Towards modeling and improving human-centered code review

Gullstrand Heander, Lo

2025

Document Version:
Publisher's PDF, also known as Version of record

Towards Modeling and Improving Human-Centered Code Review

Lo Gullstrand Heander

Licentiate Thesis, 2025

Department of Computer Science
Lund University

ISBN 978-91-8104-675-5 (printed version)
ISBN 978-91-8104-676-2 (electronic version)
ISSN 1652-4691
Licentiate Thesis 3, 2025

Department of Computer Science
Lund University
Box 118
SE-221 00 Lund
Sweden

Email: lo.gullstrand_heander@cs.lth.se
WWW: https://portal.research.lu.se/en/persons/lo-gullstrand-heander

Printed in Sweden by Tryckeriet i E-huset, Lund, 2025

ABSTRACT

Tool-based code review has been an established software engineering practice for at least a decade. However, while software development environments have improved significantly during this time with advanced features for code comprehension, refactoring, and AI support, code review tools have remained more static and are still centered around a two-way textual diff view with features similar to when the first code review tools were introduced.

With the rapid development of artificial intelligence (AI), code review is at a crucial moment. It must adapt to meet the demands of a future where more and more AI generated code needs to be reviewed, while higher efficiency demands are placed on software engineering teams. More and more capable AI models will soon make it feasible to completely automate code review or offer sophisticated AI support to human code reviewers. Complete automation could potentially offer increased efficiency, but risk losing many of the interpersonal benefits. This gives researchers and software engineers reason to stop and reflect on what the purpose and benefits of code review are and how to best preserve these benefits in the future.

In this thesis, I present a direction for modeling and improving human-centered code review, where code review tools are designed to support the human software engineer, adapt to their needs, and augment their capabilities. The contributions are a prototype for flexible code block comparisons developed using participatory design, an architecture for AI-supported code review, and a cognitive model of code review as decision-making (CRDM). Together, these contributions indicate one way toward the next generation of code review tools, practices, and processes: to use participatory design methodology, cognitive insights from the CRDM model, and AI agent-based architectures to improve code review while focusing on the needs of the human reviewers.

ACKNOWLEDGEMENTS

I would like to dedicate this licentiate thesis to my wonderful family, sharing so much happiness, mutual support, curiosity, and encouragement. My children Melvin and Leo, my bonus children Vide, Rönn, Lärke and Björk, my amazing wife Alva, and my parents Jan and Christine. You fill my life with love and inspiration!

My warmest thanks to my supervisors Emma Söderberg and Christofer Rydenfält for your guidance, inspiration, support, hard work, and knowledge. The way you are always open to discuss any aspect of research, thinking, and creativity makes working together both an inspiration and a joy.

Special thanks to my amazing friends and colleagues Andreas Bexell, Ken Engström, Carl Wolff, Sebastian Andersson, Peng Kuang, Arthur Nijdam, Anton Risberg Alaküla, Daria Rago, and Leo Esson for inspiration, discussions, and feedback. Being a research student can feel lonely at times, and I'm ever grateful for your support and community.

Further, I also want to thank all of the study participants for sharing their insights, ideas, and collaboration. For funding this thesis, I thank the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

"The creative adult is the child who has survived."
—Ursula K. Le Guin

Lo Gullstrand Heander
Lund, August 2025

List of Publications

In the introduction chapter of this thesis, the included and related publications listed below are referred to by Roman numerals.

Publications Included in the Thesis

I Lo Gullstrand Heander, Emma Söderberg, Christofer Rydenfält "Design of Flexible Code Block Comparisons to Improve Code Review of Refactored Code" In *Companion Proceedings of the 8th International Conference on the Art, Science, and Engineering of Programming*, 2024.
DOI: 10.1145/3660829.3660842

II Lo Gullstrand Heander, Emma Söderberg, Christofer Rydenfält "Support, Not Automation: Towards AI-supported Code Review for Code Quality and Beyond" In *33rd ACM International Conference on the Foundations of Software Engineering (FSE Companion '25)*, 2025.
DOI: 10.1145/3696630.3728505

III Lo Gullstrand Heander, Emma Söderberg, Christofer Rydenfält, "Code Review as Decision-Making – Building a Cognitive Model from the Questions Asked During Code Review" Submitted to *Empirical Software Engineering, Springer Science*, 2025.
DOI: 10.48550/arXiv.2507.09637

Related Publications

VI William Saranpää, Felix Apell Skjutar, Lo Gullstrand Heander, Emma Söderberg, Diederick C. Niehorster, Olivia Mattsson, Hedda Klintskog, Luke Church "GANDER: a Platform for Exploration of Gaze-driven Assistance in Code Review" In *Proceedings of the 2023 Symposium on Eye Tracking*

Research and Applications, 2023.
DOI: 10.1145/3588015.3589191

VII Andreas Bexell, Lo Gullstrand Heander, Emma Söderberg, Sigrid Eldh, Per Runeson "Exploring the Performance of ML Model Size for Classification in Relation to Energy Consumption" Submitted to *26th International Conference on Product-Focused Software Process Improvement*, 2025.

Artifacts and Demonstrations Related to the Publications

Artifact I Lo Gullstrand Heander "Prototype of Flexible Code Block Comparisons"
URL: https://figma.com/proto/KZIrsBH8DZ22ZI0B0YD2BC/GBC?hide-ui=1

FIGMA prototype demonstrating the UX design for flexible code block comparisons in Gerrit. It was used both during the design process and for design validation with the participants.

- Lo Gullstrand Heander led and did almost all of the work.

Artifact II Lo Gullstrand Heander "Replication Package for Code Review as Decision Making" DOI: 10.5281/zenodo.15758267

Replication package containing source code for statistical, temporal, and sequential analysis, as well as code book and process coded data.

- Lo Gullstrand Heander led and did almost all of the work.

Contribution Statement

	Paper I	**Paper II**	**Paper III**
Conceptualization	◐	◒	◒
Methodology	●	◐	◒
Investigation	●	◒	●
Analysis	●	◒	◒
Writing	◒	◒	◒
Visualization	●	●	●

Table 1: Author's contributions for each paper and category.

All papers included in this thesis have been co-authored with other researchers. The contributions of Lo Gullstrand Heander are illustrated in Table 1. The definitions for the headings in the table follow the CRediT [1] terminology. The dark

portion of the circles represents the individual contributions by Lo Gullstrand Heander for each paper and category:

- ◒ Lo Gullstrand Heander was a minor contributor to the work.
- ◐ Lo Gullstrand Heander was a contributor to the work.
- ◓ Lo Gullstrand Heander led and did a majority of the work.
- ● Lo Gullstrand Heander led and did almost all of the work.

CONTENTS

INTRODUCTION

1 Introduction

Code review is an established industry practice in software engineering [2]. Benefits include defect finding, code improvement, discovering alternative solutions, knowledge transfer, team awareness, improving developer process, sharing code ownership, reducing build breaks, tracking rationale, and team assessment [3]. But there are also significant challenges. Some developers view code review as a boring chore [4], unhelpful comments or other poor code review practices harm collaboration [5], and there are several misalignments between the task, the developers' needs, and the tools [6].

So far, code review tools have not evolved to meet these challenges. The first dedicated software tool for code review, ICICLE, was launched in 1990 and already contained a two-way diff view, support for code review comments, annotations by static analyzers, and distributed collaboration over the network [7]. The most popular code review tools in use today, such as GitHub, GitLab, and Gerrit, are web-based, integrated with version control, can link to continuous integration systems, and show the code in full color, but are still based on the same features and core workflows. I believe there is ample room for improvement to address the current challenges and misalignments mentioned above.

The pressure to improve code review is increasing. The 2023 Accelerate State of DevOps industry report finds that teams with faster code reviews have up to 50% higher overall software delivery performance, marking it as an important area for improvement [8]. Furthermore, with the accelerating research and application of Artificial Intelligence (AI) in software engineering, developers are predicted to spend even more time reviewing and integrating AI-generated code [9, 10]. Code review is at a crucial moment where it must adapt and become more efficient and help developers deal with increasing amounts of code to review, and at the same

time preserve and increase the interpersonal and quality benefits that made code review a valued industry practice.

The fast development of AI technologies offers both threats and opportunities for the future of code review. Although several recent initiatives explore how to completely automate code review [11–14], I argue that many of the benefits of code review are interpersonal and risk being lost if the activity is automated. Instead, improvements in code review tools and processes should preserve the benefits and values of code review, address its difficulties, and center the human software engineers involved.

Already in 1962, Douglas Engelbart [15] published a seminal report demonstrating how new technology can augment cognitive and physical abilities and allow humans to perform tasks faster and easier. The example he gives in his article is the workflow of an 'augmented architect'; a human architect working assisted by a computer system that can draw the building plans, measure angles and distances, and update visualizations in real time. When described in the article, a system like this was still decades away, but today Computer Aided Design (CAD) systems are widely used for technical design tasks. In Engelbart's view, when developing a tool, you should begin by questioning what is hard about the task and then how a tool can help. Following this principle; to design improved code review tools that augment humans' ability and efficiency, a better understanding of code review and its difficulties from a human perspective is needed.

1.1 Research Goals

This thesis explores how code review tools, practices, and processes work and how they can be improved to fit the goals and needs of the humans involved. My vision is to be part of creating a world where developers *enjoy* doing code reviews and software teams achieve better collaboration, fewer code defects, increased knowledge sharing, and stronger team cohesion. The central research question springing from this goal and vision is **"how can code reviews be made fit for purpose?"**. With *fit for purpose*, I refer to tailored to the needs and goals of the diverse roles involved in code review. Here, roles imply not only authors or reviewers, but also variants of these roles. Reviewers may have different units of attention [6], they may act as an educator in one review and a gatekeeper in the next [2], or they may just want to keep up to date with what is happening in the code base. The authors also vary in needs and roles. For example, they may be very senior on the team or they might have started last week, they might perform changes as the lead developer in a project or as a guest contributing changes in a repository outside their regular team.

Specifically, the research goal and the research question are applied to the included papers in the following ways. Paper I explores how participatory design can be used to improve the user experience of code review by designing a flexible code block comparison interface that addresses common difficulties with review of

refactored code. Paper II proposes an architecture for human-centered AI support designed to make code reviews faster and more efficient by providing reviewers with the information they need when it is needed. Finally, Paper III presents a cognitive model of code review that highlights the similarities between code review and decision-making and can be used to deepen the understanding of developers' needs and design future tool and process improvements.

1.2 Contributions

The research presented in Paper I, Paper II, and Paper III contributes:

- **C_1** Prototype of a flexible code block comparison tool (Section 5.1).
- **C_2** An architecture for AI-supported code review (Section 5.2).
- **C_3** A cognitive model of code review as decision-making (Section 5.3).

2 Modern Code Review

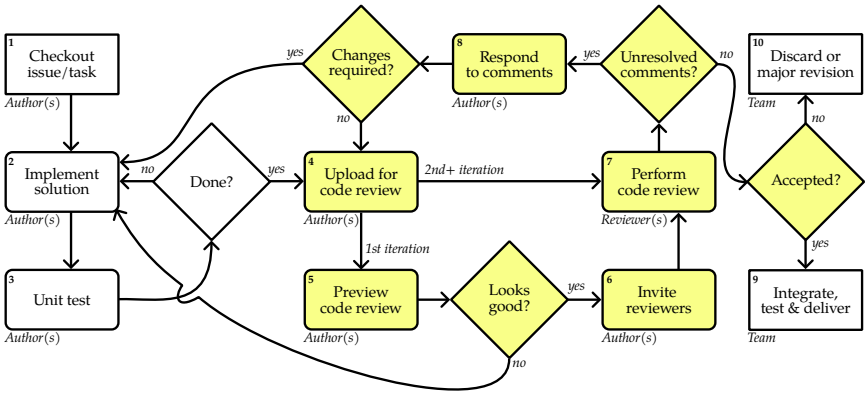

Figure 1: Example of a software development process with modern code review steps highlighted in yellow. **Takeaway:** Code review is an iterative process where authors and reviewers take turns updating and commenting until the code is accepted for integration, or, in rare cases, discarded.

In its essence, code review means that code written by one software engineer (the author) is inspected, commented upon, and eventually approved (or rejected) by one or more other software engineers (the reviewers). Today, code review is most often asynchronous and distributed, with each reviewer looking through and

commenting on the changed code when they have time and from their own workstations over a web-based code review tool such as GitLab, GitHub, Gerrit, and Critique [2].

Code review has its origins in formal (Fagan-style) code inspection meetings, where the source code was inspected following a formal process in a meeting with the author, reviewers, a moderator, a reader, and a recorder [16]. The moderator led the meeting while the reader paraphrased the code changes out loud. The reviewers gave feedback on the code, while the recorder documented the meeting for future archival and analysis. Famous projects using code inspections include the development of NASA space shuttle software [17]. As software development workflows became more informal [18] and the use of tools became ubiquitous, code inspections developed into what Bacchelli and Bird [3] defined as Modern Code Review: *"(1) informal (in contrast to Fagan-style), (2) tool-based, and that (3) occurs regularly"*.

Modern code review is an iterative process (Figure 1) that can involve several cycles of review, update, and re-review. To take you through an example of a code review process [2, 19], I will describe each step and refer to the corresponding figure nodes in parentheses. Modern code review is often performed after the issue or task (node 1) has been implemented (node 2) and tested (node 3) by the author, but before integration, acceptance testing, and delivery. The first step of the code review process is to upload the code changes to the team's code review tool (node 4). Next, the author can preview the code change inside the code review tool to check if they discover any issues that they did not spot inside their development environment (node 5). If the preview looks good, the author will invite one or more reviewers (node 6). The reviewers will usually get some kind of notification and, when they have time, they will perform the code review by writing review comments and voting for accepting or rejecting the code changes (node 7). When the code reviews are done, the author needs to address any unresolved code review comments, either by updating the code (node 2 and 3) or writing a response explaining their reasoning (node 8). If the code is updated, the review cycle often starts over with a re-review of the new state of the code changes. Finally, when all comments are resolved, the code changes are either accepted for integration and delivery (node 9) or, in rare cases, rejected and discarded (node 10).

3 Related Work

In the majority of previous research, code review has been characterized from a process perspective describing the steps and tasks involved [2]; see Section 2. Gonçalves et al. [20] are among the few studies that propose a cognitive model for code review. They present the *Code Review Comprehension Model* (Figure 2), focusing on the comprehension part of code review. Their model is based on Letovsky's cognitive model of code comprehension [21] expanded with elements

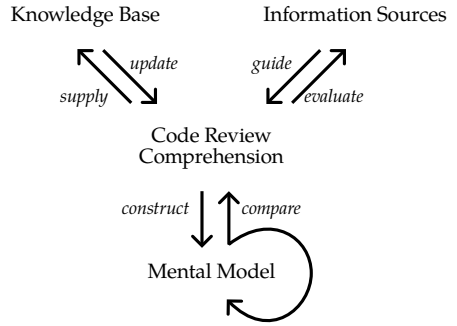

Figure 2: Code Review Comprehension Model by Gonçalves et al. [20].

that allow for iterative comparison and evaluation of changes compared to the original code, the reviewers' expectations, their experience, external information sources, team knowledge base, and their mental model. This thesis aims to model the entire code review task from start to finish, including cognitive processes that are beyond the scope of comprehension.

Several recent studies explore how to use AI and machine learning (ML) to partially or completely automate code review. Lu et al. [11] introduce LLaMA-Reviewer, where they fine-tune the publicly available LLaMA[1] large language model (LLM) to handle common code review tasks. Yu et al. [12] presents Carllm, a fine-tuned LLM that improves issue detection and comprehensibility of code review comments in automatic code review. Tang et al. [13] propose CodeAgent, an approach in which multiple AI-agents are fine-tuned for different code review tasks and then collaborate to automate the entire review. An industry example is the Google DIDACT project [14], where ML models are trained on the sequential steps in software development processes, such as code review, in order to replicate and automate them. In this thesis, I will present an analysis of the potential impact of automated code review on the benefits of code review and propose an architecture for AI-supported code review instead.

4 Methodology

Since this thesis focuses on the human aspect of code reviews, we apply an interdisciplinary perspective that recognizes that code review tools, processes, and activities exist in interaction with a team of humans and in a social environment that cannot be understood from a single lens. The dynamics of code review are complex and include elements from areas of software engineering, sociology, human-computer interaction, cognitive psychology, program analysis, and group dynamics. Therefore, I have chosen interdisciplinary methods and applied design sci-

[1] https://github.com/meta-llama/llama-models

ence [22, 23] and ethnography [24, 25] to the research questions presented in the included papers. My research is rooted in a constructivist epistemology postulating that all knowledge presented in this thesis is co-constructed together with study participants, coauthors, related research, and with you, the reader, who interprets and applies the results. An overview of the research questions, data sources, methods, and analysis approaches can be found in Table 2.

Table 2: Overview of the included papers, their research questions, data, methods, and analysis.

Title	Research Question	Data	Methods	Analysis
Paper I: Design of Flexible Code Block Comparisons to Improve Code Review of Refactored Code	**RQ$_1$** What developer experiences during code review can cause frustration? **RQ$_2$** How can code review tools be modified to improve the developer experience? **RQ$_3$** How can developers be involved in the design process to better discover, understand and design tooling improvements?	Interviews, participatory design sessions, sketches, and prototype	Participatory design	Participant validation
Paper II: Support, Not Automation: Towards AI-supported Code Review for Code Quality and Beyond	Vision for using AI to support code review and its users striving to boost all the positive effects of code review, including interpersonal effects such as knowledge transfer, team awareness, and shared code ownership.	Literature	System design	Analysis of impact on code review benefits
Paper III: Code Review as Decision-Making —Building a Cognitive Model from the Questions Asked During Code Review	**RQ$_1$** How can the cognitive process of code review be modeled from a theoretical perspective?	Transcribed think-aloud sessions and interviews	Ethnography, Interviews	Thematic analysis

4.1 Design Science

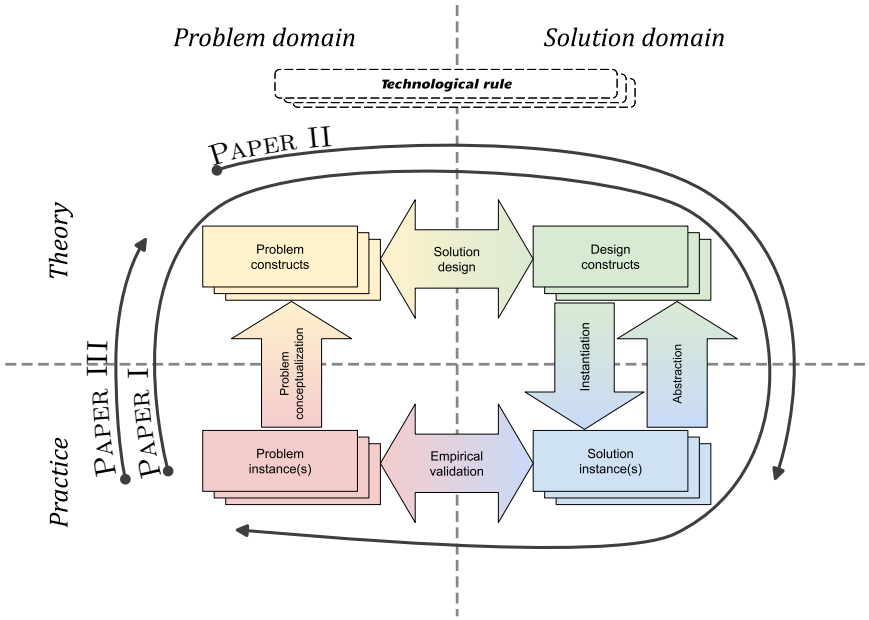

Figure 3: The included papers mapped onto problem domain, solution domain, theory, and practice.

In contrast to the Natural Science paradigm, which studies and describes the natural world, the Design Science paradigm describes the study of the artificial, the human-created [26]. Van Aken [27] defines the ultimate mission of design science as *"develop design knowledge, i.e. knowledge that can be used in designing solutions to problems in the field in question"*. Design knowledge can be created on three different abstraction levels: 1. *object design* specifies the intervention or artifact itself, 2. *realization design* creates a plan for the implementation of the intervention or artifact, and 3. *process design* outputs a process for the steps to take to create a design that solves a set of problems.. The output of any of these abstraction levels can be formulated as *technological rules*, defined in the context of design science by van Aken [27] as *"a chunk of general knowledge, linking an intervention or artifact with a desired outcome or performance in a certain field of application"*.

The design science researcher can go through several cycles of exploring a problem, building theories about the causes, conceptualizing one or more solutions, implementing solutions, and validating solutions towards the original problem. A study can start and end anywhere on this cycle. Throughout the cycle both the design process itself and the design knowledge are continually updated [28].

Engström et al. [29] formalizes the design science cycle by introducing a four-field model that separates the abstraction level into *practice* and *theory*, and the research domain into *problem domain* and *solution domain*. The model is illustrated in Figure 3, together with a mapping of the included papers to the quadrants. In the bottom-left *practice/problem-domain* quadrant, you find concrete instances of the problems. By problem conceptualization, the researcher can build theoretical problem constructs and move up to the *theory/problem-domain* quadrant. From here, the model describes an iterative solution design process that goes to the *theory/solution-domain* quadrant and creates design constructs. Through instantiation, the researcher can move down to the *practice/solution-domain* quadrant and realize one or more solution instances. Solution instances can also be abstracted back into design constructs, to analyze or improve an existing solution. From the *practice/solution-domain* quadrant, iterative empirical validation evaluates the impact of the implemented design on the problem instances. In this model, the technological rules are illustrated as spanning the problem and solution domains on a theoretical level, even if their eventual application might be practical.

Through a meta-study on the alignment between software engineering research and the design science paradigm, Engström et al. [29] conclude that the lens of design science helps to emphasize the theoretical contributions, practical relevance, novelty, and rigor of software engineering research. Runeson et al. [30] builds on these results and adds that positioning a software engineering study in relation to the design science framework helps to communicate and evaluate research contributions and limitations. The included papers in this thesis have the following positioning relative to the design science framework.

Paper I traverses a full design science framework cycle. The study starts with exploring problem instances through interviews and workshops with industry practitioners and continues with conceptualizing possible causes of a selection of the problems. From there, we create a conceptual design, build a prototype of the design, and finally validate the prototype with the practitioners.

Paper II begins from problem constructs such as misalignments, challenges in navigating the code review, and inefficiencies in the process. It continues to design a conceptual solution proposal, evaluates this based on the problem constructs, and proposes an architecture for a solution. Implementation and validation of the solution proposal is left for future work.

Paper III presents an ethnographic study that goes deep into the concrete experiences of code review. From there, a theoretical model of the cognitive process during code review is built, which can in future work be used as the basis for novel design constructs. Having a cognitive model as reference could also guide future implementations of solutions and empirical validations.

4.2 Participatory Design

Participatory design is based on the idea that involving stakeholders in the design process of a solution will lead to both a better design and also a higher degree of adoption of the new solution. As a designer, engineer, or researcher, you likely have a lot of design knowledge and skill, but the premise of participatory design is that is important from a democratic perspective and for the quality and adoption of the design that communities affected by the design are invited and deeply involved in the work [31]. Costanza-Chock [32] argues that there is an inherent power imbalance in design where the designer's world view and values are imposed on the groups affected by the design, such as users, customers, and the general public. They present the motivation and methods for *Design Justice* where design work is driven by and for affected communities and is done in a way that enhances equity and accessibility.

The Double Diamond design process [23] is one available framework for implementing participatory design. It consists of four process steps all conducted in the context of engagement and leadership, which explicitly emphasizes the participation of the community.

Discover Explore the challenge; its characteristics, the people affected, limitations, use-cases, etc. Typical actions include surveys, workshops, interviews, and observations.

Define Narrow the challenges down to a clear definition of the problem. Typical actions include requirements analysis, conceptual design, personas, and use case analysis.

Develop Explore the solution space; evaluate as many different solutions to the problem as possible. Typical actions include low-fidelity prototyping, sketching, and design workshops.

Deliver Narrow down to a final design; choose the most promising candidate solution and refine it further. Typical actions include high-fidelity prototyping, implementation, user validation, interviews, and surveys.

The process is iterative, and individual steps, sequences of steps, or the whole process is repeated until a desired outcome is reached. The Double Diamond design process is described in more detail in Paper I, as the main method used in that research study.

4.3 Ethnography

The study presented in Paper III uses ethnographic methodology to study questions and cognitive processes during code review. Ethnography originated in social science and anthropology and is a scientific method created to study and describe a culture or a group of people from the members' own point of view [33, 34]. It can

be defined as a form of participant observation in which the researcher embeds themselves into the group they wish to study. You study people in their everyday context by participating in social interactions with them with the goal of understanding them. Sharp et al. [24] explore the relevance of ethnography to software engineering research and present a practical framework on how it can be applied. Especially, they argue that it is an essential methodology to uncover *why* software engineering practices are done a certain way and recommend that ethnographic studies can and should be used to inform tool design and method development. Ethnography is work-intensive, requiring extensive note-taking, transcription, and qualitative analysis, which in practice limits the number of participants. On the other hand, it presents the possibility of studying the meanings and perspectives of the participants in depth [25] and collect very rich descriptive data [35].

5 Contributions

5.1 Prototype of Flexible Code Block Comparison Tool

The majority of the code review tools in use today (GitLab, Gerrit, GitHub, etc.) only support 2-way diff comparisons of code changes within the same file. If the author moves code between files or breaks out code blocks into a new file, a common result of refactoring, it becomes difficult and time consuming to verify whether the code was just moved or also changed in a significant way. During the initial interviews for the study described in Paper I the participants explored several common problems experienced during code review, and cross-file comparisons were highlighted as frequent and time-consuming for both the authors and reviewers. To verify the moved code, they often resorted to opening the code review tool in two separate browser windows, placing them side by side on the monitor, and manually comparing the code blocks line by line.

Using a Double Diamond design process (Section 4.2) together with industry practitioners, we explored common code review problems, narrowed the scope to comparison of refactored code, facilitated a participatory design workshop to find different solutions, created a conceptual design, implemented a prototype, and finally validated the prototype. The resulting design, contribution $\mathbf{C_1}$, combines a Lightboard toolbar where reviewers can add code blocks and a flexible comparison modal that can compare any block in the current file with any block on the Lightboard. Automatic similarity detection assists reviewers in finding blocks that are interesting to compare. During validation with industry practitioners, all participants were able to use the design without any instructions and found the functionality useful. In addition to validating the prototype itself, these results showcase the potential of using participatory design to create well-received code review tool improvements. The design process, the resulting design and the screenshots are

shown in detail in Paper I and the prototype is publicly available for testing in Artifact I[2].

5.2 Architecture for AI-Supported Code Review

Benefit [3]	Preserved by AI-automation	Preserved by AI-support
Defect finding	✓ Yes	✓ Yes
Code improvement	✓ Yes	✓ Yes
Alternative solutions	✓ Yes	✓ Yes
Knowledge transfer	✗ No	✓ Yes
Team awareness	✗ No	✓ Yes
Improve developer process	✗ No	✓ Yes
Share code ownership	✗ No	✓ Yes
Avoid build breaks	✓ Yes	✓ Yes
Track rationale	✗ No	✓ Yes
Team assessment	✗ No	✓ Yes

Table 3: Code review benefit preservation if automating code review compared to AI-supported code review (Paper II). **Takeaway:** Many of the benefits of code review risk being lost if the activity is automated.

As discussed in Section 1 and Section 3, there are recent research initiatives toward completely automating code review. In my view, automation risks losing several of the interpersonal benefits teams and organizations get from the practice. Instead, I propose working towards AI support to augment the capabilities of the human reviewers. As shown in Table 3, AI-supported code review has the potential to preserve or enhance more of the benefits of code review compared to AI-automation. In Paper II, we propose an AI agent-based architecture, contribution $\mathbf{C_2}$, to support code review and analyze its impact on the benefits of code review, as described by Bacchelli & Bird [3]. The architecture is based on a central Large Language Model (LLM) handling user interaction combined with a number of specialized AI-agents taking care of integrations with external documentation, KANBAN board, issue tracker, team chat, etc. An additional agent is responsible for storing and updating user preferences to make the system adaptable and customized to each user. In this way, context and information can be collected and shown to the user at the moment in the code review process where it is needed. The system wraps the user's favorite code review tool, such as Gerrit, GitHub, GitLab, etc., so that they can work in a familiar environment but with the benefits of customized AI support. In Figure 4, we show a user interface proposal in which

[2] `https://figma.com/proto/KZIrsBH8DZ22ZI0B0YD2BC/GBC?hide-ui=1`

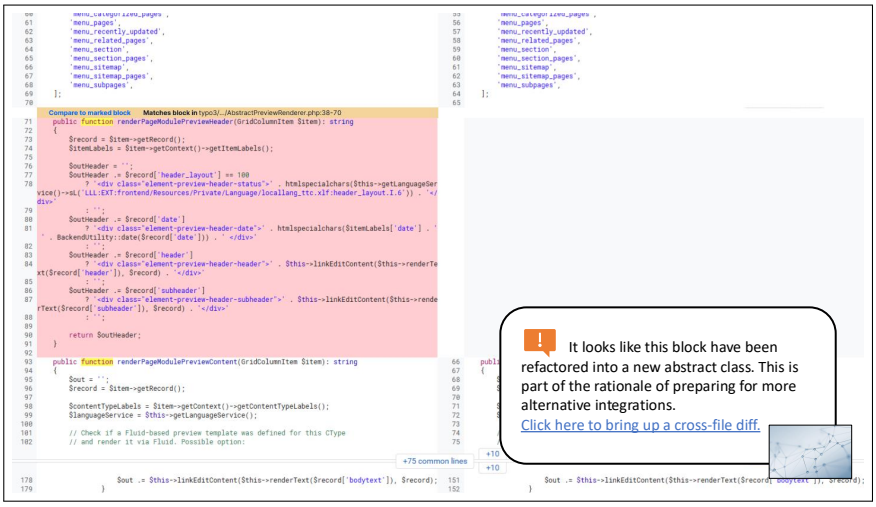

Figure 4: User interface proposal for AI-supported code review. **Takeaway:** The AI agent-based support provides contextualized insights into the review and suggest actions the reviewer can take.

the AI agent discovers that a code block is refactored into a new abstract class and suggest that this is related to parts of the code review rationale. If the user whishes to investigate further, it offers to open up the cross-file diff tool described in Section 5.1. The architecture, its rationale, and an analysis of the effects on the benefits of code review are found in Paper II.

5.3 Cognitive Model of Code Review as Decision-Making

In Paper III, we present an ethnographic study that explores the cognitive process of the entire code review task by studying the implicit and explicit questions that reviewers ask during code review sessions. Using thematic, statistical, and sequential analysis, we build the Code Review as Decision-Making (CRDM) cognitive model, contribution $\mathbf{C_3}$, which highlights the similarity between code review and recognition-primed decision-making [36]. As a reviewer, you constantly make decisions around which file to look at next, whether to write a review comment, how to phrase it, how to vote regarding the integration of the code change, if you need to run the code locally, what documentation to cross reference, and more. These decisions are based on the reviewers' expectations and experience, both with code review in general but also with the specific code base and team context they are working in and also on their understanding of the rationale and implementation of the code change.

The CRDM model has many potential applications, for example, to adapt code review tools and AI support in a way that matches the cognitive needs of the reviewers throughout the code review. The model provides an explanation to results of previous code review studies, such as the findings of Bosu et al. [37] where it was shown that even experienced programmers can take up to a year to become efficient at code review in a new team.

6 Conclusions

The three contributions $\mathbf{C_1}$, $\mathbf{C_2}$, and $\mathbf{C_3}$ together lead to some conclusions regarding the current state and future of code review.

- Participatory design approaches can, with relatively low effort, improve the user experience and efficiency of today's code review tools. These approaches amplifies the wealth of information and experience from users and aim to create designs that maximize interpersonal benefits and have broad user adoption. Validation of the prototype in Paper I shows that this approach has the potential to create useful and well-received code review features.

- An AI agent-based architecture is promising for AI-supported code review, as it can manage a diverse range of integrations with auxiliary systems and provide information customized to the context and cognitive process of the user. This keeps the human in the loop, again increasing or preserving the interpersonal benefits of code review.

- Code review can be modeled as a decision-making process, leading to, for example, better insight into user needs and guidance for tool improvements. Having a cognitive model when creating and evaluating tool and process improvements increases understanding of the impact from a human perspective and helps predict what support is needed and when.

From three different directions, these conclusions put the user experience at the center, strive to increase interpersonal benefits, engage users, and deepen understanding while improving both tools and processes. Taken together, they formulate a technological rule for one way towards the next generation of code review tools, practices, and processes.

Technological rule: To design improvements to the user experience and efficiency of modern code review, use insights from the CRDM model and AI agent-based architecture together with participatory design methodologies and focus on the needs of human reviewers.

Empirical validation of this technical rule is left for future work. However, enthusiastic participation of practitioners in the design work described in Paper I, along with positive reception when validating the prototype, gives a strong indication that participatory design is a viable methodology for code review tool improvements. The CRDM model and AI agent-based architectures open up interesting paths to find, simulate, and implement new tool improvement ideas.

7 Future Work

7.1 Empirical Validation of Technological Rule

In the future, I would like to validate the technological rule described above (Section 6) by implementing and validating an agent-based system for AI-supported code review. Using the architecture of $\mathbf{C_1}$ and the CRDM model of $\mathbf{C_2}$, the system would be designed to provide the right information and context at the right time during code review. Participatory design approaches should be used to ensure a good user experience where the support given is experienced as positive, helpful, and anchored in the software engineering community. Ideally, detailed feedback from a smaller group of participants is combined with a survey with a larger number of participants from diverse teams and organizations.

7.2 Strategies for Actions in Code Review

In the data collected from field work with Paper III there are also many examples of strategies used to answer implicit and explicit questions during code review. Some examples include how and when users decide to check out the code locally, message a colleague on the team chat, or search for external API documentation. A re-coding and thematic re-analysis of the existing ethnographic material from the angle of actions and strategies could highlight actions connected to information needs during code review. The results could show use-cases that the code review tools already support well and others where the users need to switch to other tools in order to move forward with the code review.

7.3 Motivations Behind Code Review

Intuitively, the benefits of code review should be well aligned with the motivations for the developers to submit their code for review and conduct code reviews. In practice, the motivations the developers state are sometimes parallel or even contradictory to the benefits of code review. For example, Alami et al. [38] find that the most pronounced extrinsic motivator in open-source projects is that quality code reviews improve the reputation of the reviewer. But good reputation can lead to the developer's own code changes being reviewed less thoroughly or even integrated without review, thus negating many of the purposes of code review.

In our interview data from Paper III we see a range of responses regarding motivation; from seeing code review as a mandatory chore, to a way to catch bugs before production, to an opportunity for learning and teaching. In the future, conducting a survey or interview study combined with the mining of data from Paper III could give more insight into developers' motivations for code review in industry projects, compared to open source projects. These results could be used to align processes with developer motivations and reduce lead time and friction in code reviews.

7.4 Measuring Effectiveness of Code Review

When researching improvement to code review tools, it can be elusive to determine if a particular feature or intervention achieves desired outcomes. It is easy to say that the goal is to improve code review efficiency, but that is a concept that is very hard to define and measure. Mäntylä and Lassenius [39] explore methods to define and measure defect finding during code review, but other benefits, such as knowledge sharing, team awareness, and shared ownership, lack consistent definitions and metrics. Defining metrics or indicators for efficient and effective code review would be valuable in evaluating improvements in tools and processes. Approaches could include measuring the code review task in isolation, as well as its effect on the whole software engineering process [8]. The results could allow researchers and developers to be transparent and explicit about the benefits and trade-offs involved and address the need for improvements without unintended loss of code review benefits.

INCLUDED PAPERS

PAPER I

DESIGN OF FLEXIBLE CODE BLOCK COMPARISONS TO IMPROVE CODE REVIEW OF REFACTORED CODE

Abstract

Code review occupies a significant amount of developers' work time and is an established practice in modern software development. Despite misaligments between users' goals and the code review tools and processes pointed out by recent research, the code review tooling has largely stayed the same since the early 90s. Improving these tools, even slightly, has the potential for a large impact spread out over time and the large developer community.

In this paper, we use the Double Diamond design process to work together with a team of industry practitioners to find, refine, prototype, and evaluate ways to make it easier to compare refactored code blocks and find previously hard-to-see changes in them. The results show that a flexible comparison modal integrated into Gerrit could reduce the mental load of code review on refactored code. Potentially, it could also have effects on how code is written by no longer discouraging refactoring due to it complicating the review. The user interface created in this col-

Lo Gullstrand Heander, Emma Söderberg, Christofer Rydenfält "Design of Flexible Code Block Comparisons to Improve Code Review of Refactored Code" In *Companion Proceedings of the 8th International Conference on the Art, Science, and Engineering of Programming*, 2024.
DOI: 10.1145/3660829.3660842

laborative manner was also intuitive enough for all of the participants to be able to use it without any hints or instructions.

1 Introduction

Software developers today spend between 10-20% [2, 40] of their working time doing code reviews. With the total number of software developers expected to reach 28 million people by 2024 [41], this could mean between 22-44 million hours spent doing code reviews every day! When usage is on this scale, even small improvements in code review tools and processes can have a significant effect.

Yet, the tools used have not changed in their approach since they were first introduced with ICICLE in 1990 [7]. Research has shown that there are misalignments [6] between the tools used and the desired goals, such as code quality and knowledge sharing [3].

There are not very many studies published that explore how changes or new features in the code review tools can affect the review experience or quality. Bagirov et. al. [42] investigates if the ordering of the files in the review could be improved. Baum 2019 [43] studies how code review tools could be improved using cognitive support techniques to reduce the cognitive load of the task and improve code review quality. Baum et al. [44] study the (mis)alignment between the code review task and requirements and the tools in use today. They believe that there is room for improvement and that a new generation of more specialized tools could lead to "increased review efficiency and effectiveness".

In this paper, we explore ways to improve the code review developer experience by applying a double diamond design process (Section 2) to the code review tooling. Our research questions are:

- **RQ$_1$** What developer experiences during code review can cause frustration?
- **RQ$_2$** How can code review tools be modified to improve the developer experience?
- **RQ$_3$** How can developers be involved in the design process to better discover, understand and design tooling improvements?

To answer these questions, we study the code review experience of software developers, with an established code review process in Gerrit, working at a company developing embedded systems. Through a focus group session with the developers, we identify several problems that could be addressed by improved tooling. We select one of these problems and organize a co-design workshop with the participants, focusing on coming up with a range of possible solutions. One solution, a flexible code comparison modal to compare moved and refactored code blocks, is chosen. After developing this solution into a high-fidelity prototype, we bring it back to the software developers for evaluation and feedback.

The feedback from the participants (Section 3), during both the workshops and the evaluation, suggests that this feature could simplify code reviews of moved and refactored code. It could potentially also have effects on how code is written, by no longer discouraging change and refactoring in moved code to avoid complicating the review.

2 Method

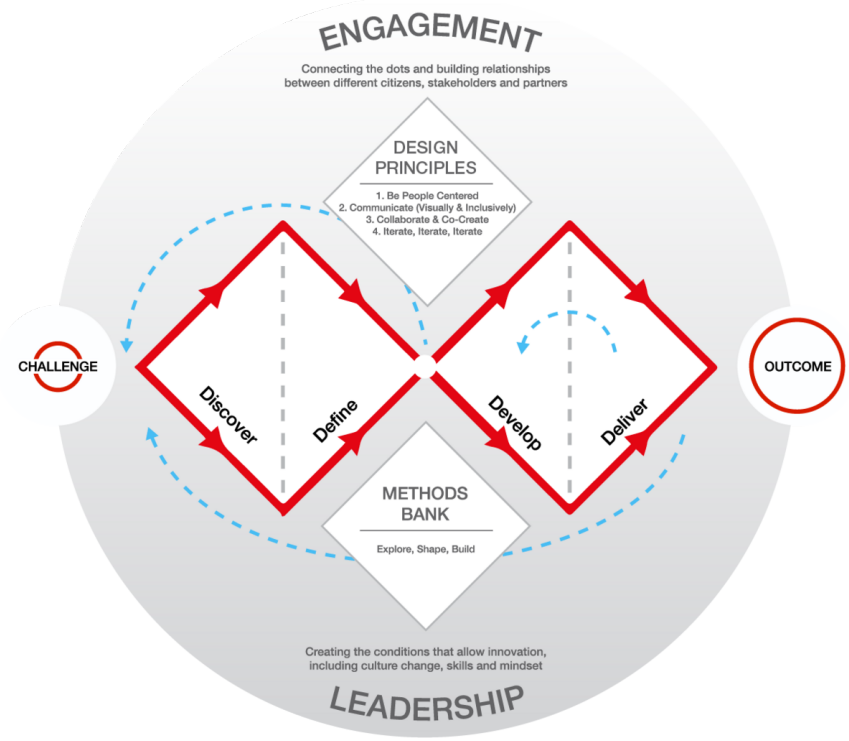

Figure 1: The Double Diamond design process.
From `https://www.designcouncil.org.uk/`

To work towards answering $\mathbf{RQ_3}$, we wanted to choose a method that involved the developers using code review tools in the design process. The purpose of this is both to increase the chance of designing something that would be genuinely useful to the developers and also to invite their experience and expertise into the project.

For these reasons, we have used the Double Diamond design process [23] (Figure 1) to structure the work. The double diamond consists of two consecutive steps both consisting of one exploratory, i.e., divergent, and one converging phase.

These phases live inside a context of Engagement and Leadership which makes the importance of involving the community explicit. These two steps form two "diamonds". The steps in the project and how they relate to the phases in the process can be seen in Table 1.

In the first diamond, the process starts with the challenge. The design project's purpose and starting point. From here, through the "Discover" phase, the purpose is to deepen the understanding of the challenge. This is a divergent phase, expanding outwards to discover more and more about the challenge, its characteristics, the people affected, the limitations, and use cases to work with. In this process, it is important to refrain from self-censoring or jumping to conclusions based on your prejudices about the problem.

The second half of the first diamond, the "Define" phase, uses the understanding of the challenge obtained in the Discover phase, narrowing it down to a clear problem definition. The definition needs to be a real and urgent problem for the intended users and manageable to tackle within the scope of the project.

The first phase of the second diamond, the expansive "Develop" phase, is an exploration of the solution space for the previously defined problem. In practice, this means developing as many different solutions to the problem as possible. Rough prototyping can take place in this phase to describe the ideas more visually.

In the second half of the second diamond, the "Deliver" phase, the purpose is to reduce the scope down to a single final design. Here the team can iterate through a variety of methods. Creating prototypes, evaluating prototypes, and finally refining the prototype to arrive at a final design choice.

Table 1: Design process steps and participants.

Discover phase		
Sep 2023	Literature review	1st author
Sep 2023	Developer meeting	5 developers
Oct 2023	Focus group	5 developers
Define phase		
Nov 2023	Definition workshop	3 faculty members
Nov 2023	Problem statement	1st author
Develop phase		
Nov 2023	Co-design workshop	5 developers
Dec 2023	Conceptual design	3 faculty members
Deliver phase		
Dec 2023	Prototype development	1st author
Dec 2023	Prototype persona verification	3 faculty members
Jan 2024	Prototype evaluations	7 developers
Jan 2024	Prototype refinement	1st author

2.1 Discover phase

In the Discover phase, we performed a literature review, a developer meeting, and a focus group (Table 1). The literature review was intended to orient the study about problems in code review as found by current research [6, 45–47] and give us an idea of what and where to explore for $\mathbf{RQ_1}$. After this, we held a first developer meeting to introduce the project to five professional developers participating in the study. During the meeting we got to know their backgrounds, the teams' code review processes, and collected informed consent for participating in the study.

The participating developers work at a medium-sized embedded software development company. Their prior experience ranges from over 20 years for one system architect, to 2-3 years for some of the software developers. They work in 4 different teams, which all have established code review practices. All new code in the teams undergoes code review by usually two other developers, who both need to approve the change before it is merged.

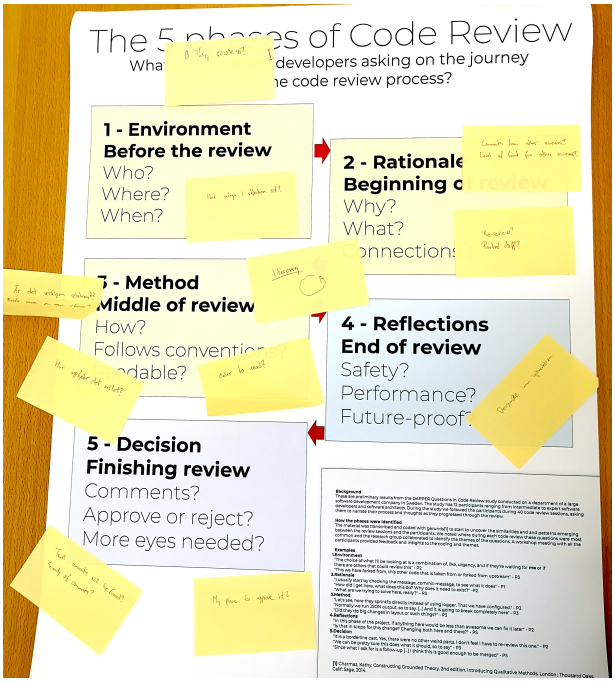

Figure 2: Post-it notes from the focus group.

In the next step, we arranged a focus group [48] with the same five developers. The output from the literature review and the developer meeting were used as input to create a design brief and an interview protocol for the focus group (Appendix I.A). The topics for the interview questions were aiming to explore $\mathbf{RQ_1}$

and **RQ$_2$**. We first explored the practices the developers used when reviewing code (i.e., "Do you read the files in a code review one time or multiple times?"), and then experiences reading code in an IDE (i.e., "How is the experience of reading and understanding new code in other tools, environments or situations?"), and finally comparing the two (i.e., "How is the experience different compared to code review in Gerrit?")

The problems and challenges discussed in the focus group were gathered as post-it notes on a big sheet of paper (Figure 2) and sorted into categories depending on if the challenge usually occurred before, in the beginning, middle, or end of looking through a new code review. The session was held in Swedish and recorded for later transcription and translation.

To balance the participation in the meeting, the facilitation was done in a way where every participant got the same time to talk about their perspective and then with a discussion where everyone could contribute before moving on to the next person's perspective. See results in Section 3.1.

2.2 Define phase

The Define phase consisted of a workshop in the research group where the sorted and categorized challenges from the focus group were described based on potential impact and feasibility to prototype within the scope and time limits of the project. Based on the description, the group prioritized the challenges to conclude which problem areas to move on with. After this, the first author put together a problem statement as input to the next phase. The results from this process are described in Section 3.2.

2.3 Develop phase

In the Develop phase, we conducted a co-design workshop [49] where five industry practitioners (four who also participated in the focus group, plus one more developer from the same company but a different development team) co-created different solution designs. The reason for choosing a co-design workshop as the method was to deeply involve the developers in designing solutions to improve their own working tools, in accordance with **RQ$_3$**.

The first author facilitated the co-design workshop and prepared hand-drawn low-fidelity mockups of the Gerrit user interface (Appendix I.B) and different kinds of widgets, buttons, and overlays on overhead film. New interfaces and ideas could be created by cutting and moving parts of the interface around. The output of the meeting was documented with photos of the mockups and notes on the ideas and principles behind them.

The ideas from the co-design workshop were elaborated through conceptual design [50] to make them more rich and substantial. Conceptual design is the definition of the metaphors, use cases, concepts, and actions that can be involved

in the design. This also included the creation of two personas [51] based on the concerns and challenges emphasized by the participants in the focus group and co-design workshop.

2.4 Deliver phase

The final Deliver phase distilled the solution ideas and conceptual designs from the Develop phase and reduced this to one high-fidelity prototype that can be evaluated towards the problem statement developed in the Define phase. The phase contained 4 steps (Table 1). First, a high-fidelity prototype was created using Figma[1], a tool for creating interactive prototypes of computer interfaces.

The prototype was verified by using the personas and their questions. After some updates, the practitioners from the focus group, co-design workshop, and one additional external developer from a different company were invited to individually test and evaluate the prototype. During the test, the participants did a code review in a private Zoom meeting without detailed instructions or guidance (see Appendix I.C. All of them reviewed the same code that contained examples of blocks moved both within a file and between different files. Their running commentary and their shared screen was recorded to evaluate the interface's usability and how well it supported the challenge selected in the Define phase. The meetings were held in Swedish and the recordings were transcribed, referenced, and translated into English where needed.

Finally, the design of the prototype was adjusted based on the results and suggestions from the evaluations.

3 Results

The result section follows the structure of the Double Diamond process and the sequence of method steps described in Table 1.

3.1 Discover phase

In the developer meeting and literature review (Section 2.1) several developers and articles mentioned file order as an important factor in code reviews. For this reason, the design brief and questions for the focus group, started out exploring this problem area.

[1] `https://www.figma.com/`

Design brief When doing a code review, the developer often has to read the files multiple times because they are presented in an order where early files are not understood until after reading files further down in the review.

Design questions:

1. In what ways could reading the code in a review be improved so that fewer passes or even a single pass through the files in the review would be enough to understand the changes?
2. Are there ways to improve the ordering or let the author convey more of the narrative when sending the code for review?

However, during the meeting our participants raised six different problem areas that they felt were more frequent and impacting their experience more. The areas are described below:

Diff problems

There are many cases when the diff algorithms break down and require tedious manual comparison word-for-word. For example, if a function is moved within a file from the bottom to the top, or maybe refactored into two separate functions, it will all show up as just deleted lines and completely different added lines. This makes it hard to see if the code was only moved or moved *and* modified. Also, if a file is renamed and then changed, or a function is moved into a different file, it is impossible to use the built-in diff tools to compare the code.

Suggestions from the focus group included manually selecting files, lines or blocks, to compare sections that the algorithms themselves don't match for diffing.

Participant 3 "I was working with this today and had to sit with two separate windows and go through it, just like, Control-F in this file and see if I find it. Is it added? Is it completely new? Or is it just moved from further down in the file?"
Participant 1 "Do we want a more semantic diff? Where you can kind of say that this has been extracted from over there or it has been moved between here and there?"

Finding similar but unchanged code

It might be the case that a code base contains several similar snippets of code and that a code change should affect all of them. In the code review tools, this is difficult to verify. There might be a forgotten snippet left in an unchanged file that

is never even shown in the review. It would be helpful to have a tool to find similar code that maybe also should have been changed or looked at.

> **Participant 1** "It is one of the things that are easiest to miss during a review, regardless of reviewing a document or code. It is like, you only look, think, and look at what *is in the diff*. Not what *should have* been there."

Lack of navigation

Lack of navigation in code review tools causes problems, such as making it difficult to go from a variable's or function's use to its definition, or finding all uses of a variable or function. In IDEs such as VS Code, this kind of navigation is easy and commonly just one or two clicks away, but the same convenience is missing from code review tools.

> **Participant 4** "I mean, say that you could just press it and «yeah, you have 5 references here» and then you see that, yeah, but the reference down there is not changed in this commit. Why? Then it would be very fast to get to that insight."

Ordering of files

When changes are big and spread out over many files, the alphabetic ordering of files in the code review tool is essentially random, in regards to how the code should be read and understood in the best way. Suggestions for how to address this problem from our participants include placing generated code last or placing the tests last. The uploader could also draw a path through the change with commentary for each file, to clarify the story told by the code under review.

> **Participant 4** "You could make it easy for yourself and just, like, let the person uploading the review decide or give a suggestion for an order. Then people can choose to go back to their own order, but you can say, kind of, that *I* suggest you look at it in this way. Then you can do it in call stack order if you like that.

Overlays and annotations

The continuous integration (CI) pipeline used by our participants already includes support from running a wide variety of testing and code analysis tools, but these results are disconnected from the code. In the best case, they are shown as a

pass/fail stoplight in the code review tool with a link to the full logs. It would be useful to show these results as inline overlays on top of the code. To see, for example, test coverage, linter warnings, execution traces, loops with frequent execution, failing tests, etc. The overlays need to be easy to select and toggle on and off so that the user interface stays easy to use. For adoption, it needs to be easy to integrate the results from the CI without modifying the linters, tests, etc.

> **Participant 5** "It would have been nice to have a code coverage overlay because then you would have been able to see that (if more tests were needed) in a completely different way."

Unchanged files

Finally, it was discussed that unchanged files are not shown at all in code reviews today. It would help to have a way both to find and navigate to unchanged files and also write review comments in them. There may be places in unchanged files that have not been changed, but that should have been changed, or that affect some parts of the changed code.

> **Participant 5** "But if you don't want to be marked as the uploader, you have to do it (commenting on an unchanged file) through URL-hacking. (...) NN does it fairly often and I do it sometimes when I realize that there is a change in a nearby file that should have been there."

3.2 Define phase

When considering the problem areas discussed in the focus group, we made a first selection of problems that can be addressed by collaborative design ($\mathbf{RQ_3}$). This selection removed *Lack of Navigation*, *Ordering of files* and *Finding similar but unchanged code*, since these problem areas would have put more of the focus on deeper code analysis instead.

The three remaining problem areas, *Diff problems*, *Overlays and annotations* and *Unchanged files* are all very interesting areas to explore under the scope of interaction design. In *diff problems*, you want an interface that is flexible in choosing the blocks to compare. It should also be intuitive to use to quickly make comparisons, and at the same time not get in the way of the classical code review interface.

For *overlays and annotations*, completely new concepts of layers would need to be designed and introduced in a way that fits well into the existing code review interface. It needs space for rich information and at the same time needs to be easy to navigate and turn on and off.

To create a design solution for *unchanged files*, the existing interface for presenting and commenting in the files could largely be reused. The challenge is

rather in the navigation and to make it clear that the unchanged file is outside the changed code.

In the end, *overlays and annotations* was estimated to be too large for the scope of this project and that *diff problems* was more important and had a greater impact on the quality and ease of reviewing than *unchanged files*. Because of this *diff problems* was chosen as the problem area to explore during the next phase.

3.3 Develop phase - co-design workshop

During the co-design workshop (Section 2.3), our industry practitioners were given hand-drawn cut-and-paste prototyping kits. With these, solution suggestions for the problem of comparing code, that the diff algorithms do not detect as moved, could be created and discussed. Two guiding principles, which should always be present, and three separate solution ideas were formulated:

Principle 1: Show context

One important principle is to always show context for both blocks. The context lines should be syntax-highlighted and displayed in a muted way but still show where the two blocks were found originally.

Principle 2: Review comments while comparing blocks

Since it will often be during these more detailed comparisons that ideas or comments about the code will be found, it is an important principle to be able to write comments right there and then. Maybe it should also be possible to view and read previous comments from other reviewers or the author, even if these were not written in the new comparison views.

Principle 3: Support comparison within and across files

The *diff problems* discovered and defined in the previous phases can occur both within the same file and across different files depending on the types of changes and refactorings done. Support for both these cases is needed to get the most benefit from the tool.

Principle 4: Integrated in code review environment

There are existing software that can do comparisons of any code blocks or texts that you choose such as Meld[2], KDiff3, git diff, etc. The issue with them is that it requires the reviewer to either check out the code locally or copy-paste the code blocks they want to compare into another window. This could switch them out of

[2]Meld is a desktop visual diff tool available for many operating systems `https://meldmerge.org/`

the code review task [6] and would require extra steps and time. To make reviewing faster and easier the tool needs to be integrated into the code review environment.

Idea 1: Scroll-lock one side

One solution idea that came up was to be able to scroll-lock one side of the diff view and then scroll only the other side, to be able to align code blocks that you want to compare so that they are next to each other. In this way, you would not have to select any lines for a block and would save the extra work and navigation of opening a new Gerrit tab and trying to place that next to the code you want to compare.

The idea is to keep the coloring and the diff the same, and just change which lines are shown next to each other. Probably some kind of snapping at line alignments would be nice. A further improvement could be a feature to mark a segment and then tell the view to scroll through matching segments on the other half.

Idea 2: Switch comparison base

Another idea is for the case of comparing blocks, or whole changes, across different changesets, for example after a revert and re-submit with changes. Here, the proposed solution is to let the user switch the base commit-id to compare against.

Code that has changed independently, by other commits, should be hidden. Only changes in the diff are highlighted so it becomes easy to compare what the difference is between the old faulty code and the new suggested changeset.

Being able to input a git commit-id manually could be a first step, with a possible extension of automatically finding suitable comparison bases that have very similar diff towards the main branch.

Idea 3: Workspace for blocks

To have a workspace where interesting blocks can be placed as they are seen in the code, such as a sidebar or a drawer, is another interesting workshop idea. These blocks could then be brought up and listed in a modal dialog to get an overview of all the interesting blocks. From here the user could compare them to each other, or search for other similar blocks to compare them to.

3.4 Develop phase - conceptual design

From the sketches and discussions during the co-design workshop, we developed a conceptual design meant to capture and enrich the metaphors, concepts, mappings, objects, and personas that can be involved in the design.

Figure 3: Developed analog film on a lightboard.
From 'Museum of Obsolete Media', used under CC BY-SA 4.0.

Metaphor: Lightboard

In analog photography, a lightboard is a flat luminous surface where the developed film can be previewed and frames compared to each other before choosing which to enlarge onto photo paper (Figure 3). It can also be used for drawing, to copy or compare art drawn on paper. In this project, the lightboard is a metaphor for a work surface where pieces that you need to illuminate or inspect can be kept and compared.

Concept: Changeset

The changeset is the central concept in Gerrit and is what you are approving or rejecting in a code review. It is submitted by an author and contains a commit message written by the author to describe the contained changes.

Concept: File

A file is part of the changeset and has a name, a path, an old and one or more new versions. The diff view in Gerrit can show the comparison between any pair of selected versions for the same file.

Concept: Line

A line in a file within the changeset. Line numbers and contents might not correspond between versions of the same file, so a line only makes sense as a concept when referring to a specific version.

Concept: Blocks

A block consists of one or more consecutive lines of code in a file. Since it is built up of lines, it also must refer to a specific version of the file.

Concept: Diff view

A view to show differences between two text-based contents. One content is designated as the older one, and the other as the newer one. Differences can be shown interleaved or in a side-by-side view.

Concept: Matches

Two blocks that either have a similarity score above a certain threshold or that the user has manually selected.

Action: Select block

Select a block by selecting lines on either the old or the new side of the file diff view.

Action: Compare blocks

Open up a comparison of a match.

Action: Comment on match

While looking at a diff view of a match, write review comments.

Action: Read comments in context

When seeing a review comment in a file, open up the match in a diff view to see the same context as the comment was written in.

Persona: Willow

Willow is new to the team and inexperienced with the particular code base. They have some experience in general software development and code review from education and previous projects, but not seasoned enough to feel super secure in a new code base and environment. Willow is part of a small team with five colleagues, two of whom are on a similar level and three who are more experienced, especially in this particular codebase. Their process is mutual peer reviews where at least one, preferably two, developers should look at and approve new changesets. When reviewing code and finding whole functions or blocks that have been

removed, Willow often asks the following questions and would like the design to help them answer them:

- Where did this code go?
- Was this block deleted or moved?

Persona: Raven

Raven has experience with many different software projects in a range of teams, tools, and programming languages. In the project she is working on, she knows most of the codebase by heart and is aware of most of the interactions and intricacies in how it interweaves with its environment. Raven is part of a small team with four colleagues, two on a similar level and two less experienced. Their process in the team is peer code review where one other developer should look at and approve all changesets. When finding files, functions, or blocks that have been moved, she often asks:

- Is this moved block identical to the original?
- Why was it moved?
- How does moving the code affect the surrounding code and projects?

3.5 Deliver phase - first high-fidelity prototype

To converge the ideas and concepts from the Develop phase we focused on **Idea 3**. **Idea 1** was discarded since it would be hard to support comparison across files intuitively with just scrolling. Also, it would require the reader to compare manually line for line without diff coloring. **Idea 2** was also discarded since it would need a functional version of **Idea 3** to start with so that wherever the blocks come from (same patchset version, other versions, or other changesets) they could be compared easily and intuitively.

The prototype[3] (Section 2.4) simulates both source code that has been moved and then modified within a single file, and code that has been broken out into a new file and modified in the process. The prototype is designed as if being fully integrated into Gerrit [**Principle 4**].

Feature I: Comparison modal

The comparison modal (Figure 4) shows a detailed diff view between two blocks of code with intra-line markings to highlight the changed parts. The context around

[3] Available at `https://figma.com/proto/KZIrsBH8DZ22ZI0B0YD2BC/GBC?hide-ui=1`

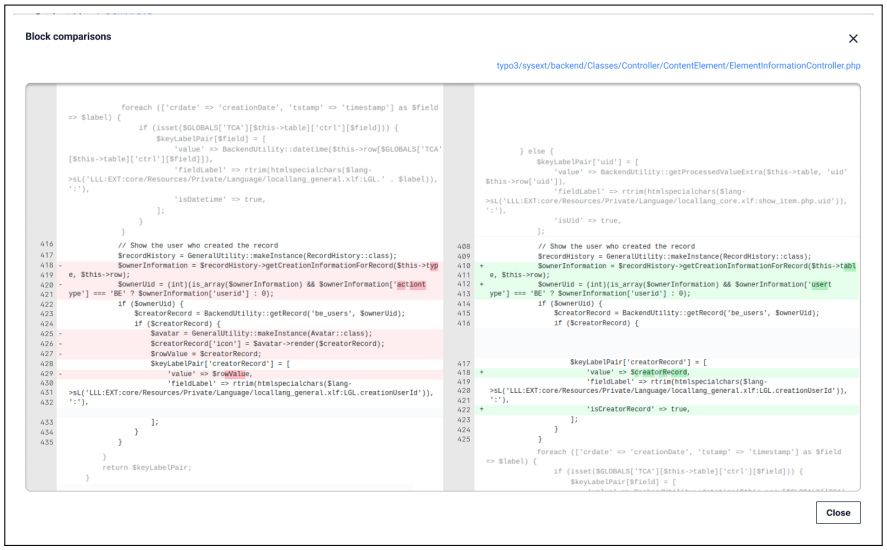

Figure 4: Modal dialog for detailed comparison of blocks.

the selected blocks is shown without background colors and with lower contrast to make it clear that it is not part of the current comparison [**Principle 1**].

Clicking on the row numbers opens up a comment text field, so the reviewer can write directly in the context that made them notice an issue or questions [**Principle 2**].

Feature II: Comparison within a single file

If code has been moved within a single file, that is detected and a hint is shown on the line above the moved code (Figure 5). The user can click the link to open the comparison modal with the two blocks loaded.

Feature III: Lightboard toolbar

If the user wants to compare blocks across files [**Principle 3**], an added or removed code block can be added to the list of interesting blocks to compare (Figure 6). This toolbar can be minimized to only take up less space and attention while reviewing and navigating the files, and then expanded to show the list of selected blocks.

Feature IV: Comparison to lightboard block

When navigating through the files under review, the current file is checked for blocks that are similar to any block that is on the lightboard. These blocks show a

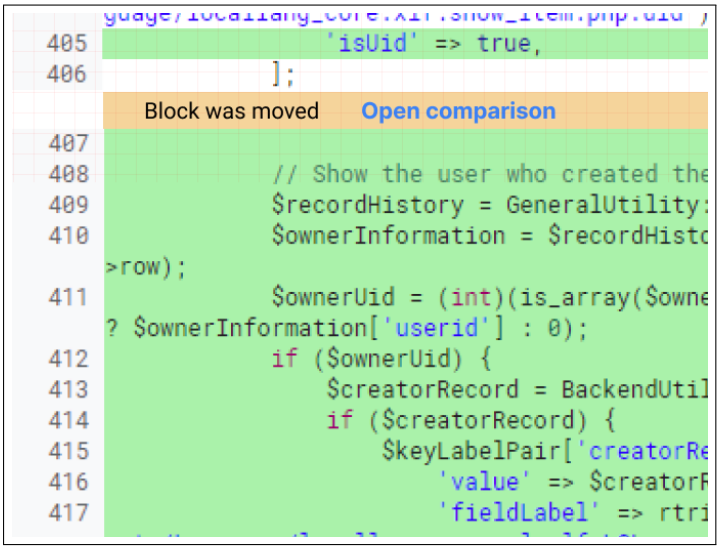

Figure 5: Moved code detected within the same file.

hint on the line above the code block (Figure 7). Clicking there will open the comparison modal dialog between the block and its closest match on the lightboard.

3.6 Deliver phase - prototype persona verification

The first sanity check of the prototype was done by the research group by checking if the questions and use cases described by our two personas, Willow and Raven, could be answered and performed using the flow in the prototype.

Willow: Where did this code go?

This question is answered by the move-detection and the headings that come up over a block [Feature II, Feature IV] and allows Willow to compare it to similar blocks in the same file, or files that have been added to their lightboard.

Willow: Was this block deleted or moved?

This question is also answered by the move-detection, where moved blocks will have headings over them showing Willow where the block was moved to or from [Feature II, Feature IV]. However, if the block is moved between different files, and the source or destination blocks are not on the lightboard, the heading will not show and it will look the same as if the block was deleted or newly created.

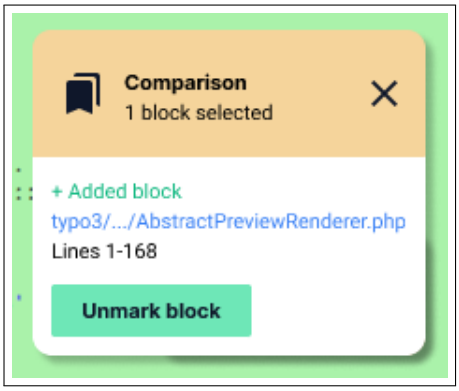

Figure 6: Floating toolbar with blocks marked as interesting for comparison.

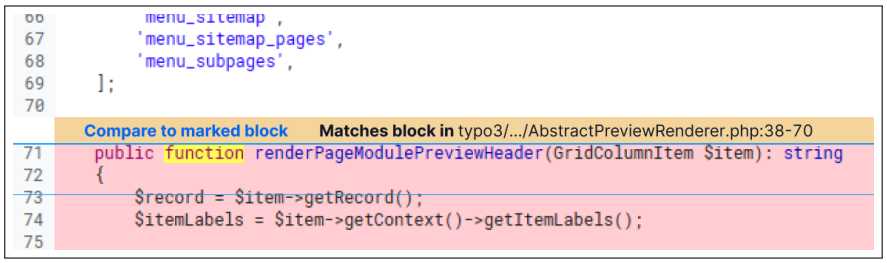

Figure 7: Moved code that matches marked block on lightboard.

Integrating a clone detection [52] tool could make it possible to scan the whole changeset for similar blocks to detect those cases.

Raven: Is this moved block identical to the original?

This question is answered by the comparison modal [Feature I]. Here, Raven can see detailed differences between the block before and after it was moved, with intra-line diff markers to highlight changes.

Raven: How does moving the code affect the surrounding code and projects?

This question is a bit more complex. The comparison modal [Feature I] should give Raven a detailed view of any changes in functionality, which will help judge the effects on surrounding code. It also shows the context before and after the blocks, so that Raven can look for potential side effects there as well.

Raven: Why was it moved?

Not supported - The design of the prototype does not give any extra help for this question. Knowing the details of the changes might in the best case give you a hint, but without a clear rationale being stated by the changeset author it is hard to infer it just from the code.

3.7 Deliver phase - prototype evaluation

The prototype evaluation (Section 2.4 and Table 1) showed that all of the 7 participants could complete the code review task, and were able to use the comparison modal dialog to clarify questions that they had about the moved and refactored code blocks without any additional instructions except for the user interface.

One thing that stood out, when analyzing the recorded evaluations, was how positive the sentiment was regarding the usefulness. For example, in the co-design workshop some of the participants commented on how the solutions they used today, e.g., opening two browser windows next to each other and comparing the code line-by-line manually, worked pretty well, but when using the prototype they expressed surprise at how much easier it was to read and validate the changes by using the new comparison modal instead.

> **Participant 3** "So I am, if anything, yes, positively surprised that it is, yeah, that it feels like it works and is maybe also not a lot of different things that needs to be developed to still kind of make a pretty big difference to the better."
> **Participant 1** "Well, when code has been moved around and when it is so easy to use so you get comfortable with it [...] it is a really big difference. And then, like I said, it makes you able to stay in Gerrit the whole time. You don't need to, as said, cut and paste into a Meld-window and figure out what happened that way."

One participant also commented that this improvement could potentially change how they write code themselves. Today, they avoid moving and refactoring code in the same changeset since it was so difficult to review. They therefore try their best first to move the code, commit that, send that for review, and only after the move is approved go ahead and also do the refactoring.

> **Participant 1** "This thing can *really* make the difference between how you today tend to only move code but not touch it. *Then* you make the actual changes. [...] One reason today is that it is hopeless to review if you do, if you don't separate that into two steps."

For the case when code had been refactored into a new file, 3 of the 7 participants needed several passes back and forth between the two files before they understood how the feature worked and how the new file could be marked for comparison and then used to verify the changes by bringing up the comparison modal in the old file.

> **Participant 6** "But then there was a feature here in the second file, file 2. It said «Mark for comparison». At first, I did not understand what to do with it. Then I understood that I could go to file number 3 and then click on «Compare to marked block»"

Comments on the functionality and interface included that it was confusing sometimes which of the selected blocks were shown on the left or the right side of the modal diff dialog. Also, 4 out of the 7 participants commented that it should be possible to mark either the new or the old blocks for comparison and add them to the lightboard in any order. When comparing in a single file, several of the participants would have wanted a visual marker linking the two blocks that were detected for comparison.

> **Participant 5** "But I, I think it would have helped to have an explaining text from both sides, absolutely."

Another more general comment from one of the participants was that this feature might mean that you see and read code in a file you have not fully visited in Gerrit yet, so when you finally get to that file it should be marked in some way that these particular lines have already been seen and potentially commented on.

> **Participant 4** "...so then if I do this review and compare and see that, well, this block looks good, then I am finished with this part. But I am also finished with the other file, I just compared to. So then it could like almost be defined as reviewed. And if I press «Next» here it is nice to know that in some way, yes like, you have already, you looked at this file just now. There is not a lot more to see."

In regards to code review comments written while inside the comparison modal, it would also be important to include a link so that the author could bring up and read the comment in the same context as it was written in so that it would be easier to understand what the comment means and how they saw it.

> **Participant 2** "But if you click on the comments in the change view do you come to this view (the comparison modal) then?"

One other idea from one of the participants was to be able to click a single link and button and bring up all relevant comparisons for a full file collected in one single modal to save the time of reading through and clicking each correspondence one by one.

> **Participant 7** "To select several from this page would have been nice so that you can see, just click, like, I want this *and* this and then look."

Overall, the participants thought that the user interface, with links above the blocks, was clear and easy to understand. However, it would take some time to learn which blocks are useful to mark for comparison and get into the habit of doing so while passing through a file. Especially, if they are not completely sure if a matching block to compare to will occur later in the review or not.

3.8 Deliver phase - updates after evaluation

After the evaluation meetings, the prototype was updated to incorporate some of the feedback, in particular:

Consistent placement of old and new versions

If a selected block is originally on the left side, it should be kept on the left side also in the comparison modal and vice-versa if it is originally on the right side. If the user chooses to compare two blocks that are both on the same side, we could place the first selected one to the right and allowing the user to flip the comparison with a button in the top part of the modal.

Explicit location of matches

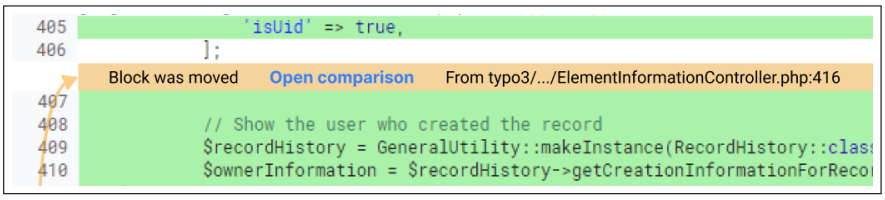

Figure 8: Moved code explicitly marked with arrow, filename, and line numbers.

Every action block with a comparison link lists the file name and the line numbers it would open a comparison with to make it more explicit what you would be comparing to. If the match is in the same file, a line with arrows is drawn linking the two blocks to give a visual marker (Figure 8).

Mark blocks for comparison from both directions

It should not matter if you encounter the old or the new file first when dealing with blocks that have been moved between files. Therefore, the links to mark a block for comparison are made available for both removed and added blocks.

4 Discussion

The goal of this study was to address the three research questions in Section 1 and through the choice of methodology, the execution and the analysis we have found new insights into all of them. Concerning $\mathbf{RQ_1}$, we found that several issues in the code review experience causes frustration for the group participating in the study. These issues are not directly related to the specific code, language, or process these developers use, so we believe that similar frustrations can be found among other developers elsewhere.

By designing and evaluating one possible modification to Gerrit, we also presented one answer to how review tools can be modified to improve the developer experience ($\mathbf{RQ_2}$). While this improvement can be seen as relatively small, we are convinced that even small improvements can have an impact since so much time is spent on code review and because the task is so cognitively demanding [53].

Finally, we explored one way to involve developers in improving their tools ($\mathbf{RQ_3}$), specifically code review tools. By centering developers' experiences and needs from the beginning, involving them both in the design and evaluation of solutions, we estimate that we have reached a prototype that is more in line with processes and flows in code review. Which helps give a larger improvement than its learning curve or distraction.

Overall, as a recommendation and reflection, we think that since the code review tools and processes today are so accepted and ingrained into modern software development practices, we are at a point where completely disrupting and re-designing these tools from the ground up would have a steep hill before reaching adoption and making a difference. One way around such adoption problems, is the way shown in this study; to make small improvements that have the potential to compound and over time, and after continued incremental development, make a big difference.

4.1 Reflection on the method

We estimate that the choice of method was a good fit for this type of study and we also discovered some ways in which the execution of it can be improved. During the co-design workshop, the participants were at first hesitant to directly change the prototype themselves. The first author, as the facilitator, had to draw, cut, and paste together the ideas in the room. Both to get things started, to keep the ideas going, and the solutions evolving.

After discussing this experience during a research group meeting, there were ideas shared about warming up the co-design team by first collaborating on designing something low-stakes and fun, like a celebratory garbage bin for redundant code. When the team has warmed up and gotten used to creating and discarding sketches together it could be easier move on to the real design task and see more confidence in the group to directly change and create prototypes. This would require 2-3 times as much time from the participants, which can be an issue in practice, but could have the potential of getting results that are more creative and more firmly anchored in the whole group.

4.2 Threats to validity

One threat to the validity of this study is that the participating software developers in the focus group and co-design workshop are a fairly small group of only 6 developers. They are also all working in the same company, albeit in different teams. While this focus group size falls within the ideal size of 4-8 participants described by Liamputtong [48], she also recommends conducting several focus groups with the same interview questions until reaching data saturation (i.e., no new ideas or data is being found). However, since the study is aimed at finding *some* (rather than all) possible improvements we believe that ideas and experiences from this group are still valid.

The study is also only valid for the code review tool Gerrit. However, since Gerrit has a large user base, this still means that the results could have broad applicability. Also, other tools in wide use today, such as GitHub and GitLab, are similar to Gerrit and there might be findings here that can be generalized to apply to them as well, but it is beyond the scope of this study to verify or evaluate that.

There is also a risk that the prototype validation has a positive confirmation bias since the developers evaluating the prototype have participated in the co-design workshop that led to its creation. In this way, they might both be more familiar with the concepts and also personally invested in the success of the prototype. To check for this we also did one extra validation with a software practitioner who has not been involved in the project at all and works for a different company than the other participants.

Finally, the prototype evaluation is also exposed to response bias [54]. We tried to counter-balance this by encouraging feedback.

4.3 Directions for future work

Paths to broader adoption and impact

Exploring the prototype of more flexible diffing between blocks further and implementing it as a feature or a plugin to Gerrit (or GitLab or GitHub) would be very interesting and something we would like to address in the future. Further validation with several other teams of industry practitioners would be needed to

refine the prototype into something generally useful. Implementing the features of the prototype would also require the integration of clone detection tools [52] to identify blocks to suggest for comparison.

To have any significant benefits compared to already available solutions and avoid context switches for the developers, we feel that the implementation would need to be available directly in the review tool. Therefore it is important to anchor the idea and the implementation in the Gerrit (or GitHub or GitLab) community. This could increase the support for a plugin implementation or possibilities for merging the feature into the tool itself. A first step here could be presenting the prototype and the thoughts behind it to the Gerrit community during the annual Gerrit User Summit.

Future design explorations

The other problem areas and solution suggestions uncovered during the workshops are also viable ideas for improving the experience of code review and reduce the cognitive load of the task while at the same time having the chance of increasing the benefits of a team's code review process. The idea of overlays or layers of meta-information from continuous integration systems, tests and source code analyzers is particularly interesting to study and explore further.

Increase understanding of cognitive demands

While many papers agree [3, 43, 55, 56] that code review seems to be a cognitively demanding activity, there have not been many studies to measure the cognitive load during code review and compare that to other tasks that are known to be demanding. It would be interesting to do a study integrating EEG measurements [57] or fNIRS [58] with participants doing code reviews of different sizes and also, for example, general problems in math or programming. This could give valuable insight into code review's cognitive demands and guide future exploration into its design.

5 Conclusions

We used the Double Diamond design process to explore how the Gerrit code review tool could improved. By hosting a focus group we found several problem areas that are common experiences when doing code reviews. The problem of comparing moved and refactored blocks that the built-in diff algorithms don't pick up was chosen for exploring solutions.

A co-design workshop with industry practitioners was held and the prototype created collectively there was then refined to a high-fidelity interactive prototype that could be evaluated in one-on-one testing sessions.

The results show that making these kinds of comparisons has the potential of improving the code review experience both by reducing the mental workload and also give higher accuracy in the comments and the analysis of the code. The user interface of the prototype was also intuitive enough for all of the participants to be able to complete the assigned task without any hints or instructions.

APPENDICES

I.A Focus Group Design Brief

Agenda for the focus group meeting (Section 2.1).

I.A.1 Goal

Collect information about challenges in understanding the changes in code reviews that span multiple files.

I.A.2 Warm-up Questions

- Which is the best tool for code reviews? Gerrit, GitHub, or GitLab?
- Which is your favorite IDE? Vim, Emacs, VS Code, others?

I.A.3 Main Questions

- Do you read the files in a code review one time or multiple times?
- Why?
- How do you choose which file you read first?
- How does it affect your review what kind of file that you read first? (API, test, etc.)
- In what way?
- What is your experience of reading and understanding new code in other tools, environments, or situations? (IDE, pair programming, explained by a colleague, on paper in a book, etc.)

- In what ways do they work?
- Where do you start reading in those cases?
- In what ways is that different compared to code review in Gerrit?
- Are there features and support from there that could have been applied also in code review tools?
- Is there anything else you would like to tell?

I.B Co-Design Workshop

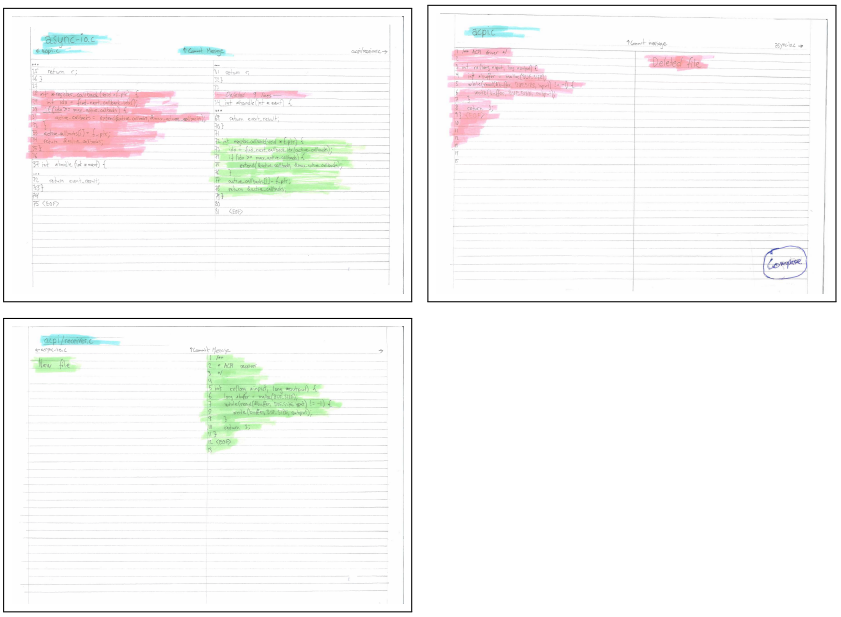

Figure I.B.1: The three base views for the co-design workshop.

The top-left corner shows code moved within the same file. The top-right corner shows a deleted file and the bottom-left corner shows an added file with similar content.

The material for the co-design workshop consisted of three pre-drawn base views (Figure I.B.1), blank paper, overhead film, scissors, tape, and pens in different colors. The base views displayed some of the use cases brought up during the focus group. The participants created simple prototypes and showcased ideas

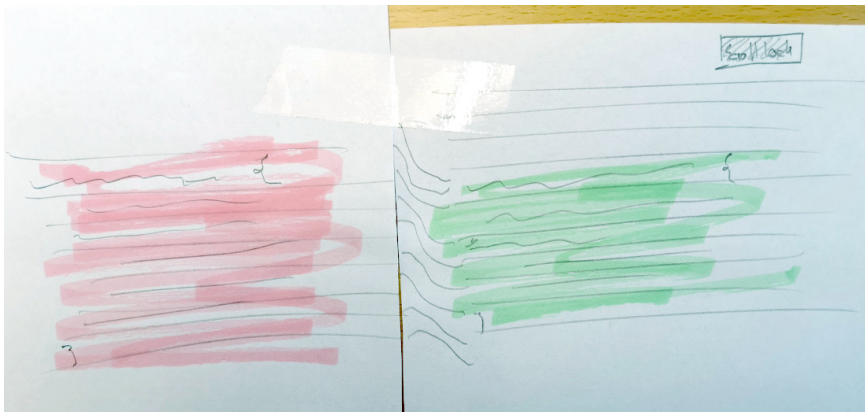

Figure I.B.2: First sketch of comparison modal.

The red area to the left symbolizes removed code, the green area to the right symbolizes added code.

by overlaying new components on top of the base views. A small example can be seen in the top-right image where the first sketch of the "Compare" action overlay is drawn on overhead film and then taped onto the base view.

In Figure I.B.2, we show an example of a rough sketch made during the workshop to show the level of fidelity that we think is feasible and appropriate for the setting. This is the first sketch of the comparison modal and shows the idea of lining up and comparing blocks that are in different places in the original Gerrit diff view.

I.C Prototype Evaluation Questions

I.C.1 Task

- You are welcome to think aloud, but I will not respond or give hints until after the task.
- Perform the code review.
- Find changes in any moved code.

I.C.2 Interview Questions

- What was your experience of trying this prototype?
- How often do you encounter the situation the prototype tries to aid in?

- How much would a fully functional version of the prototype help with the problem?
- What are the solution's greatest weaknesses?
- What are the solution's greatest strengths?

PAPER II

SUPPORT, NOT AUTOMATION: TOWARDS AI-SUPPORTED CODE REVIEW FOR CODE QUALITY AND BEYOND

Abstract

Code review is a well-established and valuable software development practice associated with code quality, interpersonal, and team benefits. However, it is also time-consuming, with developers spending 10–20% of their working time doing code reviews. With recent advances in AI capabilities, there are more and more initiatives aimed at fully automating code reviews to save time and streamline software developer workflows.

However, while automated tools might succeed in maintaining the code quality, we risk losing interpersonal and team benefits such as knowledge transfer, shared code ownership, and team awareness. Instead of automating code review and losing these important benefits, we envision a code review platform where AI

Lo Gullstrand Heander, Emma Söderberg, Christofer Rydenfält "Support, Not Automation: Towards AI-supported Code Review for Code Quality and Beyond" In *33rd ACM International Conference on the Foundations of Software Engineering (FSE Companion '25)*, 2025.
DOI: 10.1145/3696630.3728505

is used to *support* code review to increase benefits for both code quality and the development team.

We propose an AI agent-based architecture that collects and combines information to support the user throughout the code review and adapt the workflow to their needs. We analyze this design in relation to the benefits of code review and outline a research agenda aimed at realizing the proposed design.

1 Introduction

Code review is a valued practice in the software industry. The practice, originally introduced for quality improvement in the 1980s [59], is today valued for a number of properties beyond code quality. Bacchelli and Bird [3] report that developers' motivation for code review is, in order: defect finding, code improvement, alternative solutions, knowledge transfer, team awareness, improve the developer process, share code ownership, avoid build breaks, track rationale, and team assessment. Notably, at least half of these motivations are not directly about code quality but about user needs or interpersonal benefits. Thus, code review is clearly an important source of learning and education within a team.

Although code review is valued, it is also a time-consuming practice. Software developers have been reported to spend between 10-20% of their working time doing code reviews [2, 40]; with an estimated 28 million software developers during 2024 [41] this corresponds to 22-44 million hours every work day. The 2023 DORA State of DevOps report [8], focused on industry best practices, reports that optimizing code reviews is a key factor in overall developer team productivity. There is a need to continue to develop code review and its tools to improve the practice.

With more and more capable AI models available, there is an increased interest in automated code review. For example, Lu et al. [11] have introduced LLaMA-Reviewer to automate the code review task. Yu et al. [12] present Carllm for improved precision and clarity in automated code review. Tang et al. [13] introduce CodeAgent, an approach in which multiple agents collaborate to find code quality issues. Google's DIDACT project [14] trains ML models on the sequential steps in software development processes, such as code review, to automate them. Although these approaches may be able to ensure code quality in the future, we see an overhanging risk that the interpersonal and team benefits of code review will be lost in such a development.

In this paper, we present a vision for using AI to *support* code review and its users, rather than replacing the activity. We believe that we should strive to boost *all* the positive effects of code review, including interpersonal effects such as knowledge transfer, team awareness, and shared code ownership. We propose to do this by focusing on the participants of the code review process, the authors and reviewers – the users, and their needs. We envision an adaptive code review

pipeline, with improved user experience and powered by an AI agent-based architecture, that provides customized support to each code review role and context.

The contributions of this paper are an architectural design for an AI agent-based code review platform (Section 3), an analysis of the design with respect to code review benefits (Section 4), and an outline of future research to realize this vision (Section 6).

2 Today's Code Review and Its User Needs

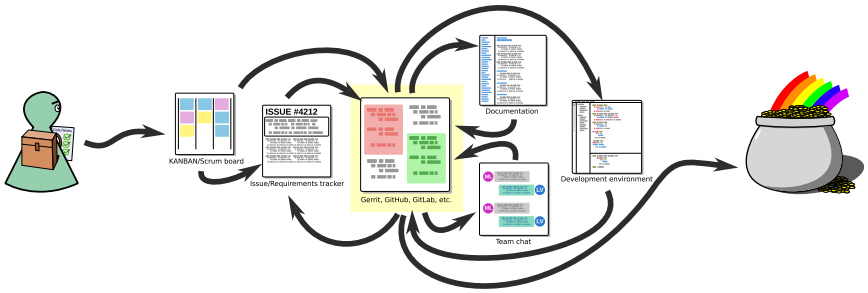

Figure 1: Illustration of navigation between tools in modern code review.

Modern code review practices are centered on a web-based code review tool, such as open source tools like Gerrit, open services like GitHub and GitLab [60], or proprietary tools like Critique [2] and CodeFlow [3]. These tools contain functionality to list code changes awaiting review, compare the changed and original code [61], write review comments, respond to review comments, and vote on the next steps. By integrating with continuous integration (CI) systems [62, 63] they can show results from automated tests, clean code with automatic formatters, and reject code based on compiler or linter errors.

There are several challenges with today's code review practices and tools. The information needed to complete the review is scattered across different systems such as issue trackers, requirements databases, KANBAN boards, team chats, API documentation, and CI reports. Different users involved in code review will have different needs, processes, and goals when using code review systems [64]. For example, the author of the change may want to view the code and the rationale behind it briefly to discover mistakes before submitting it for review. An expert reviewer might want to get an overview of architectural changes and the performance profile before and after the change. A new team member could spend extra time understanding the rationale and need to ask questions about unfamiliar patterns or APIs. Some team members might skim the rationale and code to stay up-to-date with changes in the repository, but not vote or write comments.

The reviewer must use their experience and the team processes to navigate between tools effectively and decide which steps are helpful and when [47]. Sometimes, even check out the code locally to trace variables and execute the code to verify its behavior and performance. This experience takes time to build up, and becoming effective at carrying out code reviews in a new workplace can take up to a year [37]. Even with experience, it demands time, effort, and focus [6]. There are often difficulties in understanding the rationale for the change [5] and reviewing large changes [65]. Multiple review cycles between reviewers and authors, together with long response times, can create delays affecting the overall productivity [8, 65].

Figure 1 illustrates what this can look like for a developer embarking on a code review. Carrying their experience as a backpack full of resources and a checklist with the team code review process, they switch between different tools and systems. The code review system (Gerrit, GitHub, GitLab, etc.) is in the center, and the point to return to and start from. With experience, iteration, and help from their peers, they can reach the "pot of gold" containing improved code quality, better knowledge distribution, team cohesion, and more.

3 Design Proposal

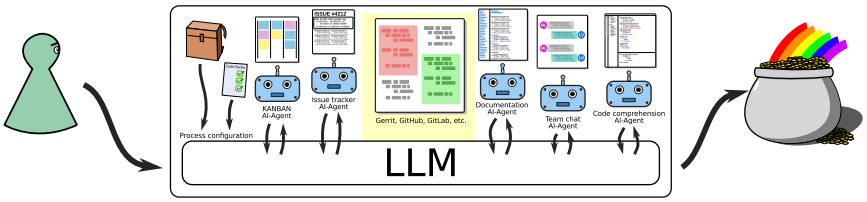

Figure 2: Proposal for a code review platform architecture driven by AI-agents.

Our design proposal is a code review platform that is built on an agent-based AI-OS architecture [66]. In the AI-OS architecture, a central LLM takes a role similar to the kernel in an operating system and is responsible for interpreting user input, planning, and coordination. Smaller AI agents connected to the central LLM manage integrations to databases and online APIs; functioning as the input, output, and memory subsystems in the analogy to an operating system. Several specialized Small Language Models are trained to create prompts, construct database queries, build API calls, and combine results [66].

In the case of code review, these agents implement integrations with the version control system, requirements database, issue tracker, continuous integration, API documentation, etc., to collect all the information needed before, during, and after code review. We envision a user interface that embeds existing and familiar tools, such as Gerrit, GitHub, GitLab, CodeFlow, etc., in the center. Information from,

guidance by, and interaction with the agent-based system is placed below and in the sidebar.

Figure 2 illustrates our envisioned code review platform architecture with an example flow. Throughout the review, the reviewer interacts with the LLM trained to act as the platform's foundation. Information about the user's preferences and the team's code review process (the backpack filled with experience and the process document from Figure 1) can be configured by the team and the user. Parts of this configuration could potentially be updated with reinforcement learning or similar approaches to make it adaptable over time and for different contexts. The LLM customizes the process for each user using the configuration and coordinates the AI agents to provide the information and support needed at the right time during code review.

For example, for an in-depth review use case, the platform could first assist with picking a change to review, given the reviewer's time constraints. Then help throughout the process of understanding the rationale of the change, connecting it to related work, reading the code changes, finding potential defects, writing constructive comments, and finally assist in making a decision on accepting the code for integration or sending it back to the author for adjustments.

4 Analysis of Design

An AI agent-based architecture has the potential to preserve or amplify all the benefits of code review, as described by Bacchelli and Bird [3], while at the same time reducing users' mental load and time spent. The agents are trained or tuned for each aspect of code review, and the adaptable nature of the platform allows it to fit the needs of different users and teams.

4.1 Defect finding, code improvement, and alternative solutions

Current and future work on automated defect finding can be integrated into this architecture as one of the AI agents. An option would be to run a model similar to CodeAgent [13], but instead of automation support the user by marking parts of the code that could contain a defect and suggesting code review comments. Other agents could be trained to look for performance improvements, refactorings, and alternative solutions.

4.2 Knowledge transfer

Knowledge transfer during code review has, for example, been shown to reduce the impact of developer turnover by exposing developers to code they have not authored [67]. Achieving this requires keeping the human in the loop, i.e. doing the

code review supported by AI rather than automated with AI. AI agents can further be used to gather information so that the user does not have to navigate different systems to piece together the rationale, system architecture, and implementation details. They can also expand on user code review comments with references to team guidelines, language conventions, design patterns, and best practices.

4.3 Team awareness

For users reading through code reviews to get awareness of ongoing work, it can be time-consuming and demanding to read through a large code diff just to understand what it does. Language models for code comprehension are developing rapidly, with many options both in open source [68] and in closed source [69]. Receiving a code summary as soon as you open the review could be enough for users looking to be aware of current changes with the option of going deeper into the changed code when needed.

4.4 Improve developer process

A unified platform for the whole code review flow, instead of manually switching between code review tool, documentation, KANBAN boards, etc. will streamline the developer process as a whole. Introducing, configuring, and refining the settings and use of the platform can also encourage teams to examine and improve their process. The adaptability of a multi-agent approach can also help situations where different teams follow different workflows but still need to work together.

4.5 Share code ownership

Software development teams are at risk of developing a blame culture [70], where developers are held personally responsible for introduced bugs. This culture can lead to a reluctance to contribute new features and undermines trust between team members. Code review can contribute to preventing or mitigating this. It is no longer solely the fault of the author if a bug is introduced, but also of all the reviewers who did not discover it.

This is another benefit that would be at risk if code reviews were automated. Supporting the author and reviewers in assessing the code and giving them good grounds for their decision using AI, but leaving the decision of approving, rejecting, or revising the code up to the users keeps the developers accountable and encourages a culture of shared ownership.

4.6 Avoid build breaks

Connecting CI with code reviews is an effective way to encourage reviewer participation and ensure passing builds [71]. This kind of system already automatically

rejects changes that break the build. An AI agent could help by suggesting a fix or pointing out the most likely causes.

4.7 Track rationale

Finding the rationale can involve piecing together information from commit message, issue tracker, product plan, recent team meetings, team chat conversations, and more. Bringing this information together and summarizing it in one place makes it easier to find the rationale behind a code change and connect it to larger goals and efforts.

4.8 Team assessment

Code reviews generate metrics such as the number of comments written, acceptance rate for posted changes, time from comment to resolution, etc. Measuring individual and team performance in software development is very difficult, and looking at reviews may provide additional insight. Our proposed design does not affect the collection of metrics, but has the potential to make code reviews more efficient, increase quality, and improve assessment metrics for the whole team.

5 Related Work

There are a few recent studies, also focused on supporting rather than automating code review, that complement the vision in this paper.

Unterkalmsteiner et al. explore providing reviewers with a better context for the code under review with a proposal called the Code Review Contextualizer [72]. Using existing literature on what developers need help with during code review, they present several parts that could be improved by data collection and assistance. Although they do not go into detail regarding what kind of technologies and architectures could be used to implement such a system, their research on use cases that should be supported is a valuable foundation for building future AI-supported code review systems.

Almedia et al. present AICodeReview, a plugin for the IntelliJ integrated development environment that takes advantage of GPT 3.5 [73]. The plugin analyzes code snippets while they are being written and identifies potential issues. Comments, resolutions, and improvement suggestions are provided in the editor. This approach can likely reduce human code review time since changes submitted for code review are hopefully of higher quality than they would have been without the AICodeReview plugin.

Wang et al. presents an AI agent-based approach to recommend which reviewers that should be assigned to each code review [74]. Their work complements the suggestion in this paper very well in that it seeks to build AI-based support systems for human reviewers instead of automating the activity. The article shows

better preliminary performance using AI agents compared to previous state of the art for reviewer recommendation.

Yang et al. investigate machine learning approaches to predict review duration and merge approval rate prior to code review [75]. This can give code change authors early feedback and give them the option to rework changes before sending them out for code review, in order to reduce lead times and number of code review cycles. They concluded that while the approach was promising, work work was needed, especially in making the feedback explanatory and actionable.

6 Research Agenda

Here, we list research activities that we believe are important for realizing our proposed design vision.

6.1 Increased understanding of user needs

Recent work improves our understanding of the causes of confusion [5], anxiety [76], and misalignments [6] in code review. This research helps to provide a deeper understanding of user needs and user experience in code review, but there is much more to study here. For example, the needs of each user in code review vary [6], and this variation goes beyond roles such as author and reviewer, and may extend into tasks such as gatekeeping [2].

6.2 Measuring effectiveness of code review

Despite the wide use of code review in industry and its time-consuming nature, there is no unified way to measure the effectiveness of code review. The primary benefit explored with regard to measuring is 'defect finding' [39]. Other benefits, such as knowledge sharing, team awareness, and shared ownership, have not been studied as extensively. With a deeper understanding of the effectiveness of code review, we can consider cases where code review is the most effective with respect to different benefits. This understanding would open up for addressing the reported industry need for optimization [8], but without an unintended loss of code review benefits.

6.3 Effective code review interaction

The interaction with today's code review tooling has stayed largely the same since the introduction of the ICICLE tool in the 90s [7], with variations of interfaces centered around textual diff views of changed files (Gerrit, GitHub, Critique, and so on). Although this user interface design helps to provide answers to questions best answered with a textual diff, they are less successful at answering questions connected to, for example, requirements or execution behavior [6]. There is room

for more exploration and innovation here to better align the interaction with user needs.

An interesting research direction would be to explore the use of collaborative and user-centered design processes [32] to take advantage of the depth of experience in the software development community. Another interesting direction would be to explore new ideas from conversational interaction design [77]. There are examples on the use of conversational interaction design for software development tools [78, 79] and recent advances in language models provide interesting new possibilities.

6.4 Effective AI integration

There are technical challenges in how best to build an AI agent-based code review pipeline, as outlined in our proposed design. One challenge is to identify suitable tasks for agents. For example, should an agent focus on one aspect of program comprehension, like code summation, or rather be trained for larger functional areas? There is also the challenge of choosing suitable models for different agents, along with data collection, training, and tuning. The models need to be integrated, and the main LLM trained to manage orchestration based on configured user needs.

An interesting direction is to identify a minimal viable use case and iterate on a smaller instance of the design with fewer AI agents. Follow best practices for AI in software development [80] and maintain close interaction with industry practitioners for rapid prototyping, early user feedback, and testing.

CODE REVIEW AS DECISION-MAKING —BUILDING A COGNITIVE MODEL FROM THE QUESTIONS ASKED DURING CODE REVIEW

PAPER III

Abstract

Code review is a well-established and valued practice in the software engineering community contributing to both code quality and interpersonal benefits. However, there are challenges in both tools and processes that give rise to misalignments and frustrations. Recent research seeks to address this by automating code review entirely, but we believe that this risks losing the majority of the interpersonal benefits such as knowledge transfer and shared ownership.

We believe that by better understanding the cognitive processes involved in code review, it would be possible to improve tool support, with out without AI,

Lo Gullstrand Heander, Emma Söderberg, Christofer Rydenfält, "Code Review as Decision-Making – Building a Cognitive Model from the Questions Asked During Code Review" Submitted to *Empirical Software Engineering, Springer Science*, 2025.
DOI: 10.48550/arXiv.2507.09637

and make code review both more efficient, more enjoyable, while increasing or maintaining all of its benefits. In this paper, we conduct an ethnographic think-aloud study involving 10 participants and 34 code reviews. We build a cognitive model of code review bottom up through thematic, statistical, temporal, and sequential analysis of the transcribed material. Through the data, the similarities between the cognitive process in code review and decision-making processes, especially recognition-primed decision-making, become apparent.

The result is the Code Review as Decision-Making (**CRDM**) model that shows how the developers move through two phases during the code review; first an orientation phase to establish context and rationale and then an analytical phase to understand, assess, and plan the rest of the review. Throughout the process several decisions must be taken, on writing comments, finding more information, voting, running the code locally, verifying continuous integration results, etc.

Analysis software and process-coded data publicly available at DOI: 10.5281/zenodo.15758266

1 Introduction

Code review is a well-established activity in modern software development valued for both quality assurance and interpersonal benefits [2, 3]. On a global scale, with more than 28 million software developers [41] spending 10%-20% of their working time reviewing code [2, 40], more than 22-44 million hours are spent on code reviews daily. However, despite the importance of code review, there are significant misalignments between the tools used and the goals and actions of software developers, which reduces efficiency and adds frustration [6]. The code review process is also challenging for many teams with a range of common antipatterns [5]. Improving code review tools and processes has huge potential benefits for the software engineering community. In addition to saving time on the code review itself, the 2023 DORA State of Devops industry report finds that teams with more efficient code review have up to 50% higher software development throughput overall [8].

However, despite the potential benefits of improvements, the tools used for code review today are very similar to the first tools introduced in the early 1990s, for example, the ICICLE tool [7]. Similarly to today's code review tools, such as GitHub[1] or Gerrit[2], ICICLE was centered around a textual diff view where comments can be added by humans or by automated analysis. During the same time, we have seen huge developments in tools for writing, navigating, and understanding software [9]. We should not leave code review behind. With the recent increase in AI assistance capabilities, there has been a growing interest in how to utilize AI-based assistance in software development tools [81]. This trend also

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

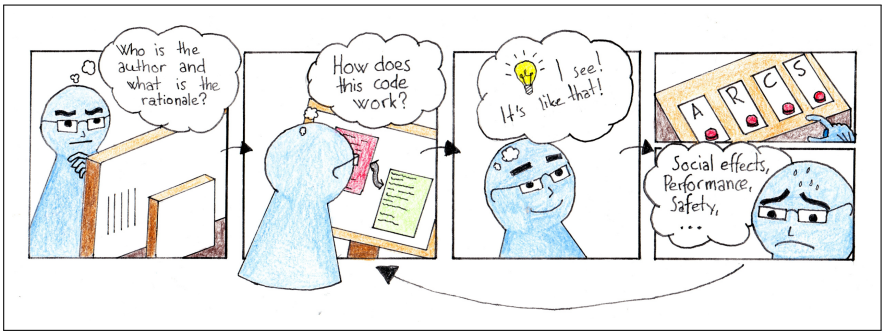

Figure 1: Illustration of steps and questions during code review. First, the developer orients themselves on the context of the code review, next asks how a part of the code change works, moves on to comprehending the code, and finally have to make a decision on how to proceed. Considering social effects, performance, safety, etc., should they **A**ccept, **R**eject, **C**omment, or **S**earch for more information. The reviewer iterates and looks at more code changes before reaching a final decision. **Takeaway:** Code review starts with an orientation phase, followed by an iterative comprehending-assessing-decision phase. Comprehending the code change is not the end goal, but rather a prerequisite for deciding how to handle the review.

extends to AI-powered improvements in code review, specifically automated code review has received a lot of attention in recent years. For example,Lu et al. [11] introduced LLaMA-Reviewer to automate the code review task. Yu et al. [12] presented Carllm for improved precision and clarity in automated code review. Tang et al. [13] proposed CodeAgent, an approach in which multiple agents collaborate to find code quality issues. Google explored how to automate code review in their DIDACT project by training ML models on each of the sequential steps [82]. There is also recent research on user experience improvements [61], AI assistance frameworks [72, 83], and innovative visualizations [84].

Although the focus on automated code review is interesting, several of the benefits of code review, such as knowledge transfer and shared code ownership, are interpersonal and risk being lost if the activity is automated [83]. To improve code review processes and tools while preserving all the important benefits it is important to understand the activity and its challenges well. However, while the practical process of code review is well researched and described [2, 60], there are few studies on the cognitive processes during code review. Gonçalves et al. [20] investigate how developers form and use their understanding of the changed code under review, building a model for code review comprehension . This is interesting work contributing to the understanding of code review, but in this paper we want to look at a wider scope beyond comprehending the changed code and study the

cognitive process of the code review activity as whole including choosing a review, writing comments, voting, looking for more information, etc.

Building a theoretical model of cognitive processes in code review grounded in interviews and observations can facilitate improvements in code review tools and processes in several ways. By analyzing existing tools to see which parts of the cognitive process they facilitate or hinder, by using the model to reason about the effects of new tool ideas, or by adapting the code review process in your organization to better match the cognitive process of the developers.

1.1 Research Questions

The goal of this study is to investigate the cognitive process during code review and how it can be modeled to increase our understanding of code review and guide future improvements to tools and processes. This leads us to the following main research question:

RQ$_1$ *How can the cognitive process of code review be modeled from a theoretical perspective?*

Supporting Research Questions

Since the cognitive processes of the developers during code review are not directly observable, we must study their actions and behaviors and use that insight to theoretically model the cognitive process. A basic assumption of this work is that developers *actions* when reviewing code are intentional, that is, that they are tied to some *meaning* that gives them direction [85], and that it is this *intentional relationship* [86] that needs to be analyzed to understand the cognitive processes involved in code review. In line with the theory of planned behavior, we also assume that the actual behavior or actions of reviewers are predicted by their intentions [87]. Furthermore, we assume that *questions asked during code review* gives an indication of the intention of the reviewer when the question is asked, and thus that they give valuable insights into the reviewers' intent or cognitive focus in different parts of the review process.

To move from a state where you know nothing about the review to a state where you are ready to vote for accepting or rejecting the code change, many questions must be answered and different aspects of the code change understood. Similarly to Letovsky's study on questions during code comprehension [21], in this paper, we study the explicit and implicit questions asked during a code review in order to build a theory about the cognitive process of the reviewers. By analyzing the patterns of which questions are asked, when questions on certain themes occur and how they relate to each other, the observed data can build the foundation for a theory of the cognitive process during code review. This gives us the following supporting research questions:

RQ$_2$ *What questions do developers ask during code review?*

RQ$_3$ *How do the questions asked during code review connect to each other and the overall code review process?*

To explore these research questions in a context as realistic as possible, we perform an ethnographic think-aloud study combined with interviews [25, 35]. The study is conducted at the software tools department of a multinational software company (Section 3). In total, we observe 34 code reviews by 10 participants and analyze the results using thematic, statistic, and sequential analysis (Section 4). The analyses form a basis for the construction of a theoretical model of the cognitive process during code review, interpreted and illustrated in Figure 1 (Section 5).

1.2 Contributions

The contributions of this research are as follows.

- A theoretical model of the cognitive process of code review closely relating code review to decision-making processes (Section 5).
- A thematic and statistical analysis of the questions asked during code review (Section 4).
- Suggested directions for future work to apply the theoretical model to improve code review processes and tools (Section 6).

2 Background and Related Work

The method, interview quotes, analysis, and theory building in this study build upon an understanding of modern code review practices, challenges, social effects during code review, the terminology of Gerrit code review tools, as well as cognitive theories around decision-making processes.

2.1 Code Review in Practice

The term "Modern code review" was popularized by Bacchelli and Bird [3], where they defined it as *"(1) informal (in contrast to Fagan-style), (2) tool-based, and that (3) occurs regularly"*. Over the years, the properties of modern code review have changed. Code review has become more formalized, and many teams have checklists and processes that, for example, define how to conduct the review, how many approvals are needed to merge the changed code and how quickly the review is expected to be done [56]. Modern code reviews are today even more centered around the code review tools used. The tools define much of the process, how the code is analyzed and read during the review, how comments and responses clarify or solve issues, and finally how the changed code is approved or rejected [2, 60].

Industry and community practices also place an increasing emphasis on performing code reviews regularly and quickly. Since code reviews are mandatory in many teams and projects, high throughput of new features and bug fixes depends on code reviews being done as soon as possible. The industry report DORA Accelerate State of Devops finds that teams with faster code reviews have up to 50% higher software delivery performance overall, marking it as an important area for improvements [8]. As a reference, Kudrjavets et al. [88] analyzed code review times in eight different large open source projects and found a median time between submission and acceptance of less than 24 hours. Sadowski et al. [2] reports a median time of less than 4 hours between submission and acceptance, and a median of 4 code reviews per developer per week at Google.

Code review has been shown to have several benefits. Both for its nominal purpose of finding and reducing software defects, but also, importantly, for code improvement, finding alternative solutions, increasing knowledge transfer, building team awareness, improving the development process, sharing code ownership, avoiding build breaks, tracking rationale, and assessing teams [2, 3]. Code review is also an efficient way to spread information, such as best practices or information about new features, in a software development organization. A recent study by Dorner et al. [89] shows that the information spreads to up to 85% of the participating developers after an average of only 3 code reviews.

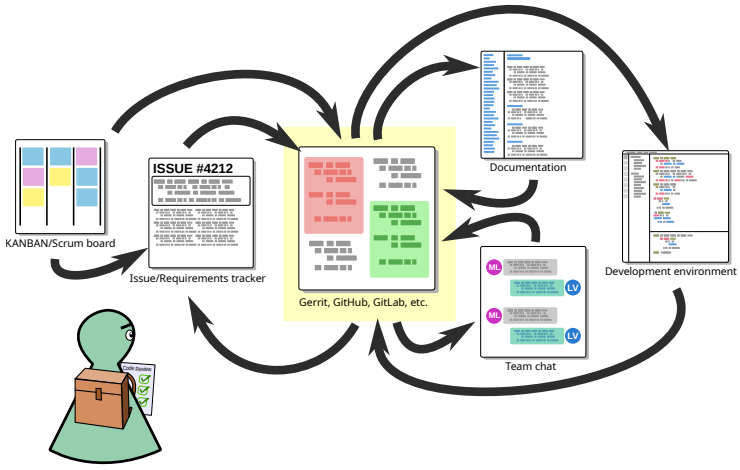

Figure 2: Illustration of the experience of navigating between different tools during code review. The backpack and checklist represent the reviewer's experience and team processes respectively. **Takeaway:** The code review tool is in the center, but cross-referencing with several other tools is necessary to understand the full context of the code review.

There are also several challenges with modern code review practices and tools.

Because of how current tools are designed, the reviewer needs to navigate between issue trackers, requirements databases, KANBAN boards, team chats, API documentation, continuous integration reports, etc., to gather the information needed to complete the code review. As illustrated in Figure 2, the reviewer must use their experience and the team processes to navigate between tools effectively and decide which steps are helpful and when [47]. This experience takes time to build up, and becoming effective at code reviews in a new workplace can take up to a year [37]. Even with experience, it demands time, effort, and focus [6]. Other challenges include understanding the rationale for the code change [5], long response times, many repetitions of reviewing the same code change, and reviewing large code changes [65].

2.2 Social and Team Effects During Code Review

There is recent work on improving code review practices and recommendations based on social and team effects in code review. Pascarella et al. [55] investigate the information needs of the reviewers during the code review by analyzing discussion threads with questions and responses in open-source projects. They find seven main categories of information needs, such as rationale and code context, and recommend ways to improve code review by better meeting these needs.

Lee et al. [76] studies anxiety and avoidance in relation to code reviews, both for code authors and reviewers. Their work outlines the main factors that contribute to code review anxiety and compel developers to avoid or procrastinate code review tasks, such as fear of judgment and criticism. They also present a CBT-based intervention that helps developers reduce anxiety after just a single session.

Coelho et al. [90] analyzes review comments and divides them into "refactoring-inducing" and "non-refactoring-inducing". They describe the factors leading to code refactorings in code changes and code review comments, since refactoring in the code review stage when the code is almost ready can be time-consuming. They give guidelines to researchers, practitioners, tool builders, and educators on how to better handle these situations and improve the code review process.

2.3 Gerrit Code Review

Gerrit Code Review[3] is the open source code review platform used by the teams in this study. It is a widely used code review tool that has its origins in Google Mondrian, the code review platform used for many of Google's internal projects. When Google released the Android project as open source[4], they wanted a code review tool with features and workflow similar to Mondrian, but built with open source

[3] https://www.gerritcodereview.com/
[4] https://source.android.com/

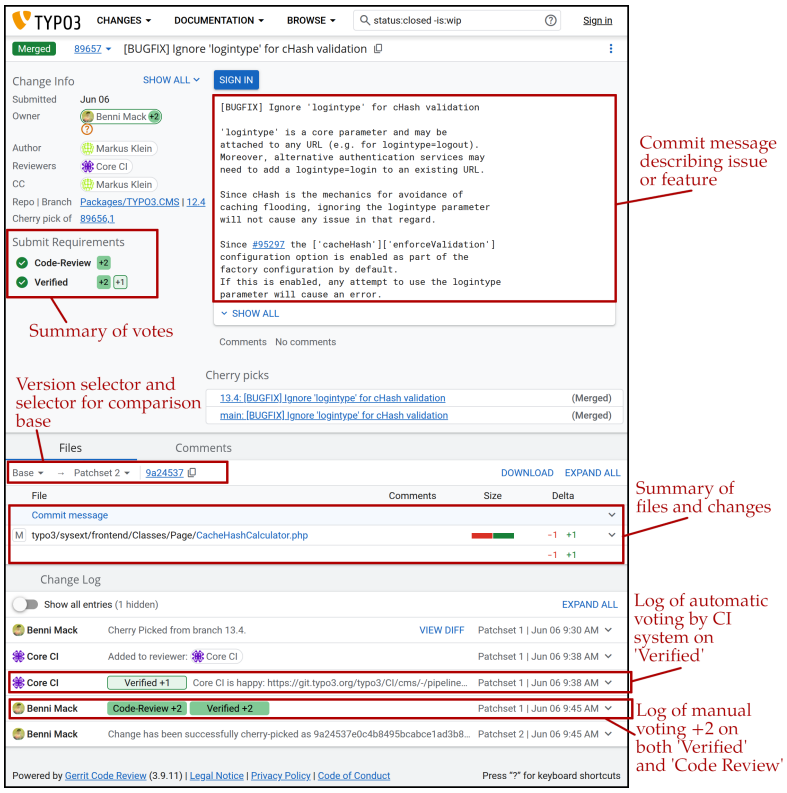

Figure 3: Annotated screenshot of Gerrit user interface. **Takeaway:** Note how the 'Change Info', 'Submit Requirements', and 'Commit message' sections gives an overview of the state and context of the code change. The log shows examples of both automated systems and human reviewers voting on different aspects.

software and using Git[5] instead of Perforce for version control. Figure 3 shows an annotated screenshot of Gerrit's user interface. Some concepts and terminology from Gerrit show up in the quotes from the study.

Changeset and Patchset A collection of one or more changed files with a commit message describing the rationale for the change is called a *changeset* in Gerrit. A changeset can have many versions, called *patchsets*, for example if the first version received some code review comments that led to updating the code and submitting a new version for re-review.

[5] `https://git-scm.com/`

Voting Gerrit is very flexible in how you can set up your workflows. While most code review software by default only allows the reviewer to accept or reject the changed code, Gerrit has a voting system instead. The administrator can create multiple labels to vote for and configure a range of numeric values for voting. By default the labels 'Code Review' and 'Verified' are available, signifying code review results and testing/verification results, respectively. Rules can be set up to allow merging the changed code only when certain voting scores are reached. In the setup used in this study, reviewers can make the decision to vote -2, -1, ± 0, +1, or +2 on the 'Code Review' label, and each reviewer's vote accumulates to the total score on the code change. The code change must reach a total code review score of +2 or higher before it is possible to merge it into the main branch. This means that a vote of -2 will effectively block the code change from being merged, -1 will strongly discourage it, ± 0 will be a neutral vote, +1 means that you approve but you want someone more to take a look, and +2 signifies approval and ready to merge. Automated tests, linters, static analyzers, etc., can also vote on the code change but usually on labels like 'Verified', 'Formatted', etc., so their votes will not be confused with the scores from human reviewers.

Commenting Reviewers can decide to leave code review comments on individual lines of code as well as for the change as a whole, independently of how they choose to vote. Comments can be questions that need clarification, suggestions for improvements, pointing out potential issues, requests for fixes, or code for alternative solutions that the author can accept with just a click. When created, comments are marked as unresolved and must manually be marked as resolved by code change author or the reviewer who wrote the comment before the code change can be merged into the main branch.

2.4 The Recognition-Primed Decision Model

Research on rational choice and decision-making has shown that in practice human rationality is quite far from living up to the standards of being absolute or globally rational (i.e., the ability to pick the objectively best choice). Instead, it should be considered bounded or local and dependent on the problem framing made by the decision maker and the currently available knowledge [26, 91–93].

However, even then, the decision-making process does not necessarily aim at the best possible solution. Rather, it has been shown that the decision maker applies a satisficing approach, that is, looks for the first solution available deemed "good enough" [26], and that heuristics and biases play an important part in the decision-making process [94].

Klein defines the ground-breaking and influential recognition-primed decision model (RPD model) [36] from research on fireground and military commanders, who need to make critical decisions often and quickly. The RPD model differs from the more traditional rational choice strategy model [91], in that it does not

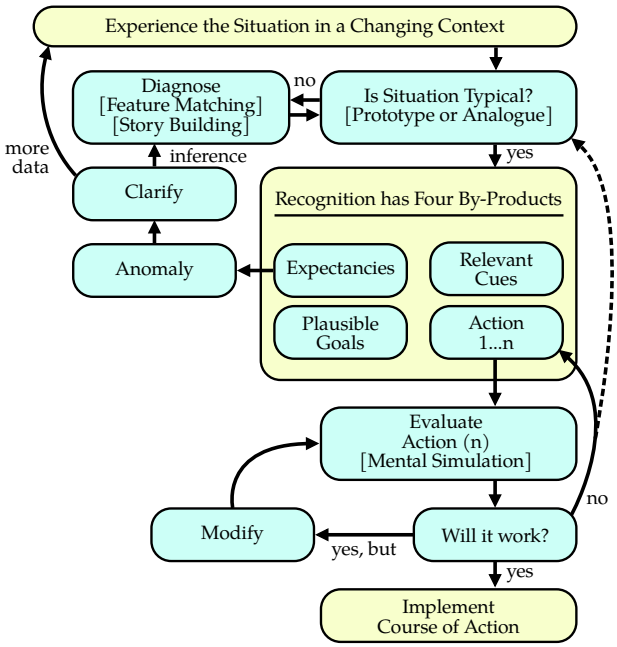

Figure 4: Recognition-primed decision (RPD) model from Klein [36]. **Takeaway:** In contrast with previous models of rational decision-making, Klein models decision-making as a process that starts from experiencing the situation and recognizing similarities and differences to previous situations. From this recognition springs possible actions that are tested, first with mental simulation and then in practice.

list all available actions and their pros and cons. Instead, the RPD model describes how decision makers use their experience to, often subconsciously, identify analogous situations and take the first action that, by experience and mental simulation, seems likely to succeed.

The RPD model (see Figure 4) is an iterative model that begins with experiencing a situation and evaluating if the situation is typical. If it is deemed typical in some aspect, i.e. recognized, this elicits expectancies, relevant cues, plausible goals, and typical actions. From here, there are two iterative flows possible. First, check if the perceived reality matches the expectations and if it does not go back, collect more data, and modify the story building until the expectations match reality. Second, evaluate possible actions by mental simulation and modify or discard the action until the first action likely to work is found. Then, this action is carried out, with the decision maker mentally prepared for some of the possible consequences. If the results differ from the mental predictions, the situation is re-evaluated and the decision maker carries out a different action they feel is likely to

work under the new circumstances. The process repeats until the desired outcome is achieved, or there are no more actions likely to work.

3 Methodology

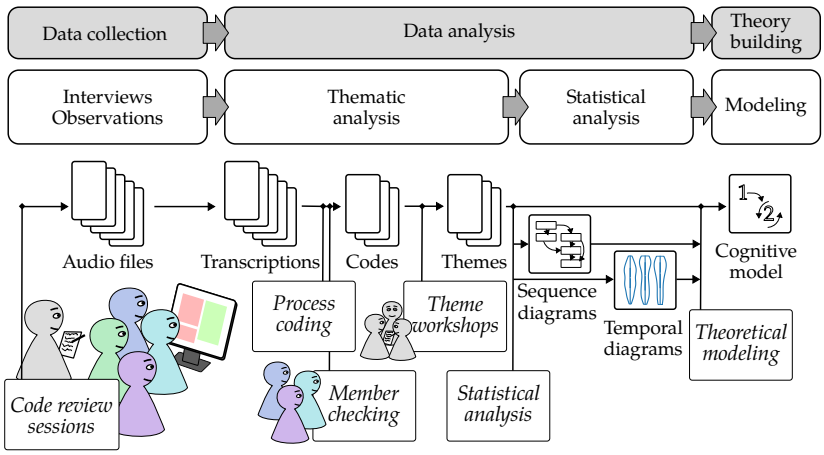

Figure 5: Process and method overview. **Takeaway:** The theoretical output, a cognitive model of code review, is built upon data collection from real code review sessions followed by thematic and statistic analysis.

In Figure 5, we present an overview of the process and methods used in this study. The study design is based on constructivist epistemology applied to ethnographic methodology, as described by Williamson [25], with the purpose of co-constructing useful and applicable models and theories together with the study participants. As described by Sharp et al. [24], ethnography applied to software engineering is well suited to explore not only what practitioners do but also *why* they do it. Sharp et al. also find that the results of ethnographic studies can deepen knowledge on social and human aspects of software engineering, inform improvements to software engineering tools, lead to process development, and point out directions for future research. All of which are goals for the contributions of this study.

Specifically, to understand the questions asked during code reviews ($\mathbf{RQ_2}$, $\mathbf{RQ_3}$) and the corresponding cognitive processes ($\mathbf{RQ_1}$), we designed an ethnographic think-aloud study combined with interviews [35]. We worked with software developers in the industry and captured the participants' normal way of reviewing as closely as possible. Our goal was to help developers feel comfortable and view the study as an exploration, and not an evaluation, of their code review techniques, habits, and skills. To achieve this, the first author worked from the

same office as the participants for several weeks to get to know the participants and to be available whenever someone needed to do a code review.

The first author sat right next to the software developers while they conducted real code reviews on their usual workstations, in their own workplace, and in their team context. Everyone was fully informed about the research, and we asked the participants to treat the first author as a newly hired developer and openly explain their ways of working. In this role, we could observe code reviews and the questions asked by developers and document their thought processes and strategies to choose code reviews, ask and answer questions, write review comments, and conclude the code reviews.

The first author has worked as a software developer, project manager, team lead, etc., in similar companies for over 15 years and has extensive first-hand experience with code reviews. This background contributed to creating good rapport between the researcher and the participants. Encouraging participants to be more detailed, vocal, and open in describing their code reviews, backgrounds, and processes.

3.1 Study Context

To achieve depth in our interviews and code review sessions, we presented the study idea to a multinational software company where we already had an active industry-academia collaboration agreement. The second author contacted the company to ask for a meeting in which we could present the study, and after the company accepted the invitation, the first author gave a presentation about the background of the study, its goals, and its methodology.

In agreement with the company, we decided to focus the study on their tools department. The tools department has around 30 developers with different levels of experience divided into 8 teams and working in several programming languages. This department was chosen because it would give a broad view of different practices and levels of experience and because it works almost exclusively with open-source software, allowing us to conduct the study and report the findings more openly.

Code Review Process

All teams use Gerrit as their code review platform (Section 2.3), and code reviews are mandatory for all code changes. Depending on the size of the team, one or two other team members must approve every code change before it can be merged. The majority of the repositories have automatic linting, code formatting, unit tests, and continuous integration (CI), eliminating the most basic code review comments and allowing reviewers to focus on higher-level issues. The process requires all team members to review code at least once a week and to try and check their code review inbox daily so that code changes do not get stuck for too long waiting for review.

3.2 Participants

Table 1: Role, experience, and weekly time spent on code review for the participants in the study.

Participant	Role	Code review exp.	Role exp.	Weekly code reviews
P01	Developer	3 years	3 years	1 hour
P02	SW Architect	14 years	9 years	2 hours
P03	Developer	14 years	9 years	2 hours
P04	Developer	1 year	1 year	0.75 hours
P05	Developer	6 years	1 year	5 hours
P06	Developer	2 years	2 years	5-10 hours
P07	Team Lead	5 years	5 years	2-3 hours
P08	Team Lead	16 years	7 years	6 hours
P09	Developer	11 years	11 years	4-8 hours
P10	Developer	17 years	13 years	5 hours
Average		8 years	6 years	4 hours
Std.dev.		6 years	4 years	2 hours

During the duration of the study, 10 software developers, from the tools department mentioned above, participated in interviews and code review sessions. Their role, experience in the role, experience with code reviews, and average time spent on code reviews weekly are found in Table 1. No participants left the study during or after field work. The participants are all of Swedish nationality and have at least a Bachelor degree. The teams involved each have 2-7 developers and work according to the agile software development methodology. In accordance with the team process, all participants did code reviews at least every week with many of the participants reviewing code every workday.

3.3 Data Collection

To test our study design and the interview protocol, we conducted a pilot interview and code review session at a small local software company. The pilot session went smoothly and gave us no reason to change the study setup. Data from the pilot study were kept for reference and comparison, but excluded from data analysis and results.

For each participant, we collected informed consent for the participation in the research study and interviewed them about their background, experience with code review, and role in the team. Whenever a participant had a pending code review to carry out, the first author sat next to them, observing their work and asking them to think aloud about what they were doing during the code review sessions. We asked questions or noted their actions aloud to record them in the sound file and to encourage the participants to explain and reflect on what they were doing and why. The code review sessions and the interviews were recorded using a dictaphone.

3.4 Data Analysis

To analyze the material in this study, we used the principles of Williamson's constructivist ethnographic research [25] and thematic analysis following the general process described by Braun and Clarke [95, 96]:

1. *Familiarizing yourself with your data:* We transcribed and annotated the recorded material and performed member-checking to validate the data.
2. *Generating initial codes:* We coded the transcriptions using process coding.
3. *Searching for themes:* Two authors organized the process codes into themes independently.
4. *Reviewing themes:* All authors reviewed and analyzed the themes until interpretative convergence.
5. *Defining and naming themes:* We refined the theme naming by studying excerpts from transcriptions.
6. *Producing the report:* We analyzed themes using statistical and sequential analysis.

Data Transcription

The first author manually transcribed the sound files from all interviews and code review sessions into Markdown-formatted text. The text files were annotated with information about the participant, the time and date, and the beginning and end of each code review session. The interviews were conducted in Swedish mixed with many software engineering terms and anglicisms. Care had to be taken when transcribing the material to preserve the meaning and intent faithfully. Due to data privacy agreements with the company and for language reasons, we did not use automated transcription software.

Participant Feedback & Member-Checking

To verify emerging themes from the data and establish an approach for coding the finished transcripts, we invited all participants to a member-checking focus group meeting [97]. The meeting was planned and facilitated by the first author and participants P01, P02, P05, P06, P09, and P10 attended. The meeting was recorded with a tabletop dictaphone to ensure a clear recording of everyone's voice.

During the meeting, we presented examples from the transcriptions for each emerging theme. The participants then discussed whether they recognized the situations in the quotes and what their experiences were like in similar situations. There were also general discussions on the processes and challenges in code review. The first author took notes that were reviewed during the meeting by the

participants. These notes and the audio recording of the meeting were combined into meeting minutes that described the points to be taken into account during the continued analysis of the transcribed data.

Initial Coding

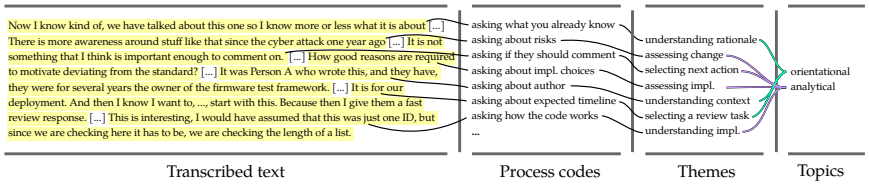

Figure 6: Example of process coding showing excerpts of transcribed text, a small selection of process codes, and mappings onto themes and topics. **Takeaway:** The transcribed text is abstracted into process codes, process codes are abstracted into themes, and finally themes are abstracted into topics.

Coding, in qualitative methods, is the practice of assigning a label or code to each sentence or segment of a text [35]. The purpose is to increase the abstraction level of the text to facilitate the data analysis and comparison of different parts of the material. Different ways of coding will highlight different aspects of the original data. Coding is often done repeatedly, first by abstracting text into codes, then systemizing the codes into themes, and then grouping the themes into topics (see Figure 6). Since the study aims to understand the developers' cognitive process during code review, we chose process coding both for the initial coding of the transcribed data and for constructing the themes. The first author did the initial coding, with authors two and three reviewing the coding and suggesting changes.

Process coding is a methodology where the first word in every code must start with a verb in the gerund form (ending with *-ing* in English) [98]. This form of coding is designed to capture the action and intention behind each coded segment [35], making it especially suited to uncover underlying questions and processes. Its name comes from how it results in a timeline of actions, a process.

Identifying Themes

The first and second authors independently grouped the process codes into themes to get two contrasting starting points. Based on these two sets of themes, all three authors discussed and worked through differences in workshop meetings. In these meetings, process codes, suggested themes, and excerpts from the transcripts were used. We drew connections between the sets of themes, regrouped process codes, and discussed until we reached interpretive convergence [98]. Finally, using the transcripts, we refined the naming of the themes to capture and communicate the intent of the segments they covered.

Statistical Analysis

When codes and themes were established, we wrote a Python program to analyze the data using temporal and sequential analysis. The sequential analysis determined the transition rate from each theme of questions to another. We gathered the transition rates into a transition table and constructed a sequence diagram of the code review process using the top 3 most common incoming and outgoing transitions with a rate higher than 0, see Figure 8.

For temporal analysis, we positioned each theme in relation to the start and endpoints of each code review. Sometimes, code reviews were paused or interrupted due to meetings, the end of the work day, or other more urgent code reviews. In the analysis program, we detected these interruptions and gathered the codes and themes for each code review in a linear flow. We normalized the duration of each review, which ranged from a couple of minutes to almost an hour, to a scale from 0 at the beginning of the review to 1 at the end. We analyzed the distribution of each theme, as well as the process codes contained, over the duration of the code reviews.

3.5 Ethical Considerations

Contributing to the research study would take significant time and effort for the participants, and neither they nor the company could be reimbursed or compensated for this. Participation was done with informed consent and on an explicitly voluntary basis. For participants, the upside would be learning something new about how they perform code reviews by explaining it to someone else, a change of pace in their workday, and the feeling of contributing to research.

During sessions and interviews, statements might come up that could be negatively interpreted by colleagues or the employer. To protect the participants, we pseudonymized all interviews and keep audio recordings and transcriptions confidential except for excerpts used to exemplify codes and themes. To allow other researchers to verify and replicate our study, it would be ideal to publish all collected data, but as discussed above, this is not possible for privacy and confidentiality reasons. The interview protocols, the data analysis program and the process-coded data set are available in the replication package (Section 8).

4 Results

The study includes 10 participants (Table 1); 7 out of 10 described their role as ‘Developer’, 2 as ‘Team Lead’, and 1 as ‘Software Architect’. The minimum work experience among the participants, both in their role and in code reviews, was 1 year. The most experienced participant had worked with code reviews for 17 years and in their current role for 13 years. The average was 8 years of code review experience and 6 years of experience in their current role.

Table 2: Overview of code review sessions from which we collected data. Note that a coded segment roughly corresponds to a paragraph in the transcription.

Participant	# Segments	Duration (min.)	Outcome
P01	27	26	vote ±0 with comments
P01	7	4	vote +2
P01	17	20	vote ±0 with comments
P01	4	2	vote +2
P01	2	1	vote +2
P02	98	48	vote +1 with comments
P02	7	9	vote +1 with comments
P02	32	6	vote +1
P02	166	75	vote -1 with comments
P03	141	49	vote -1 with comments
P03	15	5	vote +2
P03	7	2	vote +2 with comments
P04	45	42	vote +1 with comments
P05	50	38	vote +1 with comments
P05	15	21	vote +1
P05	15	6	vote +2
P05	9	6	vote +2
P05	49	32	vote +1 with comments
P05	15	37	vote +1 with comments
P06	70	49	vote ±0 with comments
P07	75	41	vote ±0 with comments
P08	20	6	vote +2 with comments
P08	34	9	vote +2
P08	37	17	vote +1 with comments
P09	14	9	vote +1 with comments
P09	25	17	vote +2 with comments
P09	13	7	vote -1 with comments
P10	16	4	vote +1 with comments
P10	16	7	vote +1
P10	43	26	vote -1 with comments
P10	3	1	vote +1
P10	61	21	vote ±0 with comments
P10	4	1	vote +2
P10	7	4	vote +1
Total	1159	648	
Average	34	19	
Std.dev.	38	18	

We gathered data from a total of 34 code reviews (Table 2). The mean code review duration was 19 minutes with a standard deviation of 18 minutes. The shortest review took 1 minute, while the longest one took 75 minutes. During all

the recorded code reviews, we process-coded a total of 1159 segments, meaning an average of 116 coded segments per participant and an average of 34 segments per review.

The recorded sessions show a clear majority of positive votes (Table 1); 26 of 34 code reviews receive a vote of at least +1. Often positive votes are given even when the reviewer wrote code review comments that they wanted to be addressed, either by updating the code or explaining the current implementation. Many participants say that the teams have a culture of trust and in general vote for merging the code and trust the author to address comments in a good way without needing re-review.

> *"To force them to fix this little issue and then them having to wait for me to get back and approve feels very silly. So I leave a comment, set +2. [...] So if, because I trust them to fix it, I don't have to come back and look at it [again]."* —P06, post code review interview

> *"You can trust that people will fix it in a satisfactory way. I don't need to look at it again. It is just a waste of time. Especially if you have several people, it becomes, like, it adds lead time. So then you can vote +1 or +2 if you anyway think that 'oh, I trust that this person will do, do something good regarding my comment'. And then we have configured it so [...] you cannot, you cannot submit the change when you have unresolved comments."* —P02

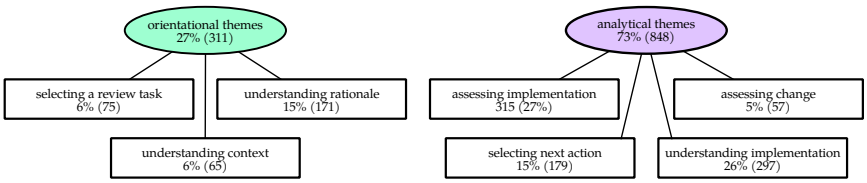

Figure 7: Topics, their contained themes, and the number of occurrences in the data set. **Takeaway:** Over 70% of the questions fall under the analytical themes indicating a majority of the code review effort is spent here.

In our thematic analysis (see Section 3.4) we constructed 157 process-codes to abstract explicit or implicit question underlying the 1159 segments. During the process-coding, we reached data saturation after coding around 3/4 of the transcribed material and after this very rarely encountered new process codes. We grouped the process-codes into 7 themes encoding commonality in underlying intentions. The detailed mapping from process codes to themes can be found in the replication package (Section 8). The three most common themes are *assessing implementation*, *understanding implementation*, and *selecting next action*, which were found 315, 297, and 179 times, respectively, in the transcriptions.

In turn, the themes are grouped into two topics: *orientational themes*, and *analytical themes*. Figure 7 shows the topics, the enclosed themes, and the number of occurrences in the process-coded segments. The orientational themes revolve around how to make sense of the context, framing, and rationale of the review, while the analytical themes are about understanding the changed code, evaluating it, planning the next action, and making a decision about the change as a whole. The analytical themes are the most common and make up 73% of the process-coded segments, with the orientational themes covering the remaining 27%.

4.1 Orientational Themes

In the transcribed data, we found many different kinds of questions asked to explore the context, framing, and rationale of the code review; orienting the code review task in relation to author, repository, expected timeline, programming language, rationale for the code change, available time for the reviewer, etc. This topic spans three themes: *selecting a review task*, *understanding context*, and *understanding rationale*.

Theme: Selecting a Review Task

Questions about how to pick which code change to review. This involves asking about the amount of time you have available as a reviewer, the urgency of different code changes, social factors, competence, and interests. Many reviews begin from the list showing all code changes that are waiting for code review, where the reviewer picks one based on available time, code change size, and urgency.

> *"I get an overview. And here I can somehow, now it is only 3 reviews, but for the case where it would have been, I don't know, say 10 reviews. And somewhere, I have, you have 8 hours per day. Then I have to prioritize somehow what, what it is I will look at. [...] I try to somewhere check a bit what it is that... yes, but what is kind of the most important? Everybody wants to get their code out, but what, what is the prio-order? If I, like, know Person C is working on something that is, that they have been doing for a long time and they really want to get it out. And it is the last thing in the stack, then I will rather take that than for example this. Then I know the other one is not, is like, it is not as urgent."* —P05

> *"And then if there are several things [to code review] you can maybe make time for a couple of the small ones but have to leave the big ones, or you take one big [code review] and leave the small ones."* —P07

A more experienced participant with a software architect role also factored in if their expertise was required, or if someone else could review the code instead.

> *"So the choice of what I will look at is kind of the combination of, like, urgency and if I, if it is like, so to speak, are they waiting for me or are there others that can do it?"* —P02

One participant kept it simple and usually picked the code change that had been waiting for the longest time.

> *"I usually take the oldest."* —P01

Theme: Understanding Context

Questions to orient the change and the review task in regards to who the author of the change is, what programming language it is written in, which repository it is, how long ago the change was posted, the expected timeline for deploying the change, previous reviews, related work, etc. In one review, the reviewer asks about who the author is and their background and concludes that the changed code will probably have tests in place:

> *"It was Person A who wrote this, and they have, they were for several years the owner of the firmware test framework. [. . .] So that's good, you don't have to nag them to write tests at least."* —P02

During another session, a reviewer asks about repository, author, and author's recent work.

> *"I check repo and person. Because then I know a little, I know for this repo it is mostly Person B working in it right now. And then I can, like, infer that, ok, I know what Person B is up to and that gives me some context still. . . "* —P05

One reviewer checks for previous code review comments.

> *"So that, also the others have had some, a number of comments already."* —P09

For a code change with previous reviews and multiple versions (patchsets in Gerrit terminology), one participant asks which version they have seen before and sets that version as the base version for comparisons.

> *"I would probably do the same thing here. . . no, right, this is a re-review so then I think I will compare with patchset 1 that is the, the one I reviewed last time."* —P02

Theme: Understanding Rationale

Understanding *why* a change has been made and what goals it is trying to achieve. In one example, the reviewer directly asks about the rationale and how it compares to their expectations.

> *"... otherwise it would have been interesting to know 'what are we trying to solve here, really?'. And it seems like maybe the scope has become a bit bigger, now they have done a bunch of other stuff"* —P03

For another participant, it is the first thing they ask about when starting a code review.

> *"So then you will, first of all, check if I understand this. Because that is often a thing for me with his stuff. I mean, do I know what he wants to do here?"* —P04

Many participants investigate the rationale by reading the original issue in the issue tracker (JIRA in the case of this company).

> *"Right, let's see what it says... [reading quietly, from JIRA] Yes, so then I see that this is also just a part towards making us more automated."* —P10

> *"If we have an issue then it is, then I often think it is good to go in here and check because you can get some background. A bit more. People are different in how much you want to write in your commit-message and stuff."* —P04

> *"OK, now we go to the JIRA-issue."* —P01

4.2 Analytical Themes

Many questions in the transcripts involve analytical themes aiming towards solving the code review task, i.e., finding defects, writing comments, voting, etc. These are questions that seek to understand and evaluate the implementation details of the changed code, plan the next action to take in the code review, and also to evaluate the change as a whole. This topic spans four themes: *assessing change*, *assessing implementation*, *selecting next action*, and *understanding implementation*.

Theme: Assessing Change

In this theme, there are questions about whether the change as a whole, regardless of the specifics of the implementation, meets the reviewer's expectations. The reviewer questions security, performance, interoperability, rationale, compatibility with future development plans, etc. One reviewer asks about the impact of the changed code:

> *"Because this is that kind of change. Here, it is the total opposite of the last one, this is something that affects a lot of people. It is a critical piece (of code)."* —P10

One example involves questioning what would happen if the change was deployed as-is.

> *"In practice we could have rolled out this change today. It is just that we probably would have gotten support tickets asking 'how does this actually work?' "* —P07

A common example is general reasoning about the risks involved with the change as a whole, regardless of the specific implementation choices.

> *"The thing is, what are the risks with this change? Either it doesn't work for unknown reasons or it could, it would maybe create, maybe create a huge amount of tickets in JIRA. It is maybe not great to overload JIRA either. But we would have noticed it pretty quickly..."* —P04

Theme: Assessing Implementation

Asking questions about whether the implementation follows the code conventions for the project, is readable, is correct, has bugs, meets the rationale, etc. An example is a session where the reviewer dislikes the implementation choices in the tests.

> *"I'm not very fond of what he has done here, in that he uses his own struct-type for the tests."* —P02

Another reviewer questions the safety of the import statements used.

> *"So now we have imported something from the backend, from a backend project, so to speak. It is maybe not always safe to do that from the frontend without, what you call like, there is something called isomorphic javascript. [..] It is doubtful if you can do what is done here."* —P03

Asking about or commenting on error handling and system messages is also common.

> *"To just say 'failed to run command' is not a very good error message in my opinion."* —P02

Questions verifying that the code looks reasonable and follows expectations is also frequently found.

> *"Here we check, check the format, check if things look reasonable. 'New host replace old machine', ok. And this looks relatively reasonable."* —P05

Theme: Selecting Next Action

Questions about what the next step in a review is. For example, on whether to read related documentation, run the code locally, read the code again, talk to the author, reference the issue tracker, and more. One strategy encountered in the code review sessions is to look for the entry point and read the code in the order it is called during execution.

> *"Then I try to identify, if I look at the file list, to find, like, the top of the call stack. So you do not start by going deep down into, like, a leaf function."* —P02

Asking if they should write a comment or not is common and often involves reasoning about the implementation.

> *"Do we want a database connection directly from the service layer? It might be that we have that in other places but that I have just forgotten. In that case, like, it is like that. But we will write a comment on it."* —P06

Choosing how to vote involves different strategies. One participant checks the overview over all code review comments to inform their decision and to remember questions they had about the code change.

> *"And now, sometimes I have not really decided how I will, what I will do [when voting]. But then you can get an overview here over what, where you can see, but is that... ? [looking at a preview of all code review comments] Mostly, mostly a lot of small... and then it was, what was the thing again... ? Something I did not like a lot that I was going to look at later, what was it now again... ?"* —P02

Reviewers often take into account the size of the code change and they want other reviewers to look at it when reasoning about how to vote. Either deciding they want at least one more approval:

> *"Mmm, the change is so small that you could set +2 here. Just to say that is it OK. But in this case I still think that I would like to set +1. Because I would like, when you set +1, you are saying 'looks good to me, but someone else must approve'."* —P05

... or that their own review is sufficient:

> *"This is such a small thing so then I feel like this, are two [pairs of] eyes required for this? Except mine? No, here I somehow trust Person D [commit author] and myself so then I think that, yes, we might as well set +2."* —P05

. . . or when they are the second reviewer and give the final approval:

> *"Yes, yes, and here they have voted code review +1, so then I could set +2 actually. Because there is no reason [not to] since I think it looks reasonable."* —P09

Theme: Understanding Implementation

This theme includes questions about, i.e., the execution flow through the code, call signatures, variable declarations, comparing the code before and after the change, etc. One participant traces the execution of the code and decides to check it out in their IDE.

> *"Then he does a 'findOne'. Then I think I want to look at it in my IDE here. Because I wonder what is going one. Let's see, let's see. . . "* —P06

There are also general questions about how the code works.

> *"I'm just trying to understand what, what the change does"* —P01

> *"No, I don't know what the hell it does."* —P02

Another reviewer traces the execution to finally understand how it fits together.

> *"Umm, this became a bit strange because here you have, here we get a context from the. . . or, yes, really from the event-pipeline as a whole that calls this method to handle an activity-started event and there we get a context that can contain a timeout or a cancellation. So we need to. . . and then I get this, but the transaction as a whole has, does not get any context. But we do send the context in. . . Ah, right! When we look up, when we fetch a build!"* —P02

In other cases, something that looks like a serious mistake makes the reviewer ask why the code does not crash.

> *"So this seems weird. It should actually, it should crash there with a key violation in the database, I think, because you are not meant to be able to register the same activity on the same build attempt more than once."* —P02

4.3 Transitions Between Themes

In Figure 8, we visualize the results of the sequential analysis (Section 3.4) as a sequence diagram. The graph shows the themes and the transition rate of the questions being asked shifting from one theme to another. In addition to the themes,

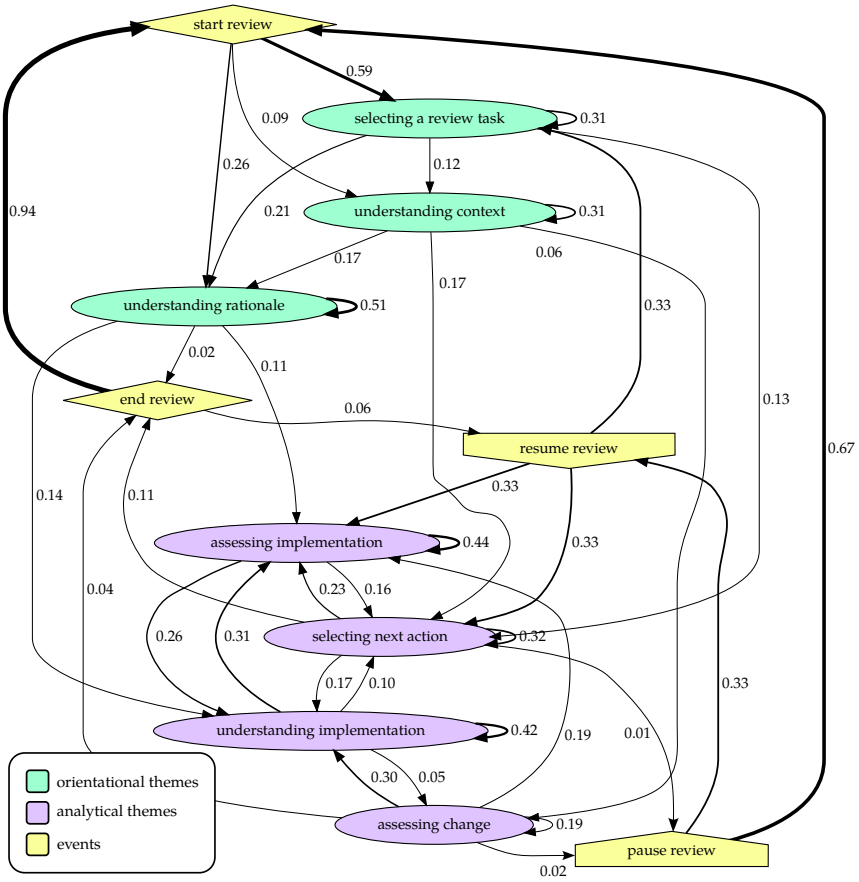

Figure 8: Transitions between themes during code review. **Takeaway:** After starting the review, there is a linear flow through the three orientational themes followed by an iterative loop through the four analytical themes.

the events ‘start review’, ‘pause review’, ‘resume review’, and ‘end review’ are included as reference points for the process. The 3 most frequent outgoing and incoming edges are plotted. A full list of transition rates can be found in Figure 9. The transition rates are normalized to 1 over the outgoing edges, while the sum over the incoming edges can be higher or lower. For example, for less frequent themes such as *end review*, which occurs at most once for every review, the sum of the incoming transition rates is much lower than 1.

In the diagram, we can see that, right after starting a code review session, the most common action (60% of cases) is to select which code change to review. When the review task is selected and its scope and expectations are known, the

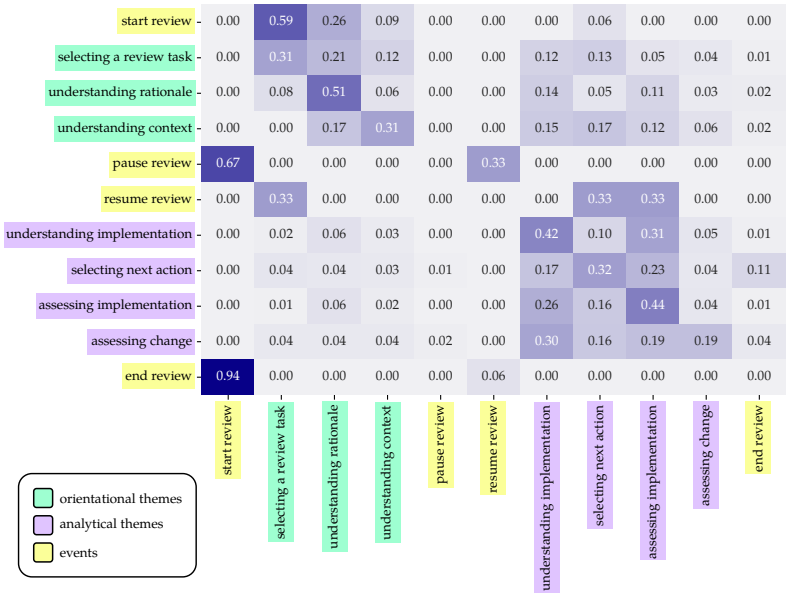

Figure 9: Transition probabilities between themes during code review. **Takeaway:** Three denser clusters appear, one in the top left quadrant between the orientational themes, one in the bottom right quadrant with the analytical themes and one less dense in the top right quadrant showing the transition from orientational to analytical themes.

questions first shift to understanding the context and then to understanding the rationale for the change. If the reviewer has already decided what to review before starting the session, they usually go straight to trying to understand the rationale, maybe because in these cases they already know the context.

These first three themes are all more orientational in nature, asking about, for example, expectations, author, related work, and rationale. Also note that while the reviewer often iterates on every theme, they never move backward once they have started asking questions on the next theme. We believe that this is because there is a causal link between these questions. You need to have selected what to review before it is meaningful or possible to learn the context and rationale of what you are reviewing. Likewise, if you after understanding context and rationale go back and pick another code change to review, it means that you are ending the current review and starting a new review session. You can see in the diagram that this actually happened in our dataset, although very few times, as indicated by the arc going from *understanding rationale* straight to *end review*.

After that, the reviewer moves on to the remaining four themes; *assessing implementation*, *selecting next action*, *understanding implementation*, and *assessing*

change. These themes are more analytical compared to the initial three themes, and revolve around understanding, assessment, and planning. In the sequence diagram, they form a series of connected loops, and we can see that the reviewer moves iteratively between them.

Notably, the fact that the *selecting next action* questions are so central in the iterative loop highlights that code review is not a linear and straightforward activity where the reviewer just reads the diffs of the changed files from beginning to end. Instead, the reviewer constantly plans what to do next and will often revisit the same files and lines many times during a review.

Interrupted Reviews

If a review is interrupted for some reason, in two thirds of the cases it is resumed by going directly to *assessing implementation* or *selecting next action* and then into the iterative loop described above. In one third of the cases, we see that even if the reviewer has read some of the code before they were interrupted, they start over from the beginning and read all of the code again to gain a full picture of the code change and be able to reach a decision on how to vote. Also, even if the reviewer goes straight into assessing implementation, they will often start over and read from the first code diff in the change.

> *"I have, I prefer to try to review the whole change in one [sitting]. Because if I get half-way and then get interrupted and have to go do something else, yes, I have a tendency to get lost a little bit. You have to go through everything, it is faster the second time but you still have to do it."* —Participant 7

4.4 Distribution of Themes during Code Review

In Figure 10 we show the distribution of themes over the normalized timeline of a code review where 0 and 1 represent the start and the end of the review session, respectively (see Section 3.4). The themes are sorted ascending by the mean timestamp.

Supporting the patterns seen in Section 4.3, we see that the themes of the two topics have similar characteristics within each topic but separate characteristics between topics. Sorting by mean timestamp separates the themes into two groups aligned with the topics. The orientational themes *selecting a review task*, *understanding rationale*, and *understanding context* occur from the beginning to about the midpoint of the review with a mean timestamp of $0.2 - 0.3$. The analytical themes *understanding implementation*, *selecting next action*, *assessing implementation*, and *assessing change* have a timestamp distribution that is centered around the middle and extends to the end of the code review with a mean timestamp of $0.5 - 0.6$.

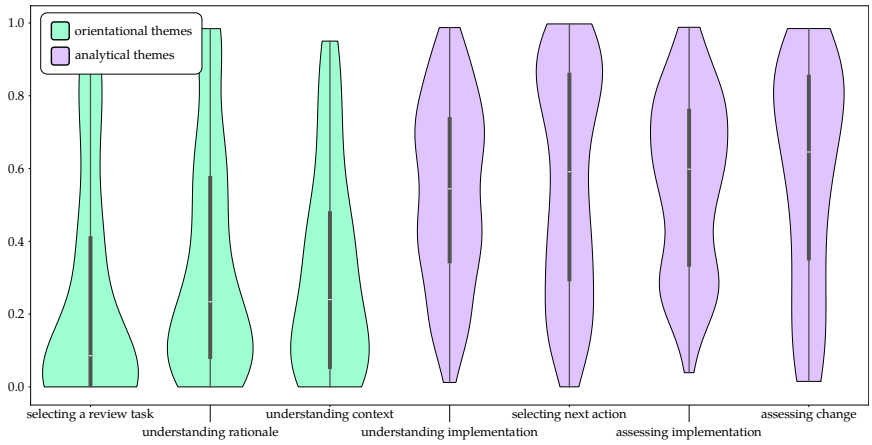

Figure 10: Distribution of themes relative to the beginning (0) and end (1) of the code review. **Takeaway:** The orientational themes and the analytical themes form two groups respectively. Each group having similar timestamp distributions internally, but distinct from the other group.

5 Theory

Here, we present a theoretical model of code review as a decision-making process, first identifying observed phases and then mapping our observations to the RPD model described by Klein [36] (Section 2.4).

5.1 The Two Phases of Code Review

When thematic, temporal, and sequential analyses of the questions asked during code review are combined, two distinct phases emerge. A linear phase at the beginning of the review that we call *orientation phase*, and an iterative phase from the middle to the end of the review that we call *analytical phase*.

The Orientation Phase

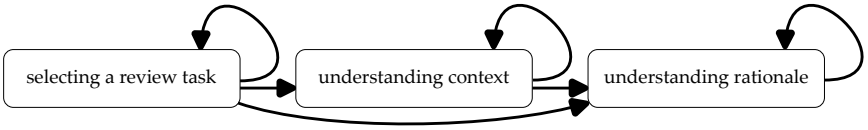

Figure 11: Orientation phase of code review. **Takeaway:** In this phase, reviewers explore each theme with several questions before moving forward to the next. All transitions move forward and the questions are related to rationale, context, and expectations.

In the orientation phase of code review (see Figure 11) the reviewer asks about and establishes context, scope, rationale, and expectations. The phase consists of the three themes *selecting a review task*, *understanding context*, and *understanding rationale*. The themes fall under the functional topic of orientational themes, share early mean timestamps in the temporal analysis, and mostly forward transitions in the sequential analysis.

The reviewer begins by asking one or more questions under the *selecting a review task* theme, for example, asking about how big the code change is, which code base it is in, continuous integration (CI) status, priority, and urgency. Next, they continue with the theme *understanding context* asking questions about the code change author, repository, programming language, and type of change (bug fix, feature, refactoring, etc.). Finally, the reviewers ask questions on the theme *understanding rationale*. For example, about the commit message content, issue description, feature requirements, and recent team conversations. Sometimes, the context is well known, from the team stand-up meeting or other recent discussions, and the reviewer skips directly to *understanding rationale*. Each theme can be repeated multiple times with several questions exploring the theme, but once the questions transition to the next theme, the reviewer rarely goes back again.

The Analytical Phase

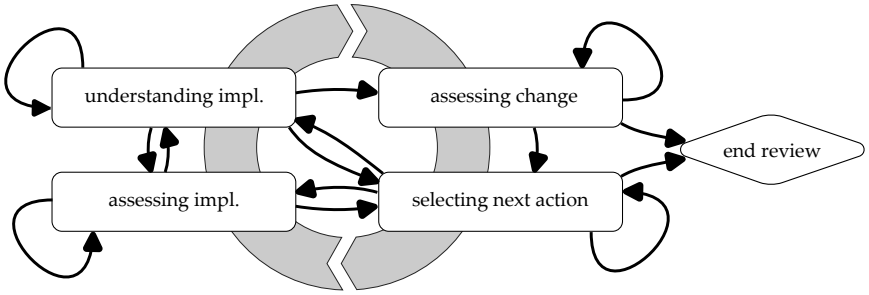

Figure 12: Analytical phase of code review. **Takeaway:** The analytical phase iterates over understanding implementation, assessing implementation, assessing the change, and planning the next step. Both each theme and also the full cycle are repeated several times before the review is done.

In the analytical phase of code review (see Figure 12), the reviewer iteratively builds an understanding of the changed code, evaluates the implementation and the change as a whole based on this new understanding, and plans what action to take next. The phase consists of the four themes *understanding implementation*, *assessing implementation*, *assessing change*, and *selecting next action*. The themes all fall under the functional topic of analytical themes, share late mean timestamps in the temporal analysis, have cyclic transitions between them in the sequential analysis, and have very few transitions going back to the themes in the orientation phase.

The reviewer could enter the phase on any of the themes. After entering the analytical phase, the most common scenario is that the reviewer iterates through all the themes several times before reaching a *selecting next action* question in which the reviewer decides to end the review. The final question is often about a summary of the comments the reviewer has written or about how to vote regarding the code change.

5.2 Introducing a Model of Code Review as Decision-Making

The expected outcome of code review is to reach a decision on voting for or against merging the changed code [2]. On the way there, the participants in the study take several smaller decisions around writing review comments, phrasing of comments, reading external sources, reading changed source code, checking out the code locally, choosing next steps, etc. We can also see that while the cognitive model of code review comprehension [20] is confirmed by our data and can be mapped to the theme *understanding implementation* as a whole and to the mental model-

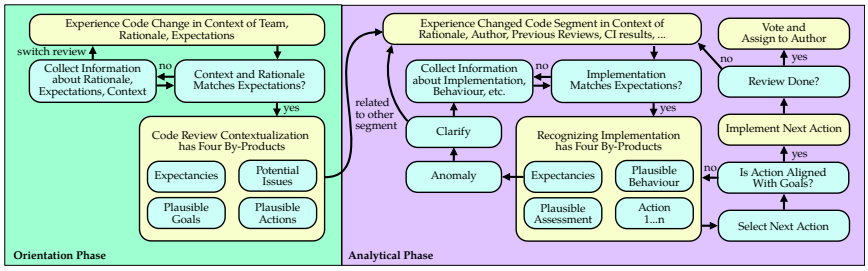

Figure 13: The Code Review as Decision-Making (**CRDM**) cognitive model. **Takeaway:** Code review can be modeled as two linked decision-making processes. The first is preparatory and establish context, plausible goals, and plausible actions. The second takes decisions on what actions to take during the review, and how to vote regarding the integration of the code change. It bases these decisions in understanding the change through both implementation details and as part of a larger system.

ing parts of the theme *selecting next action*, it is just a part of a larger cognitive process.

This leads us to re-frame code review as a kind of decision-making process. Specifically, the orientation and analytical phases in code review can be mapped onto the RPD model introduced by Klein [36] (Section 2.4). In Figure 13, we illustrate the Code Review as Decision-Making (**CRDM**) cognitive model mapping the orientation and analytical phases of code review onto one preparatory and one complete RPD model, respectively. Connecting the themes, topics, and phases from our data analysis to the RPD model allows us to construct a cognitive model for the code review task as a whole. A model in which the code reviewers' experience helps them spot potential issues, find confusing or problematic sections of code, formulate effective review comments, look for more information, and vote appropriately for accepting or rejecting the changed code based on their mental simulation of software and team behavior.

In the orientation phase, the reviewer asks questions about the rationale and context of the changed code in order to build a story around the change and what it tries to do and why. This creates expectancies around the changed code, potential issues to look out for, plausible goals of the review, and plausible actions. If the context or the rationale does not seem to match expectations, the reviewer collects more information, for example, by reading in the issue tracker or asking over the team chat. Since the reviewer usually does not vote or comment during this phase, it is modeled with the preparatory first half of the RPD model.

After the orientation phase, the reviewer enters the analytical phase that can be modeled with a complete cycle through the RPD model. The reviewer experiences the changed code in view of the context established in the orientation phase,

reacts to how well it matches their expectations, evaluates if it is in line with the rationale as they have understood it, and chooses their next action. If the changed code triggers a cue for potential issues or does not seem to match the rationale, the reviewer takes action such as writing a review comment, reading external sources, asking for clarification, or voting for rejecting the changed code. When the rationale and the code are coherent, they might perform mental simulation of both what could happen if the code was deployed and of how their colleagues will respond to review comments and voting choices. Finally, after as many iterations as needed, the reviewer implements their decisions on how to vote, which review comments to write, what to communicate in other channels, and what to do next.

Together, the two phases in the CRDM model form a cohesive cognitive process that covers the entire code review task. The orientation phase equips the reviewer with context and rationale for the code change, while the analytical phase enables iterative evaluation, understanding, and action planning. In total, it models code review as a dynamic, experience-driven decision process that includes but goes beyond comprehension.

6 Discussion

Our results and theoretical modeling show that code review has much in common with decision-making processes, specifically the RPD model by Klein [36]. Mapping the cognitive phases found during code review onto the RPD model gives us a novel cognitive model ($\mathbf{RQ_1}$) of Code Review as Decision-Making; the CRDM model (Section 5.2). This model can explain and predict some observations from empirical studies of code review. For example, since recognition-primed decision-making requires extensive experience of analogous situations, it can explain how even experienced programmers can take up to a year to become effective at code review in a new workplace [37]. It is not just about code comprehension (a well-developed skill for an experienced programmer), but also about building up a mental index of patterns in a new code base and organization. Furthermore, misalignments between current code review tools and developer needs [6] could be extrapolated from current code review tools, which center the code diff-view and thereby the *understanding implementation* part of the code review. This design leaves it up to the developer to plan the review, gather decision-making information, explore the context, and understand the rationale, thus showing that current tools are not aligned with all the needs and goals of users.

The CRDM cognitive model also allows insight into how code review compares to more well-studied processes such as decision-making, reflection, and learning. Being able to relate research results from other disciplines through the theoretical model can inspire future research, give insight into the challenges of code review, and indicate directions for improvements in code review tools (Section 6.4).

6.1 Insights from Thematic Analysis

Looking at the levels of topics and themes ($\mathbf{RQ_2}$) gives us insight into the relative frequency of questions asked during code review. On topic level, the balance is roughly 70/30 between questions with analytical themes and orientational themes; emphasizing the analytical nature of code review while still pointing out context and orientation as an imprescriptible part.

At the theme level, questions seeking to understand the implementation, rationale, and context account for slightly less than half of the total questions asked. These themes can be related to processes of comprehension, as described by both Letovsky [21] and Gonçalves et al. [20]. Also, on this level, these results show that while comprehension is a very important part of code review, there is more involved to complete the task. Planning, decision-making, and assessing account for the other half of the questions asked. Notably, a significant part of the analytical work during code review consists of *selecting next action*, which is concerned with planning and decision-making and in a way is a metacognitive theme.

6.2 Classification of Questions During the Code Review

When analyzing topics and themes as a sequential process during code review ($\mathbf{RQ_3}$), we see interesting patterns in how participants move between themes in their questions. Letovsky [21] classifies questions asked during a cognitive process into five groups; *Why*, *How*, *What*, *Whether*, *Discrepancy*. From the thematic and sequential analysis, we see that the first orientation phase is dominated by the *Why* and *What* questions exploring the rationale and context of the code review. In the analytical phase, this emphasis shifts to *How*, *Whether*, and *Discrepancy* questions that explore how the code works, whether it could have desired or undesired behaviors, and assessing it for discrepancies with expectations and code guidelines. That the questions are different in the two phases we conjecture, also when applying Letovsky's classification system in addition to our own thematic coding, supports the division into orientation and analytical phases.

6.3 Comparison to Previous Models

Existing models for code review focus on and describe the organizational process [2, 3, 99], but to our knowledge only one model of the cognitive process has been presented; the code review comprehension model of Gonçalves et al. [20], which deliberately only models a part of the entire code review process; comprehending code changes. For this part of the review, our results support their findings. Especially, the reviewer iteratively uses information sources, a knowledge base, and their own mental model to drive comprehension during the code review. The CRDM model then expands the scope significantly by describing orientation processes before starting to read the changed code and the recurring decision-making needs throughout the review.

6.4 Future Work

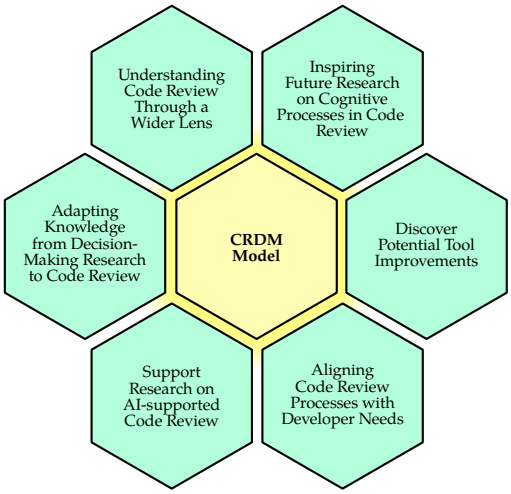

Figure 14: Illustration of potential applications of the CRDM cognitive model. Each surrounding hexagon represents a potential application of the model, such as adapting knowledge from decision-making research, improving code review tools, and supporting future research. The center hexagon represents the CRDM cognitive model itself.

As illustrated in Figure 14, the CRDM model and the results leading up to the model can have several applications for future research and development.

Adapting Knowledge from Decision-Making Research Given the similarities between code review and decision-making processes, code review tools could learn a lot from decision support systems (DSS). Liu et al. [100] presents a meta-study of over 100 papers on DSS and shows the advantages of Integrated Decision Support Systems (IDSS) where the decision support is integrated into existing systems and processes. This would be an interesting way forward for code review tools. For example, knowledge graph-based or agentic IDSS could be integrated into existing tools to support developers throughout their workflow and make code review more effective and efficient.

Discover Potential Tool Improvements Today's code review tools come from a legacy of code inspection meetings and have, on a feature level, not changed much since the first ever software for code review, ICICLE, was introduced in 1990 [7]. By analyzing the needs of developers during the different steps in the CRDM model, the features of popular code review tools [101, 102], and recent re-

search on misalignment between code review tools and user goals [6] we conclude that today's code review tools have significant room for improvement.

Reevaluating code review tools from the lens the CRDM model, or incorporating ideas and features from DSS software discussed above could give benefits to development teams around the world. In particular, the tools' focus on the code diff view and inline review comments centers its support mostly around the two themes *understanding implementation* and *assessing implementation*. Overall, the *orientation phase* as a whole receives less support, and users often choose to leave the code review tool and look for information in issue tracker, team chat, external documentation, etc., to better understand the context and rationale.

Support Research on AI-supported Code Review As discussed in Section 1, more and more research and development in academia and industry is directed towards supporting or replacing code review with AI models or agents. Many researchers are calling out the importance of preserving the human perspective and emphasizing AI technology that supports rather than replaces current software engineering practices [83, 103]. A cognitive model can be a useful foundation in finding areas where current tools give insufficient support and where AI models and agents could augment the capabilities of the human engineers. Training the CRDM model into an agentic workflow could also provide pre-emptive support and information and guide developers through the code review task.

Understanding Code Review Through a Wider Lens In much of the literature today, code review is seen as a process of comprehension and defect finding. While that is certainly part of the truth, we would like to challenge and expand this view and treat code review as more closely related to decision-making. By applying this wider lens, we believe that there are new insights and perspectives to be found that might, for example, change the way we teach code review to new developers and the way we design tools.

Aligning Code Review Processes with Developer Needs As well as misalignments between the developer needs and the tools used, Söderberg et al. [6] also finds misalignments between the process itself and responsibilites and outcomes of code review. Perhaps there are ways to adapt the accepted code review processes to better match the needs and cognitive process of the developers involved in the process.

Inspiring Future Research on Cognitive Processes To our knowledge, few cognitive models of code review have been published today. We hope by presenting methodology and results from building the CRDM model that we can inspire future research in studying software engineering processes from a cognitive lens. Certainly, there are parts of general code review, such as which actions

reviewers take to resolve their questions, that warrant further study. It could also be illuminating to study code review in different kinds of companies, open source projects, development methodologies, and team sizes.

7 Threats to Validity

LeCompte and Goetz present a comprehensive investigation of both internal and external threats to be considered for ethnographic research [104]. They separate threats to reliability, which concerns to what degree the study is reproducible, and to validity, describing the accuracy of the conclusions in relation to empirical reality. In the following subsections, we use the threats identified by LeCompte and Goetz to analyze the threats in this study.

7.1 External Reliability

The external reliability of an ethnographic study is affected by *position of the researcher in the study*, *choices of the informants*, *social situations and conditions*, *analytic constructs and premises*, and *methods of data collection and analysis* [104]. The researcher conducting the field work has extensive experience in software engineering and code reviews, contributing to good rapport between the researcher and the participants. This experience facilitated open sharing of thoughts and experiences as well as understanding of specific terms, jargon, and practices (Section 3). In the research team, we also have one more member with long industry and code review experience, as well as one researcher without industry experience but with extensive knowledge in human factors, cognition, and interaction studies. This contributed to building an understanding and theory that is valid from both an insider and an outsider perspective.

Regarding informant choices, there is inherent bias in the fact that people who volunteer to participate in ethnographic research studies are introspective and insightful about their own thinking and actions to a greater degree than the average in most groups [104]. In our case, choosing to work with the outward-facing software tools department that collaborates with external open source communities and with all other internal departments gives a bias towards people who are communicative, outgoing, and used to describing their work to outsiders. This increased the depth and detail we could achieve in our collected data. We think the risk that less extroverted participants would follow a fundamentally different code review process is small.

All participants and teams in this study have strong similarities (Section 3.2). Developers with, for example, a different cultural and educational background working in large teams using waterfall methodology might have a different approach to code reviews. We have tried to mitigate this risk by actively choosing participants ranging from inexperienced to very experienced and with different roles in their teams.

The social context and setting for the code reviews were at the developers' regular workplace, on their own computer, monitor, and desk to create conditions for realistic results. Our analytical constructs are based on our constructivist epistemological view, as well as the process codes used to encode the transcripts (Section 3). The process codes follow accepted coding practices and are published in the replication package, see Section 8 Finally, data collection and analysis were performed using common practices in qualitative studies, such as audio recording, transcription, process coding, and statistical analysis [35, 98]. Transcribing the recorded interviews carry the risk of subtly shifting the meaning, since spoken and written language is interpreted slightly differently and the transcriptions lack prosody and tone of voice. We mitigated this by adding notes in the transcription where the meaning would otherwise be ambiguous.

7.2 Internal Reliability

The internal reliability of an ethnographic study is affected by *low-inference descriptors*, *multiple researchers*, *participant researchers*, *peer examination*, and *mechanically recorded data* [104]. To achieve low-inference descriptors, the source materials for analysis in the study were verbatim transcriptions of the recorded code review sessions with little to no inference. Multiple researchers were involved in the interpretation of the data. Two researchers independently did the thematic coding of the material. All three researchers discussed process coding, thematic coding, and topics until agreement [98]. The process coding and its interpretation were further verified with the participants through a member-checking workshop (Section 3.4). For peer examination, we note that Gonçalves et al. [20] describes a process, albeit within a more narrow scope, that confirms our model in the parts where they overlap. Finally, data were recorded using digital dictaphones for voice clarity, and the original recordings are archived at the university.

7.3 External Validity

The external validity of an ethnographic study is affected by *selection effects*, *setting effects*, *history effects*, and *construct effects* [104]. To address selection effects, we selected participants with different age, experience, and roles, but working for the same company and team. They follow the same or very similar code review guidelines. This contributes to results from different participants being comparable. Since the observations were carried out in the same office with members of the same development team, the settings were very similar. We think that the social effects of the setting, group, and researcher are comparable between the interviews. Regarding history effects and construct effects, all participants have a comparable cultural and educational background, which contributes to the validity of comparing their data. Further, the same process-coding and thematic coding

was used for all recordings and participants, again contributing to comparability between participants.

7.4 Internal Validity

The internal validity of an ethnographic study is affected by *history and maturation*, *observer effects*, *selection and regression*, *mortality*, and *spurious conclusions* [104]. In relation to history and maturation, code review was a well established and mature practice in the development team that participated in the study, and their guidelines remained the same throughout the field work. We may have had observer effects in that participants might have put in more effort than usual into the code reviews. That is, participants may have spent more time in code review and may have been more meticulous with comments and approvals, to be perceived as competent by the researcher and their peers. During field work, we tried to mitigate observer effects by being neutral and curious about any approach the participants took. While we did select as diverse participants as possible from the members of the participating development teams, they do have similar cultural and educational backgrounds. It cannot be ruled out that participants with different culture, education, role, employer, role, etc. would also have different strategies during code review. No participants left the study (or the team) during the field work. Finally, conclusions and theories were built bottom-up from the results of thematic and statistical analysis to avoid drawing spurious conclusions from our field observations.

8 Conclusions

We studied questions asked during code review using an ethnographic think-aloud study combined with interviews (Section 3). The study included 10 participants and a total of 34 code reviews. We performed thematic analysis of the transcribed interviews followed by temporal and sequential analysis. Through this analysis, we discovered patterns in the kinds of questions that reviewers asked during code reviews; when the questions were asked, and how the questions connected to each other.

From our thematic analysis we identified 2 topics containing a total of 7 themes (Figure 7). Temporal and sequential analysis indicates that code review can be modeled by two phases (Section 5), a linear *orientation phase* (Figure 11) followed by an iterative *analytical phase* (Figure 12). During the orientation phase, the reviewer seeks information about the expectations, rationale, and context of the code change within and outside the code review tool. Once those factors are understood, the reviewer enters the assessment phase. Here, they iterate by seeking to understand the implementation, assessing the change, assessing the implementation, and planning their next action until the code review is finished.

The similarities in the dynamics during these two phases with decision-making processes in general, and in particular the RPD model defined by Klein [36], lead us to propose the Code Review as Decision-Making (CRDM) cognitive model (Section 5.2). In this model, we reframe code review as a decision-making process, providing new perspectives on the practice, its effects, avenues for future research, and ideas on how tools can evolve to support code review in a better way.

Notes

Data availability Anonymized data on the level of process coding that support the findings of this study are openly available in the replication package below. Due to sensitivity reasons, full recordings and transcriptions are not openly available and are available from the corresponding author upon reasonable request.

Code availability The program code for data analysis is available in the replication package at DOI: 10.5281/zenodo.15758266

BIBLIOGRAPHY

[1] L. Allen, A. O'Connell, and V. Kiermer, "How can we ensure visibility and diversity in research contributions? How the Contributor Role Taxonomy (CRediT) is helping the shift from authorship to contributorship," *Learned Publishing*, vol. 32, no. 1, pp. 71–74, 2019. doi:10.1002/leap.1210

[2] C. Sadowski, E. Söderberg, L. Church, M. Sipko, and A. Bacchelli, "Modern Code Review: A Case Study at Google," in *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, ser. ICSE-SEIP '18. New York, NY, USA: Association for Computing Machinery, 2018, pp. 181–190, event-place: Gothenburg, Sweden. doi:10.1145/3183519.3183525

[3] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in *2013 35th International Conference on Software Engineering (ICSE)*, 2013, pp. 712–721. doi:10.1109/ICSE.2013.6606617

[4] C. D. Egelman *et al.*, "Predicting developers' negative feelings about code review," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. Seoul South Korea: ACM, 2020, pp. 174–185. doi:10.1145/3377811.3380414

[5] M. Chouchen *et al.*, "Anti-patterns in modern code review: Symptoms and prevalence," in *Proceedings of the International Conference on Software Analysis, Evolution and Reengineering*, ser. SANER, 2021, pp. 531–535. doi:10.1109/SANER50967.2021.00060

[6] E. Söderberg, L. Church, J. Börstler, D. Niehorster, and C. Rydenfält, "Understanding the Experience of Code Review: Misalignments, Attention, and Units of Analysis," in *Proceedings of the International Conference on Evaluation and Assessment in Software Engineering 2022*, ser. EASE '22. Association for Computing Machinery, 2022, pp. 170–179, event-place: Gothenburg, Sweden. doi:10.1145/3530019.3530037

[7] L. Brothers, V. Sembugamoorthy, and M. Muller, "ICICLE: Groupware for Code Inspection," in *Proceedings of the 1990 ACM Conference on Computer-Supported Cooperative Work*, ser. CSCW '90. New York, NY, USA: Association for Computing Machinery, 1990, pp. 169–181, event-place: Los Angeles, California, USA. doi:10.1145/99332.99353

[8] DORA, "Accelerate State of DevOps 2023," DORA, Tech. Rep., 2023. Available: https://dora.dev/research/2023/dora-report/

[9] C. Bird *et al.*, "Taking Flight with Copilot: Early Insights and Opportunities of AI-Powered Pair-Programming Tools," *Queue*, vol. 20, no. 6, pp. 35–57, 2023, place: New York, NY, USA Publisher: Association for Computing Machinery. doi:10.1145/3582083

[10] D. Sobania, M. Briesch, C. Hanna, and J. Petke, "An Analysis of the Automatic Bug Fixing Performance of ChatGPT," 2023. doi:10.48550/ARXIV.2301.08653

[11] J. Lu, L. Yu, X. Li, L. Yang, and C. 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)*, 2023, pp. 647–658. doi:10.1109/ISSRE59848.2023.00026

[12] Y. Yu *et al.*, "Fine-Tuning Large Language Models to Improve Accuracy and Comprehensibility of Automated Code Review," *ACM Trans. Softw. Eng. Methodol.*, vol. 34, no. 1, 2024. doi:10.1145/3695993

[13] X. Tang *et al.*, "CodeAgent: Autonomous Communicative Agents for Code Review," in *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 2024, pp. 11 279–11 313. doi:10.18653/v1/2024.emnlp-main.632

[14] Google Research. (2023) Large sequence models for software development activities. Available: https://research.google/blog/large-sequence-models-for-software-development-activities/

[15] D. C. Engelbart, "Augmenting Human Intellect: A Conceptual Framework," Stanford Research Institute, Menlo Park, California, Tech. Rep. AFOSR-3223, 1962.

[16] M. E. Fagan, "Design and code inspections to reduce errors in program development," *IBM Systems Journal*, vol. 15, no. 3, pp. 182–211, 1976. doi:10.1147/sj.153.0182

[17] W. Florac, A. Carleton, and J. Barnard, "Statistical process control: analyzing space shuttle onboard software process," *IEEE Software*, vol. 17, no. 4, pp. 97–106, Aug. 2000, (Accessed 2025-08-17). doi:10.1109/52.854075

[18] N. B. Ruparelia, "Software development lifecycle models," *ACM SIGSOFT Software Engineering Notes*, vol. 35, no. 3, pp. 8–13, 2010. doi:10.1145/1764810.1764814

[19] N. Jørgensen, "Putting it all in the trunk: incremental software development in the FreeBSD open source project," *Information Systems Journal*, vol. 11, no. 4, pp. 321–336, 2001. doi:10.1046/j.1365-2575.2001.00113.x

[20] P. W. Gonçalves, P. Rani, M.-A. Storey, D. Spinellis, and A. Bacchelli, "Code Review Comprehension: Reviewing Strategies Seen Through Code Comprehension Theories," in *2025 IEEE/ACM 33rd International Conference on Program Comprehension (ICPC)*, 2025, pp. 589–601. doi:10.1109/ICPC66645.2025.00068

[21] S. Letovsky, "Cognitive Processes in Program Comprehension," *Journal of Systems and Software*, vol. 7, no. 4, pp. 325–339, 1987. doi:10.1016/0164-1212(87)90032-X

[22] Y. Rogers, *Interaction Design - Beyond Human-Computer Interaction*, 6th ed. New York: John Wiley & Sons Inc, 2023.

[23] Design Council, "Double Diamond framework for innovation," https://www.designcouncil.org.uk/our-resources/framework-for-innovation/, 2023, [Online; accessed 25-October-2023].

[24] H. Sharp, Y. Dittrich, and C. R. B. de Souza, "The Role of Ethnographic Studies in Empirical Software Engineering," *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 786–804, 2016. doi:10.1109/TSE.2016.2519887

[25] K. Williamson, "Research in Constructivist Frameworks Using Ethnographic Techniques," *Library Trends*, vol. 55, no. 1, pp. 83–101, 2006. doi:10.1353/lib.2006.0054

[26] H. A. Simon, *The sciences of the artificial*, 3rd ed. Cambridge, MA: MIT Press, 1996.

[27] J. E. V. Aken, "Management Research Based on the Paradigm of the Design Sciences: The Quest for Field-Tested and Grounded Technological

Rules," *Journal of Management Studies*, vol. 41, no. 2, pp. 219–246, 2004. doi:10.1111/j.1467-6486.2004.00430.x

[28] A. R. Hevner, S. T. March, J. Park, and S. Ram, "Design Science in Information Systems Research," *MIS Quarterly*, vol. 28, no. 1, pp. 75–105, 2004. doi:10.2307/25148625

[29] E. Engström, M.-A. Storey, P. Runeson, M. Höst, and M. T. Baldassarre, "How software engineering research aligns with design science: a review," *Empirical Software Engineering*, vol. 25, no. 4, pp. 2630–2660, 2020. doi:10.1007/s10664-020-09818-7

[30] P. Runeson, E. Engström, and M.-A. Storey, "The Design Science Paradigm as a Frame for Empirical Software Engineering," in *Contemporary Empirical Methods in Software Engineering*, M. Felderer and G. H. Travassos, Eds. Cham: Springer International Publishing, 2020, pp. 127–147. doi:10.1007/978-3-030-32489-6_5

[31] G. Bjerknes, P. Ehn, and M. Kyng, Eds., *Computers and democracy: a Scandinavian challenge*, repr ed. Aldershot, Hampshire: Avebury, 1989.

[32] S. Costanza-Chock, *Design Justice: Community-Led Practices to Build the Worlds We Need*. The MIT Press, 2020.

[33] R. M. Emerson, R. I. Fretz, and L. L. Shaw, *Writing ethnographic fieldnotes*, ser. Chicago guides to writing, editing, and publishing. University of Chicago Press, 2011.

[34] M. Hammersley and P. Atkinson, *Ethnography : principles in practice*, 4th ed. London :: Routledge, London, 2019.

[35] K. Charmaz, *Constructing grounded theory*, 2nd ed., ser. Introducing qualitative methods. London, UK: Sage, 2014, OCLC: ocn878133162.

[36] G. Klein, *Sources of Power : How People Make Decisions*. Cambridge, Mass., USA: The MIT Press, 1998.

[37] A. Bosu, M. Greiler, and C. Bird, "Characteristics of Useful Code Reviews: An Empirical Study at Microsoft," in *Proceedings of the 12th Working Conference on Mining Software Repositories*, ser. MSR '15. IEEE Press, 2015, pp. 146–156, event-place: Florence, Italy. doi:10.5555/2820518.2820538

[38] A. Alami, M. Leavitt Cohn, and A. Wasowski, "Why Does Code Review Work for Open Source Software Communities?" in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 1073–1083. doi:10.1109/ICSE.2019.00111

[39] M. V. Mäntylä and C. Lassenius, "What Types of Defects Are Really Discovered in Code Reviews?" *IEEE Transactions on Software Engineering*, vol. 35, no. 3, pp. 430–448, 2009. doi:10.1109/TSE.2008.71

[40] A. Bosu and J. 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*. IEEE, 2013, pp. 133–142. doi:10.1109/ESEM.2013.23

[41] Statistia, "Number of software developers worldwide in 2018 to 2024," 2023, accessed 2025-01-15. Available: https://www.statista.com/statistics/627312/worldwide-developer-population/

[42] F. Bagirov, P. Derakhshanfar, A. Kalina, E. Kartysheva, and V. 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*. ACM, 2023, pp. 188–191. doi:10.1145/3593434.3593462

[43] T. Baum, "Cognitive-support code review tools : improved efficiency of change-based code review by guiding and assisting reviewers," Ph.D. dissertation, Universität Hannover, 2019, publisher: Hannover : Institutionelles Repositorium der Universität Hannover.

[44] T. Baum and K. Schneider, "On the Need for a New Generation of Code Review Tools," in *Product-Focused Software Process Improvement*, P. Abrahamsson, A. Jedlitschka, A. Nguyen Duc, M. Felderer, S. Amasaki, and T. Mikkonen, Eds. Springer International Publishing, 2016, vol. 10027, pp. 301–308, series Title: Lecture Notes in Computer Science. Available: http://link.springer.com/10.1007/978-3-319-49094-6_19. doi:10.1007/978-3-319-49094-6_19

[45] E. Söderberg, L. Church, J. Börstler, D. C. Niehorster, and C. Rydenfält, "What's Bothering Developers in Code Review?" in *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, ser. ICSE-SEIP '22. Association for Computing Machinery, 2022, pp. 341–342, event-place: Pittsburgh, Pennsylvania. doi:10.1145/3510457.3513083

[46] O. Kononenko, O. Baysal, and M. W. Godfrey, "Code Review Quality: How Developers See It," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. Association for Computing Machinery, 2016, pp. 1028–1038, event-place: Austin, Texas. doi:10.1145/2884781.2884840

[47] L. MacLeod, M. Greiler, M.-A. Storey, C. Bird, and J. Czerwonka, "Code Reviewing in the Trenches: Challenges and Best Practices," *IEEE Software*, vol. 35, no. 4, pp. 34–42, 2018. doi:10.1109/MS.2017.265100500

[48] P. Liamputtong, *Focus Group Methodology: Principle and Practice*. SAGE Publications Ltd,, 2011.

[49] M. Steen, "Co-Design as a Process of Joint Inquiry and Imagination," *Design Issues*, vol. 29, no. 2, pp. 16–28, 2013. doi:10.1162/DESI_a_00207

[50] J. Johnson and A. Henderson, "Conceptual models: begin by designing what to design," *interactions*, vol. 9, no. 1, pp. 25–32, 2002.

[51] Y.-n. Chang, Y.-k. Lim, and E. Stolterman, "Personas: from theory to practices," in *Proceedings of the 5th Nordic conference on Human-computer interaction: building bridges*, 2008, pp. 439–442.

[52] Q. U. Ain, W. H. Butt, M. W. Anwar, F. Azam, and B. Maqbool, "A Systematic Review on Code Clone Detection," *IEEE Access*, vol. 7, pp. 86 121–86 144, 2019, conference Name: IEEE Access. doi:10.1109/ACCESS.2019.2918202

[53] T. Baum, K. Schneider, and A. Bacchelli, "Associating working memory capacity and code change ordering with code review performance," *Empirical Software Engineering*, vol. 24, no. 4, pp. 1762–1798, 2019. doi:10.1007/s10664-018-9676-8

[54] N. Dell, V. Vaidyanathan, I. Medhi, E. Cutrell, and W. Thies, "" Yours is better!" participant response bias in HCI," in *Proceedings of the sigchi conference on human factors in computing systems*, 2012, pp. 1321–1330.

[55] L. Pascarella, D. Spadini, F. Palomba, M. Bruntink, and A. Bacchelli, "Information Needs in Contemporary Code Review," *Proceedings of the ACM on Human-Computer Interaction*, vol. 2, no. CSCW, pp. 1–27, 2018. doi:10.1145/3274404

[56] P. W. Gonçalves, E. Fregnan, T. Baum, K. Schneider, and A. Bacchelli, "Do explicit review strategies improve code review performance? Towards understanding the role of cognitive load," *Empirical Software Engineering*, vol. 27, no. 4, p. 99, 2022. Available: https://doi.org/10.1007/s10664-022-10123-8. doi:10.1007/s10664-022-10123-8

[57] P. Antonenko, F. Paas, R. Grabner, and T. Van Gog, "Using Electroencephalography to Measure Cognitive Load," *Educational Psychology Review*, vol. 22, no. 4, pp. 425–438, 2010. doi:10.1007/s10648-010-9130-y

[58] F. A. Fishburn, M. E. Norr, A. V. Medvedev, and C. J. Vaidya, "Sensitivity of fNIRS to cognitive state and load," *Frontiers in Human Neuroscience*, vol. 8, 2014. doi:10.3389/fnhum.2014.00076

[59] A. F. Ackerman, P. J. Fowler, and R. G. Ebenau, "Software inspections and the industrial production of software," in *Proc. of a symposium on Software validation: inspection-testing-verification-alternatives*, 1984, pp. 13–40.

[60] N. Davila and I. Nunes, "A systematic literature review and taxonomy of modern code review," *Journal of Systems and Software*, vol. 177, p. 110951, 2021. doi:10.1016/j.jss.2021.110951

[61] L. Gullstrand Heander, E. Söderberg, and C. Rydenfält, "Design of Flexible Code Block Comparisons to Improve Code Review of Refactored Code," in *Companion Proceedings of the 8th International Conference on the Art, Science, and Engineering of Programming*, ser. Programming '24. ACM, 2024, p. 57–67. doi:10.1145/3660829.3660842

[62] C. Sadowski, J. Van Gogh, C. Jaspan, E. Soderberg, and C. Winter, "Tricorder: Building a program analysis ecosystem," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 598–608. doi:10.1109/ICSE.2015.76

[63] A. Ljungberg, D. Åkerman, E. Söderberg, G. Lundh, J. Sten, and L. Church, "Case study on data-driven deployment of program analysis on an open tools stack," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2021, pp. 208–217. doi:10.1109/ICSE-SEIP52600.2021.00030

[64] R. G. Kula *et al.*, "Using Profiling Metrics to Categorise Peer Review Types in the Android Project," in *2012 IEEE 23rd International Symposium on Software Reliability Engineering Workshops*. IEEE, 2012, pp. 146–151.

[65] E. Doğan and E. Tüzün, "Towards a taxonomy of code review smells," *Information and Software Technology*, vol. 142, p. 106737, 2022. doi:10.1016/j.infsof.2021.106737

[66] C. Packer *et al.*, "MemGPT: Towards LLMs as Operating Systems," 2024. Available: http://arxiv.org/abs/2310.08560

[67] F. Hajari, S. Malmir, E. Mirsaeedi, and P. C. Rigby, "Factoring Expertise, Workload, and Turnover into Code Review Recommendation," 2023. doi:10.48550/arXiv.2312.17236

[68] B. Hui *et al.*, "Qwen2.5-Coder Technical Report," 2024. doi:10.48550/ARXIV.2409.12186

[69] E. Chen, R. Huang, H.-S. Chen, Y.-H. Tseng, and L.-Y. Li, *GPTutor: A ChatGPT-Powered Programming Tool for Code Explanation*. Springer Nature Switzerland, 2023, pp. 321–327.

[70] F. Hein, "The Blame Game," *IEEE Software*, vol. 15, no. 6, pp. 89–91, 1998.

[71] M. M. Rahman and C. K. Roy, "Impact of Continuous Integration on Code Reviews," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 2017, pp. 499–502.

[72] M. Unterkalmsteiner, D. Badampudi, R. Britto, and N. B. Ali, "Help Me to Understand this Commit!-A Vision for Contextualized Code Reviews," in *Proceedings of the 1st ACM/IEEE Workshop on Integrated Development Environments*, 2024, pp. 18–23. doi:10.1145/3643796.3648447

[73] Y. Almeida *et al.*, "AICodeReview: Advancing Code Quality with AI-enhanced Reviews," *SoftwareX*, vol. 26, p. 101677, 2024.

[74] L. Wang *et al.*, "Unity Is Strength: Collaborative LLM-Based Agents for Code Reviewer Recommendation," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2024, pp. 2235–2239. doi:10.1145/3691620.3695291

[75] L. Yang *et al.*, "A preliminary investigation on using multi-task learning to predict change performance in code reviews," *Empirical Software Engineering*, vol. 29, no. 6, p. 157, 2024. doi:10.1007/s10664-024-10526-9

[76] C. S. Lee and C. M. Hicks, "Understanding and effectively mitigating code review anxiety," *Empirical Software Engineering*, vol. 29, no. 6, p. 161, 2024. doi:10.1007/s10664-024-10550-9

[77] H. Dubberly and P. Pangaro, *Cybernetics and Design: Conversations for Action*. Springer International Publishing, 2019, pp. 85–99.

[78] L. Church, E. Söderberg, and A. McCabe, "Breaking down and making up-a lens for conversing with compilers," in *Psychology of Programming Interest Group Annual Workshop 2021*, 2021.

[79] S. I. Ross, F. Martinez, S. Houde, M. Muller, and J. D. Weisz, "The Programmer's Assistant: Conversational Interaction with a Large Language Model for Software Development," in *Proceedings of the 28th International Conference on Intelligent User Interfaces*. ACM, 2023, pp. 491–514.

[80] Y. Wang *et al.*, "Agents in Software Engineering: Survey, Landscape, and Vision," 2024. Available: https://arxiv.org/abs/2409.09030

[81] DORA, "Accelerate State of DevOps 2024," DORA, Tech. Rep., 2024. Available: https://dora.dev/research/2024/dora-report/

[82] A. Froemmgen *et al.*, "Resolving Code Review Comments with Machine Learning," in *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice*. ACM, 2024, pp. 204–215. doi:10.1145/3639477.3639746

[83] L. Gullstrand Heander, E. Söderberg, and C. Rydenfält, "Support, Not Automation: Towards AI-supported Code Review for Code Quality and Beyond," in *In 33rd ACM International Conference on the Foundations of Software Engineering (FSE Companion '25)*, 2025. doi:10.1145/3696630.3728505

[84] A. Krause-Glau, L. Damerau, M. Hansen, and W. Hasselbring, "Visual Integration of Static and Dynamic Software Analysis in Code Reviews via Software City Visualization," 2024, version Number: 1. doi:10.48550/ARXIV.2408.08141

[85] P. Dourish, *Where the action is: the foundations of embodied interaction*. MIT Press, Cambridge, Mass., 2001.

[86] R. Rosenberger and P.-P. Verbeek, "A field guide to postphenomenology," *Postphenomenological investigations: Essays on human-technology relations*, pp. 9–41, 2015.

[87] I. Ajzen, "The theory of planned behavior," *Organizational Behavior and Human Decision Processes*, vol. 50, no. 2, pp. 179–211, 1991. doi:10.1016/0749-5978(91)90020-T

[88] G. Kudrjavets, A. Kumar, N. Nagappan, and A. Rastogi, "Mining Code Review Data to Understand Waiting Times between Acceptance and Merging: An Empirical Analysis," in *Proceedings of the 19th International Conference on Mining Software Repositories*, ser. MSR '22. Association for Computing Machinery, 2022, pp. 579–590, event-place: Pittsburgh, Pennsylvania. doi:10.1145/3524842.3528432

[89] M. Dorner, D. Mendez, K. Wnuk, E. Zabardast, and J. Czerwonka, "The upper bound of information diffusion in code review," *Empirical Software Engineering*, vol. 30, no. 1, p. 2, 2025. doi:10.1007/s10664-024-10442-y

[90] F. Coelho, N. Tsantalis, T. Massoni, and E. L. G. Alves, "A qualitative study on refactorings induced by code review," *Empirical Software Engineering*, vol. 30, no. 1, p. 17, 2025. doi:10.1007/s10664-024-10560-7

[91] H. A. Simon, "A Behavioral Model of Rational Choice," *The Quarterly Journal of Economics*, vol. 69, no. 1, pp. 99–118, 1955. Available: http://www.jstor.org/stable/1884852

[92] A. Tversky and D. Kahneman, "The framing of decisions and the psychology of choice," *science*, vol. 211, no. 4481, pp. 453–458, 1981. doi:10.1126/science.7455683

[93] S. W. Dekker, "Reconstructing human contributions to accidents: the new view on error and performance," *Journal of safety research*, vol. 33, no. 3, pp. 371–385, 2002. doi:10.1016/S0022-4375(02)00032-4

[94] A. Tversky and D. Kahneman, "Judgment under Uncertainty: Heuristics and Biases: Biases in judgments reveal some heuristics of thinking under uncertainty." *science*, vol. 185, no. 4157, pp. 1124–1131, 1974. doi:10.1126/science.185.4157.1124

[95] V. Braun and V. Clarke, "Using thematic analysis in psychology," *Qualitative Research in Psychology*, vol. 3, no. 2, pp. 77–101, 2006. doi:10.1191/1478088706qp063oa

[96] V. Clarke and V. Braun, "Thematic Analysis," *The Journal of Positive Psychology*, vol. 12, no. 3, pp. 297–298, 2017. doi:10.1080/17439760.2016.1262613

[97] L. Birt, S. Scott, D. Cavers, C. Campbell, and F. Walter, "Member Checking: A Tool to Enhance Trustworthiness or Merely a Nod to Validation?" *Qualitative Health Research*, vol. 26, no. 13, pp. 1802–1811, 2016. doi:10.1177/1049732316654870

[98] J. Saldaña, *The coding manual for qualitative researchers*, 3rd ed. Thousand Oaks, CA: Sage Publications, 2015.

[99] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan, "An empirical study of the impact of modern code review practices on software quality," *Empirical Software Engineering*, vol. 21, no. 5, pp. 2146–2189, 2016. doi:10.1007/s10664-015-9381-9

[100] S. Liu, A. H. B. Duffy, R. I. Whitfield, and I. M. Boyle, "Integration of Decision Support Systems to Improve Decision Support Performance," *Knowledge and Information Systems*, vol. 22, no. 3, pp. 261–286, 2010. doi:10.1007/s10115-009-0192-4

[101] GitHub Inc., "GitHub Code Review," 2025, accessed 2025-02-27. Available: https://github.com/features/code-review

[102] Gerrit, "User Guide," 2025, accessed 2025-02-27. Available: https://gerrit-review.googlesource.com/Documentation/intro-user.html

[103] D. Russo *et al.*, "Generative AI in Software Engineering Must Be Human-Centered: The Copenhagen Manifesto," *Journal of Systems*

and Software, vol. 216, p. 112115, 2024, publisher: Elsevier BV. doi:10.1016/j.jss.2024.112115

[104] M. D. LeCompte and J. P. Goetz, "Problems of Reliability and Validity in Ethnographic Research," *Review of Educational Research*, vol. 52, no. 1, pp. 31–60, 1982. doi:10.3102/00346543052001031