*, the average change is 73.80%-102.04% for various P_D settings.

- Project stakeholders can use *CSR* with a dynamic method for identifying effective range for the P_D setting. The dynamic method provides better performance for risk-averse and risk-balanced reviewer recommendation strategies while not hurting the risk-tolerant strategy's performance.

5.9 Chapter Summary

In this chapter, we discuss another challenge in using CRR systems: the quality of recommended reviewers. Our study demonstrates that accepting CRR system suggestions can increase defect proneness, potentially compromising project safety. We then present an adjustable mitigation strategy that works well with existing approaches and two dynamic methods for configuring settings based on the project's state. In the next chapter, we explore Automatic Code Review (ACR) systems and assess the relevance of human reviewers and these tools in the current ACR process.

Part III

Relevance of Code Reviewer Recommendation Systems

Chapter 6

Studying the Interrogative Comments Posed by Review Comment Generators

> **Note.** An earlier version of the work in this chapter is under submission in the IEEE Transactions on Software Engineering (TSE) Journal.

6.1 Introduction

Despite many benefits, code review is a time-consuming [17, 77] and error-prone [113] process. With the emergence new AI techniques, some tool builders aim to automate the code review process, potentially excluding human involvement. Consequently, certain tools like Code Reviewer Recommendation (CRR) systems may become obsolete. These efforts led to the creation of automation tools such as Review Comment Generator (RCG), which can automatically generate code review comments [61, 147]. The goal of RCGs is to provide more timely, consistent, and objective feedback than human reviewers [161].

Although RCGs aim to emulate human reviewers in comment generation, they are not without limitations. Indeed, a common type of comment is *interrogative*, i.e., asking questions of other review participants [188]. While state-of-the-art task-specific RCGs [89, 90, 162] may pose questions, they cannot comprehend the author responses, and hence, cannot follow up like human reviewers. While, Large Language Model (LLM) can potentially

follow-up the review discussion threads and mitigate this limitation, incorrectly answering author responses can hinder useful discussions [169], leading to less productive code reviews.

Prior work has shown that RCGs do generate interrogative comments [188], but despite their importance, a comprehensive study is still lacking [35]. Our study aims to fill this gap by conducting quantitative and qualitative analyses of RCG-generated interrogative comments. More specifically, we *quantify* the prevalence and regularity of RCG-generated interrogative comments, and *characterize* their similarity to those of humans. For this chapter, we use code review records from October 2018 to August 2023 to build a dataset from the Gerrit project,[1] which maintains a high standard for code review, resulting in 172,919 code review comments.

- **Quantitative Analyses (Section 6.3).** We find that a median of 15.61% of comments generated by task-specific RCGs are interrogative, whereas 65.26% and 47.67% of DBRX and LLaMA2 comments (i.e., LLM-based RCGs) are interrogative, respectively. Comparatively, 39.91% of human-submitted comments in our corpus are interrogative, suggesting that task-specific RCGs do not pose questions frequently enough, and that LLM-based RCGs may be overcorrecting. We also study whether RCGs and humans ask questions about same code changes using Fisher's exact test. The result reveals that, indeed, there is an association among RCG-generated and human-submitted comments in their mood, i.e., whether the comment is declarative or interrogative. Finally, despite the critical nature of discussion-inducing code changes, i.e., code changes with discussion threads that start with interrogative comments, we find that RCG behaviour in these scenarios is irregular and erratic, i.e., the rate of RCG-generated interrogative comments varies more across the studied Merge Request (MR) instances.
- **Qualitative Analysis (Section 6.4).** We observe that a considerably larger share of questions (13.86%–22.11%) focus on the rationale for code changes when RCGs generate comments rather than humans (1.94%). Humans discuss logical code flow more often (54.37% vs. RCGs 38.64% to 42.17%) and predominantly use questions for suggestions (63.11%), whereas RCGs tend to request additional context (56.84%–84.09%). Furthermore, humans often employ rhetorical questions (8.74%) and hypotheticals (4.85%), whereas task-specific RCGs do not. Our LLM-based RCG experiments show that LLMs outperform task-specific RCGs in hypothetical inquiries (12.65% for GPT-4), but lag in rhetorical questioning (2.27% for LLaMA2). Also, we

[1] https://gerrit-review.googlesource.com/

propose a discussion thread response generation as a new ACRs task, and evaluated the performance of LLM-based RCGs on this task. Our results shows that LLM-based RCGs can be helpful in identifying whether the discussion is resolve. However, if they need to follow up with the author, their performance is sub-optimal.

We conclude that while state-of-the-art RCGs could aid the review process by improving tasks like exception handling, they cannot replace humans in generating interrogative review comments. Specifically, they exhibit a disparity with respect to human behaviour in generating interrogative comments, their limited linguistic ability impacts comment quality, and they have a tendency to request information without offering as many recommendations as humans. Although LLM-based RCGs show promise in addressing some of the linguistic limitations of RCGs, they do not consistently improve performance across different types of interrogative comments. We recommend using RCGs as complementary tools rather than as replacements for human reviewers. Thus, their benefits can be harnessed while mitigating their shortcomings.

6.2 Dataset Preparation

In this section, we describe how we create dataset for our analysis. Figure 6.1 shows the steps which consists of: (1) Data Collection, (2) Data Cleaning, and (3) Review Generation components. Below, we describe each component.

6.2.1 Data Collection

We begin by selecting a subject community on which to focus our efforts. Our study aims to use RCGs trained on human code reviews to generate code review comments, allowing us to analyze the prevalence and patterns of questions that are generated. To that end, we must first select a development community that produces a large number of high-quality human-submitted code reviews. Human-submitted code reviews are necessary both to fine-tune RCGs and to compare their results against a baseline. While multiple communities may satisfy our criteria, we choose the Gerrit community for our study because it provides us with the opportunity to analyze how RCGs perform in a near-ideal case. Indeed, the Gerrit community contains a large amount of review data (1,852 code reviews and 15,000 code review comments in 2022), has contributors representing organizations of influence (e.g., Google, Cisco, and Spotify), and tends to provide review records that are well linked

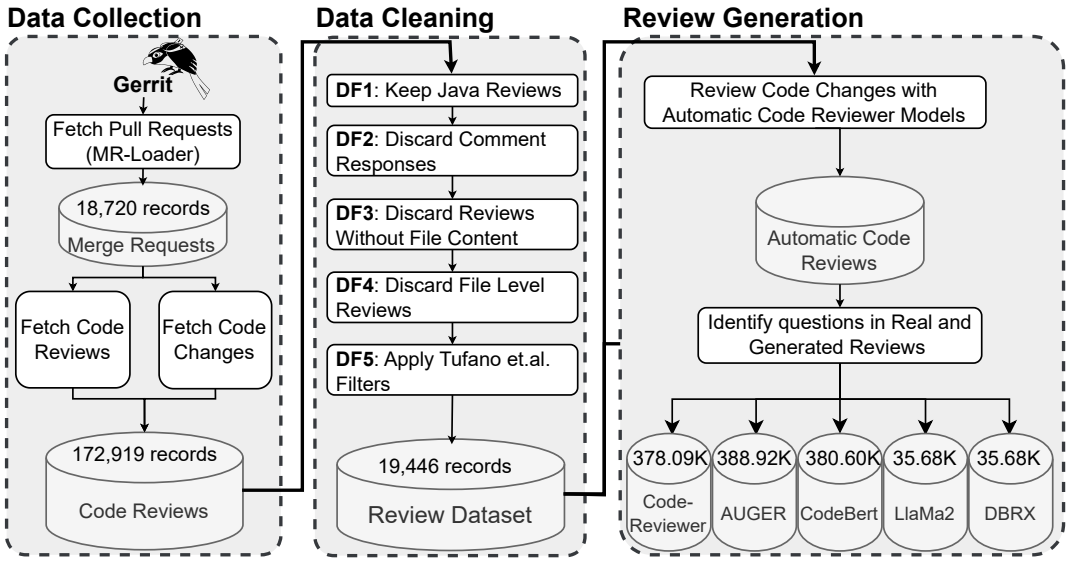


Figure 6.1: The overview of the data preparation procedure.

to the commits on which they were performed. Focusing on the Gerrit community does pose clear risks to external validity but allows control over internal validity threats, such as understanding of data linkage properties, highlighting a subjective trade-off between these validity types among researchers as studied previously [142]. Furthermore, the Gerrit community has been the subject of previous studies [24] as the its maintainers exhibit a stringent commitment to practicing code review, and comments consistently display both quality and quantity.

To start our data collection, we use MR-Loader[2] to obtain the Gerrit community's raw MR data. As the primary quality gate through which all code changes of the community flow, MRs are a prime source of data for our study. Indeed, all the details related to a change, including the code review comments, can be accessed through MRs. We also collect metadata, such as the author and committer of each code change, to use in our data analysis.

We collect this data from the Gerrit community from the first available MR (2018-10-21) until the date when we performed the data collection (2023-08-22). Using this data, we filter out MRs from projects unrelated to the development of the Gerrit system itself.

[2] https://github.com/JetBrains-Research/MR-loader

This results in a dataset containing 172,919 code review comments. To complement the MR data, we leverage the Gerrit API[3] to download the content of the files under discussion before a change and the code diff on which the reviewers commented.

6.2.2 Data Cleaning

After we collect our data, we apply five *Data Filters* (DFs) to mitigate noise that is irrelevant to our study (e.g., comments like "lgtm"). We present our five DFs below.

- **DF1**: We filter out comments left on files other than Java files by identifying files with the `.java` extension. We concentrate on Java files because two of the three studied RCGs are trained on Java code review data and provide feedback only on Java code changes. 77,529 code review comments survive this filter.
- **DF2**: We filter out response comments, i.e. comments posted in reply to another comment, using a Gerrit API flag that indicates whether a comment is in reply to another comment. We remove response comments because non LLM-based RCGs only review code changes and do not engage in follow-up discussions. Similar filters are common in previous studies [89, 162] to prepare the data for training and inference. After **DF2**, our dataset contains 37,656 code review comments that survive.
- **DF3**: We filter out records that do not have retrievable file content before the code change. For each code review comment, we issue a Gerrit API request to retrieve related file content. We remove the small subset of cases where the API request is unsuccessful because the data cannot be processed by the studied RCGs. After **DF3**, our dataset contains 37,625 code review comments.
- **DF4**: We filter out file-level reviews, i.e. comments that are not connected to any line of code but are instead about a higher-level concept in that file.[4] We do this by identifying the comments that have a non-null file name yet no comment lines specified. We remove these code review records because existing RCGs perform the review on functions, sub-functions blocks, or lines of code. Therefore, these high-level comments on the files cannot be produced by the studied RCGs. 37,133 code review comments survive this filter.

[3] https://gerrit-review.googlesource.com/Documentation/

[4] Sample comment: https://gerrit-review.googlesource.com/c/gerrit/+/351075/comment/2408abef_e5ea576c/

- **DF5**: Our final filter preprocesses the comments using the replication packages from Tufano *et al.* [162]. Specifically, we use their `code/Analyzer.py` and `code/Cleaner.py` files.[5] This filters out non-English comments and comments on code portions that are not part of a function, such as comments on imports. This filter also removes empty comments after the removal of emojis and links. In the end, 19,446 code review comment records survive this filter.

We do not apply restrictions on comment length since (1) any comment may contain important information, and (2) the main objective of this dataset is to study generated code review comments, not training RCGs. Our dataset and the resulting 19,446 code review records are available online in our replication package.[6]

6.2.3 Code Review Comment Generation

Using the dataset collected in our data preparation steps, we use the selected task-specific and LLMs-based RCGs to generate code review comments based on the given code changes. Task-specific RCGs are trained to excel in comment generation with a specific input format and have not undergone Reinforcement Learning from Human Feedback (RLHF), a technique used to align intelligent agents with human preferences. On the other hand, LLMs are more generalist models that interact through natural language interfaces but, as their name suggests, typically require more resources. Below, we elaborate on each type in more detail.

Task-specific RCGs

The selected task-specific RCGs differ in their input format. While both AUGER and CodeReviewer adopted the Text-To-Text-Transfer Transformer (T5) model [125], each chooses one of its many existing variations [98], which require different inputs. AUGER's input requires a function that contains commented code, allowing it to use this context to comment on issues at the function level [89]. CodeReviewer, on the other hand, uses inputs in the form of code diffs. Thus, while we use off-the-shelf versions of AUGER [89] and CodeReviewer [90], we tailor the inputs to each model. Unlike CodeReviewer and AUGER, CodeBert, as a general-purpose model, can ingest a code change as-is; however, it should be further fine-tuned [162]. Thus, we use the pre-trained CodeBert model and

[5] https://github.com/RosaliaTufano/code_review_automation
[6] https://doi.org/10.5281/zenodo.13301078

train it for another 50,000 epochs on the Tufano *et al.* dataset [162] to perform code reviews. To prevent data leakage from fine-tuning to testing phase, we remove the duplicate common entries when evaluating our results for this model. Furthermore, to minimize implementation errors, we use the code provided by Zhou *et al.* [188] and only modify it to add top-k sampling [41].

Our diverse selection of RCGs allows us to generate a wide range of code review comments. Similar to prior work [89, 90], we run experiments for all of our studied RCGs on our dataset for top-k results. We use k values of 1, 3, 6, and 10, similar to prior work [89].

Next, we use a rule-based heuristic to identify whether the comment mood is interrogative, building upon the rules introduced by Zhou *et al.* [188] and add NLTK[7] to improve the accuracy by separating sentences in a review comment. The source code of the heuristic used in this chapter is publicly accessible as part of our replication package.[6] To verify the performance of this heuristic, we draw a random sample of 377 comments and inspect the output of the heuristic for each one manually. We find that the heuristic yields an AUC of 0.96 for human review comments (95% confidence level, 5% error margin). Using this heuristic, we individually classify both the actual comments and each of the top-k generated code review comments of each of the studied RCGs. After collecting and classifying the model outputs as interrogative or declarative, we analyze them to address our RQs.

Large Language Model

For code review comment, each LLM is run with the $k = 1$ setting, prompted to adopt the persona of a code reviewer [172], and leverages two-shot prompting [28, 120] to enhance model efficacy. We provide two examples, an interrogative and a declarative comment, to prevent model bias and pose questions only when necessary. These examples are selected from a randomly chosen set of code reviews and inspected to ensure that the issues raised in the comments are confined to the code change and do not reference materials outside the scope of the change. We choose not to fine-tune our chosen LLMs since they have already been trained on extensive code review data. The input comprises a code hunk, similar to CodeReviewer [90], which is known for its superior performance in comment generation among RCGs [99]. To ensure an unbiased comparison, no additional context is provided. After presenting the prompt and code change, we record the LLM's output for further analysis.

[7] https://www.nltk.org/

6.2.4 Discussion Thread Response Generation

While RCGs in current form cannot follow up on discussion threads, LLMs are well suited to the task. Therefore, we introduce a new task for automatic code review, discussion thread response generation, which refers to responding to the discussion thread after a new comment is placed if needed, or marking the thread as resolved. As this experiment required the interactive behaviour, which is an attribute of LLMs, we employ only our studied LLM-based RCGs. For this task, we select code changes accompanied by review threads initiated with interrogative comments. We provide each model with the code change hunk, commented code, reviewer's comment, and author's response, instructing the LLM to determine whether the initial concern is resolved and, if needed, to continue the discussion and generate a response. Then, we record the LLM-generated response and compare it to the actual progression of the thread, where the two first authors assess the similarity of the comment using a five-point Likert scale.

6.3 Quantitative Analyses

In this section, we quantitatively study interrogative comments generated by RCGs during code reviews. Below, we describe our approach and then present the results.

6.3.1 Approach

We first evaluate the prevalence of RCG-generated interrogative comments by measuring their frequency. Then, to measure the similarity of interrogative comments generated by RCGs and humans, we calculate the density of interrogative comments per MR for both RCGs and human reviewers, i.e., the rate at which interrogative comments are posed in each MR. We report the results of our analysis using top-k values ($k \in 1, 3, 6, 10$ for task-specific and k=1 for LLM-based models) for each of our studied task-specific RCGs to explore the impact of k on their behaviour. We test various k settings because higher k settings improve the quality of comments by providing a more lax guess budget and increasing the diversity of suggestions [41]. Experimenting with different k settings [89, 162] is common in prior work, with larger k settings typically yielding better performance.

We explore whether RCGs imitate human behaviour when posing interrogative comments. To evaluate the association between comments generated by RCG and those posed

by humans, Fisher's exact test [164] is employed. Additionally, to compare the density of interrogative comments generated by RCGs and humans across the MRs, the Mann-Whitney U test [100] is used. Also, Shannon's entropy [140] is used to measure the regularity of RCG behaviour.

When we evaluate the prevalence of LLM-generated interrogative comments, we focus on the k=1 setting because larger k settings incur greater hardware costs. To provide a rough estimate, running all dataset records with the k=1 setting of the GPT-4 Turbo LLM would cost approximately $2.3K USD, assuming that only a single run was necessary. For these practical reasons, we limit our LLM experiment to only one value of k and use DBRX as a freely available alternative for GPT-4. Similar to the evaluation of task-specific RCGs, we study the association between the mood of comments produced by humans and LLMs. We use Fisher's exact test with alpha set to 0.0036, adjusted using the Bonferroni correction [33].

We also conduct an analysis of code review comments with discussion threads that have at least one response because these discussions are (1) closely linked to questions during the code review process [35], and (2) play an essential role in knowledge sharing and design conversations within the code review process [6, 169, 182]. Discussion-inducing code changes in the context of RCG-generated interrogative comments are of interest because RCGs have the potential to disrupt valuable code review discussions. Two scenarios may arise in such cases: (1) RCGs ask questions similar to humans, sparking productive discussions that require active RCG engagement after the authors respond, or (2) RCGs ask the different (perhaps even incorrect) questions, potentially stifling informative discussions. While the former scenario necessitates RCGs with a deep understanding of the code, strong reasoning, and comprehension capabilities, which are currently lacking, the latter is more detrimental, as it can hinder valuable project documentation [6]. We study whether LLMs can address this shortcoming by applying them to such cases and qualitatively analyzing their responses. Furthermore, we compare the behaviour of RCGs when face these code changes and inspect to see if RCGs also treat them differently, similar to humans. This comparison is crucial since these code changes have led to beneficial discussions, and treating them similarly to other changes may indicate oppressing useful discussions that could have started otherwise.

Identify Discussion Inducing Changes

To identify these code changes, we first extract comment threads, specifically those with more than one reply from the code review records. During this process, we filter out single-word responses to focus on informative discussions related to code changes, excluding minimal responses like "Acknowledged."[8] We then flag the corresponding code and the initial comment that initiated the changes as discussion-inducing.

6.3.2 Results

Tables 6.1 and 6.3 show the prevalence of interrogative comments and the regularity of RCG behaviour in generating them across k settings for the studied RCGs. Below, we present our observations.

How often do Review Comment Generators generate interrogative comments?

15.61% of comments generated by the task-specific RCGs are interrogative. Interrogative review comments account for 0.91% to 27.54% of all generated comments, with medians of 15.78%, 19.50%, and 10.15% for AUGER, CodeReviewer, and CodeBert, respectively. The overall median across all values of k is 15.61%. Regarding the LLM-based RCGs, 65.26% and 47.67% of the DBRX and LLaMA2-generated review comments are interrogative, respectively. To provide a benchmark, our mined dataset of human review comments from the Gerrit project contains interrogative comments at a median rate of 39.91%. Among the studied task-specific RCGs, AUGER has the highest interrogative comment frequency at 15.30% for k=1, with CodeReviewer and CodeBert at 4.11% and 0.91%, respectively. The range of these rates increase to between 6.07%–15.62% for k=3, 14.18%–23.38% for k=6, and 15.60%–27.54% for k=10. CodeBert's relative rate of interrogative comments, especially for k=1, can be attributed to its base model, which is primarily trained for generating code rather than natural language. Although pre-training CodeBert for 50K epochs enhances comment quality for larger k settings, when focusing only on the most likely solution (k=1), the generated comments mainly consist of code fragments with limited natural language text. Consequently, only a few would have questions within the CodeBert-generated comments.

[8] https://gerrit-review.googlesource.com/c/gerrit/+/430619/comments/c74c5adb_ebc49a32

LLM-based RCGs pose questions more frequently than task-specific models, especially in the case of DBRX. We suspect that this is due to their limited contextual knowledge about the given change hunks, which lead to confusion. We explore the intentions and types of questions that LLMs pose in more detail in Section 6.4.

We repeat the previous analysis focusing exclusively on discussion-inducing code changes. We find that the rate of interrogative comments ranges from 0.91% to 28.02%, with a median of 15.60% for all comments generated by task-specific RCGs. This indicates a slight increase in rates compared to our assessment of all comments. As for LLM-based RCGs, DBRX and LLaMA2 generate interrogative comments 67.62% and 46.64% of the time, indicating a +2.36 and -1.03 percentage point difference. As a benchmark, human-submitted comments in discussion-inducing code changes increased by 18.25 percentage points to reach 58.16% when all comments are considered. This result remarks that RCGs do not treat discussion-inducing changes, similar to humans.

Table 6.1 shows that larger k settings tend to produce higher rates of interrogative comments for CodeBert and CodeReviewer, whereas this rate remains relatively consistent with a variance of 13.95% for AUGER. This stable or increasing rate for RCGs highlights the prevalence of these comments, since the studied RCGs either always ask questions under all k settings at considerable rates (AUGER) or ask questions when one relaxes the constraints on the number of guesses (CodeBert and CoreReviewer). Thus, while interrogative comments posed by human reviewers typically invite authors to engage with reviewers, share knowledge, and defend decisions that were made during development [169], current RCGs are not poised to follow up on the response to interrogative comments. Therefore, having a high rate of interrogative comments may hinder the usefulness of RCGs in the code review process.

RCGs generate interrogative comments a median of 15.61% of the time for task-specific and 65.26% and 47.67% for DBRX and LLaMA2-based RCGs, respectively. Since current RCGs do not actively participate in discussions, the rates at which they pose interrogative comments may limit their capacity to generate productive conversations in the code review process.

Do Review Comment Generators and Human Reviewers Raise Interrogative Comments for Similar Code Changes?

To shed light on RCG behaviour, we examine whether RCGs behave similarly to humans by correlating their interrogative comments with those of humans. Fisher's exact test [164]

is used to assess the comment mood, checking for nonrandom associations between RCG-generated and human comments' mood. We repeat this for each top-k setting and apply the Bonferroni method [33] to correct for multiple comparisons.

Table 6.2 presents p-values and odds ratios from this test, revealing that two of the three studied task-specific RCGs exhibit an association between generated and human comments in terms of sentence mood for different k settings. Specifically, for k settings of 3, 6, and 10, CodeReviewer and for k settings 6 and 10 CodeBert show significant associations with human rates of interrogative comments, whereas AUGER shows no association for any k setting. While in all but one case the significant associations have odds ratios greater than one, they are all small numbers. This indicates that even when RCG-generated and human-submitted comment moods are correlated, the relationship is weak.

Regarding the LLM-based RCGs, our observations show that DBRX-generated comments significantly co-occur with human comments, unlike LLaMA2. We suspect that while enterprise LLM-based RCGs may ask questions in the same cases as humans, the problem of hindering beneficial discussions remains a concern unless they can also effectively follow up on discussion points. Furthermore, one should question the similarity between questions asked by the models and humans. We explore these aspects further in Section 6.4.

The association of comment mood between human-submitted and RCG-generated comments diminishes when only considering discussion-inducing changes. Table 6.2 shows that, compared to the same setting for all comments, either the significant association is lost or the odds ratio decreases. It means that comment moods differ more among discussion-inducing comments than among other review comments.

To complement our analysis,RCG interrogative comment density versus human reviewers. A Mann-Whitney U test shows significant differences in distributions between studied RCG- and human-submitted interrogative comments in all experiments. Contrasting this observation with the results on the association of the comment moods, it appears that RCGs, indeed, behave differently than humans when generating interrogative comments.

Table 6.1: Rate of generated interrogative comments for studied RCGs.

RCG \ Type	All	Discussion-inducing	top_k
AUGER	15.30%	15.52%	1
	16.14%	16.31%	3
	15.96%	15.89%	6
	15.60%	15.64%	10
CodeBert	0.91%	0.91%	1
	6.08%	6.22%	3
	14.21%	14.85%	6
	19.61%	20.57%	10
CodeReviewer	4.12%	4.18%	1
	15.62%	15.57%	3
	23.38%	24.00%	6
	27.54%	28.02%	10
DBRX	65.26%	67.62%	1
LLaMA2	47.67%	46.64%	1

Table 6.2: Odds ratios and Fisher's exact test p-values for RCGs are shown. A corrected p-value below 0.0018 (*) indicates significance. Ratios > 1 or < 1 imply positive or negative associations, respectively, with significant ones in bold.

	odds ratios								p-values							
	All comments				Discussion-Inducing comments				All comments				Discussion-Inducing comments			
	top-1	top-3	top-6	top-10	top-1	top-3	top-6	top-10	top-1	top-3	top-6	top-10	top-1	top-3	top-6	top-10
AUGER	1.0178	1.0001	1.0020	1.0038	1.0762	1.0153	1.0367	1.0378	6.6931e-01	1.0000e+00	9.0257e-01	7.6428e-01	2.7679e-01	6.9533e-01	1.7836e-01	7.5635e-02
CodeBert	1.1597	1.0564	**1.1150**	**1.0716**	1.1979	**1.1031**	**1.0973**	**1.0550**	5.4031e-02	1.9305e-03	**3.5884e-37***	**1.9010e-32***	1.8091e-01	**7.7971-04***	**3.2262e-11***	**1.7494e-08***
CodeReviewer	1.0061	**0.9613**	**1.0546**	**1.0270**	1.0122	**0.9296**	1.0165	1.0187	8.6718e-01	**8.5961e-04***	**1.1110e-13***	**3.9885e-07***	8.5674e-01	**1.5052e-04***	1.5752e-01	3.0259e-02
DBRX-Based	**1.1366**	-	-	-	1.1249	-	-	-	**2.26e-08***	-	-	-	0.0024	-	-	-
LLaMA2-Based	0.9653	-	-	-	0.9873	-	-	-	0.1074	-	-	-	0.7305	-	-	-

Table 6.3: Entropy of the density of interrogative comments for all and discussion-inducing code changes.

RCG \ Type	All		Discussion-inducing		top_k
	entropy	p-value	entropy	p-value	
AUGER	0.297	0.00	0.337	$< .001$	1.0
	0.431	$< .001$	0.456	$< .001$	3.0
	0.520	$< .001$	0.546	$< .001$	6.0
	0.578	$< .001$	0.593	$< .001$	10.0
CodeBert	0.068	0.00	0.073	0.00	1.0
	0.353	0.00	0.389	0.00	3.0
	0.553	$< .001$	0.576	$< .001$	6.0
	0.641	$< .001$	0.671	$< .001$	10.0
CodeReviewer	0.169	0.00	0.154	0.00	1.0
	0.470	$< .001$	0.497	$< .001$	3.0
	0.613	$< .001$	0.641	$< .001$	6.0
	0.673	$< .001$	0.707	$< .001$	10.0
DBRX	0.555	$< .001$	0.567	$< .001$	1.0
LLaMA2	0.402	$< .001$	0.399	$< .001$	1.0

We repeated this test for discussion-inducing code changes. Notably, the distribution of RCG-generated interrogative comments remains largely unchanged by filtering out the non-discussion-inducing code changes. To verify that RCGs, unlike humans, treat discussion-inducing code changes similar to the rest of the changes, we perform a Mann-Whitney U test over RCG-generated comments for task-specific changes and all the changes to explore whether the density of interrogative comments differs. The test yielded p-value=0.8852, indicating that the null hypothesis (i.e., both samples are drawn from the same distribution) cannot be rejected. Based on this observation, when faced with a discussion-inducing code change, RCGs treat them with no difference, leading to potentially missing out on useful discussion in code review.

Neither task-specific nor LLMs-based RCGs generate interrogative comments like humans. RCG-generated and human-submitted interrogative comments differ in terms of frequency of interrogative comment and their density per MR. Moreover, RCGs treat discussion-inducing changes similar to other changes, potentially diminishing the depth of review discussions.

When do Review Comment Generators generate interrogative comments?

Given that RCG-generated interrogative comments differ from the humans', we compute the entropy of normalized interrogative comments per MR, i.e., count of interrogative comments over the top-k, as a measure of regularity for task-specific RCGs. A higher entropy indicates that the number of RCG-generated interrogative comments varies more from one MR to another, making their overall behaviour more irregular. Table 6.3 presents the normalized entropy of interrogative comment density using Equation 6.1:

$$\text{Normalized Entropy} = \frac{-\sum_{i=1}^{n} rate_i \times \log_2 rate_i}{\log_2(\text{number of MRs})} \tag{6.1}$$

Furthermore, this table presents the p-values yielded from the Mann-Whitney U test to explore statistically whether the distribution of the generation rate of interrogative comments is different for RCGs and humans in each setting.

In Table 6.3, we observe that the p-value is less than the corrected alpha=0.0018 all the time, indicating that human-submitted and RCG-generated interrogative comments have different regularity in their rate per MR. Also, we notice an increase in k results in higher entropy for RCG-generated interrogative comments. While prior studies suggest that larger k settings lead to better performance [89, 162], predicting RCGs' behaviour in generating interrogative comments may be challenging due to this irregularity.

We separately investigate the regularity of RCGs for discussion-inducing changes. Higher entropy in interrogative comments, as shown in Table 6.3, suggests less regularity in generating them, thus having more challenge predicting the RCG behaviour.

While considering larger top-k suggestions leads to higher quality RCG-generated comments, it also increases the entropy of interrogative comments in task-specific RCGs, making it increasingly harder to speculate about the RCG behaviour. Thus, RCGs not only pose questions with which authors might not be able to interact, but they also pose them with patterns that differ from humans. Additionally, while predicting the behaviour of task-specific RCGs in generating interrogative comments for discussion-inducing code changes is more important than normal code changes, the higher entropy shows that it is, in fact, more challenging.

6.4 Qualitative Analyses

In this section, we study the types and intentions of the RCG-generated interrogative comments for both RCGs and LLMs by comparing them with human-submitted comments. We analyze the content of interrogative comments and categorize them to shed light on the questions-posing behaviour of RCGs and LLMs. Below, we describe our approach, followed by our results.

6.4.1 Approach

We adopt a catalogue of comment categories from prior work [35, 117] to compare the types and intentions of interrogative code review comments generated by RCGs with those of human reviewers. The catalogue is derived from two sources—one for comment types [117] and one for interrogative comment intentions [35]. The white rows of Table 6.4 present the twelve comment categories proposed by Ochodek *et al.* [117] to describe different types of code review comments, whereas the grey rows show the three new comment categories that emerged from our dataset but did not exist in the categories proposed by Ochodek *et al.* [117]. Similarly, Table 6.5 presents the five primary intents behind review queries that were identified by Ebert *et al.* [35]. We leverage the types of questions posed by RCGs to further uncover how RCG can aid in code review.

Since inspecting all of the interrogative comments is impractical, we draw a random sample containing both task-specific RCG- and human-submitted code review comments. We then apply blended coding [52] to the sample, initially using established categories for comment types [117] and intentions [35], while also integrating new categories to capture emergent trends. The following sections will detail our sampling method and the blended coding approach that we use.

Table 6.4: Code review comment types (Ochodek *et al.* [117]), with the inclusion of new types identified in our analysis (in gray).

Comment Type	Description
code design	Code review comments related to the structural organization of the code (e.g., class design).
code style	Code review comments pertaining to the code's layout and readability.
code naming	Code review comments focusing on the conventions used for naming variables, functions, classes, etc.
code logic	Code review comments that discuss the logic and operations within the code, such as algorithms.
code data	Code review comments that address the handling and usage of data (e.g., variables) within the code.
code api	Code review comments on the use and evolution of APIs within the codebase.
code doc	Code review comments that concern documentation and commentary in the source code.
compatibility	Comments related to compatibility with operating systems, tools, and various versions.
config/.../review	Code review comments about the process of submitting and reviewing patches and commits.
code purpose	Code review comments about the necessity for changes, typically to clarify the intention behind code segments.
code exception	Code review comments related to exception handling within the code.
code testing	Code review comments related to existing tests or the need for new tests.

Sampling

Similar to prior studies [89, 162], our analyses focus on initial review comments and exclude subsequent responses in the review threads. Given our goal of identifying the types and intentions of interrogative comments, our attention is narrowed to questions posed by either human reviewers or RCGs. In our dataset, the code review comments are categorized into three groups based on who produced the comment: (1) human reviewers only, (2) task-specific RCGs only, or (3) both human reviewers and task-specific RCGs. We randomly

Table 6.5: Code review comment intentions (Ebert *et al.* [35])

Intention	**Description**
Suggestions	Inquiries that subtly propose a course of action.
Requests	Questions seeking details such as explanation related to the code under review.
Hypothetical Scenarios	Questions that construct a potential situation which may not have been previously considered.
Rhetorical Questions	Questions paired immediately with their answers, serving to emphasize a point.

Table 6.6: Distribution of generated comment types for interrogative comments.

Comment Type	Human	RCGs	GPT-4	LLaMA2
code logic	54.37%	40.00%	42.17%	38.64%
code design	14.56%	7.37%	7.23%	6.82%
code naming	9.71%	8.42%	3.01%	2.27%
code data	3.88%	5.26%	4.82%	6.82%
code testing	3.88%	1.05%	3.61%	0.00%
code api	2.91%	2.11%	3.61%	0.00%
code exceptions	2.91%	7.37%	14.46%	20.45%
config/../review	1.94%	3.16%	0.60%	0.00%
code purpose	1.94%	22.11%	13.86%	15.91%
code style	1.94%	3.16%	3.61%	4.55%
code doc	0.97%	0.00%	2.41%	4.55%
compatibility	0.97%	0.00%	0.60%	0.00%

Table 6.7: Distribution of question intentions for generated interrogative comments.

Comment Intention	Human	RCGs	GPT-4	LLaMA2
Suggestions	63.11%	40.00%	14.46%	13.64%
Requests	23.30%	56.84%	72.89%	84.09%
Rhetorical questions	8.74%	3.16%	0.00%	2.27%
Hypothetical scenario	4.85%	0.00%	12.65%	0.00%

select samples from each category and merge these into a composite sample set to ensure representation from all types of review comments. We then use GPT-4- and LlAMA2-based RCGs to generate LLM-based responses for the sampled set. Moreover, to ensure that LLMs have not been exposed to the code changes during their training, we augment this set with 30 additional changes for LLM-based RCGs, balanced between interrogative and declarative comments, that were introduced after the cutoff date for the LLM training period at the time they are used.

Blended Coding

We employ a blended coding approach [52] to label the sampled set of interrogative comments. This strategy allows us to leverage categories from prior studies while we still have the flexibility to create new categories.

Two coders with related expertise inspect each entry in the sample, focusing on interrogative review comments by both reviewers and RCGs. The coders also label false positives (i.e., non-interrogative comments) as a separate class. Furthermore, for consistency, in multi-question comments, the coders only label the first question. As a first step, in a preliminary collaborative session, for the task-specific RCGs, the coders label 100 interrogative comments, separate from the sampled set, to lay the common ground for the initial categories. For LLM-based RCGs, coders label only 50 samples to refine guidelines, as these comments are more structured due to the superior language skills of LLMs. Subsequently, each coder independently labels the remaining comments in batches of 50, alternating between human reviewers and RCG-generated comments. In addition to the content of interrogative comments, the coders use the comment context, such as the referenced changes to the code, the type of RCG, and the responses to human reviewer comments. After each batch of 50 comments, the coders meet to resolve discrepancies and refine the emerging categories. For disagreements, coders explain their rationales and jointly decide. The first author arbitrates unresolved disputes. This iterative process con-

tinues until our saturation [15, 53] criterion is satisfied, i.e., no new codes are identified across two consecutive batches. This criterion is met after labelling 150 samples of entries.

To assess the reliability of the coding task, we measure the inter-rater reliability using Cohen's Kappa score across the two coders. We repeat the measurement for both distinct coding tasks for the four sources of comment generation, i.e., categorizing the type and intention of generated comments by task-specific RCGs, the GPT-based RCG, the LLaMA2-based RCG, as well as the human reviewers. For task pairs of comment type and interrogative intention, we obtain Kappa score pairs of (0.49, 0.45), (0.67, 0.66), (0.60, 0.50), and (0.61, 0.43) for task-specific, GPT-4, LLaMA2, and human reviewers, respectively. These scores reflect substantial agreement on comment type and thread response decisions for both LLMs. The scores for comment type tasks show moderate agreement for task-specific RCGs, and substantial agreement for the other two RCGs [84]. For the interrogative comment question, the Kappa score for human-submitted comments is interpreted as substantial agreement while the other RCGs have a moderate agreement rate. Lower Kappa scores in some tasks were attributed to the high diversity of possible labels and noise from hard-to-parse generated comments.

6.4.2 Results

Tables 6.6 and 6.7 summarize the results of the coding task. We discuss our findings with respect to comment type and intention below.

Comment Type: What types of comments can be observed within the scope of generated interrogative comments?

Table 6.4 highlights three new emergent categories of comment types (cells with a gray background). Conversely, our analysis does not reveal any examples of types *code io*, *code doc*, *compatibility*, *rule def*, and *config building/installing* in either human or RCG comments.

LLM-based RCGs pose more documentation-related questions (2.41% and 4.55% of all generated interrogative comments for GPT-4 and LLaMA2, respectively) than both human reviewers (0.97% of all human-submitted interrogative comments) and task-specific RCGs, which do not ask this type of question. This suggests that LLM-based RCGs could serve as vigilant overseers for code documentation quality.

RCGs pose more questions about exceptions than human reviewers. While task-specific RCGs are notably adept at generating such interrogative comments (7.37% of all their gen-

erated interrogative comments), LLMs ask about exceptions even more frequently (14.46% and 20.45% for GPT-4 and LLaMA2, respectively). We find that these task-specific and LLM-based RCGs are especially adept at raising exception-handling concerns for common APIs, such as file I/O operations, whereas human reviewers excel at pinpointing complex issues, such as neglected edge cases.

Human reviewers more frequently question the logic behind code changes (54.37%), such as conditional placement or potential oversights in handling edge cases, compared to RCGs (38.64%-42.17%). In contrast, RCGs' (22.11%) and LLMs' (13.86% and 15.91% for GPT-4 and LLaMA2, respectively) questions often concentrate on the purpose for the changes, possibly reflecting their more limited grasp of the code's broader context. Human reviewers are often among the core developers of the projects [109] or have past involvement with the modified files and subsystem in the code change [155]. By using this prior context on the changed files, they can provide more in-depth interrogative comments on improving the code logic. RCGs lack this context. As a result, they often question the rationale behind a code change rather than the logic of the code being changed. Moreover, providing all of the available code and documentation may not resolve this problem since the additional context may be misleading, yielding poorer results while imposing higher computation costs [97]. Based on this observed behaviour, our findings suggest that humans are better suited to review code changes involving complex logic. They often question the logic of the changes, leading to potentially useful discussions. On the other hand, RCGs, whose questioning style diverges from human reviewers', probe the logic underlying a change considerably less frequently and ask more frequently about the purpose.

While RCGs excel at identifying unhandled exceptions, they fall short in contextual understanding, frequently using interrogative comments to seek context about code changes (54.37%).

Intention: What are the perceived intentions behind generated interrogative comments?

During labeling, we encounter all intent categories [35] except *Attitudes and emotions*.

Human reviewers primarily (63.11% of sampled comments) use interrogative comments to provide suggestions, whereas task-specific (56.84%) and LLMs-based (72.89% for GPT-4 and 84.09% for LLaMA2) RCGs primarily ask questions to gain more information or justifications for code changes. For example, in a specific code change,[9] a reviewer suggests,

[9] https://gerrit-review.googlesource.com/c/gerrit/+/194420

"*Cannot be list.sort(comparing(GpgKeyInfo::id))?*", while CodeBert comments, "*This is a bit confusing. Does this work for a single GpgKeyInfo?[...]*", reflecting confusion possibly due to limited context. This outcome supports our previous observation concerning the comment types. Moreover, since current RCGs do not engage in the resulting discussions and, thus, do not use the information provided by the author, these comments hinder the productive use of RCGs. While LLMs do not face this limitation, this may still hinder their usefulness since, if they do not provide a suggestion, the responsibility to devise one rests solely with the authors.

Indeed, *a considerable portion of LLM-based RCG inquiries request more information about the code change (72.89% for GPT-4 and 84.09% for LLaMA2), prompting authors to propose solutions rather than offering them like human reviewers.* This trend may be partially attributed to a relative lack of context compared to human reviewers, who request additional information in 23.30% of interrogative comments. It also exceeds the rate at which task-specific RCGs seek information (56.84% of times). Although this trait is beneficial if LLMs continue the discussion or have relevant context at hand, authors may have to respond to many questions when addressing reviews, potentially prolonging the code review process. Even when authors provide the requested information in their responses, LLMs do not always engage with discussion threads, resulting in wasted time and unresolved comments.

Human reviewers occasionally (4.85%) propose hypothetical scenarios to probe potential issues—a practice that RCGs do not replicate, with the exception of GPT-4-based RCG (12.65%). For example, in a specific code change, a reviewer inquires, "*Should we reload plugins in dependency order if the caller gave us more than one and one depends on the other??*"[10] This question highlights a scenario potentially missed by the code author. This illustrates the limited capacity of RCGs to consider the relevant project context to draw attention to possible defects that authors may have overlooked. While task-specific RCGs are trained on human-submitted comments, since they lack the context of the project under scrutiny and the necessary capacity to assess the code for potential hypothetical scenarios, they cannot ask hypothetical questions that can help draw attention to potential bugs or future issues. Our qualitative analysis reveals that the LLaMA2-based RCG similarly lacks the capability for this task. Meanwhile, GPT-4-based RCG is capable of asking such types of questions, asking them more often than humans. This showcases the need to carefully select the type of RCGs based on the desired outcomes.

Our intention-based analysis highlights that most of the studied RCGs have a limited understanding of the code's context. While LLM-based RCGs could mitigate this issue

[10] https://gerrit-review.googlesource.com/c/gerrit/+/54428

using transfer learning techniques [62], they still require significant advancements in reasoning [184] to effectively simulate the nuanced thought processes of human reviewers, especially concerning logical reasoning [10, 14, 96]. Thus, human reviewers still provide unique benefits for code changes that rely heavily on historical context or external information, such as bug fixes or integration with private APIs. Reliance on RCGs for such reviews may lead to an ineffective cycle of rationale-seeking code review comments.

While rhetorical questions are used by humans (8.74%) to draw author attention [166], strengthen reviewer arguments, and aid in convincing authors [119] to take action, task-specific (3.16%) and LLM-based (0% for GPT-4 and 2.27% in LLaMA2) RCGs seldom employ this technique. Even if RCGs could express the same concepts in a declarative manner, this could potentially hamper the communication of ideas and increase author resistance to perceiving mistakes.

This finding, though subtle, suggests that task-specific RCGs do not fully leverage the expressive capacity of natural language. Over time, this limitation could impede development velocity by diminishing RCG effectiveness in highlighting the significance of some comments over others through careful word choice. On the other hand, LLM-based RCGs may perform better in this regard at the cost of increased latency and resource consumption [67].

Human reviewers predominantly use interrogative comments (63.11%) to offer suggestions, highlighting the subtle differences in review strategies. Their second most common intention is to request more information about the change (23.30%). Conversely, task-specific RCG comments mainly request more information (56.84%), with recommendations being the next most common purpose (40%). LLM-based RCGs follows the same pattern with GPT-4 and LLaMA requesting more information 72.89% and 84.09% of the time and making suggestions 14.46% and 13.64% of the time, respectively.

6.5 Automatic Code Review Proposed Task: Discussion Thread Response Generation

Code Review Discussion Thread Response. We propose this task as a part of the code review automation and focus our anlysis on discussion threads that were initiated by a human posing an interrogative comment, and were classified as a discussion thread

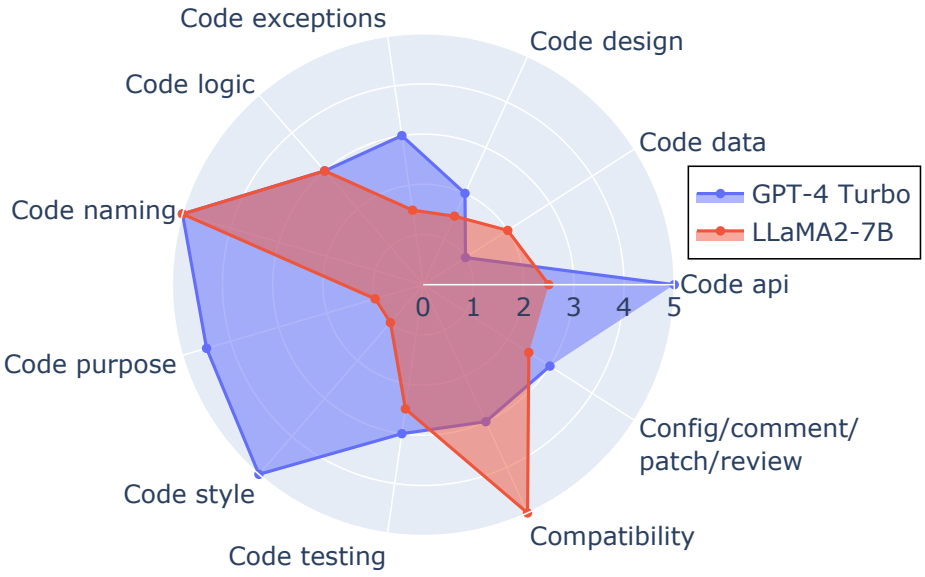


Figure 6.2: similarity scores (median) of responses of LLM-based RCGs to comment threads.

according to our definition in Section 6.3.1. We inspect the generated comments for their similarity to human reviewer comments using a five-point Likert scale, where zero indicates no similarity and five indicates a match. Each coder independently labels responses in batches of 50 from the sampled dataset if the entry meets these criteria. After each batch, the coders resolve discrepancies. Overall, 60 of the studied entries meet these criteria. Figure 6.2 plots the similarity between the generated responses and human comments.

While LLM-based RCGs can effectively address discussions centered on comment types, such as identifier naming, the alignment with human responses diminishes in threads initiated by other comment types, such as design. To further analyze our results, we calculated the Area Under Curve (AUC) for the LLM-based RCGs predictions for thread resolution. The resulting AUCs are 72.34% for GPT-4 and 42.04% for LLaMA2. These observations reveal that current LLMs are promising in emulating reviewer responses; however, the type of comment and the intention of the initial reviewer question play a key role. Moreover, Figure 6.2 shows that the choice of the LLM influences performance, with GPT-4 outperforming LLaMA2 by at least one unit on the Likert scale in six of the categories, and only underperforming in two categories.

LLM-based RCGs show promise in following up on interrogative discussion threads in code review, especially for topics such as naming. Since their performance varies substantially based on the type of comment and intention of the thread, LLM-based RCGs should not yet be considered a direct replacement for human reviewers.

6.6 Threats to Validity

Construct Validity threats undermine measurement effectiveness [173]. One construct threat is imposed by the heuristic that we use to identify interrogative comments. To assess its accuracy, we manually label 377 review comments and observe that our heuristic achieves a 0.894 kappa score, indicating almost perfect agreement.

While hallucinations [1, 10] could be a problem for our study, we did not notice any systematic examples of such errors during our inspection in Section 5. Therefore, this should not be a concern for our observations in this study. Correctness of the questions asked by the RCGs is another threat. Unfortunately, due to the high volume of interrogative comments and the lack of domain knowledge for the Gerrit project, coders cannot verify whether interrogative comments related to a code change are correct. While the incorrect questions can waste the authors' time and impose unnecessary workload, in this study, our focus is primarily on comment mood, whether they are correct or not. However, hallucinations and the correctness of interrogative comments are interesting future studies.

Internal Validity threats relate to uncontrolled confounding factors [173]. Of concern is the subjective judgment of coders in qualitative tasks. To mitigate this, we adhered to best practices [23, 81, 139] for qualitative analysis. Disagreements between coders were resolved through collaborative discussion until a consensus was reached. Cohen's Kappa inter-rater reliability scores indicate moderate to perfect agreement. Given the complexity of selecting labels from lists of 15 and 5 categories for the type and intent of comments, respectively, such agreement levels are often deemed acceptable [34, 82, 163]. Also, to balance coder subjectivity and allow new categories to emerge, we employed a blended coding approach [52].

External Validity threats impact the generalizability of the findings. One such threat is our focus on a single community. While we acknowledge that this weakens generalizability, the Gerrit community is selected due to its long-standing commitment to performing a rigorous review process. This makes the Gerrit community an exemplar for other communities that consistently practice code reviews over time, thus lending relevance to our findings.

6.7 Conclusions and Lessons Learned

In this chapter, we study interrogative comments generated by RCGs. To that end, we use three state-of-the-art task-specific RCGs [43, 89, 90] and three LLMs to generate code review comments and quantitatively and qualitatively analyze the interrogative ones. Below, we distill lessons for both development and research communities.

- **On Development.** ***Human reviewers are still necessary for submitting interrogative comments during code review, as neither task-specific RCGs nor LLM-based ones can fully replace them.*** Indeed, when RCGs pose questions, they primarily inquire about the rationale behind changes (56.84%, 72.89% 84.09% for task-specific, GPT-4-, and LLaMA2-based RCGs, respectively), unlike human reviewers who more often propose solutions (63.11%). We conjecture that projects can benefit from presenting RCG-generated comments to human reviewers during the review process. This is supported by our results where RCGs more frequently ask questions related to code exceptions (7.37%, 14.46%, and 20.45% for task-specific, GPT-4-, and LLaMA2-based RCGs, respectively) and documentation (2.41% for GPT-4 and 4.55% for LLaMA2) than humans (0.97% and 2.91% for documentation and exceptions, respectively). This approach can complement human reviewers, leading to a broader review process. Indeed, similar approaches have been effective in enhancing the code review process [69, 168]. Furthermore, while early work suggests that LLMs have limitations in logical reasoning [14, 96], and that current LLM-based RCGs do not outperform task-specific RCG [99], we find that LLM-based models mitigate some issues of current RCGs such as interacting with the authors. Specifically, in cases where human reviewers request information, LLM-based RCGs can resolve discussion threads similarly to humans (72.34% AUC for GPT-4). This suggests that using LLMs in review discussions could reduce developer workload more effectively than task-specific RCGs.
- **On Research.** ***Researchers should explore ways to control the generation of interrogative comments and propose methods to automate the new proposed task of discussion thread response generation.*** The median share of interrogative comments for task-specific RCGs is 15.61%. This noticeable share highlights the importance of these questions. Additionally, the studied RCGs do not consistently mimic humans in submitting interrogative comments, making their behaviour more irregular and erratic. Since task-specific RCGs do not currently participate in follow-up discussions, the usefulness of this large portion of generated comments, especially for discussion-inducing changes, is hindered, or worse, these

tools may stifle beneficial conversations. Thus, we recommend that research focus on either reducing the current conversation-impeding interrogative comments from these models or integrating additional context into the model inputs. This enhancement could enable RCGs to understand and respond to the various facets of a change more effectively. Also, LLMs such as GPT-4 show promise in the review automation proposed task for responding to the threads in specific types of comments and determining resolved threads (72.34% AUC for GPT-4, but only 42.04% AUC for LLaMA2 for the resolution task). Future research should determine how to choose (and further improve) RCGs to respond to the review discussions as a task in automatic code review generation that has not been feasible previously.

6.8 Chapter Summary

In this chapter, we investigate whether ACR tools are advanced enough to render CRR systems and other human recommendation tools in code review obsolete. We observe that, despite promising results in specific tasks, such as determining whether a code review discussion is resolved, ACR processes have not yet replaced human involvement. Therefore, we conclude that continued investment in improving code review automation approaches and addressing their challenges would be beneficial in the long run.

Part IV

Final Remarks

Chapter 7

Conclusion and Future Work

Code review is an important step in improving proposed code changesets. Code review automation approaches have been proposed to speed up this time-consuming process and to help developers keep up with the rapid pace of modern software development.

While in theory, these tools promise to boost the software development process, they can fall short when deployed. In this thesis, we empirically study challenges and explore the relevance of these approaches with the emergence of Automatic Code Review (ACR) approaches.

The primary goal of this thesis is to better understand the practical challenges of using automated code review suggestions. To achieve this goal, we harness the information from Version Control System (VCS) platforms on historical data and within the review platforms such as Gerrit[1] and GitHub.[2] We formulate and evaluate the following thesis statement:

Thesis Statement

Practical challenges in code review processes diminish the usefulness of code review automation. A multi-faceted approach to address issues like reviewer staleness and the bug-proneness of changesets will improve the interplay between human and automation, thereby enhancing automation outcomes.

We empirically study some of the practical challenges that practitioners face when employing the code review automation approaches and explore whether Code Reviewer Recommendation (CRR) systems are obsolete considering the current state of Automatic Code Review (ACR) process. Our studies confirm that CRR approaches face practical challenges, such as staleness and risks to project safety. These issues can erode the trust of

developers and discourage the adoption of these approaches [185]. Furthermore, our last study demonstrates that code review automation approaches remain relevant despite recent advancements in ACR, highlighting the importance of continued investment in improving these systems. Below, we reiterate some of the core findings of these studies:

7.1 Contributions and Findings

1. Part II: Limitations of CRR System
   (a) **Unintended negative impact of CRR systems:** Despite the main goal of code review suggestion systems to speed up the review process, practitioners who deploy these systems encounter challenges, such as stale reviewers (Chapter 4) and risky reviewer recommendations (Chapter 5).
   (b) **Identifying the effect of considering contribution recency on the staleness of CRRs:** While considering the recency of contributions can mitigate the risk of recommending stale reviewers, the complex behaviour of multi-objective CRR systems can overshadow this impact if not evaluated independently (Chapter 4).
   (c) **Understanding the impact of ignoring the changeset bug-proneness:** CRR systems may compromise project safety by suggesting less experienced reviewers for risky changesets due to their multi-faceted behaviour. While this approach can be beneficial for non-risky code changes, it undermines project safety when dealing with bug-prone changesets (Chapter 5).
   (d) **Mitigation strategy adaptability:** While there is no silver bullet to resolve existing challenges, enabling practitioners to adjust the provided CRRs with flexible and simple configurations can improve the adaptability of these systems. Therefore, all our mitigation strategies include a setting to adjust depending on the project state of development, as well as guidance on the potential side effects of possible configurations (Chapter 4 and Chapter 5).
   (e) **Enabling the reuse of existing CRR systems:** Our proposed mitigation strategies for addressing stale reviewers and enhancing project safety can be integrated with existing CRR systems to improve the staleness (Chapter 4) and safety (Chapter 5) of the recommended reviewers.
2. Part III: Practicality of CRR Systems

(a) **Relevance of automatic review suggestion systems:** Neither state-of-the-art task-specific nor LLM-based Review Comment Generator (RCG) systems can replace human reviewers when it comes to asking questions during code review.

(b) **Understanding behaviour of RCGs concerning interrogative comments:** RCGs differ from humans when reviewing a PR, particularly when the code change induces review discussion. This difference is evident in both the quantity and quality of questions. For instance, human reviewers often use interrogative questions to suggest solutions when pointing out problems, whereas RCGs do this noticeably less often in the sample dataset. Instead, they more often ask about the purpose of the code change (Chapter 6).

(c) **Performance of LLMs on following-up review threads:** LLMs' performance on discussion thread resolution shows promising results. However, they regress when they are required to follow up on the thread and respond to the author (Chapter 6).

7.2 Prospects for Future Research

This thesis makes a meaningful contribution to understanding the practical challenges that automatic code review suggestion tools are facing, yet, many other questions remain open to future research. Below, we outline several promising directions for further work.

7.2.1 Assessing the Validity of Our Findings in Different Software Development Settings

Although we explore some of the practical challenges in employing CRR systems in various stages of their recommendation, we do not investigate the extent to which these challenges could be generalized to other code review suggestion tools such as code review comment recommenders [61]. We believe one avenue for future works to explore is to investigate the applicability and validity of our findings in different code review settings, such as in large-scale, closed-source environments, to ensure that these challenges stand true in those settings and that our proposed solutions are robust and generalizable across different types of projects and organizational settings.

7.2.2 Comprehensive Code Reviewer Recommendation improvement Toolkit Development

In this thesis, we identify and address specific challenges faced by CRR systems. Future work should aim to integrate these solutions, as well as prior challenges [109], into a comprehensive toolkit that can manage multiple aspects of the code review process, such as the risk of the proposed changeset and the knowledge turnover rate of the project.

7.2.3 Assessing the Impact of Employing Improved Predictors for Stale Reviewers

Predicting stale reviewers accurately is crucial for maintaining an efficient review process. This thesis proposes a simple time-based filtering strategy to mitigate recommendation staleness. Future research should focus on better predictors for identifying and filtering the top-recommended stale reviewers while trying to minimize the exclusion of available developers.

7.2.4 Surveying the Usefulness of Mitigation Strategies

We have validated our proposed mitigation strategies by simulating the review process throughout the project history. However, we believe a positive affirmation from developers who use our techniques is a complementary stage to this study. Conducting surveys to assess their practicality and effectiveness would provide valuable feedback which can be used in developing a comprehensive CRR toolkit.

7.2.5 Development of Task-Specific or Large Language Model-based Models to Follow Up on Discussion Threads

While we investigate the performance of three LLMs from two different sizes on the proposed task of following up on the code review discussion for ACR, we believe developing task-specific models or fine-tuning existing models can lead to better results both for code review thread resolution and responding to the thread. This research would involve training models on large datasets of code review discussions to learn how to generate meaningful follow-up questions and responses. It may also benefit from some levels of human feedback

to align the responses with human expectations. In addition, these models should be capable of understanding the context and subtle differences between the ongoing discussions, helping to maintain the flow of communication and ensuring that important issues are addressed.

7.2.6 Assessing the similarity of Human-submitted and Machine-generated Comments

We study interrogative comments as one type of comment that plays a crucial role in understanding code review. However, a comprehensive assessment of the RCG-generated comments against human-submitted comments could yield interesting results. This study would especially be useful since we can now measure the semantic similarity of comments using embedding models [176], which allow for a quantitative assessment of the quality and relevance of these comments.

7.2.7 Developing an Automatic Code Reviewer Selection Model

Another interesting area of exploration is the development of a new family of CRRs that assesses the submitted code changeset and suggests whether the change can be reviewed automatically or, due to factors such as being error-prone, a human reviewer is required. This new family of CRRs can help balance the workload and ensure critical changes receive adequate human attention.

References

[1] Ashish Agarwal, Clara Wong-Fannjiang, David Sussillo, Katherine Lee, and Orhan Firat. Hallucinations in neural machine translation. In *ICLR*. 2018.

[2] Wisam Haitham Abbood Al-Zubaidi, Patanamon Thongtanunam, Hoa Khanh Dam, Chakkrit Tantithamthavorn, and Aditya Ghose. Workload-aware reviewer recommendation using a multi-objective search-based approach. In *Proceedings of the 16th ACM International Conference on Predictive Models and Data Analytics in Software Engineering*, pages 21–30, 2020.

[3] Hamda Hasan AlBreiki and Qusay H Mahmoud. Evaluation of static analysis tools for software security. In *2014 10th International Conference on Innovations in Information Technology (IIT)*, pages 93–98. IEEE, 2014.

[4] Guilherme Avelino, Eleni Constantinou, Marco Tulio Valente, and Alexander Serebrenik. On the abandonment and survival of open source projects: An empirical investigation. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–12. IEEE, 2019.

[5] Muhammad Ilyas Azeem, Qiang Peng, and Qing Wang. Pull request prioritization algorithm based on acceptance and response probability. In *2020 IEEE 20th International Conference on Software Quality, Reliability and Security (QRS)*, pages 231–242. IEEE, 2020.

[6] Alberto Bacchelli and Christian Bird. Expectations, outcomes, and challenges of modern code review. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 712–721. IEEE, 2013.

[7] Deepika Badampudi, Michael Unterkalmsteiner, and Ricardo Britto. Modern code reviews—survey of literature and practice. *ACM Transactions on Software Engineering and Methodology*, 32(4):1–61, 2023.

[8] Farid Bagirov, Pouria Derakhshanfar, Alexey Kalina, Elena Kartysheva, and Vladimir Kovalenko. Assessing the impact of file ordering strategies on code review process. In *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering*, pages 188–191, 2023.

[9] Vipin Balachandran. Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 931–940. IEEE, 2013.

[10] Yejin Bang, Samuel Cahyawijaya, Nayeon Lee, Wenliang Dai, Dan Su, Bryan Wilie, Holy Lovenia, Ziwei Ji, Tiezheng Yu, Willy Chung, et al. A multitask, multilingual, multimodal evaluation of chatgpt on reasoning, hallucination, and interactivity. *arXiv preprint arXiv:2302.04023*, 2023.

[11] Lingfeng Bao, Xin Xia, David Lo, and Gail C Murphy. A large scale study of long-time contributor prediction for github projects. *IEEE Transactions on Software Engineering*, 47(6):1277–1298, 2019.

[12] Lingfeng Bao, Zhenchang Xing, Xin Xia, David Lo, and Shanping Li. Who will leave the company?: a large-scale industry study of developer turnover by mining monthly work report. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 170–181. IEEE, 2017.

[13] Olga Baysal, Oleksii Kononenko, Reid Holmes, and Michael W Godfrey. The influence of non-technical factors on code review. In *2013 20th working conference on reverse engineering (WCRE)*, pages 122–131. IEEE, 2013.

[14] Lukas Berglund, Meg Tong, Maximilian Kaufmann, Mikita Balesni, Asa Cooper Stickland, Tomasz Korbak, and Owain Evans. The reversal curse: LLMs trained on "a is b" fail to learn "b is a". In *The Twelfth International Conference on Learning Representations*, 2024.

[15] H Russell Bernard, Amber Wutich, and Gery W Ryan. *Analyzing qualitative data: Systematic approaches*. SAGE publications, 2016.

[16] Christian Bird, Nachiappan Nagappan, Brendan Murphy, Harald Gall, and Premkumar Devanbu. Don't touch my code! examining the effects of ownership on software quality. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 4–14, 2011.

[17] Amiangshu Bosu and Jeffrey C Carver. Impact of peer code review on peer impression formation: A survey. In *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 133–142. IEEE, 2013.

[18] Amiangshu Bosu, Jeffrey C Carver, Christian Bird, Jonathan Orbeck, and Christopher Chockley. Process aspects and social dynamics of contemporary code review: Insights from open source development and industrial practice at microsoft. *IEEE Transactions on Software Engineering*, pages 56–75, 2016.

[19] Larissa Braz, Christian Aeberhard, Gül Çalikli, and Alberto Bacchelli. Less is more: supporting developers in vulnerability detection during code review. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1317–1329, 2022.

[20] Larissa Braz and Alberto Bacchelli. Software security during modern code review: the developer's perspective. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 810–821, 2022.

[21] Zhenzhen Cao, Sijia Lv, Xinlong Zhang, Hui Li, Qian Ma, Tingting Li, Cheng Guo, and Shikai Guo. Structuring meaningful code review automation in developer community. *Engineering Applications of Artificial Intelligence*, 127:106970, 2024.

[22] H Alperen Çetin, Emre Doğan, and Eray Tüzün. A review of code reviewer recommendation studies: Challenges and future directions. *Science of Computer Programming*, page 102652, 2021.

[23] K Charmaz. Constructing grounded theory, 2014.

[24] Moataz Chouchen, Ali Ouni, Jefferson Olongo, and Mohamed Wiem Mkaouer. Learning to predict code review completion time in modern code review. *Empirical Software Engineering*, 28(4):82, 2023.

[25] Motaz Chouchen, Ali Ouni, Mohamed Wiem Mkaouer, Raula Gaikovina Kula, and Katsuro Inoue. Recommending peer reviewers in modern code review: a multi-objective search-based approach. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion*, pages 307–308, 2020.

[26] WJf Conover and Ronald L Iman. On some alternative procedures using ranks for the analysis of experimental designs. *Communications in Statistics-Theory and Methods*, pages 1349–1368, 1976.

[27] Fred J Damerau. A technique for computer detection and correction of spelling errors. *Communications of the ACM*, 7(3):171–176, 1964.

[28] Hai Dang, Lukas Mecke, Florian Lehmann, Sven Goller, and Daniel Buschek. How to prompt? opportunities and challenges of zero-and few-shot learning for human-ai interaction in creative applications of generative models. *arXiv preprint arXiv:2209.01390*, 2022.

[29] Steven Davies, Marc Roper, and Murray Wood. Comparing text-based and dependence-based approaches for determining the origins of bugs. *Journal of Software: Evolution and Process*, pages 107–139, 2014.

[30] Manoel Limeira de Lima Júnior, Daricélio Moreira Soares, Alexandre Plastino, and Leonardo Murta. Developers assignment for analyzing pull requests. In *Proceedings of the 30th annual ACM symposium on applied computing*, pages 1567–1572, 2015.

[31] Emre Doğan and Eray Tüzün. Towards a taxonomy of code review smells. *Information and Software Technology*, 142:106737, 2022.

[32] Emre Doğan, Eray Tüzün, K Ayberk Tecimer, and H Altay Güvenir. Investigating the validity of ground truth in code reviewer recommendation studies. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–6. IEEE, 2019.

[33] Olive Jean Dunn. Multiple comparisons among means. *Journal of the American statistical association*, 56(293):52–64, 1961.

[34] Felipe Ebert, Fernando Castor, Nicole Novielli, and Alexander Serebrenik. Confusion detection in code reviews. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 549–553. IEEE, 2017.

[35] Felipe Ebert, Fernando Castor, Nicole Novielli, and Alexander Serebrenik. Communicative intention in code review questions. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 519–523. IEEE, 2018.

[36] Nasir U Eisty and Jeffrey C Carver. Developers perception of peer code review in research software development. *Empirical Software Engineering*, pages 1–26, 2022.

[37] Jayalath Ekanayake, Jonas Tappolet, Harald C Gall, and Abraham Bernstein. Tracking concept drift of software projects using defect prediction quality. In *International Working Conference on Mining Software Repositories*, pages 51–60. IEEE, 2009.

[38] Ahmed Elnaggar, Wei Ding, Llion Jones, Tom Gibbs, Tamas Feher, Christoph Angerer, Silvia Severini, Florian Matthes, and Burkhard Rost. Codetrans: Towards cracking the language of silicon's code through self-supervised deep learning and high performance computing. *arXiv preprint arXiv:2104.02443*, 2021.

[39] Vahid Etemadi, Omid Bushehrian, and Gregorio Robles. Task assignment to counter the effect of developer turnover in software maintenance: A knowledge diffusion model. *Information and Software Technology*, 143:106786, 2022.

[40] Michael E Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 38(2.3):258–287, 1999.

[41] Angela Fan, Mike Lewis, and Yann Dauphin. Hierarchical neural story generation. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 889–898, 2018.

[42] Mikołaj Fejzer, Piotr Przymus, and Krzysztof Stencel. Profile based recommendation of code reviewers. *Journal of Intelligent Information Systems*, 50(3):597–619, 2018.

[43] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

[44] Matthieu Foucault, Marc Palyart, Xavier Blanc, Gail C Murphy, and Jean-Rémy Falleri. Impact of developer turnover on quality in open-source software. In *Proceedings of the 2015 10th joint meeting on foundations of software engineering*, pages 829–841, 2015.

[45] Enrico Fregnan, Larissa Braz, Marco D'Ambros, Gül Çalıklı, and Alberto Bacchelli. First come first served: the impact of file position on code review. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 483–494, 2022.

[46] Milton Friedman. A comparison of alternative tests of significance for the problem of m rankings. *The Annals of Mathematical Statistics*, pages 86–92, 1940.

[47] Ian X. Gauthier, Maxime Lamothe, Gunter Mussbacher, and Shane McIntosh. Is Historical Data an Appropriate Benchmark for Reviewer Recommendation Systems? A Case Study of the Gerrit Community. In *Proc. of the International Conference on Automated Software Engineering (ASE)*, page To appear, 2021.

[48] Ian X Gauthier, Maxime Lamothe, Gunter Mussbacher, and Shane McIntosh. Is historical data an appropriate benchmark for reviewer recommendation systems?: A case study of the gerrit community. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 30–41. IEEE, 2021.

[49] Georgios Gousios, Martin Pinzger, and Arie van Deursen. An exploratory study of the pull-based software development model. In *Proceedings of the 36th international conference on software engineering*, pages 345–355, 2014.

[50] Georgios Gousios, Margaret-Anne Storey, and Alberto Bacchelli. Work practices and challenges in pull-based development: the contributor's perspective. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 285–296. IEEE, 2016.

[51] Georgios Gousios, Andy Zaidman, Margaret-Anne Storey, and Arie Van Deursen. Work practices and challenges in pull-based development: The integrator's perspective. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 358–368. IEEE, 2015.

[52] Melissa E Graebner, Jeffrey A Martin, and Philip T Roundy. Qualitative data: Cooking without a recipe. *Strategic Organization*, 10(3):276–284, 2012.

[53] Greg Guest, Emily Namey, and Mario Chen. A simple method to assess and report thematic saturation in qualitative research. *PloS one*, 15(5):e0232076, 2020.

[54] Qi Guo, Junming Cao, Xiaofei Xie, Shangqing Liu, Xiaohong Li, Bihuan Chen, and Xin Peng. Exploring the potential of chatgpt in automated code refinement: An empirical study. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pages 1–13, 2024.

[55] Anshul Gupta and Neel Sundaresan. Intelligent code reviews using deep learning. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'18) Deep Learning Day*, 2018.

[56] Christoph Hannebauer, Michael Patalas, Sebastian Stünkel, and Volker Gruhn. Automatically recommending code reviewers based on their expertise: An empirical comparison. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 99–110, 2016.

[57] Ahmed E Hassan. Predicting faults using the complexity of code changes. In *2009 IEEE 31st international conference on software engineering*, pages 78–88. IEEE, 2009.

[58] Vincent J Hellendoorn, Jason Tsay, Manisha Mukherjee, and Martin Hirzel. Towards automating code review at scale. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1479–1482, 2021.

[59] Kim Herzig, Sascha Just, and Andreas Zeller. The impact of tangled code changes on defect prediction models. *Empirical Software Engineering*, 21:303–336, 2016.

[60] Thong Hoang, Hoa Khanh Dam, Yasutaka Kamei, David Lo, and Naoyasu Ubayashi. Deepjit: an end-to-end deep learning framework for just-in-time defect prediction. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 34–45. IEEE, 2019.

[61] Yang Hong, Chakkrit Tantithamthavorn, Patanamon Thongtanunam, and Aldeida Aleti. Commentfinder: a simpler, faster, more accurate code review comments recommendation. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 507–519, 2022.

[62] Wenlong Huang, Fei Xia, Ted Xiao, Harris Chan, Jacky Liang, Pete Florence, Andy Zeng, Jonathan Tompson, Igor Mordatch, Yevgen Chebotar, et al. Inner monologue: Embodied reasoning through planning with language models. In *Conference on Robot Learning*, pages 1769–1782. PMLR, 2023.

[63] Mark A Huselid. The impact of human resource management practices on turnover, productivity, and corporate financial performance. *Academy of management journal*, 38(3):635–672, 1995.

[64] Daniel Izquierdo-Cortazar, Gregorio Robles, Felipe Ortega, and Jesus M Gonzalez-Barahona. Using software archaeology to measure knowledge loss in software projects due to developer turnover. In *2009 42nd Hawaii International Conference on System Sciences*, pages 1–10. IEEE, 2009.

[65] Jing Jiang, Jia-Huan He, and Xue-Yuan Chen. Coredevrec: Automatic core member recommendation for contribution evaluation. *Journal of Computer Science and Technology*, 30(5):998–1016, 2015.

[66] Jing Jiang, Yun Yang, Jiahuan He, Xavier Blanc, and Li Zhang. Who should comment on this pull request? analyzing attributes for more accurate commenter recommendation in pull-based development. *Information and Software Technology*, 84:48–62, 2017.

[67] Jean Kaddour, Joshua Harris, Maximilian Mozes, Herbie Bradley, Roberta Raileanu, and Robert McHardy. Challenges and applications of large language models. *arXiv preprint arXiv:2307.10169*, 2023.

[68] Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E. Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. A large-scale empirical study of just-in-time quality assurance. *IEEE Trans. Softw. Eng.*, pages 757–773, June 2013.

[69] Farshad Kazemi, Maxime Lamothe, and Shane McIntosh. Exploring the notion of risk in code reviewer recommendation. In *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 139–150. IEEE, 2022.

[70] Farshad Kazemi, Maxime Lamothe, and Shane McIntosh. Characterizing the prevalence distribution and duration of stale reviewer recommendations. *IEEE Transactions on Software Engineering*, 2024.

[71] Maurice G Kendall et al. The advanced theory of statistics. vols. 1. *The advanced theory of statistics. Vols. 1.*, 1948.

[72] SayedHassan Khatoonabadi, Diego Elias Costa, Rabe Abdalkareem, and Emad Shihab. On wasted contributions: Understanding the dynamics of contributor-abandoned pull requests: A mixed-methods study of 10 large open-source projects. *ACM Transactions on Software Engineering and Methodology*, 2021.

[73] SayedHassan Khatoonabadi, Diego Elias Costa, Rabe Abdalkareem, and Emad Shihab. On wasted contributions: understanding the dynamics of contributor-abandoned pull requests–a mixed-methods study of 10 large open-source projects. *ACM Transactions on Software Engineering and Methodology*, 32(1):1–39, 2023.

[74] Jungil Kim and Eunjoo Lee. Understanding review expertise of developers: A reviewer recommendation approach based on latent dirichlet allocation. *Symmetry*, page 114, 2018.

[75] Kisub Kim, Xin Zhou, Dongsun Kim, Julia Lawall, Kui Liu, Tegawendé F Bissyandé, Jacques Klein, Jaekwon Lee, and David Lo. How are we detecting inconsistent

method names? an empirical study from code review perspective. *arXiv preprint arXiv:2308.12701*, 2023.

[76] Sunghun Kim, E James Whitehead, and Yi Zhang. Classifying software changes: Clean or buggy? *IEEE Transactions on software engineering*, 34(2):181–196, 2008.

[77] Oleksii Kononenko, Olga Baysal, and Michael W Godfrey. Code review quality: how developers see it. In *Proceedings of the 38th International Conference on Software Engineering*, pages 1028–1038, 2016.

[78] Oleksii Kononenko, Olga Baysal, Latifa Guerrouj, Yaxin Cao, and Michael W Godfrey. Investigating code review quality: Do people and participation matter? In *2015 IEEE international conference on software maintenance and evolution (ICSME)*, pages 111–120. IEEE, 2015.

[79] Oleksii Kononenko, Tresa Rose, Olga Baysal, Michael Godfrey, Dennis Theisen, and Bart De Water. Studying pull request merges: a case study of shopify's active merchant. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, pages 124–133, 2018.

[80] Vladimir Kovalenko, Nava Tintarev, Evgeny Pasynkov, Christian Bird, and Alberto Bacchelli. Does reviewer recommendation help developers? *IEEE Transactions on Software Engineering*, pages 710–731, 2018.

[81] Klaus Krippendorff. *Content analysis: An introduction to its methodology*. Sage publications, 2018.

[82] Andrey Krutauz, Tapajit Dey, Peter C Rigby, and Audris Mockus. Do code review measures explain the incidence of post-release defects? case study replications and bayesian networks. *Empirical Software Engineering*, 25:3323–3356, 2020.

[83] Maciej Kula. Metadata embeddings for user and item cold-start recommendations. *arXiv preprint arXiv:1507.08439*, 2015.

[84] J Richard Landis and Gary G Koch. The measurement of observer agreement for categorical data. *biometrics*, pages 159–174, 1977.

[85] Triet Huynh Minh Le, David Hin, Roland Croft, and M Ali Babar. Deepcva: Automated commit-level vulnerability assessment with deep multi-task learning. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 717–729. IEEE, 2021.

[86] Valentina Lenarduzzi, Vili Nikkola, Nyyti Saarimäki, and Davide Taibi. Does code quality affect pull request acceptance? an empirical study. *Journal of Systems and Software*, 171:110806, 2021.

[87] Heng-Yi Li, Shu-Ting Shi, Ferdian Thung, Xuan Huo, Bowen Xu, Ming Li, and David Lo. Deepreview: automatic code review using deep multi-instance learning. In *Advances in Knowledge Discovery and Data Mining: 23rd Pacific-Asia Conference, PAKDD 2019, Macau, China, April 14-17, 2019, Proceedings, Part II 23*, pages 318–330. Springer, 2019.

[88] Jian Li, Pinjia He, Jieming Zhu, and Michael R Lyu. Software defect prediction via convolutional neural network. In *2017 IEEE international conference on software quality, reliability and security (QRS)*, pages 318–328. IEEE, 2017.

[89] Lingwei Li, Li Yang, Huaxi Jiang, Jun Yan, Tiejian Luo, Zihan Hua, Geng Liang, and Chun Zuo. Auger: automatically generating review comments with pre-training models. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1009–1021, 2022.

[90] Zhiyu Li, Shuai Lu, Daya Guo, Nan Duan, Shailesh Jannu, Grant Jenks, Deep Majumder, Jared Green, Alexey Svyatkovskiy, Shengyu Fu, et al. Automating code review activities by large-scale pre-training. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1035–1047, 2022.

[91] Hongliang Liang, Yue Yu, Lin Jiang, and Zhuosi Xie. Seml: A semantic lstm model for software defect prediction. *IEEE Access*, 7:83812–83824, 2019.

[92] Bin Lin, Gregorio Robles, and Alexander Serebrenik. Developer turnover in global, industrial open source projects: Insights from applying survival analysis. In *2017 IEEE 12th International Conference on Global Software Engineering (ICGSE)*, pages 66–75. IEEE, 2017.

[93] Bo Lin, Shangwen Wang, Zhongxin Liu, Yepang Liu, Xin Xia, and Xiaoguang Mao. Cct5: A code-change-oriented pre-trained model. *arXiv preprint arXiv:2305.10785*, 2023.

[94] Hong Yi Lin and Patanamon Thongtanunam. Towards automated code reviews: Does learning code structure help? In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 703–707. IEEE, 2023.

[95] Jakub Lipcak and Bruno Rossi. A large-scale study on source code reviewer recommendation. In *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 378–387. IEEE, 2018.

[96] Hanmeng Liu, Ruoxi Ning, Zhiyang Teng, Jian Liu, Qiji Zhou, and Yue Zhang. Evaluating the logical reasoning ability of chatgpt and gpt-4. *arXiv preprint arXiv:2304.03439*, 2023.

[97] Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics*, 12:157–173, 2024.

[98] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *ACM Computing Surveys*, 55(9):1–35, 2023.

[99] Junyi Lu, Lei Yu, Xiaojia Li, Li Yang, and Chun Zuo. Llama-reviewer: Advancing code review automation with large language models through parameter-efficient fine-tuning. In *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*, pages 647–658. IEEE, 2023.

[100] Thomas W MacFarland, Jan M Yates, Thomas W MacFarland, and Jan M Yates. Mann–whitney u test. *Introduction to nonparametric statistics for the biological sciences using R*, pages 103–132, 2016.

[101] Laura MacLeod, Michaela Greiler, Margaret-Anne Storey, Christian Bird, and Jacek Czerwonka. Code reviewing in the trenches: Challenges and best practices. *IEEE Software*, pages 34–42, 2017.

[102] Saifullah Mahbub, Md Easin Arafat, Chowdhury Rafeed Rahman, Zannatul Ferdows, and Masum Hasan. Reviewranker: A semi-supervised learning based approach for code review quality estimation. *arXiv preprint arXiv:2307.03996*, 2023.

[103] Mika V Mäntylä and Casper Lassenius. What types of defects are really discovered in code reviews? *IEEE Transactions on Software Engineering*, 35(3):430–448, 2008.

[104] Shane McIntosh and Yasutaka Kamei. Are Fix-Inducing Changes a Moving Target? A Longitudinal Case Study of Just-In-Time Defect Prediction. *IEEE Transactions on Software Engineering*, page 412–428, 2018.

[105] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E Hassan. The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 192–201, 2014.

[106] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E Hassan. An empirical study of the impact of modern code review practices on software quality. *Empirical Software Engineering*, pages 2146–2189, 2016.

[107] Courtney Miller, David Gray Widder, Christian Kästner, and Bogdan Vasilescu. Why do people give up flossing? a study of contributor disengagement in open source. In *IFIP International Conference on Open Source Systems*, pages 116–129. Springer, 2019.

[108] Ehsan Mirsaeedi and Peter C. Peter, 2020. RelationalGit.

[109] Ehsan Mirsaeedi and Peter C Rigby. Mitigating turnover with code review recommendation: Balancing expertise, workload, and knowledge distribution. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 1183–1195, 2020.

[110] Md Rakib Hossain Misu, Aleksandar Saša Janjanin, Zhiqiang Bian, Valentin-Sebastian Burlacu, and Naum Anteski. Ada: a tool for visualizing the architectural overview of open-source repositories. In *Proceedings of the 11th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis*, pages 30–35, 2022.

[111] Audris Mockus. Organizational volatility and its effects on software defects. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 117–126, 2010.

[112] Audris Mockus and James D Herbsleb. Expertise browser: a quantitative approach to identifying expertise. In *Proceedings of the 24th international conference on software engineering. icse 2002*, pages 503–512. IEEE, 2002.

[113] Yukasa Murakami, Masateru Tsunoda, and Hidetake Uwano. Wap: Does reviewer age affect code review performance? In *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*, pages 164–169. IEEE, 2017.

[114] Jaechang Nam. Survey on software defect prediction. *Department of Compter Science and Engineerning, The Hong Kong University of Science and Technology, Tech. Rep*, 2014.

[115] Mathieu Nassif and Martin P Robillard. Revisiting turnover-induced knowledge loss in software projects. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 261–272. IEEE, 2017.

[116] Edmilson Campos Neto, Daniel Alencar Da Costa, and Uirá Kulesza. The impact of refactoring changes on the szz algorithm: An empirical study. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 380–390. IEEE, 2018.

[117] Miroslaw Ochodek, Miroslaw Staron, Wilhelm Meding, and Ola Söder. Automated code review comment classification to improve modern code reviews. In *International Conference on Software Quality*, pages 23–40. Springer, 2022.

[118] Luca Pascarella, Fabio Palomba, and Alberto Bacchelli. Fine-grained just-in-time defect prediction. *Journal of Systems and Software*, pages 22–36, 2019.

[119] Richard E Petty, John T Cacioppo, and Martin Heesacker. Effects of rhetorical questions on persuasion: A cognitive response analysis. *Journal of personality and social psychology*, 40(3):432, 1981.

[120] Chanathip Pornprasit and Chakkrit Tantithamthavorn. Gpt-3.5 for code review automation: How do few-shot learning, prompt design, and model fine-tuning impact their performance? *arXiv preprint arXiv:2402.00905*, 2024.

[121] Chanathip Pornprasit and Chakkrit Kla Tantithamthavorn. Jitline: A simpler, better, faster, finer-grained just-in-time defect prediction. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 369–379. IEEE, 2021.

[122] Huilian Sophie Qiu, Yucen Lily Li, Susmita Padala, Anita Sarma, and Bogdan Vasilescu. The signals that potential contributors look for when choosing open-source projects. *Proceedings of the ACM on Human-Computer Interaction*, 3(CSCW):1–29, 2019.

[123] Sophia Quach, Maxime Lamothe, Bram Adams, Yasutaka Kamei, and Weiyi Shang. Evaluating the impact of falsely detected performance bug-inducing changes in jit models. *Empirical Software Engineering*, pages 1–32, 2021.

[124] Sophia Quach, Maxime Lamothe, Yasutaka Kamei, and Weiyi Shang. An empirical study on the use of szz for identifying inducing changes of non-functional bugs. *Empirical Software Engineering*, pages 1–25, 2021.

[125] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *The Journal of Machine Learning Research*, 21(1):5485–5551, 2020.

[126] Mohammad Masudur Rahman, Chanchal K Roy, and Jason A Collins. Correct: code reviewer recommendation in github based on cross-project and technology experience. In *Proceedings of the 38th international conference on software engineering companion*, pages 222–231, 2016.

[127] Gopi Krishnan Rajbahadur, Shaowei Wang, Gustavo A Oliva, Yasutaka Kamei, and Ahmed E Hassan. The impact of feature importance methods on the interpretation of defect classifiers. *IEEE Transactions on Software Engineering*, 48(7):2245–2261, 2021.

[128] Zeeshan Rasheed, Malik Abdul Sami, Muhammad Waseem, Kai-Kristian Kemell, Xiaofeng Wang, Anh Nguyen, Kari Systä, and Pekka Abrahamsson. Ai-powered code review with llms: Early results. *arXiv preprint arXiv:2404.18496*, 2024.

[129] Mehvish Rashid, Paul M Clarke, and Rory V O'Connor. A systematic examination of knowledge loss in open source software projects. *International Journal of Information Management*, 46:104–123, 2019.

[130] Soumaya Rebai, Abderrahmen Amich, Somayeh Molaei, Marouane Kessentini, and Rick Kazman. Multi-objective code reviewer recommendations: balancing expertise, availability and collaborations. *Automated Software Engineering*, pages 301–328, 2020.

[131] Filippo Ricca, Alessandro Marchetto, and Marco Torchiano. On the difficulty of computing the truck factor. In *International Conference on Product Focused Software Process Improvement*, pages 337–351. Springer, 2011.

[132] Peter C Rigby, Yue Cai Zhu, Samuel M Donadelli, and Audris Mockus. Quantifying and mitigating turnover-induced knowledge loss: case studies of chrome and a project at avaya. In *Proceedings of the 38th International Conference on Software Engineering*, pages 1006–1016, 2016.

[133] Linda Rising and Norman S Janoff. The scrum software development process for small teams. *IEEE software*, 17(4):26–32, 2000.

[134] Martin P Robillard. Turnover-induced knowledge loss in practice. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1292–1302, 2021.

[135] Gema Rodríguez-Pérez, Gregorio Robles, Alexander Serebrenik, Andy Zaidman, Daniel M Germán, and Jesus M Gonzalez-Barahona. How bugs are born: a model to identify how bugs are introduced in software components. *Empirical Software Engineering*, pages 1294–1340, 2020.

[136] Christoffer Rosen, Ben Grawi, and Emad Shihab. Commit guru: Analytics and risk prediction of software commits. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 966–969. ACM, 2015.

[137] Shade Ruangwan, Patanamon Thongtanunam, Akinori Ihara, and Kenichi Matsumoto. The impact of human factors on the participation decision of reviewers in modern code review. *Empirical Software Engineering*, 24(2):973–1016, 2019.

[138] Caitlin Sadowski, Emma Söderberg, Luke Church, Michal Sipko, and Alberto Bacchelli. Modern code review: a case study at google. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, pages 181–190, 2018.

[139] Aubrey Scheopner Torres, Jessica Brett, Joshua Cox, and Sara Greller. Competency education implementation: Examining the influence of contextual forces in three new hampshire secondary schools. *AERA Open*, 4(2):2332858418782883, 2018.

[140] Claude Elwood Shannon. A mathematical theory of communication. *ACM SIGMOBILE mobile computing and communications review*, 5(1):3–55, 2001.

[141] Pratyush N Sharma, John Hulland, and Sherae Daniel. Examining turnover in open source software projects using logistic hierarchical linear modeling approach. In *Open Source Systems: Long-Term Sustainability: 8th IFIP WG 2.13 International Conference, OSS 2012, Hammamet, Tunisia, September 10-13, 2012. Proceedings 8*, pages 331–337. Springer, 2012.

[142] Janet Siegmund, Norbert Siegmund, and Sven Apel. Views on internal and external validity in empirical software engineering. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 9–19. IEEE, 2015.

[143] Devarshi Singh, Varun Ramachandra Sekar, Kathryn T Stolee, and Brittany Johnson. Evaluating how static analysis tools can reduce code review effort. In *2017 IEEE symposium on visual languages and human-centric computing (VL/HCC)*, pages 101–105. IEEE, 2017.

[144] Jing Kai Siow, Cuiyun Gao, Lingling Fan, Sen Chen, and Yang Liu. Core: Automating review recommendation for code changes. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 284–295. IEEE, 2020.

[145] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? *ACM sigsoft software engineering notes*, pages 1–5, 2005.

[146] Giriprasad Sridhara, Sourav Mazumdar, et al. Chatgpt: A study on its utility for ubiquitous software engineering tasks. *arXiv preprint arXiv:2305.16837*, 2023.

[147] Miroslaw Staron, Mirosław Ochodek, Wilhelm Meding, and Ola Söder. Using machine learning to identify code fragments for manual review. In *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 513–516. IEEE, 2020.

[148] Anton Strand, Markus Gunnarson, Ricardo Britto, and Muhmmad Usman. Using a context-aware approach to recommend code reviewers: findings from an industrial case study. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*, pages 1–10, 2020.

[149] Emre Sülün, Eray Tüzün, and Uğur Doğrusöz. Rstrace+: Reviewer suggestion using software artifact traceability graphs. *Information and Software Technology*, 130:106455, 2021.

[150] Richard TAY. Correlation, variance inflation and multicollinearity in regression model. *Journal of the Eastern Asia Society for Transportation Studies*, pages 2006–2015, 2017.

[151] K Ayberk Tecimer, Eray Tüzün, Hamdi Dibeklioglu, and Hakan Erdogmus. Detection and elimination of systematic labeling bias in code reviewer recommendation systems. In *Evaluation and Assessment in Software Engineering*, pages 181–190. ACM New York, NY, USA, 2021.

[152] Patanamon Thongtanunam, Raula Gaikovina Kula, Ana Erika Camargo Cruz, Norihiro Yoshida, and Hajimu Iida. Improving code review effectiveness through reviewer

recommendations. In *Proceedings of the 7th International Workshop on Cooperative and Human Aspects of Software Engineering*, pages 119–122, 2014.

[153] Patanamon Thongtanunam, Shane McIntosh, Ahmed E Hassan, and Hajimu Iida. Revisiting code ownership and its relationship with software quality in the scope of modern code review. In *Proceedings of the 38th international conference on software engineering*, pages 1039–1050, 2016.

[154] Patanamon Thongtanunam, Chanathip Pornprasit, and Chakkrit Tantithamthavorn. Autotransform: Automated code transformation to support modern code review process. In *Proceedings of the 44th International Conference on Software Engineering*, pages 237–248, 2022.

[155] Patanamon Thongtanunam, Chakkrit Tantithamthavorn, Raula Gaikovina Kula, Norihiro Yoshida, Hajimu Iida, and Ken-ichi Matsumoto. Who should review my code? a file location-based code-reviewer recommendation approach for modern code review. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 141–150. IEEE, 2015.

[156] Zeynep Ton and Robert S Huckman. Managing the impact of employee turnover on performance: The role of process conformance. *Organization Science*, 19(1):56–68, 2008.

[157] Marco Torchiano, Filippo Ricca, and Alessandro Marchetto. Are web applications more defect-prone than desktop applications? *International journal on software tools for technology transfer*, pages 151–166, 2011.

[158] Adam Tornhill and Markus Borg. Code red: the business impact of code quality-a quantitative study of 39 proprietary production codebases. In *Proceedings of the International Conference on Technical Debt*, pages 11–20, 2022.

[159] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

[160] Rosalia Tufano, Ozren Dabić, Antonio Mastropaolo, Matteo Ciniselli, and Gabriele Bavota. Code review automation: strengths and weaknesses of the state of the art. *IEEE Transactions on Software Engineering*, 2024.

[161] Rosalia Tufano, Simone Masiero, Antonio Mastropaolo, Luca Pascarella, Denys Poshyvanyk, and Gabriele Bavota. Using pre-trained models to boost code review automation. In *Proceedings of the 44th International Conference on Software Engineering*, pages 2291–2302, 2022.

[162] Rosalia Tufano, Luca Pascarella, Michele Tufano, Denys Poshyvanyk, and Gabriele Bavota. Towards automating code review activities. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 163–174. IEEE, 2021.

[163] Asif Kamal Turzo and Amiangshu Bosu. What makes a code review useful to opendev developers? an empirical investigation. *Empirical Software Engineering*, 29(1):6, 2024.

[164] Graham JG Upton. Fisher's exact test. *Journal of the Royal Statistical Society: Series A (Statistics in Society)*, 155(3):395–402, 1992.

[165] Erik Van Der Veen, Georgios Gousios, and Andy Zaidman. Automatically prioritizing pull requests. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 357–361. IEEE, 2015.

[166] Lara Varpio. Using rhetorical appeals to credibility, logic, and emotions to increase your persuasiveness. *Perspectives on medical education*, 7:207–210, 2018.

[167] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[168] Manushree Vijayvergiya, Małgorzata Salawa, Ivan Budiselić, Dan Zheng, Pascal Lamblin, Marko Ivanković, Juanjo Carin, Mateusz Lewko, Jovan Andonov, Goran Petrović, et al. Ai-assisted assessment of coding practices in modern code review. *arXiv preprint arXiv:2405.13565*, 2024.

[169] Giovanni Viviani, Calahan Janik-Jones, Michalis Famelis, Xin Xia, and Gail C Murphy. What design topics do developers discuss? In *Proceedings of the 26th Conference on Program Comprehension*, pages 328–331, 2018.

[170] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *EMNLP*, 2021.

[171] Mairieli Wessel, Alexander Serebrenik, Igor Wiese, Igor Steinmacher, and Marco A Gerosa. Effects of adopting code review bots on pull requests to oss projects. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 1–11. IEEE, 2020.

[172] Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith, and Douglas C Schmidt. A prompt pattern catalog to enhance prompt engineering with chatgpt. *arXiv preprint arXiv:2302.11382*, 2023.

[173] Hyrum K Wright, Miryung Kim, and Dewayne E Perry. Validity concerns in software engineering research. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, pages 411–414, 2010.

[174] Jifeng Xuan, Yan Hu, and He Jiang. Debt-prone bugs: technical debt in software maintenance. *arXiv preprint arXiv:1704.04766*, 2017.

[175] Kai-Hsiang Yang, Tai-Liang Kuo, Hahn-Ming Lee, and Jan-Ming Ho. A reviewer recommendation system based on collaborative intelligence. In *2009 IEEE/WIC/ACM International Joint Conference on Web Intelligence and Intelligent Agent Technology*, pages 564–567. IEEE, 2009.

[176] Xin Ye, Hui Shen, Xiao Ma, Razvan Bunescu, and Chang Liu. From word embeddings to document similarities for improved information retrieval in software engineering. In *Proceedings of the 38th international conference on software engineering*, pages 404–415, 2016.

[177] Xin Ye, Yongjie Zheng, Wajdi Aljedaani, and Mohamed Wiem Mkaouer. Recommending pull request reviewers based on code changes. *Soft Computing*, pages 5619–5632, 2021.

[178] Ying Yin, Yuhai Zhao, Yiming Sun, and Chen Chen. Automatic code review by learning the structure information of code graph. *Sensors*, 23(5):2551, 2023.

[179] Yue Yu, Huaimin Wang, Vladimir Filkov, Premkumar Devanbu, and Bogdan Vasilescu. Wait for it: Determinants of pull request evaluation latency on github. In *2015 IEEE/ACM 12th working conference on mining software repositories*, pages 367–371. IEEE, 2015.

[180] Yue Yu, Huaimin Wang, Gang Yin, and Charles X Ling. Reviewer recommender of pull-requests in github. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 609–612. IEEE, 2014.

[181] Yue Yu, Huaimin Wang, Gang Yin, and Tao Wang. Reviewer recommendation for pull-requests in github: What can we learn from code review and bug assignment? *Information and Software Technology*, pages 204–218, 2016.

[182] Farida El Zanaty, Toshiki Hirao, Shane McIntosh, Akinori Ihara, and Kenichi Matsumoto. An empirical study of design discussions in code review. In *Proceedings of the 12th ACM/IEEE international symposium on empirical software engineering and measurement*, pages 1–10, 2018.

[183] Motahareh Bahrami Zanjani, Huzefa Kagdi, and Christian Bird. Automatically recommending peer reviewers in modern code review. *IEEE Transactions on Software Engineering*, pages 530–543, 2015.

[184] Matej Zečević, Moritz Willig, Devendra Singh Dhami, and Kristian Kersting. Causal parrots: Large language models may talk causality but are not causal. *Transactions on Machine Learning Research*, 2023.

[185] Jiyang Zhang, Chandra Maddila, Ram Bairi, Christian Bird, Ujjwal Raizada, Apoorva Agrawal, Yamini Jhawar, Kim Herzig, and Arie van Deursen. Using large-scale heterogeneous graph representation learning for code review recommendations at microsoft. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 162–172. IEEE, 2023.

[186] Xunhui Zhang, Yue Yu, Gousios Georgios, and Ayushi Rastogi. Pull request decisions explained: An empirical overview. *IEEE Transactions on Software Engineering*, 2022.

[187] Minghui Zhou and Audris Mockus. What make long term contributors: Willingness and opportunity in oss community. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 518–528. IEEE, 2012.

[188] Xin Zhou, Kisub Kim, Bowen Xu, DongGyun Han, Junda He, and David Lo. Generation-based code review automation: How far are we. In *2023 IEEE/ACM 31st International Conference on Program Comprehension (ICPC)*. IEEE, May 2023.

[189] Kun Zhu, Shi Ying, Nana Zhang, and Dandan Zhu. Software defect prediction based on enhanced metaheuristic feature selection optimization and a hybrid deep neural network. *Journal of Systems and Software*, 180:111026, 2021.

[190] Thomas Zimmermann and Nachiappan Nagappan. Predicting defects using network analysis on dependency graphs. In *Proceedings of the 30th international conference on Software engineering*, pages 531–540, 2008.

Part V

Supporting Materials and Appendices

APPENDICES

Appendix A

Experiment details and Supporting Materials for "Studying the Staleness of Code Reviewer Recommendation Systems"

Table A.1: Precision, Recall, and F-Score for the applied time-based filter as a mitigation plan for different approaches on different settings.

Approach	K	$P_{ContributionGap}$	project	Precision	Recall	F-score
LearnRec	1	1 month	Roslyn	100.00	79.43	88.53
LearnRec	2	1 month	Roslyn	100.00	89.70	94.57
LearnRec	3	1 month	Roslyn	100.00	92.51	96.11
LearnRec	1	3 months	Roslyn	100.00	78.17	87.75
LearnRec	2	3 months	Roslyn	100.00	89.39	94.40
LearnRec	3	3 months	Roslyn	100.00	92.33	96.01
LearnRec	1	6 months	Roslyn	100.00	76.23	86.51
LearnRec	2	6 months	Roslyn	100.00	88.95	94.15
LearnRec	3	6 months	Roslyn	100.00	92.10	95.89
LearnRec	1	1 year	Roslyn	100.00	67.04	80.27
LearnRec	2	1 year	Roslyn	100.00	87.12	93.12

Continued on next page

Table A.1 continued from previous page

Approach	K	$P_{ContributionGap}$	Project	Precision	Recall	F-score
LearnRec	3	1 year	Roslyn	100.00	91.07	95.33
LearnRec	1	1 month	Rust	100.00	100.00	100.00
LearnRec	2	1 month	Rust	100.00	100.00	100.00
LearnRec	3	1 month	Rust	100.00	100.00	100.00
LearnRec	1	3 months	Rust	100.00	100.00	100.00
LearnRec	2	3 months	Rust	100.00	100.00	100.00
LearnRec	3	3 months	Rust	100.00	100.00	100.00
LearnRec	1	6 months	Rust	100.00	100.00	100.00
LearnRec	2	6 months	Rust	100.00	100.00	100.00
LearnRec	3	6 months	Rust	100.00	100.00	100.00
LearnRec	1	1 year	Rust	100.00	100.00	100.00
LearnRec	2	1 year	Rust	100.00	100.00	100.00
LearnRec	3	1 year	Rust	100.00	100.00	100.00
LearnRec	2	1 month	Kubernetes	100.00	99.93	99.97
LearnRec	3	1 month	Kubernetes	100.00	95.00	97.44
LearnRec	1	3 months	Kubernetes	100.00	100.00	100.00
LearnRec	2	3 months	Kubernetes	100.00	99.93	99.97
LearnRec	3	3 months	Kubernetes	100.00	94.75	97.31
LearnRec	1	6 months	Kubernetes	100.00	100.00	100.00
LearnRec	2	6 months	Kubernetes	100.00	99.93	99.96
LearnRec	3	6 months	Kubernetes	100.00	94.15	96.99
LearnRec	1	1 year	Kubernetes	100.00	100.00	100.00
LearnRec	2	1 year	Kubernetes	100.00	99.89	99.95
LearnRec	3	1 year	Kubernetes	100.00	91.61	95.62
cHRev	1	1 month	Roslyn	100.00	3.90	7.51
cHRev	2	1 month	Roslyn	100.00	5.01	9.54
cHRev	3	1 month	Roslyn	100.00	6.03	11.37
cHRev	1	3 months	Roslyn	100.00	3.69	7.12
cHRev	2	3 months	Roslyn	100.00	4.64	8.88
cHRev	3	3 months	Roslyn	100.00	5.43	10.31
cHRev	1	6 months	Roslyn	100.00	2.57	5.02
cHRev	2	6 months	Roslyn	100.00	3.47	6.71
cHRev	3	6 months	Roslyn	100.00	4.19	8.04

Continued on next page

Table A.1 continued from previous page

Approach	K	$P_{ContributionGap}$	Project	Precision	Recall	F-score
cHRev	1	1 year	Roslyn	100.00	1.29	2.54
cHRev	2	1 year	Roslyn	100.00	2.13	4.18
cHRev	3	1 year	Roslyn	100.00	2.74	5.33
cHRev	1	1 month	Rust	100.00	5.90	11.15
cHRev	2	1 month	Rust	100.00	8.82	16.21
cHRev	3	1 month	Rust	100.00	10.43	18.89
cHRev	1	3 months	Rust	100.00	4.94	9.41
cHRev	2	3 months	Rust	100.00	7.49	13.94
cHRev	3	3 months	Rust	100.00	8.97	16.47
cHRev	1	6 months	Rust	100.00	3.93	7.57
cHRev	2	6 months	Rust	100.00	5.95	11.24
cHRev	3	6 months	Rust	100.00	7.25	13.52
cHRev	1	1 year	Rust	100.00	2.36	4.62
cHRev	2	1 year	Rust	100.00	3.77	7.26
cHRev	3	1 year	Rust	100.00	4.68	8.95
cHRev	1	1 month	Kubernetes	100.00	7.18	13.40
cHRev	2	1 month	Kubernetes	100.00	9.42	17.21
cHRev	3	1 month	Kubernetes	100.00	10.46	18.94
cHRev	1	3 months	Kubernetes	100.00	5.82	11.00
cHRev	2	3 months	Kubernetes	100.00	8.00	14.81
cHRev	3	3 months	Kubernetes	100.00	8.81	16.19
cHRev	1	6 months	Kubernetes	100.00	4.30	8.25
cHRev	2	6 months	Kubernetes	100.00	6.10	11.49
cHRev	3	6 months	Kubernetes	100.00	6.78	12.70
cHRev	1	1 year	Kubernetes	100.00	1.99	3.90
cHRev	2	1 year	Kubernetes	100.00	3.19	6.18
cHRev	3	1 year	Kubernetes	100.00	3.58	6.92
Sofia	1	1 month	Roslyn	100.00	2.98	5.78
Sofia	2	1 month	Roslyn	100.00	3.76	7.25
Sofia	3	1 month	Roslyn	100.00	4.35	8.33
Sofia	1	3 months	Roslyn	100.00	2.84	5.52
Sofia	2	3 months	Roslyn	100.00	3.47	6.70
Sofia	3	3 months	Roslyn	100.00	3.87	7.44

Continued on next page

Table A.1 continued from previous page

Approach	K	$P_{ContributionGap}$	Project	Precision	Recall	F-score
Sofia	1	6 months	Roslyn	100.00	1.95	3.82
Sofia	2	6 months	Roslyn	100.00	2.52	4.92
Sofia	3	6 months	Roslyn	100.00	2.87	5.57
Sofia	1	1 year	Roslyn	100.00	0.99	1.96
Sofia	2	1 year	Roslyn	100.00	1.51	2.97
Sofia	3	1 year	Roslyn	100.00	1.78	3.49
Sofia	1	1 month	Rust	100.00	3.39	6.56
Sofia	2	1 month	Rust	100.00	5.36	10.18
Sofia	3	1 month	Rust	100.00	6.51	12.22
Sofia	1	3 months	Rust	100.00	2.69	5.24
Sofia	2	3 months	Rust	100.00	4.44	8.50
Sofia	3	3 months	Rust	100.00	5.50	10.42
Sofia	1	6 months	Rust	100.00	2.10	4.11
Sofia	2	6 months	Rust	100.00	3.44	6.64
Sofia	3	6 months	Rust	100.00	4.33	8.30
Sofia	1	1 year	Rust	100.00	1.01	1.99
Sofia	2	1 year	Rust	100.00	1.95	3.82
Sofia	3	1 year	Rust	100.00	2.55	4.97
Sofia	1	1 month	Kubernetes	100.00	5.73	10.84
Sofia	2	1 month	Kubernetes	100.00	7.56	14.06
Sofia	3	1 month	Kubernetes	100.00	8.46	15.60
Sofia	1	3 months	Kubernetes	100.00	4.52	8.65
Sofia	2	3 months	Kubernetes	100.00	6.31	11.87
Sofia	3	3 months	Kubernetes	100.00	7.01	13.10
Sofia	1	6 months	Kubernetes	100.00	3.17	6.15
Sofia	2	6 months	Kubernetes	100.00	4.67	8.93
Sofia	3	6 months	Kubernetes	100.00	5.28	10.02
Sofia	1	1 year	Kubernetes	100.00	1.46	2.89
Sofia	2	1 year	Kubernetes	100.00	2.43	4.75
Sofia	3	1 year	Kubernetes	100.00	2.76	5.37
WRLREC	1	1 month	Roslyn	15.91	80.72	26.59
WRLREC	2	1 month	Roslyn	16.87	67.65	27.01
WRLREC	3	1 month	Roslyn	17.76	60.18	27.42

Continued on next page

Table A.1 continued from previous page

Approach	K	$P_{ContributionGap}$	Project	Precision	Recall	F-score
WRLREC	1	3 months	Roslyn	15.79	81.29	26.45
WRLREC	2	3 months	Roslyn	16.72	67.79	26.82
WRLREC	3	3 months	Roslyn	17.51	60.35	27.14
WRLREC	1	6 months	Roslyn	15.85	79.75	26.45
WRLREC	2	6 months	Roslyn	16.47	67.44	26.47
WRLREC	3	6 months	Roslyn	17.06	61.24	26.68
WRLREC	1	1 year	Roslyn	15.56	80.88	26.11
WRLREC	2	1 year	Roslyn	16.18	69.34	26.24
WRLREC	3	1 year	Roslyn	16.70	61.26	26.25
WRLREC	1	1 month	Rust	17.24	57.01	26.48
WRLREC	2	1 month	Rust	21.16	53.13	30.27
WRLREC	3	1 month	Rust	24.74	50.71	33.25
WRLREC	1	3 months	Rust	17.18	57.03	26.41
WRLREC	2	3 months	Rust	20.95	53.49	30.10
WRLREC	3	3 months	Rust	24.57	50.60	33.08
WRLREC	1	6 months	Rust	17.09	56.80	26.28
WRLREC	2	6 months	Rust	20.79	53.34	29.92
WRLREC	3	6 months	Rust	24.30	50.78	32.87
WRLREC	1	1 year	Rust	17.03	56.43	26.17
WRLREC	2	1 year	Rust	20.58	53.70	29.75
WRLREC	3	1 year	Rust	23.99	50.67	32.56
WRLREC	1	1 month	Kubernetes	22.13	94.71	35.87
WRLREC	2	1 month	Kubernetes	20.38	90.70	33.28
WRLREC	3	1 month	Kubernetes	18.76	85.28	30.75
WRLREC	1	3 months	Kubernetes	22.08	95.05	35.83
WRLREC	2	3 months	Kubernetes	20.37	90.60	33.26
WRLREC	3	3 months	Kubernetes	18.71	84.93	30.67
WRLREC	1	6 months	Kubernetes	22.09	94.82	35.83
WRLREC	2	6 months	Kubernetes	20.29	91.23	33.20
WRLREC	3	6 months	Kubernetes	18.63	85.82	30.61
WRLREC	1	1 year	Kubernetes	22.06	95.12	35.82
WRLREC	2	1 year	Kubernetes	20.26	90.72	33.13
WRLREC	3	1 year	Kubernetes	18.59	85.56	30.54

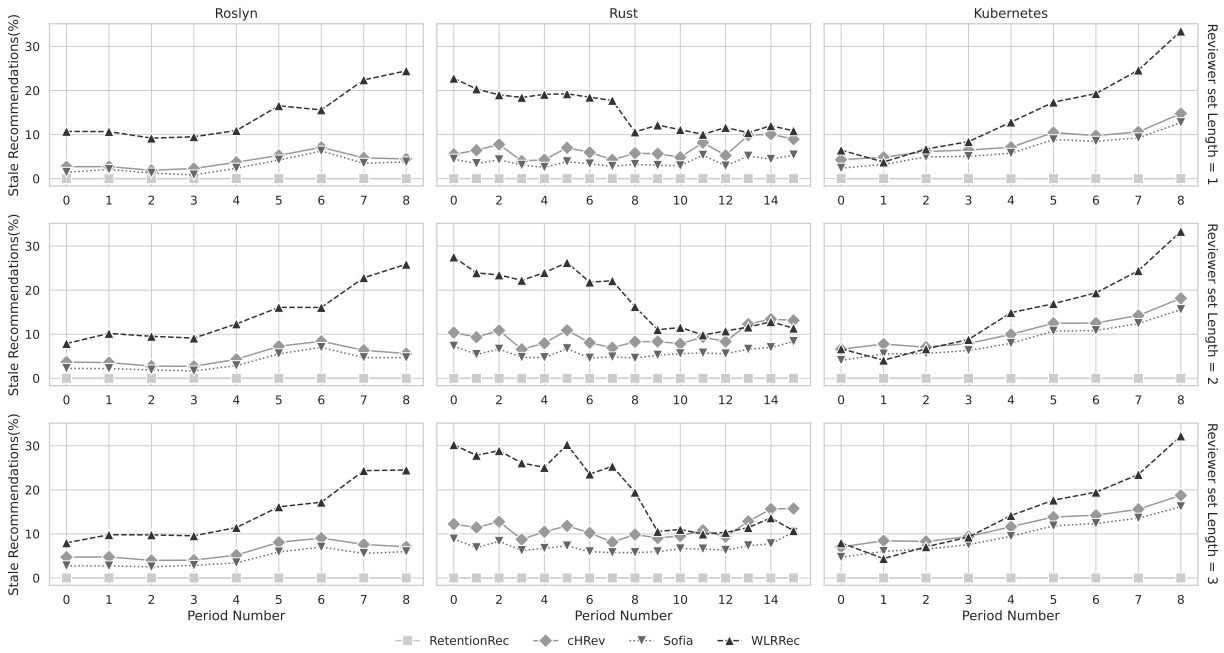


Figure A.1: The proportions of stale to all recommendations (y-axis). The period numbers are normalized, with zero representing the oldest period.

Table A.3: Reduction of SRR in all the recommendations when time-based filter is applied for different Settings.

Approach	$P_{ContributionGap}$	Project	Stale Rec. Ratio (%)	Stale Rec. Ratio Improv.(%)
LearnRec	1 month	Roslyn	7.93%	91.15%
LearnRec	3 months	Roslyn	19.63%	78.09%
LearnRec	6 months	Roslyn	29.08%	67.54%
LearnRec	1 year	Roslyn	55.09%	38.51%
LearnRec	1 month	Rust	7.52%	92.48%
LearnRec	3 months	Rust	21.56%	78.44%
LearnRec	6 months	Rust	38.06%	61.94%
LearnRec	1 year	Rust	66.32%	33.68%
LearnRec	1 month	Kubernetes	8.91%	90.86%
LearnRec	3 months	Kubernetes	27.36%	71.94%

Continued on next page

Approach	$P_{ContributionGap}$	Project	Stale Rec. Ratio (%)	Stale Rec. Ratio Improv.(%)
LearnRec	6 months	Kubernetes	50.45%	48.26%
LearnRec	1 year	Kubernetes	78.07%	19.93%
cHRev	1 month	Roslyn	0.44%	92.16%
cHRev	3 months	Roslyn	1.04%	81.49%
cHRev	6 months	Roslyn	2.31%	58.86%
cHRev	1 year	Roslyn	3.81%	32.28%
cHRev	1 month	Rust	1.13%	88.65%
cHRev	3 months	Rust	2.77%	72.13%
cHRev	6 months	Rust	4.62%	53.60%
cHRev	1 year	Rust	7.15%	28.13%
cHRev	1 month	Kubernetes	1.17%	88.79%
cHRev	3 months	Kubernetes	3.17%	69.58%
cHRev	6 months	Kubernetes	5.42%	48.03%
cHRev	1 year	Kubernetes	8.20%	21.44%
Sofia	1 month	Roslyn	0.32%	92.39%
Sofia	3 months	Roslyn	0.76%	81.62%
Sofia	6 months	Roslyn	1.77%	57.33%
Sofia	1 year	Roslyn	2.84%	31.40%
Sofia	1 month	Rust	0.69%	88.83%
Sofia	3 months	Rust	1.74%	71.84%
Sofia	6 months	Rust	2.84%	54.17%
Sofia	1 year	Rust	4.44%	28.22%
Sofia	1 month	Kubernetes	0.97%	88.56%
Sofia	3 months	Kubernetes	2.58%	69.57%
Sofia	6 months	Kubernetes	4.39%	48.20%
Sofia	1 year	Kubernetes	6.58%	22.42%
WLRRec	1 month	Roslyn	1.67%	89.45%
WLRRec	3 months	Roslyn	4.16%	73.64%
WLRRec	6 months	Roslyn	7.74%	50.97%
WLRRec	1 year	Roslyn	12.38%	21.62%
WLRRec	1 month	Rust	2.06%	88.89%
WLRRec	3 months	Rust	4.97%	73.20%
WLRRec	6 months	Rust	8.52%	54.03%
WLRRec	1 year	Rust	12.49%	32.61%

Continued on next page

Approach	$P_{ContributionGap}$	Project	Stale Rec. Ratio (%)	Stale Rec. Ratio Improv.(%)
WLRRec	1 month	Kubernetes	2.35%	87.87%
WLRRec	3 months	Kubernetes	6.19%	68.10%
WLRRec	6 months	Kubernetes	10.19%	47.47%
WLRRec	1 year	Kubernetes	14.79%	23.77%

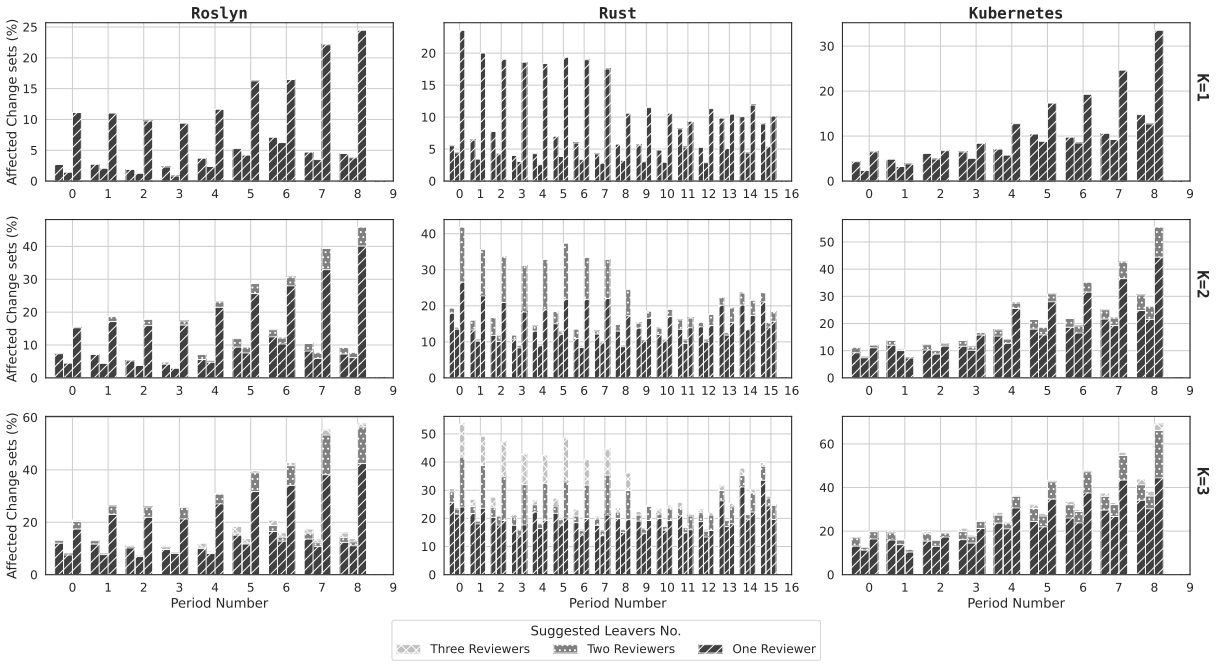


Figure A.2: Percentage of changesets impacted by stale recommendations. Each set of bars shows results for cHRev (left) [183], Sofia [109](middle), and WLRRec [2](right) in each period.

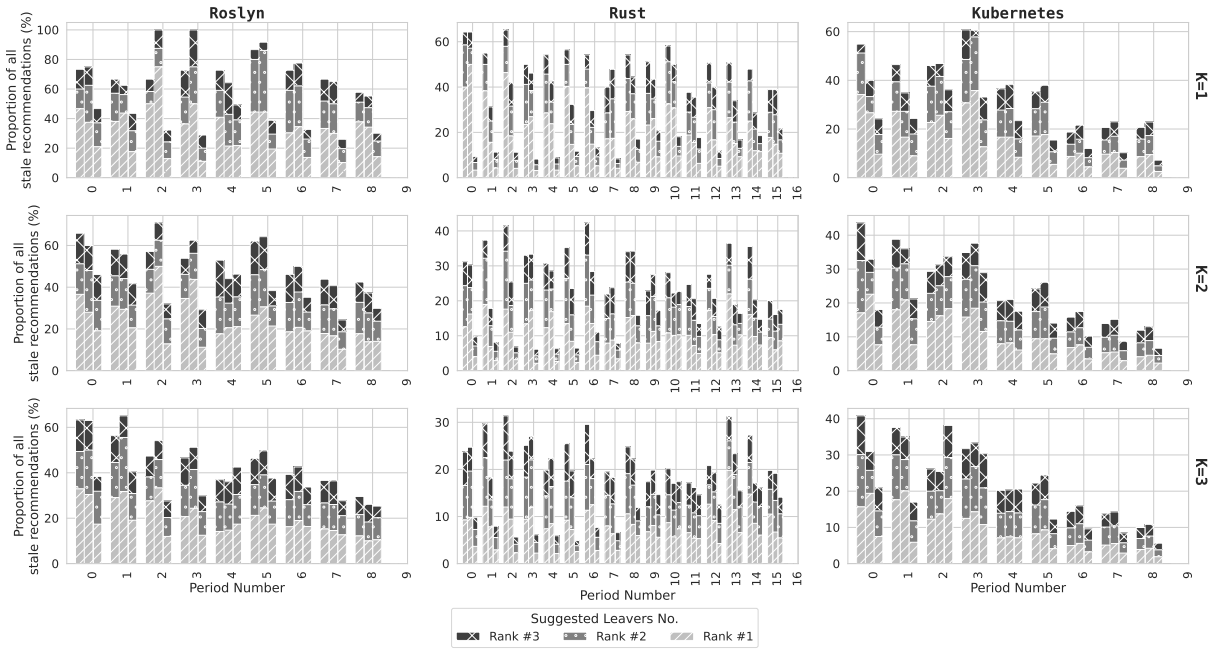


Figure A.3: The top-3 most frequently stale recommendations and their share of all suggested leavers for cHRev (leftmost bar), Sofia (middle bar), and WLRRec (right bar) for various periods under different conditions

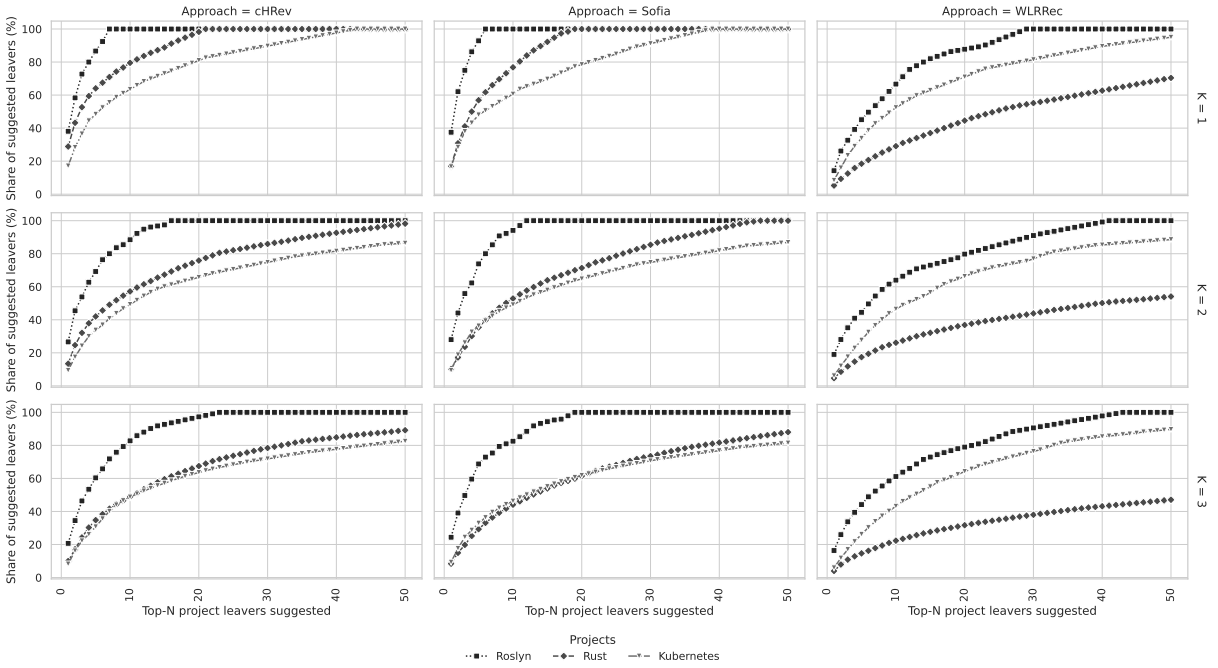


Figure A.4: Share of top-N significant leavers for different review set sizes (K) for different projects and approaches. The increase of Share, considering more top-N project leavers, increased the logarithmic trend.

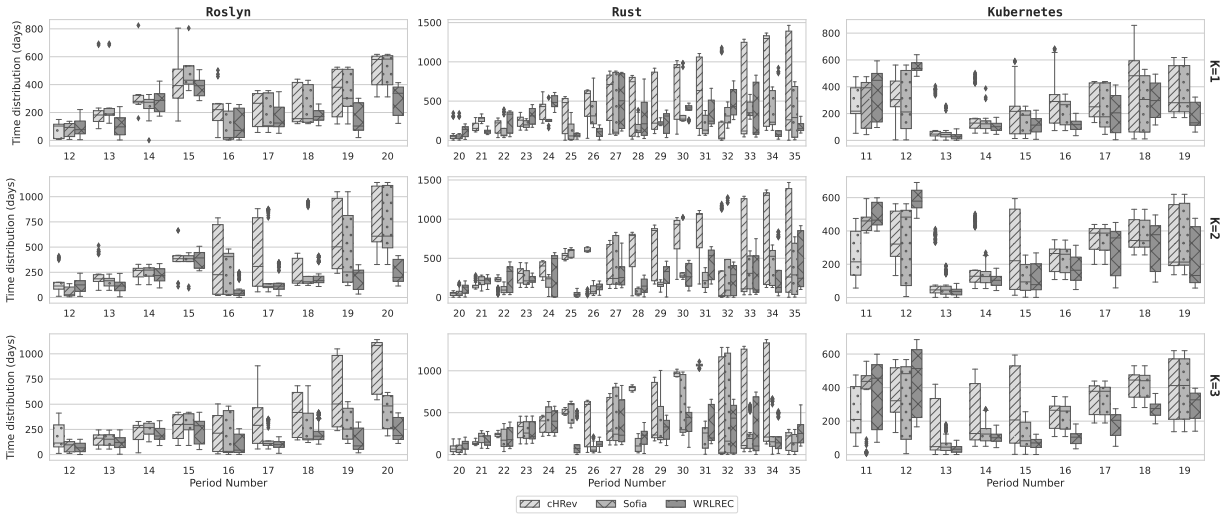
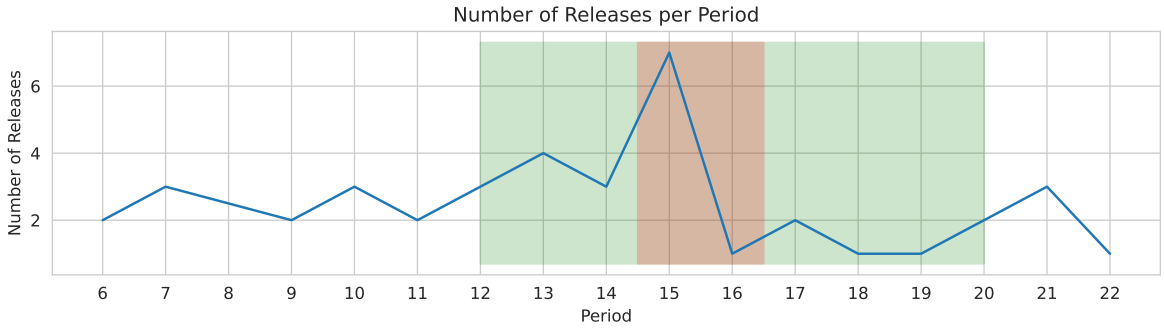


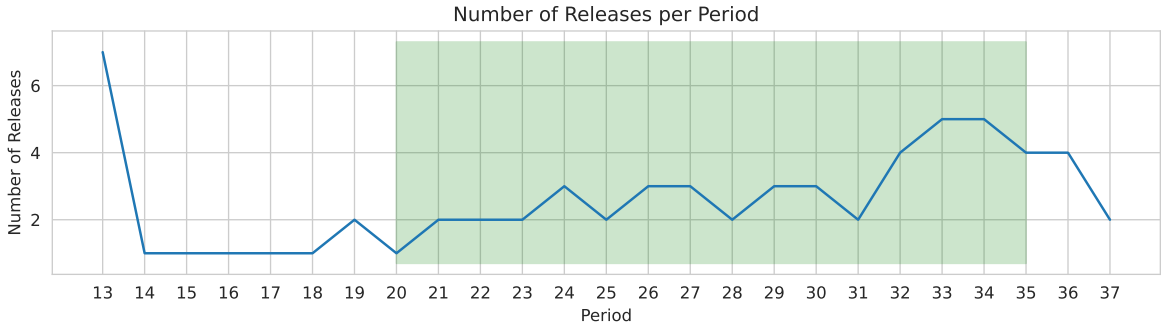
Figure A.5: The distribution of lingering time for the top-3 reviewers over quarterly periods.

Table A.2: The Developers' Work Load Ratio (DWLR) for top-3 reviewers of each Approach. This indicate that by limiting the $P_{ContibutionGap}$ the top-3 reviewers are recommended more often.

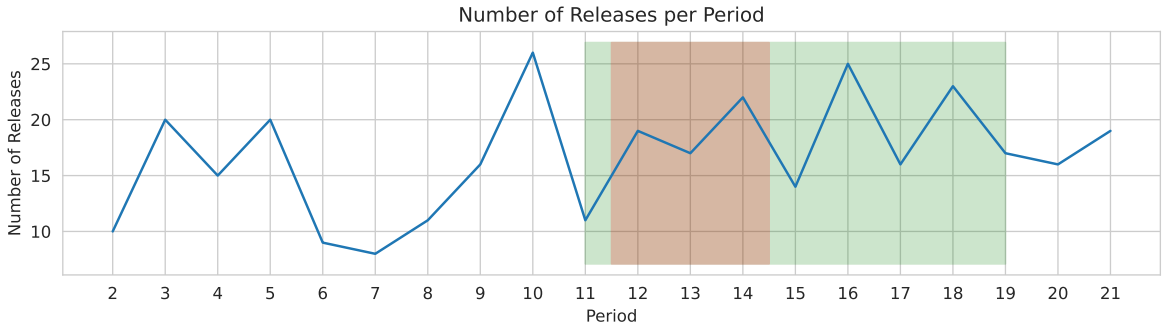
mean	std	Q1	median	Q3	max	Approach	Strategy Name	ContibGap
26.11	21.47	11.76	19.15	33.33	100.00	cHRev	Time-based Filter	1 month
23.42	19.78	10.79	16.21	28.57	100.00	cHRev	Time-based Filter	3 months
21.19	17.63	9.76	14.81	25.00	100.00	cHRev	Time-based Filter	6 months
18.55	15.61	8.61	13.33	22.22	100.00	cHRev	Time-based Filter	1 year
17.01	14.77	7.93	12.29	19.28	100.00	cHRev	No Filter	-
26.97	21.54	12.50	20.00	33.33	100.00	Sofia	Time-based Filter	1 month
23.34	19.80	10.71	16.67	27.87	100.00	Sofia	Time-based Filter	3 months
21.30	18.57	9.76	15.00	25.00	100.00	Sofia	Time-based Filter	6 months
18.88	16.64	8.57	13.10	23.08	100.00	Sofia	Time-based Filter	1 year
17.21	16.04	7.30	12.00	21.05	100.00	Sofia	No Filter	-
17.82	14.74	8.63	13.41	21.43	100.00	WLRRec	Time-based Filter	1 month
14.96	13.15	7.14	11.11	18.22	100.00	WLRRec	Time-based Filter	3 months
13.05	11.89	6.56	9.69	16.33	100.00	WLRRec	Time-based Filter	6 months
11.15	11.53	5.04	8.33	13.33	100.00	WLRRec	Time-based Filter	1 year
9.24	10.50	4.07	6.59	10.87	100.00	WLRRec	No Filter	-



(a) Roslyn

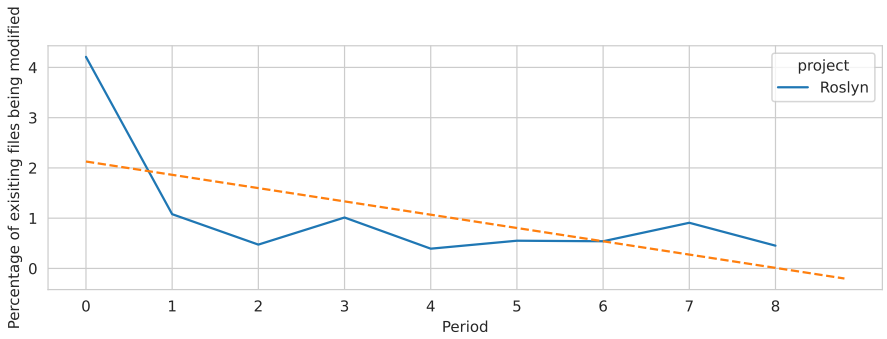


(b) Rust

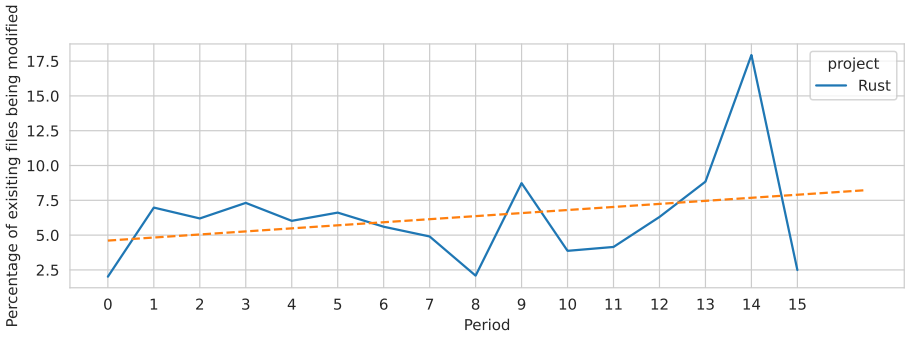


(c) Kubernetes

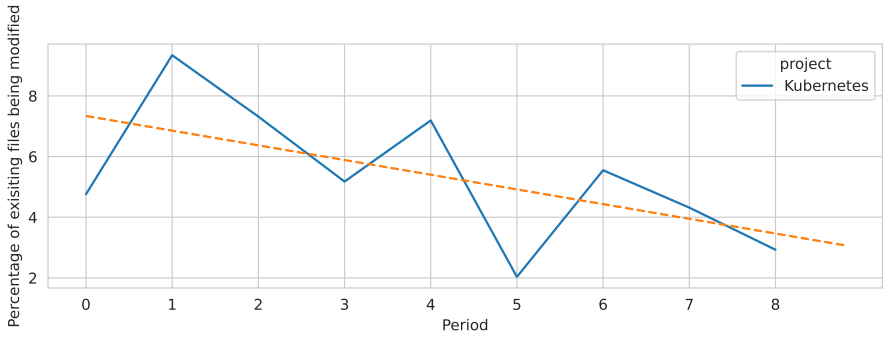
Figure A.6: Number of releases per period for per period and per project. The green shades show the studied period (more than 80% review rate), and the red shades show the sudden drops in top-3 reviewers' share in stale recommendations.



(a) Roslyn

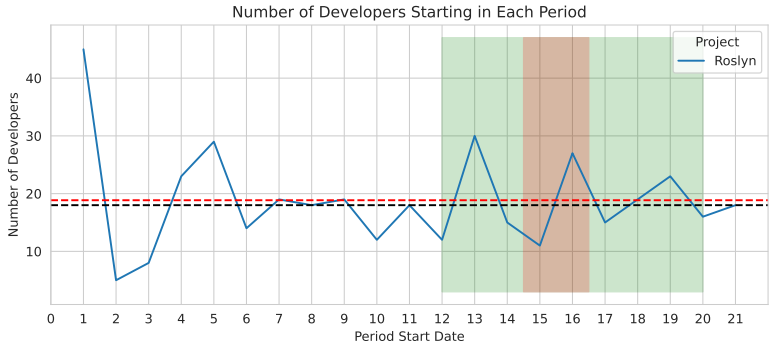


(b) Rust

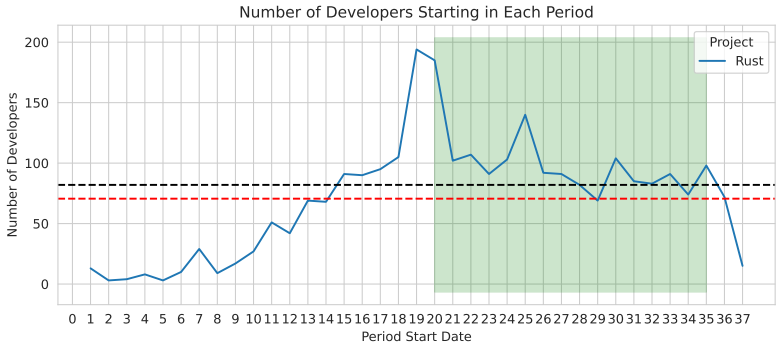


(c) Kubernetes

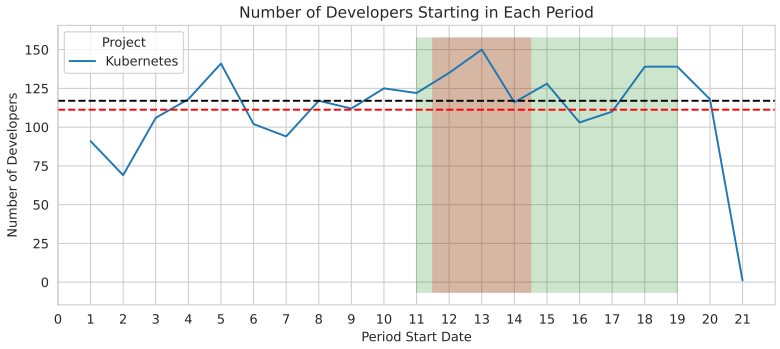
Figure A.7: The percentage of existing files being modified per period per project. The orange dashed line is the Linear extrapolation in each graph, showing the trend. The periods are normalized, and only studied periods are shown.



(a) Roslyn

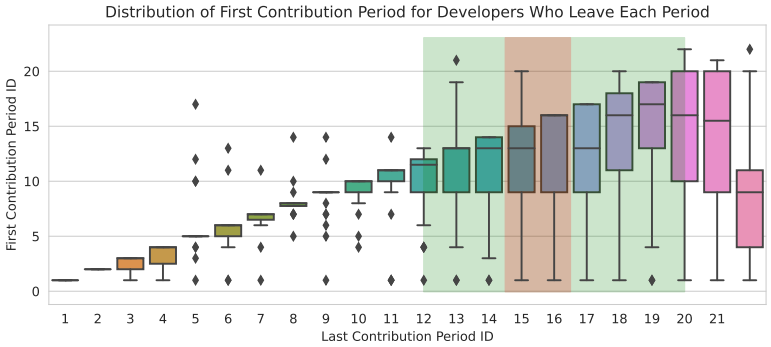


(b) Rust

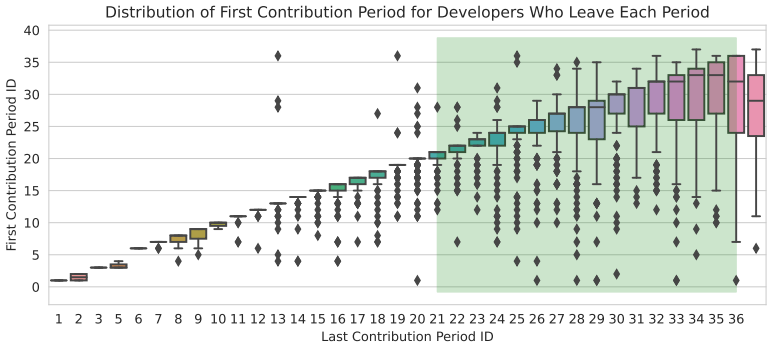


(c) Kubernetes

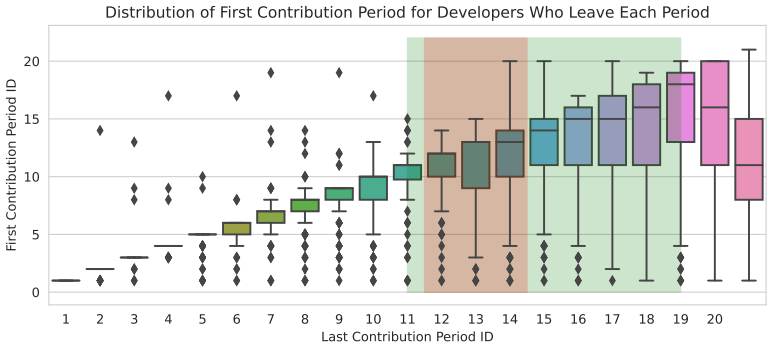
Figure A.8: Rate of new developers joining each project per period. The red and black dashed lines indicate the median and average numbers over these periods, respectively. The green shades show the studied period (more than 80% review rate), and the red shades show the sudden drops in top-3 reviewers' share in stale recommendations.



(a) Roslyn



(b) Rust



(c) Kubernetes

Figure A.9: Contributions of developers who left each project per period. The green shades show the studied period (more than 80% review rate), and the red shades show the sudden drops in top-3 reviewers' share in stale recommendations.

Appendix B

Experiment details and Supporting Materials for "Exploring the notion of risk in Code Reviewer Recommendation Systems"

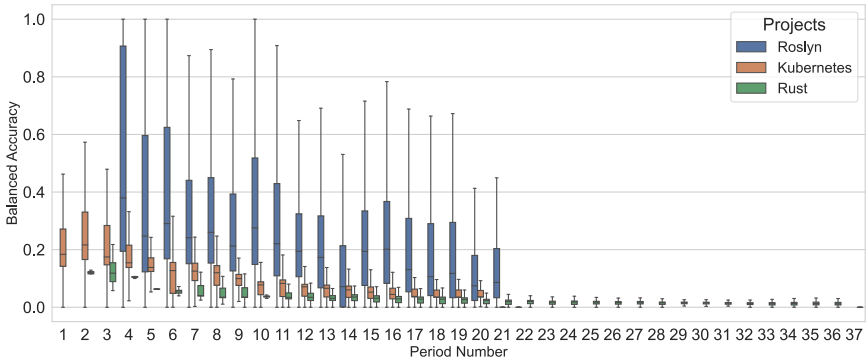


Figure B.1: The distribution of predicted defect probability of different projects.