



CHALMERS
UNIVERSITY OF TECHNOLOGY



UNIVERSITY OF GOTHENBURG



Rethinking Code Review Workflows with LLM Assistance

An Exploratory Study on LLM Integration in Practice

Master's Thesis in Computer science and engineering

Björn Borgar Magnússon
Fannar Steinn Aðalsteinsson

MASTER'S THESIS 2025

Rethinking Code Review Workflows with LLM Assistance

An Exploratory Study on LLM Integration in Practice

Björn Borgar Magnússon
Fannar Steinn Aðalsteinsson



UNIVERSITY OF
GOTHENBURG



CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2025

Rethinking Code Review Workflows with LLM Assistance: An Exploratory Study
on LLM Integration in Practice

Björn Borgar Magnússon
Fannar Steinn Aðalsteinsson

© Björn Borgar Magnússon & Fannar Steinn Aðalsteinsson, 2025.

Supervisor: Chih-Hong Cheng, Department of Computer Science and Engineering

Advisor 1: Adam Nirving Davidsson, WirelessCar AB

Advisor 2: Mislav Milicevic, WirelessCar AB

Examiner: Gregory Gay, Department of Computer Science and Engineering

Examiner-in-practice: Lucas Gren, Department of Computer Science and Engineering

Master's Thesis 2025

Department of Computer Science and Engineering

Chalmers University of Technology and University of Gothenburg

SE-412 96 Gothenburg

Telephone +46 31 772 1000

Cover: An abstract icon illustrating the integration of AI into the code review process, featuring a magnifying glass over interconnected code blocks arranged within square brackets. The design symbolizes pattern recognition, code structure, and assisted analysis.

The image was generated using DALL · E, a generative AI tool capable of producing visual content from textual prompts. The image was created with the prompt: “Review phase two of the thesis and the description of the tool we are going to create for it. Now generate a logo/icon for the tool that is relevant and connected to the study and the tool functionality. Have the background transparent.”

Typeset in L^AT_EX

Gothenburg, Sweden 2025

Björn Borgar Magnússon & Fannar Steinn Aðalsteinsson
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Code reviews are a critical yet time-consuming aspect of modern software development, increasingly challenged by growing system complexity and the demand for faster delivery. This thesis presents a study conducted at WirelessCar Sweden AB, combining an exploratory field study of current code review practices with a field experiment involving two variations of an LLM-assisted code review tool. The field study identifies key challenges in traditional code reviews, including frequent context switching, insufficient contextual information, and highlights both opportunities (e.g., automatic summarization of complex pull requests) and concerns (e.g., false positives and trust issues) in using LLMs. In the field experiment, two prototype variations were developed: one offering LLM-generated reviews upfront and the other enabling on-demand interaction. Both utilize a semantic search pipeline based on retrieval-augmented generation to assemble relevant contextual information for the review, thereby tackling the uncovered challenges. Developers evaluated both variations in real-world settings: AI-led reviews are overall more preferred, while still being conditional on the reviewers' familiarity with the code base, as well as on the severity of the pull request.

Keywords: AI in Software Engineering; Human-AI Collaboration; Software Engineering Practices; Large Language Models (LLMs); Code Review; Empirical Software Engineering; Retrieval-Augmented Generation

Acknowledgements

We would like to thank our supervisor, Chih-Hong Cheng, for his invaluable guidance throughout the course of this thesis. We are also thankful to our examiner-in-practice, Lucas Gren, for his constructive comments and feedback.

We are grateful to WirelessCar AB for the opportunity to carry out this thesis in collaboration with the company. In particular, we would like to thank our advisors, Adam Nirving Davidsson and Mislav Milicevic, for their continuous support, insights, and responsiveness, as well as Hanna Lund for her assistance and encouragement in enabling our work within the organization.

We would also like to extend our heartfelt thanks to everyone at WirelessCar who participated in our study, both those who shared their time in interviews and those who took part in the field experiment. Your engagement and openness were essential to the success of our research.

Finally, we thank our families and friends for their understanding and encouragement during this journey.

Björn Borgar Magnússon & Fannar Steinn Aðalsteinsson, Gothenburg, June 2025

Declaration on the use of Generative AI

During the thesis work, the authors utilized ChatGPT for minor grammatical corrections and to enhance clarity in phrasing. The tool did not generate substantial original content. All content, analysis, and conclusions remain exclusively those of the authors, who bear full responsibility for the publication.

Björn Borgar Magnússon & Fannar Steinn Aðalsteinsson, Gothenburg, June 2025

Contents

List of Figures	xv
List of Tables	xvii
1 Introduction	1
1.1 Problem Description	2
1.2 Purpose of the Study	3
1.3 Research Questions	3
1.4 Significance of the Study	4
1.5 Thesis Outline	4
2 Background	5
2.1 Modern Code Review Practices	5
2.2 Challenges in Modern Code Review	6
2.3 AI in Software Engineering	7
2.3.1 AI as a Collaborative Partner in Software Development	7
2.3.2 Productivity Gains and Quality Impact	8
2.3.3 Implementation Challenges	8
2.3.4 Future Potential of AI	9
2.4 Large Language Models: Foundations and Deployment	9
2.4.1 Training and Adaptation	9
2.4.2 Cloud-Based and Locally Hosted Models	9
2.5 Retrieval-Augmented Generation (RAG) Architecture	10
2.5.1 Data indexing	10
2.5.2 Retrieval	11
2.5.3 Generation	11
2.5.4 Advantages of RAG	11
2.5.5 Limitations of RAG	11
2.6 Agentic LLM Systems	11
2.6.1 Tool-Driven Reasoning	12
2.6.2 The ReAct Framework	13
3 Related Work	15
3.1 AI-Assisted Code Reviews	15
3.2 Recent Empirical Studies	16
3.3 Challenges and Limitations of AI-Driven Code Reviews	18
3.4 Summary and Gaps	19

4	Method	21
4.1	Research Design Overview	21
4.2	Literature Review Process	23
4.3	Research Setting	24
4.4	Phase 1: Field Study	24
4.4.1	Focus Areas	24
4.4.2	Data Collection	25
4.4.3	Data Analysis	27
4.5	Phase 2: Field Experiment	28
4.5.1	Focus Areas	29
4.5.2	Experiment Setup	29
4.5.3	Artifact Implementation	31
4.5.3.1	System Architecture and Technology Choices	32
4.5.3.2	Data Preparation and Indexing	34
4.5.3.3	Agentic Structure and Prompt Engineering	34
4.5.3.4	Summary	39
4.5.4	Data Collection	39
4.5.5	Data Analysis	41
5	Results	43
5.1	Results from Phase 1	43
5.1.1	Observed Code Review Process	43
5.1.2	Common Challenges in Code Reviews	46
5.1.3	Current Use of AI in Software Development	47
5.1.4	Potential AI Use Cases in Code Review	47
5.1.5	Summary	48
5.2	Results from Phase 2	49
5.2.1	Accuracy, Reliability, and Trust	50
5.2.2	Efficiency and Thoroughness	51
5.2.3	Design Expectations and Limitations	53
5.2.4	Usage Contexts and Interaction Patterns	55
5.2.5	Summary	57
6	Discussion	59
6.1	Addressing the Research Questions	59
6.1.1	RQ1	59
6.1.2	RQ2	62
6.2	Reflections on Assistant Behavior and Experimental Fidelity	66
6.3	Implications	68
6.4	Threats to Validity	69
6.4.1	Construct Validity	70
6.4.2	Internal Validity	70
6.4.3	External Validity	71
6.4.4	Reliability	72
6.5	Future Work	73
7	Conclusion	75

Bibliography	77
A Appendix	I
A.1 Phase 1 - Data	I
A.1.1 Interview Question List	I
A.1.2 Identified Themes and Codes from Thematic Analysis: Phase 1	II
A.2 Phase 2 - Data	IV
A.2.1 Post-Experiment Interview Question List	IV
A.2.2 Identified Themes and Codes from Thematic Analysis: Phase 2	V
A.3 Functional Requirements for AI assistant	VII
A.4 Example of AI Assistant Thought Process	X

List of Figures

4.1	Flowchart detailing the flow of the research	22
4.2	Screenshot of the chat interface in Mode B, reviewing a pull request from an open-source project at https://github.com/ogen-go/ogen/pull/1440	33
4.3	Agentic tool structure in Co-Reviewer mode.	36
4.4	Agentic tool structure in Interactive Assistant mode.	36
4.5	Overview of the RAG query processing pipeline.	37
5.1	Code review workflow at WirelessCar	44

List of Tables

4.1	Overview of Interview Participants	26
4.2	Example Coding of a Quote from Phase 1 Transcripts	28
4.3	Overview of Participants for Phase 1 & 2	40
4.4	Example Coding of a Quote from Phase 2 Transcripts	42
5.1	Identified themes and their descriptions from analysis of interview data from Phase 1.	44
5.2	Identified themes and their descriptions from analysis of data from Phase 2.	49

1

Introduction

Code review is an important practice in software engineering, it plays a critical role in keeping up code quality, detecting defects, and knowledge sharing within teams [1]. As modern software systems become ever more complex and the demand for greater speed in development increases, traditional code review processes face significant challenges. These include being highly resource-intensive, prone to human error, and contributing to developer burnout [1]. Addressing these challenges is essential to improving the efficiency and effectiveness of software engineering practices.

The breakthrough of Large Language Models (LLMs) in recent years has introduced a new opportunity to address these issues. LLMs, such as GPT-based models, have demonstrated remarkable capabilities in assisting with various software engineering tasks, such as code generation, bug detection, and enhancing coding practices [2]. These models leverage advanced natural language processing techniques to assist engineers in both traditional routine tasks and more complex creative tasks [2]. This could, at the same time, improve the outcome and reduce manual effort. While LLM integration in various aspects of software development shows promising results, its integration into the code review process remains a relatively unexplored area with exciting potential.

Despite their potential, there are some challenges presented when it comes to integrating LLMs into code review processes. These include the need to, for example, balance automation with human expertise, ensure contextual understanding, and address biases in training data [3]. Additionally, as LLM technology continues to evolve fast, there is a need for empirical research to establish best practices for effective and sustainable use of LLMs in code review workflows [3].

This thesis explores how LLMs can be meaningfully integrated into the human aspects of code review within real-world software development teams. The study is conducted in collaboration with WirelessCar AB, a software development company (where some but not all teams are permitted to utilize AI in development). By observing current practices within the company along with executing a field experiment, it aims to provide insights for software development teams, as well as contributing to the research on human-AI collaboration in software development. The findings are expected to benefit software development teams considering AI integration while offering researchers further insights into AI collaboration in software

engineering practices.

1.1 Problem Description

Code review is a critical practice in software engineering, essential for ensuring code quality, detecting defects, and facilitating knowledge sharing among team members. The process of code review consists of a review of a contribution made by a code author by one or more of their peers before the code is committed to a version control system [3]. However, traditional code review processes, which depend heavily on manual effort, are increasingly challenged by the growing complexity of software systems and the need for faster development cycles. These reviews are time-consuming, prone to human fatigue, and often place a heavy burden on experienced reviewers, contributing to inefficiencies and developer burnout [1]. Additionally, code reviews that are not efficient and executed poorly can result in a more negative work environment, stand in the way of further development, and provide a false sense of code quality [4]. Even with the code reviews partly automatic, with, for example, tools that can automatically fix formatting problems, the code review process still costs thousands of developer-years each year, and therefore, all minor improvements in the process can result in a significant impact [3].

LLMs have emerged as a promising tool to address these challenges. LLMs are advanced artificial intelligence systems trained on massive datasets of both text and code. That enables the models to process and generate both human-like text and code from natural language descriptions. Recent LLMs have demonstrated remarkable capabilities in various domains, including coding tasks such as code generation, bug detection, and code completion [2]. The evolution from pre-trained language models (PLMs) to LLMs, driven by larger model sizes and datasets, has significantly enhanced their performance. LLMs are therefore transforming how we interact with language and information, and they have the potential to assist in various tasks, including software engineering practices [2]. By assisting in identifying issues, providing feedback, and supporting reviewers in evaluating code, LLMs have the potential to enhance the efficiency and quality of code reviews.

Despite this promise, the integration of LLMs into the code review process presents some challenges. LLMs often struggle with understanding broader architectural context and developer intent [3]. False positives and negatives further require human oversight. Bias in training data can also result in unfair or inaccurate assessments [5]. Additionally, the rapid evolution of coding standards necessitates frequent updates to keep LLMs effective. Finally, balancing automation with human expertise is critical to preserve human judgment while leveraging LLM efficiency [3].

While LLMs are increasingly used in software engineering, there is a lack of clear guidelines on how to effectively integrate these tools into code review workflows. Without such guidance, teams risk inefficient integration, reduced trust in LLMs, or reliance on tools that fail to meet their needs.

1.2 Purpose of the Study

Today, AI technology is being introduced to increasingly more parts of software development processes and has been shown to be capable of taking on the role of a collaborator and of performing specific functions [6]. One part of the software development process that seems to have less experience with AI collaboration is the code review process. It should benefit software developers to know if it would be beneficial to introduce AI assistance to those review processes and, if so, in what ways. Therefore, the purpose of this study is to explore design considerations and developer preferences for integrating LLMs into the code review process to enhance efficiency and developer experience. The study combines observational research on current practices with experimental evaluations of LLM-assisted processes. By collecting qualitative data from developer interactions and interviews, the research seeks to understand how AI assistance is perceived and experienced during code review. The findings aim to inform efforts to integrate AI into workflows in ways that support developer needs and practices.

1.3 Research Questions

This study investigates how LLMs can be meaningfully integrated into modern code review workflows to improve developer experience and potentially support review efficiency. The research is structured around 2 core questions: one diagnostic and one exploratory. The first question (RQ1) focuses on understanding current code review practices and identifying opportunities for AI assistance, while RQ2 explores how developers perceive and interact with LLM-based tools during review tasks.

RQ1: What practices, challenges, and expectations characterize modern code review processes, and where do developers see potential for AI-based assistance?

This question aims to identify how code reviews are currently performed, what challenges developers face, and how AI can be introduced in the code review process, what tasks it can effectively support, and the optimal balance between automation and human involvement.

RQ2: How do developers perceive LLM-assisted code review tools, and what is the preferred interaction?

This question explores the qualitative aspects of AI usage, including developer trust, satisfaction, and potential barriers to adoption, by considering different interaction modes. By answering this question, insights are provided into the usability, limitations, and acceptance of AI-assisted code reviews, helping to inform design decisions and future development of such tools.

1.4 Significance of the Study

This study has the potential to refine and extend existing knowledge about how LLMs can be effectively incorporated into software engineering workflows, specifically code reviews. Substantively, it addresses the challenges and opportunities of integrating LLMs into code review processes, which is an important but resource-intensive component of the software development process. Methodologically, the study applies a two-phase approach, combining a field study and a field experiment, to explore LLM usage. While the setting is based on real-world data from a cooperating company, the experimental design remains semi-controlled, with findings that suggest promising directions rather than definitive practices.

The results of this research have practical implications for software development teams by highlighting design considerations and patterns that may inform best practices for using LLMs to improve code review processes. Applying these insights may support improvements in code review workflows by guiding the integration of AI assistance into developer practices. For researchers, the results will offer more insights into the cooperation of AI and human expertise in software engineering, possibly contributing to theoretical frameworks on AI assistance and decision-making in the field. In particular, the study contributes empirical insight into the perceived value and limitations of AI assistance in code review tasks. This also offers more knowledge to the field of human-AI collaboration. By addressing the discussed concerns, the study supports the evolution of modern software engineering practices by examining how LLM-based tools may be integrated into collaborative development workflows.

1.5 Thesis Outline

The thesis starts by introducing the research problem and highlighting the significance of AI-assisted code reviews. It defines the problem and clarifies the core objectives of the research. Chapter 2 provides essential background on modern code review practices and covers the theoretical foundation where principles behind large language models and related technologies are discussed. Chapter 3 reviews related work, showing where the current research fits within the existing research conducted on the topic, along with highlighting known gaps in AI-driven code reviews. The research methods and analysis are then described in Chapter 4 in two phases before the results are presented in Chapter 5. Finally, Chapter 6, addresses the research questions and discusses limitations and future work before the conclusions are presented in Chapter 7.

2

Background

The following chapter presents the foundational concepts and literature that support this research. It provides an overview of modern code review practices, highlights key challenges identified by recent research, and explores how AI, particularly large language models, may offer new ways to enhance the review process. This contextual grounding will serve as the basis for the methods and analysis that follow.

2.1 Modern Code Review Practices

Modern code review (MCR) is a crucial aspect of the software development lifecycle, evolving significantly from the formal, time-intensive code inspections of the past [7]. The code review process can vary a lot, and development teams tend to apply the traditional process to their known habits and rules of working. Essentially, a code review is a process of iteratively reviewing and rewriting a piece of code before it is merged into the main codebase or published to the next environment in the development pipeline [8]. Today, it is a more lightweight, asynchronous, time-efficient, tool-supported process. The focus is on ensuring code quality, knowledge sharing, and team collaboration. The shift from formal inspections to more agile and tool-assisted processes has been driven by the need for greater efficiency and flexibility in the code review process. Today popular tools such as Gerrit¹, Phabricator², ReviewBoard³, GitHub pull request⁴, Critique⁵, and CodeFlow⁶ are utilized to enable streamlined interactions between authors and reviewers [7, 8].

A traditional modern code review begins when a developer authors a code change and submits it to the review tool, typically the developer also provides a concise description of the change's purpose [8]. The author might invite specific reviewers to review the code or allow anyone with appropriate access to self-assign to the review. The designated reviewer then accesses the tool asynchronously to inspect the diffs (line-by-line code modifications) and adds inline comments where they spot

¹<https://www.gerritcodereview.com/>

²<https://secure.phabricator.com/book/phabricator/article/differential/>

³<https://www.reviewboard.org/>

⁴<https://github.com/features/code-review>

⁵Used internally at Google, not available to public.

⁶<https://www.getcodeflow.com/>

potential defects or opportunities for improvements [9]. The author can respond to the comments (if there are any) by updating the code, uploading a new code change, and iterating on the reviewer feedback [8]. Once the comments have been addressed, the reviewer must explicitly approve the change before it is integrated into the main codebase. In Google’s terminology, this is referred to as “LGTM” or “Looks Good To Me” [9]. The tools utilized also facilitate automated checks or analysis (for example, static analysis or conflict detection), but the final approval is typically dependent on human oversight [8]. Overall, this rapid and lightweight review process allows for frequent feedback on smaller code patches and treats code reviews as a group problem-solving exercise, moving beyond the strict, defect-focused method of the past [9, 8]. Recent studies also highlight the role of modern code review not only in identifying defects but also in enhancing team communication and fostering a shared understanding of the codebase. Additionally, MCR has been shown to support practices like early detection of architectural issues and continuous improvement of design patterns within agile workflows [10].

2.2 Challenges in Modern Code Review

Several challenges have been identified in modern code review practices that reduce their effectiveness. Some argue that code review activities are the longest step in integrating new code into the product in development, creating bottlenecks and increasing the effects of code review challenges [1]. Studies have shown that a considerable amount of code reviews are not considered useful by the authoring developer, raising concerns given that developers can spend 10–15% of their time on review tasks [8]. Interrupting a developer’s primary development task to perform a review also entails context switching, which can further reduce efficiency. Some also question whether every code change truly requires the same level of in-depth review to be effective [1].

Additionally, it has been noted that many code reviews fail to meet their goals regarding flagging functional defects, proposing code improvement, and suggesting alternative solutions, but are, however, considered useful when it comes to maintainability, knowledge sharing, and relationship building. Developers tend to disagree whether these latter benefits justify the time and effort required for the process [8], especially since the review serves multiple purposes, it becomes hard to pinpoint exactly where they bring the most value [1]. In some cases, this can result in quick and superficial feedback, especially when reviewers face pressure to provide rapid comments and thus focus on easily identifiable maintainability concerns rather than conducting in-depth investigations required to uncover more subtle or critical defects [7, 8].

A major challenge is also understanding the code that is being reviewed, which includes knowing the reason for the code change and having a sufficiently detailed and accurate description of the context. In many cases, this description is unclear or incomplete, making it difficult for the reviewers to grasp what prompted the change. This is highlighted when reviewers encounter new, unfamiliar files or new compo-

nents, and they note that it takes considerably more time to interpret the code's purpose, often requiring them to read additional files beyond the code changes to achieve a better context [7]. Despite the time investment, it is common that many reviews highlight lower-level maintainability concerns while they fail to uncover defects. Identifying the defects can require even more time investment and a better knowledge of the broader code context, which can be hard to achieve for reviewers, particularly when the patch size is large. This often results in quick and shallow feedback without deep, substantive comments [8].

Finally, the presentation of the feedback can also be crucial, where rude or inappropriate feedback can lower morale, reduce the value of the process, and cause conflicts, discouraging diversity and inclusion. Low reviewer agreement, missing rationale, scant discussion of non-functional requirements, and unfamiliarity with the existing codebase are further factors that degrade review quality, as does the varying frequency of collaboration between reviewers and code authors [8]. However, understanding alone is the key to a successful code review. Even when time is not a limiting factor, the absence of fundamental understanding, relevant experience, and adequate skill remains the primary obstacle in the review process. Without these core competencies, secondary challenges such as scheduling or time constraints cannot be meaningfully addressed. [7, 1]. Some even argue that the high cost, in terms of both time and manpower, may not result in a proportional benefit when it comes to preventing defects. While code reviews are standard practice today, guidelines on exactly when, where, and how to apply them remain unclear, suggesting an ongoing need to refine code review workflows [1].

2.3 AI in Software Engineering

Recent studies show that the integration of Artificial Intelligence (AI) into software engineering has transformed numerous aspects of the development lifecycle, from code generation and testing to project planning and maintenance [6, 11]. This section explores how AI, particularly LLMs, is increasingly viewed not just as a tool but as a collaborative partner in development work. It examines empirical evidence on productivity and quality improvements, identifies key implementation challenges, and outlines the emerging potential of AI.

2.3.1 AI as a Collaborative Partner in Software Development

Research has shown that the collaboration of humans and AI in software engineering is beneficial, and the view of AI collaboration is shifting from being considered as a tool to being considered as a collaborative partner. This suggests a more integrated approach in including AI and changes the way software engineers approach their work. Now, AI is considered to not only contribute to routine tasks but also take part in the creative and problem-solving parts of the work. Successful collaboration, however, requires effective communication with the AI collaboration tool, such as clearly stating its role, such as being a senior backend developer, a software tester,

or a specific programming language expert. This allows the AI model to focus on designated areas of the development process, optimize its efficiency, and helps it to propose innovative solutions. Having the communication iterative and in a software engineering context is also crucial, where clear and concise input improves both the AI's response and the overall user experience. Utilizing those practices has shown the versatility of AI in software engineering areas such as code generation, assistance across multiple programming languages, and debugging complex logic. These abilities, when combined with human oversight, have been shown to significantly enhance both the efficiency and quality of software development. However, the importance of maintaining human expertise remains important, particularly in areas requiring deep contextual understanding and critical judgment [6].

2.3.2 Productivity Gains and Quality Impact

Empirical studies highlight significant improvement in developer productivity and software quality when AI-powered tools are used in software engineering workflows. In a 2024 study involving 1000 engineers at ANZ Bank, researchers found that developers using GitHub Copilot⁷ completed coding tasks 42.36% faster than those who did not use it. The study also showed statistically significant reductions in bugs and code smells [11]. Similarly, another study showed that when comparing code error rates between the experimental group leveraging deep learning methods and the control group using traditional methods, the experimental group had a 5% error rate while the control group had 30% [12].

AI's role in software testing has also led to dramatic gains in efficiency. In a study researching AI-powered test automation tools, researchers found that test maintenance time was reduced by 95% while overall test accuracy was improved [13]. Furthermore, adopting AI-assisted development workflows has resulted in a 45% reduction in code development time alongside a 60% increase in code coverage [14].

2.3.3 Implementation Challenges

Despite clear benefits, integrating AI into software development workflows presents some challenges, ranging from trust and security concerns to over-reliance on AI.

One significant risk is over-reliance and misplaced trust in AI tools. A study involving 238 software engineers used poisoned LLMs that were trained with vulnerable code and found that developers misplaced trust in AI-generated code that led to security vulnerabilities [15]. Similarly, issues related to data quality and model interpretability have been identified. Poor data quality can lead to inconsistent performance of AI models, while the black box nature of models can make it difficult to diagnose and address performance issues [16].

Maintaining best practices in AI-generated code is another challenge. Research has shown that AI-generated test cases can often prioritize quantity over quality, as well as lack domain-specific insights, resulting in redundant or irrelevant tests [13].

⁷<https://github.com/features/copilot>

2.3.4 Future Potential of AI

AI-powered tools have already been shown to be valuable assets in software engineering, enhancing productivity, test automation, and debugging. Empirical studies show that AI integration leads to faster task completion, fewer coding errors, and increased test coverage [11, 13]. However, AI's role in code reviews remains relatively underexplored. As AI keeps evolving, identifying best practices for its effective integration into code review processes will be important in leveraging its full potential.

2.4 Large Language Models: Foundations and Deployment

Large Language Models are deep learning models designed to process and generate human-like text. These models are built upon transformer architectures and benefit significantly from large-scale training data. Training on these large datasets allows them to understand intricate linguistic patterns and produce contextually accurate responses [2].

2.4.1 Training and Adaptation

The training of LLMs typically follows a two-stage process that includes pre-training and fine-tuning. During pre-training, the model is exposed to a vast amount of textual data using self-supervised learning techniques. During this process, the model learns general language patterns, syntax, and semantic relationships. Fine-tuning then refines the model's capabilities by exposing it to a certain dataset with a specific purpose, optimizing it for targeted applications such as code generation. For general-purpose models, fine-tuning is more often in the form of instruction and alignment tuning, where the models are tuned to be better at following human instructions and to not produce undesirable content [2].

2.4.2 Cloud-Based and Locally Hosted Models

The deployment of LLMs can be categorized into cloud-based and local/edge-hosted models, each having its own set of pros and cons. Cloud-based LLMs offer significant scalability and usage-based pricing. They benefit from automatic scaling, failure recovery mechanisms, and have access to large-scale computational resources, making them more powerful and accurate. These attributes make cloud deployments ideal for handling extensive workloads and supporting complex tasks. However, cloud-based models can introduce latency and may pose challenges related to data privacy and sustained operational cost [17].

On the other hand, local models provide enhanced privacy by ensuring that data stays on the users' devices, reducing third-party exposure and reliance. They also offer lower latency since the models do not rely on external networks, making them ideal for applications requiring real-time responsiveness [18]. Nevertheless, local

deployments are constrained by hardware, requiring substantial memory and computational power for effective operation. Additionally, the quantization techniques necessary for running large models on edge devices may lead to reduced accuracy, impacting the reliability of the generated output [18]. Platforms such as Ollama⁸ and LlamaIndex⁹ provide accessible frameworks for running local models, enabling easy management and retrieval capabilities without reliance on cloud-based solutions.

2.5 Retrieval-Augmented Generation (RAG) Architecture

Retrieval-Augmented Generation (RAG) is a technique that enhances large language models by retrieving relevant information from an external knowledge base. The retrieved data is then used to help the model generate more informed and contextually accurate responses [19]. The main objective of RAGs is to combine the parametric knowledge from LLMs with non-parametric knowledge of external sources, allowing for specialized capabilities of the models without having to re-train the LLM. This can be very useful for use cases where the data is fast-changing or more specific. RAG aims to improve answer accuracy, reduce hallucinations, and produce verifiable outputs, particularly in knowledge-specific tasks [19]. An example is the Vul-RAG framework, which applies a knowledge-level RAG approach to vulnerability detection tasks [20]. By constructing a domain-specific knowledge base from existing CVE data, Vul-RAG ensures that LLMs can leverage updated references for more reliable answers. This demonstrates how RAG methods can incorporate newly available information that may not be captured in the LLM’s original training data [20].

The underlying theory of RAG can be understood through a three-phase framework: indexing, retrieval, and generation. The implementation described in this section follows the Naive RAG approach, which represents the basic form of retrieval-augmented generation. However, other variations exist, such as Advanced RAG and Modular RAG, which introduce more sophisticated retrieval and integration mechanisms [19].

2.5.1 Data indexing

This initial phase focuses on preparing the external sources for retrieval. The process starts with collecting sources such as documents, code repositories, or, in the case of “Vul-RAG”, existing CVE instances [20]. This data is then processed, cleaned, and converted into plain text. The data then goes through a process called chunking, in which the extracted text is divided into smaller sections. This is essential because LLMs have limited context windows, and the sizes of these chunks can significantly impact performance. Research explores various chunking strategies, and the optimal size can depend on content, the embedding model, and the expected queries [19].

⁸<https://ollama.com/>

⁹<https://www.llamaindex.ai/>

Each chunk is then converted into a vector embedding using an embedding model. These embeddings capture the semantic meaning of the text. The embeddings are then stored in a data index designed for quick and frequent similarity searches [19].

2.5.2 Retrieval

In the retrieval stage, the aim is to identify the most relevant information chunks given a user’s query. It starts with embedding the query with the same or a compatible embedding model as the indexed data. Then, a similarity search is performed between the query embedding and the indexed data. The similarity search selects top-k document chunks that exhibit the highest similarity to the query embedding [19].

2.5.3 Generation

In the final stage, the retrieved context and the user query are combined and fed into the LLM. The LLM then generates an answer based on this augmented information. The system can be configured to rely primarily on the retrieved context or allow it to also utilize its internal knowledge [19].

2.5.4 Advantages of RAG

The key advantage of RAG is its ability to incorporate up-to-date knowledge, unlike a typical LLM, whose knowledge is fixed at the time of training. By retrieving information from an up-to-date knowledge base, RAG can access and utilize information that was not available during the training. Compared to finetuning a model, RAG offers more flexibility. The local knowledge base can be updated, expanded, or tailored to specific needs without requiring expensive retraining of the model since updating a knowledge base is more efficient than retraining an entire model [19]. These points make RAG a promising approach to knowledge-intensive tasks such as code review, where access to relevant project context is crucial.

2.5.5 Limitations of RAG

Despite its advantages, RAG faces several key challenges that can impact effectiveness. A primary challenge of RAG is that retrieval can be suboptimal, with irrelevant or outdated chunks being included while important ones are missed, leading to hallucinations or insufficient context for accurate responses. The augmentation process may also introduce redundancy or stylistic inconsistency, and the model may fail to synthesize retrieved content effectively. RAG may also lead to increased latency as RAG involves data retrieval, while models without RAG can respond instantly [19].

2.6 Agentic LLM Systems

Agentic LLMs are defined as large language models that can reason, act, and interact [21]. These systems use an LLM to analyze a problem through reasoning,

develop a solution strategy, and carry out that strategy by invoking a set of available tools [22]. Unlike traditional LLMs, which generate responses based on a single, static prompt, agentic LLMs operate within a control loop. This loop allows the model to determine what steps are needed to accomplish a task, decide when and how to act, and recognize when to stop [22]. This autonomy is enabled through a combination of tool awareness, planning capabilities, and multi-step reasoning mechanisms. Agentic systems typically interact with external resources, allowing them to retrieve information, reflect on results, and adapt their behavior. Some also incorporate memory across steps or sessions, enabling longer-term adaptation. These interactions may generate new data or results, which can in turn influence further reasoning or action selection [21].

This differs significantly from traditional or non-agentic LLMs, which are typically passive and stateless. These models only respond to direct prompts without persistent memory or adaptive behavior and do not have access to external tools [21, 22]. They lack the ability to plan ahead or to act autonomously, making them reliant on user-driven prompt engineering for complex tasks. In contrast, agentic LLMs are capable of managing problems that span multiple steps and adjusting to the outcomes of their previous actions [22].

A key mechanism agentic LLMs use to simulate autonomy is an iterative, step-by-step reasoning process, often inspired by how humans solve complex problems. Rather than attempting to solve every task in a single inference, the model decomposes the problem into smaller components and progresses through intermediate reasoning stages instead of providing a direct, potentially flawed answer [21]. This approach, often referred to as chain-of-thought (CoT) prompting, encourages the model to “think out loud” by communicating its internal logic in sequential steps. With this, the model gains the ability to monitor its own reasoning and adjust its strategy based on emerging outcomes. This planning can either be explicit, where the agent formulates a multi-step plan and executes it incrementally, or implicit, with the model deciding on the next best action after each iteration [22]. At each stage, the agent interprets the current state, reasons about the next appropriate action, and then takes that action (often using a tool or querying some data resource) before observing the result and integrating it into the next reasoning cycle [21]. This dynamic process enables the agent to incorporate feedback, refine its strategy, and, if necessary, rerun or revise failed steps [22]. The ability to simulate such reflection, correction, and step-by-step process is a core part of the flexible problem-solving ability that distinguishes agentic LLMs from traditional, stateless LLMs [21].

2.6.1 Tool-Driven Reasoning

A central feature that distinguishes agentic LLMs from traditional models is their ability to interact with external tools. Tools provide the model with a means to interact with its environment, enabling capabilities such as querying APIs, retrieving documents, or executing specific functions that go beyond just language generation [22]. However, for an agent to correctly employ tools, it must also be able to decide when to use them. This decision-making occurs during inference and is gov-

erned by internal reasoning processes that guide when and how to invoke tools [21]. By exposing tools in a clear, structured format, such as function signatures or API schemas, the system enables the LLM to independently select and invoke the appropriate tool based on task context [22].

In addition to invoking tools, the agent must also parse tool outputs and adapt its behaviour accordingly. This includes integrating results into future reasoning steps, adjusting plans based on observed outcomes or retrying failed operations under strategies like informed retries (which append error messages and failed outputs to help the LLM self-correct) or external retries (which offload error correction to a separate LLM with expert context) [22]. Additionally, agentic LLMs often tackle problems that exceed the scope of a single inference. They approach complex tasks through sequences of reasoning and action, where intermediate results shape subsequent steps. This capacity to handle multi-step tasks, supported by ongoing planning and evaluation, allows for more flexible and reliable problem solving as the agent can adjust its strategy based on new information. To support this kind of iterative reasoning and dynamic interaction, agentic LLMs benefit from a structured control flow that governs how thoughts are generated, tools are invoked, and outputs are integrated into the decision-making process [21].

2.6.2 The ReAct Framework

ReAct, short for Reason and Act, is a prompt-based framework that is designed to integrate both reasoning and acting within LLMs for general task solving. Unlike traditional prompting strategies, which require LLMs to complete all reasoning before taking any action, ReAct allows models to alternate between thinking (verbal reasoning traces) and acting (task-specific operations) in an alternating sequence. This design supports dynamic reasoning by allowing the model to update its internal state as it progresses through a task. The defined ReAct loop consists of three core components: thought, action, and observation. The thought is an internal reasoning step expressed in natural language that decomposes goals, injects commonsense knowledge, tracks progress, adapts strategies, extracts information from observations, and handles exceptions without altering the external environment. The action is a domain-specific step that queries or manipulates external resources (e.g., invoking tools to query data or communicate with external APIs). Finally, the observation contains the feedback or the result received after executing an external action. Together, these components form a repeated cycle of thought, action, and observation, in which the model iteratively builds knowledge, executes commands, and adjusts its plan in response to outcomes. The task-solving process typically consists of multiple thought-action-observation steps. The interplay between internal reasoning and external action allows ReAct agents to move beyond static output generation, enabling them to function as decision-makers within a repeatable loop that organizes reasoning and actions [23].

Compared to CoT prompting, which is static and internally focused, ReAct introduces an interactive and externally aware reasoning pattern. While CoT elicits internal reasoning, it cannot engage with external environments, making it prone to

hallucinations and error propagation due to its reliance on internal representations alone. In contrast, ReAct integrates reasoning and acting into a unified, iterative loop, allowing the model to query external resources and adapt its behaviour based on feedback. This architecture has been shown to bring several benefits. First, it increases transparency by producing interpretable sequences of thoughts and actions that enable users to trace the whole decision-making process and distinguish between internal knowledge and retrieved information. Second, it introduces adaptability since the loop allows agents to dynamically maintain and revise plans in response to changing conditions or any unexpected outcome. Finally, ReAct has been shown to achieve superior performance on a range of multi-step tasks where it outperforms baselines in both knowledge-based settings (where all necessary information is included in the input) and interactive settings (where the model must act to retrieve information from external tools or environments). These advantages make ReAct a big milestone in the development of more capable and trustworthy agentic LLM systems [23].

3

Related Work

The goal of this chapter is to situate this research within the broader landscape of AI-assisted code reviews. Firstly, it discusses previous work on applying AI techniques, particularly LLMs, to improve or automate parts of the review process. Secondly, empirical findings from recent AI-driven review experiments are highlighted, focusing on how such approaches impact reviewer behavior and code quality. Then, challenges and open questions regarding the integration of AI tools are discussed before the last section summarizes the main research gaps that this research addresses.

3.1 AI-Assisted Code Reviews

Traditional code review practices, especially in large organizations, are well-established and tool-assisted. However, a significant part of the process still relies heavily on human reviewers. Recent studies argue that AI tools offer the potential to automate the most mundane aspects of the code review process, providing timely feedback to authors and freeing up human reviewers to concentrate on higher-level issues like functionality, design, and architecture [3]. Previous studies have explored ways to automate parts of the code review process. One study involved a tool that learned from past reviews to predict changes a reviewer might suggest. It then generated a revised version of the code, allowing developers to inspect the automatically implemented modifications before submitting it for manual review. In the same study, researchers also tested a tool that automatically implemented reviewer suggestions derived from natural language comments, since that should help the reviewer clearly show what they mean and help the developer understand the suggestion better. The transformer-based models utilized in this study represented a promising first step toward automating code review tasks, but their transformations were relatively simple. That motivates the need for more advanced models capable of handling complex scenarios [24].

Recently, more advanced models have been explored in this context, particularly LLMs, and they have demonstrated potential to enhance various aspects of the code review process. This is, for example, by providing suggestions to the reviewer, predicting problems, and facilitating improved code quality. This has also been shown to have the potential to offer actionable feedback beyond what basic static analysis tools provide, where basic syntax errors or well-known bug patterns are

highlighted. Developers who have tested LLM tools to assist with code reviews have also mentioned them to be a valuable resource for both learning and development. This indicates that the AI tools are not only improving the review process and its result but also deepening the developer’s knowledge [25]. In a study conducted at Google, they introduced a specialized tool, AutoCommenter¹, which utilizes LLMs for these purposes, and it displayed promising results [3].

Previous research shows that using LLMs in code reviews can enhance software quality and improve developers’ knowledge by promoting a better understanding of best practices and efficient coding techniques [25]. Models specifically trained for code reviews have been shown to help in optimizing the code and introducing alternative approaches, thereby enhancing code quality. Additionally, they have been able to identify various issues from minor bugs to significant code smells across different programming languages [25]. The potential for LLMs to identify security vulnerabilities has also been pointed out, as it can significantly enhance the security of the software, and such vulnerabilities are commonly overlooked in manual reviews [26]. Previous studies have also highlighted benefits such as increased efficiency and speed, improved consistency and accuracy, and a reduced reviewer workload [3]. Additionally, with these AI tools come increased learning opportunities for developers. For example, AutoCommenter can provide specific feedback with links to relevant documentation, real-time feedback, and guidance towards better understanding and application of best practices [3]. If the comparison of existing LLMs is investigated, both open-source and proprietary models like ChatGPT² tend to perform well on code review tasks. It has also been shown that techniques like chain-of-thought prompting, where the model is guided through a structured reasoning process, improve the performance of those proprietary LLMs [26].

3.2 Recent Empirical Studies

A recent study was performed to investigate how automatic code reviews generated by an advanced LLM affect the code review process in practice. In a controlled study with just under 30 developers who performed code reviews in Java and Python, the researchers compared manual code review with automated code review. The manual code review is traditional, where only the developer reviews the code, while in the automatic version, the reviewer is provided with a code review that was automatically generated by the LLM. Additionally, the results were compared with a hypothetical comprehensive review where the reviewer was provided with a code review that already pointed out all the quality issues that had been injected into the code. This was to investigate the effect a perfect AI assistant would have on the code review process. The OpenAI GPT-4 model was utilized, but not provided with a broad context, only the code that was up for review. The reviewers tended to confirm the suggestions provided by the LLM, or on average, 89% of them. When analyzing the results, no notable time savings seemed to emerge from either automated

¹Not available to the public.

²<https://chatgpt.com/>

approach. The developers still had to read, interpret, and verify the automatically generated results. Both automatic approaches tended to highlight more total issues, particularly lower-severity issues, but there was no improvement in catching critical, high-severity issues compared to the fully manual reviews. Additionally, reviewers who were fed with automatically generated reviews often demonstrated behavioral bias where they focused more on the lines flagged by the LLM and overlooked other unmentioned portions of the code. This bias supports the idea of AI suggestions steering human reviewers to accept or concentrate on the system’s priorities rather than their own. It was also noticed in the same study that automated reviews did not significantly boost reviewers’ confidence nor enhance their overall understanding of the code being reviewed, since the LLM only pointed out possible problems without assisting with comprehension of the code itself [27].

Another recent study relevant to AI-driven code reviews is an interview-based investigation comparing developers’ experiences when receiving code review feedback from an LLM versus a human reviewer (a peer-led review). In that study, participants wrote code that was then reviewed by both a designated human reviewer and by the ChatGPT 4.0 model, which was prompted to provide an “expert-level” critique of the submitted code. The code authors were then asked to reflect on how they perceived and engaged with each type of review feedback. Findings reveal that while the LLM-generated reviews can be detailed, they often impose a higher cognitive load on the developers. Participants noted that the LLM’s suggestions sometimes lacked conciseness and context. This required additional time to parse and understand the feedback compared to the easier-to-interpret peer reviews that could be more concise and tailored to the team’s coding standards or architectural conventions. However, developers reported that LLM-generated feedback was generally less emotionally taxing because of the model’s consistently polite tone and absence of personal criticism. In a human-generated review, a harsh or overly blunt comment could trigger negative emotional responses, while LLM feedback did not show that variability. Additionally, some participants expressed concerns about the AI’s lack of deeper repository knowledge, with one interviewee saying that “AI might not be completely aware of what the context of the code you are writing ... what are your future goals, so how the codebase should be.” However, despite all those differences, the developers engaged in similar sense-making steps where they evaluated the review’s merits, testing suggestions, and deciding whether or not to adopt them, regardless of whether the comment came from a human or the LLM [28].

It can be noted that the study authors concluded that context plays a critical role in shaping the usefulness of the LLM feedback. Since ChatGPT was only given a code snippet rather than the entire repository of relevant project details, it struggled to provide insights aware of the overall architecture of the project or evaluate the broader implications [28]. This also aligns with the previously discussed study that observed that LLM reviews can inadvertently focus on smaller issues while missing more severe ones [27]. This motivates approaches such as a RAG-based approach, where the LLM has access to an entire repository and additional pull-request data to improve contextual depth when generating code review feedback or summary.

3.3 Challenges and Limitations of AI-Driven Code Reviews

Despite the potential benefits outlined in previous sections, recent empirical work also identifies key limitations and challenges relating to LLM-driven code reviews. Recent studies highlight how challenging it is to integrate LLMs into the review workflow and that their performance does not always outperform the human reviewer [27].

One limitation is LLMs' inability to fully understand the broader architectural context or developer intent. This can result in inaccurate or irrelevant suggestions. As a result, important project-specific or design-level issues may go unnoticed [28].

Additionally, LLMs can generate false positives and false negatives. These inaccuracies make human oversight important [3]. Moreover, some studies indicate that LLMs do not necessarily improve the detection of high-severity issues, introducing concerns that critical bugs might still go unnoticed [27].

Evaluating the feedback provided by LLMs has also been shown not to be straightforward, particularly when it comes to descriptions of security vulnerabilities [26]. Additionally, the feedback can be long and overwhelming, making it hard to grasp and requiring more time to comprehend. Beyond length, developers also worry about a low signal-to-noise ratio, where many trivial or less relevant comments overshadow critical feedback [28].

As previously mentioned, the LLM output can also cause bias and alter how human reviewers explore the code. This raises the question of whether AI tools support deeper inspection of unfamiliar code or instead reduce the thoroughness of the review [27]. For instance, AI-suggested highlights may steer the reviewer's attention towards them, leading to unreviewed or under-reviewed code segments.

Another challenge is bias in training data, leading to unfair or inaccurate code assessments. Research has shown that by training LLMs on code review-specific datasets and oversampling high-quality reviews in the training data, significant improvements in the relevance and accuracy of recommendations were observed [5]. However, these gains depend heavily on the type of LLM in use, whether it is a general-purpose model or one explicitly trained on code repositories for the purpose of performing code reviews.

Trust is also a significant factor. Developers may not fully trust LLM-generated feedback and often feel compelled to validate each suggestion, which can undermine the intended efficiency gains [28].

Furthermore, the fast-changing nature of software development is also a challenge, with coding standards evolving rapidly, requiring AI tools to be regularly updated and retrained to stay relevant [3].

Finally, balancing automation and human expertise is essential, as over-reliance on AI tools can lead to a decline in human expertise and critical thinking. In fact, many researchers argue that complete automation is neither realistic nor desirable, as it undermines knowledge-sharing and collaboration among developers [27].

3.4 Summary and Gaps

As discussed, LLM-assisted code reviews tend to result in faster, more consistent feedback and even the potential for advanced tasks like identifying security vulnerabilities. However, empirical studies also reveal challenges, such as developers often investing additional efforts in interpreting or validating LLM suggestions, and critical issues may remain undetected when AI is not provided with a broader project context. Additionally, although peer-based reviews can introduce emotional and communication challenges, LLM-driven reviews pose new trade-offs concerning trust and interpretability, emphasizing that human oversight remains crucial. This confirms that AI-driven code reviews should currently serve as a complementary rather than a replacement step to the human reviewer, supporting developers in their routine checks on code changes but still requiring human vigilance for critical or subtle issues.

Despite these insights and growing interest in AI-assisted code reviews, the literature lacks consensus on how best to integrate LLMs into existing review workflows in ways that genuinely improve efficiency without undermining human expertise. Another gap in the literature is the lack of research on supplying models with the broader architectural context during code reviews. Additionally, while early findings indicate that AI-driven suggestions can accelerate the review process, there is still limited understanding of how to integrate AI insights without reducing the knowledge-sharing benefits.

This research addresses these gaps by first examining how code reviews are currently conducted in practice, identifying pain points and opportunities for AI support. Building on this foundation, it explores how an AI code review assistant equipped with external data sources and used through different interaction modes may assist developers during review. A particularly underexplored question this study targets is how developers prefer to interact with LLMs in the review process, whether through up-front summaries, on-demand queries, or other forms of engagement. Rather than evaluating technical performance, the focus is on understanding how developers perceive, interact with, and reflect on AI assistance, to inform the development of best practices on integrating AI into human code review.

3. Related Work

4

Method

The following chapter presents the overall research design and methods. It starts by outlining the fundamental research strategy and shows how it builds upon both established software engineering practices and specific considerations of AI integration. It then describes how the study is divided into two main phases, with one for mapping the existing code review processes and another for conducting a field experiment with LLM integrations into the workflow. This approach is intended to ensure that the exploration of LLM assistance is based on empirical observation, resulting in practical insights for teams that seek to integrate AI-assisted code reviews.

The research is conducted in cooperation with the software development company WirelessCar, where their resources, processes, and developer teams are used as subjects of the research.

4.1 Research Design Overview

Grounded in the literature review, this study adopts a two-phase research design that is aimed at understanding both the current code review processes in a natural environment and then intervening with an AI-based solution to observe the effects (see Figure 4.1 for the research flowchart). To situate these phases within established software engineering research methodology, the well-established ABC framework for research strategies in software engineering [29] is drawn upon. The ABC framework emphasizes that studies focusing on actors (here, the developers), their behavior, and the context in which they operate can be represented with two dimensions. Firstly, how much the researcher intrudes on the setting (obtrusiveness), and secondly, how generalizable the findings aim to be [29].

4. Method

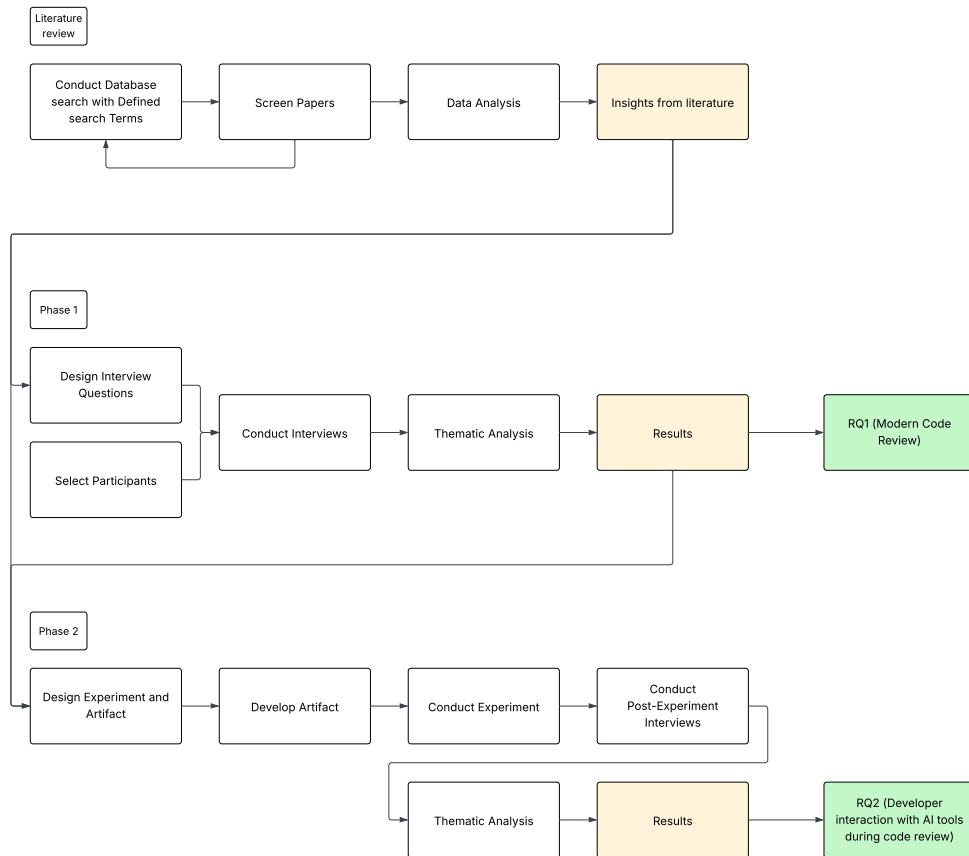


Figure 4.1: Flowchart detailing the flow of the research

Phase 1 is designed as a field study. In such a study, the researchers operate in a pre-existing, real-world environment without actively controlling or modifying its parameters. The benefit of this approach is its capacity to capture the realistic context, enabling a deep, situated understanding of the phenomenon under investigation, which in this case is the code review process at WirelessCar. However, field studies do not typically result in statistical generalizations or precise measurements of cause-and-effect relationships. Instead, they prioritize rich detail and natural insight on how developers conduct reviews, which challenges they face, and identify where in the process AI may add value [29].

Phase 2 transitions to a field experiment. Such a study still occurs in a natural environment but introduces an intervention. In this case, the intervention is the introduction of an LLM-based tool for code reviews to study the practical effects on review efficiency and developer experience. Field experiments also aim for real-world realism but rank higher on obtrusiveness than a field study since researchers manipulate certain conditions or introduce new tools to an otherwise unchanged workflow. As a result, the findings are not broadly generalizable, and real-world settings introduce many confounding factors [29]. However, this design should provide insights into how an AI assistant might meaningfully alter reviewer behavior under authentic conditions.

Together, these two phases, a field study followed by a field experiment, offer a holistic empirical approach. In phase 1, the current code review practices are mapped, which generates a hypothesis about how AI could potentially help. In phase 2, those ideas are tested in a real-world setting by introducing new tools and solutions to the process. While this design might not seek precision like found in controlled laboratory experiments or the ability to universally generalize findings, it aims to capture realistic context and investigate real-world impact. By utilizing and referencing the ABC framework [29], the methodological choices in each phase are distinguished, and the approach is grounded in empirical software engineering research strategies.

4.2 Literature Review Process

While the literature review is presented in the Background and Related Work chapters, the review served not only to contextualize the study but also to inform the design of the interview questions in Phase 1 and help guide the design of the experiment in Phase 2.

The literature review process aimed to systematically collect, analyze, and synthesize existing research on code reviews, Large Language Models, AI in software engineering, and other relevant technologies related to this study. Due to the cutting-edge nature of these technologies, there was an emphasis on the most current and relevant research available. The literature search was iterative, and additional searches were performed throughout the project timeline. This approach allowed for the inclusion of the most recent publications.

The process followed this methodology:

1. **Search Strategy:** A set of search terms was defined through preliminary exploration and prior domain knowledge. Primary keywords include but are not limited to: code review, AI-assisted code review, LLM, Large language models, local LLMs, Ollama, Retrieval-augmented Generation, RAG, and AI in software engineering. Searches were conducted using digital databases and search engines such as Google Scholar, IEEE Xplore, Semantic Scholar, Elicit, and others.
2. **Inclusion and Exclusion Criteria:** Due to the rapidly evolving nature of AI and related fields, priority was given to the latest publications on those topics. Only English-language articles were considered. Relevance to the study's core topics was evaluated.
3. **Screening process:** Initial document filtering was done based on abstracts, conclusions, and light reading of material. Further filtering was then performed with a more detailed reading of the material.
4. **Data Analysis:** Selected articles were read thoroughly with important information noted and grouped based on key topics and findings.

4.3 Research Setting

The research was conducted in close collaboration with WirelessCar Sweden AB¹, a software development company based in Gothenburg, Sweden. WirelessCar specializes in connected vehicle services and provides digital solutions to automotive clients around the world. The company operates in cross-functional development teams, each typically responsible for the end-to-end development and maintenance of specific subsystems. Within these teams, it is common that the same developer who authors a piece of code is also responsible for reviewing code contributions from teammates, ensuring both ownership and familiarity with the codebase.

Throughout the entire duration of the study, the researchers were physically present at the WirelessCar office in Gothenburg. WirelessCar supported the research by granting limited access to specific internal resources relevant to the study’s scope. The company also actively encouraged developer participation in both the field study and field experiment phases. This collaborative setup provided a unique opportunity to explore AI-assisted code review integration in a real-world software development environment.

4.4 Phase 1: Field Study

The first phase involved an exploratory case study at WirelessCar to thoroughly understand existing manual code review practices and identify specific pain points and potential AI application areas. Semi-structured interviews were conducted, and thematic analysis [30] was used to analyze the qualitative data that was collected. The following section will describe the research process, methods, and choices.

4.4.1 Focus Areas

The primary focus of Phase 1 was to capture a comprehensive picture of the code review process and environment at WirelessCar. Therefore, the goal was to identify typical workflow steps and key pain points as well as clarify why certain inefficiencies arise, how success is evaluated, and where AI might offer possible improvements. Specifically, the study aimed to map the following aspects:

1. **Document the current review workflow**

- Understand each stage of the code review process (e.g., PR creation, assignment, feedback, merging).
- Determine which tools are used and how reviews are initiated and tracked.

2. **Reveal process challenges and inefficiencies**

- Identify any “pain points” in the process.

¹<https://www.wirelesscar.com/>

- Investigate how reviewers balance their thoroughness with speed, and whether any review steps often get skipped or truncated.

3. Assess how success is defined and measured

- Examine if (and how) teams track outcomes such as defect rates, review turnaround, or reviewer satisfaction in their traditional workflow.
- Evaluate the level of formality in measuring review effectiveness and whether any systematic metrics exist.

4. Investigate the importance of broader code context

- Determine how much architectural or broader project insight is deemed necessary for an effective review.
- Understand the existing strategies that developers use to gain context.

5. Explore current AI usage and future needs

- Investigate any prior experiences with AI-based development tools and how developers perceive their benefits or drawbacks.
- Gather direct feedback on which review tasks, like summarizing changes or identifying inconsistencies, are considered suitable for AI assistance.

6. Capture the technical and organizational environment

- Understand the daily development ecosystem: e.g., what data is attached to pull requests, how code is integrated and deployed, and how team members communicate.
- Document the specific toolchain (IDEs, CI/CD systems, and communication channels) that shapes how reviews are requested and performed.

4.4.2 Data Collection

To gain insights, semi-structured interviews were conducted. That format balances consistency where all participants receive a core set of similar questions with flexibility where researchers can ask follow-up or clarifying questions as fits the flow of the interview [31]. The interview questions were developed around domains such as code review processes, challenges in code reviews, measuring code review success, and current and potential AI use cases to cover the previously mentioned objectives. The list of interview questions used can be seen in Appendix A.1.1. The interviews were designed to fit comfortably within a 30-minute timeframe. However, in practice, the interviews ranged from 15 to 40 minutes each, with most lasting approximately half an hour.

Researchers mostly employed convenience sampling by interviewing people within

the cooperating company who were able and willing to participate [32]. Participants were recruited via announcements on Slack² channels and informal Slack messages that announced the study’s purpose and form of the interview. Interviewees were sought who handled different parts of the code and were reviewers at varying seniorities. Additionally, some interviewees were from the same development team to gain multiple perspectives from the same team. This sampling enabled insights into a relatively wide range of review practices.

Table 4.1: Overview of Interview Participants

Participant ID	Role	Team
P1	Quality Assurance Specialist	Team A
P2	Application Developer	Team A
P3	Software Engineer	Team A
P4	Software Engineer	Team A
P5	Security Engineer	Team B
P6	Software Engineer	Team C
P7	Software Engineer	Teams D & B

A total of seven participants were interviewed and are listed in Table 4.1. The interviewees varied in gender and age, and the sample included software engineers, developers, security engineers, and quality assurance specialists across four development teams. Teams ranged in size from under five members to around sixteen members. The interviewees had different areas of expertise, with varying levels of experience and skill. The goal was to include individuals who could discuss both strategic and day-to-day review practices. Data saturation was reached after the seventh interview, as no new information or themes were emerging. This decision is grounded in the principle that qualitative data collection can be considered sufficient when additional interviews fail to yield new insights [33]. Data saturation is defined as the point at which no new themes are observed in the data, and studies show that although saturation often occurs within twelve interviews, the basic elements of major themes are frequently present as early as six [33]. The current sample is somewhat heterogeneous in terms of role and team, and the consistency of responses across participants suggested that the core aspects had been adequately captured. Although an eighth interview had been scheduled, the participant canceled, but given that saturation had already been achieved, it was decided not to reschedule the interview.

All interviews were held in English and conducted in the participant’s active work environment, either on-site at WirelessCar’s office or remotely via Microsoft Teams (two participants were interviewed remotely). This alignment with genuine working conditions is consistent with a field study approach since it allows developers to

²<https://slack.com/>

reference real pull requests, team communication channels, and examples of ongoing tasks. They could describe, for example, how a large refactoring PR or an urgent bug fix impacted their review process in real-time. Conducting interviews in this natural setting helps reinforce the authenticity of participant responses and results in findings consistent with everyday workflows at WirelessCar.

All participants provided informed consent before the interviews. The participants were informed about the purpose of the study, how the data would be used, and their right to withdraw at any time.

4.4.3 Data Analysis

This study applied a theoretical thematic analysis [30], guided by the focus areas of the field study and prior engagement with literature on code review practices and AI in software engineering. The analysis was deductive as it was structured around predefined domains such as review processes, challenges, and AI tools. However, it also incorporated somewhat of inductive elements where new insights emerged organically from the participant responses. Thematic analysis was considered appropriate for this study due to its flexibility and suitability for identifying patterns of meaning across qualitative datasets, especially when exploring practical experiences and expectations in a specific organizational context [30].

The analysis was conducted at a semantic level, where it focused on the explicit statements expressed by the interviewees rather than interpreting latent meanings or underlying ideologies. It also took a realistic stance, assuming the interviewees' descriptions reflected their actual experiences and practices. Finally, the goal of the analysis was to deliver a thorough examination of specific themes, rather than a broad description of the entire dataset, which is in accordance with the focus on code review workflows and potential AI integrations [30].

The thematic analysis followed the six-phase thematic analysis process, which includes: (1) familiarization with the data, (2) generating initial codes, (3) searching for themes, (4) reviewing themes, (5) defining and naming themes, and (6) producing the report [30]. Importantly, this process is not strictly linear but recursive in nature, allowing movement back and forth between phases as needed. All interviews were recorded and transcribed using two devices to ensure accuracy and prevent data loss. Transcriptions were generated using Otter AI³, which provided a text-based representation of the discussions and automatically assigned speakers. Those transcripts are the primary source of qualitative data. In the first step, transcripts were then read through multiple times to correct transcription errors to make them more readable and develop a general understanding of the responses. Additionally, during those initial readings, notable and recurring ideas were noted down. Then, initial codes were generated using mainly deductive approaches, organized according to initial analytical focus areas (e.g., review practices, challenges, AI use). Additional codes were introduced inductively to capture emergent concepts

³<https://otter.ai/>

identified during data familiarization. These codes were then grouped into potential themes based on conceptual similarity. Those themes were iteratively reviewed and refined by comparing them with their associated codes and with the full dataset to ensure the internal coherence of each theme and distinctiveness. Steps 2 through 5 (coding, searching, reviewing, and defining themes) were repeated iteratively to refine both the theme structure and the interpretation of data. During this process, some themes were split or merged as needed. Once the structure was stabilized, the themes were clearly defined, named, and documented to describe their analytic scope. An example of how a quote was assigned a code that then belongs to a theme can be seen in Table 4.2. Finally, the findings were reported in Section 5.1, where each theme is analyzed and discussed, supported by selected quotes that illustrate key points and offer direct insights into developer experiences.

Table 4.2: Example Coding of a Quote from Phase 1 Transcripts

Quote	Code	Theme
<i>“As soon as you need to context switch, even if it’s just a three-minute thing, it’s 20 minutes of lost time.” [P7]</i>	Context switching interrupts productivity	Code Review Challenges

All coding was conducted collaboratively by two researchers. After individually familiarizing themselves with the data and noting down code ideas, the researchers conducted the full coding process together. Each transcript was reviewed line-by-line in joint sessions where codes were discussed, refined, and applied by consensus. This collaborative approach ensured that both researchers had a shared understanding of each code’s meaning and use. Codes were systematically recorded in a shared digital workspace, where all active codes, descriptions, and associated quotations were tracked and updated across iterations. While no formal codebook with definitions and inclusion/exclusion criteria was created in advance, this shared code list functioned as an evolving, informal codebook that supported consistent application across the dataset.

4.5 Phase 2: Field Experiment

The second phase assesses how software developers at WirelessCar experience and evaluate the integration of a software artifact featuring AI assistance into their code review process. Building on the challenges and improvement opportunities identified during the field study in Phase 1, this phase investigates how different interaction modes with AI assistance affect the review workflow, developer preferences, and perceived usefulness.

Rather than focusing on performance metrics or direct comparisons to existing tools, the emphasis was on capturing qualitative insights into how developers interact with the AI assistant, what kinds of support they find valuable, and how such a tool might

be ideally integrated into real-world code review practices. This was investigated by exposing participants to different interaction modes, such as AI-led guidance and on-demand assistance.

4.5.1 Focus Areas

The primary focus of Phase 2 was to investigate how software developers at WirelessCar experience, evaluate, and interact with LLM-based assistance during code review. Specifically, the experiment investigates the following aspects:

1. Compare AI-assisted and traditional review experiences

- Explore how developers perceive the AI-assisted review process relative to their existing workflow.
- Capture whether developers believe the AI assistance adds value or introduces friction compared to their usual tools and methods.

2. Evaluate developer experience across different AI interaction modes

- Compare how participants experience two different interaction modes: AI-led guidance and on-demand assistance.
- Identify the preferred interaction style and under what circumstances.
- Examine the perceived usefulness of each mode.

3. Identify design implications and best practices

- Provide insights for when and how to embed AI assistance in code review.
- Identify risks in AI-assisted code reviews and how to mitigate them.

4.5.2 Experiment Setup

This section outlines the experimental setup used in Phase 2 of the study. The field experiment was conducted to evaluate the developers' experiences with LLM-based code review assistance under two distinct interaction modes that were selected based on the findings from the Phase 1 field study (see Section 5.1). As indicated by the interview data in that earlier phase, developers most frequently identified a need for clearer up-front summaries of code changes as well as on-demand explanations for specific architectural or contextual details.

In each iteration of the experiment, a single participant engaged in a traditional code review scenario, conducted within their familiar development environment using their usual tools and platforms. In addition to these standard resources, the participant was provided with access to an AI assistant, implemented as described in Section 4.5.3. The task assigned to the participant was to perform two code reviews, each on a different pull request from two separate repositories within the

company. The participant followed a different interaction mode with the AI assistant for each review. All sessions took place in conditions that mirrored each participant’s normal working situation as closely as possible.

The two interaction modes are described below:

- **Mode A: Co-Reviewer** – In this mode, the AI assistant automatically generated a summary of the code under review, highlighting major changes and any potential points of interest or concern, before the reviewer started their own examination. The reviewer could then use this information to guide their review, optionally asking the AI-assistant follow-up questions. The participant could query the AI for clarification or more details about the summarized areas. This mode directly targeted the challenge previously identified as lacking immediate context, particularly for large or complex PRs.
- **Mode B: Interactive Assistant** – In this mode, the reviewer reviewed the code in their typical manner. The AI assistant did not proactively provide a summary or suggest issues upfront. Instead, the reviewer was free to consult the AI on demand, requesting clarifications about specific parts of the code or asking higher-level architectural questions. Reviewers were encouraged to ask targeted queries rather than requesting an overall summary of the changes. Here, the AI assistant only responded when explicitly prompted. This design choice not only reflected the Phase 1 feedback on needing a lightweight, on-demand tool but also addressed an identified challenge in related studies, where automatically highlighted lines can cause reviewers to miss other important areas [27]. By making the AI passive, reviewers maintained their usual workflow and examined the entire codebase without unconsciously depending on the AI’s initial hints.

These two different modes ensured that different types of interaction with the AI assistant were explored during the experiment. They also allowed for comparison of results with respect to who initiated the interaction in the code review process, how the presence or absence of initial suggestions from the AI influenced the reviewer’s strategy, and how the reviewer’s actions shaped the AI’s responses.

To minimize review variability while still enabling comparison across interaction types, two specific pull requests of similar size and complexity were selected from the WirelessCar codebase. The selected pull requests each involved a moderate amount of change, requiring genuine reviewer effort without being excessively large or trivial. Before the first experiment session, a pilot run was conducted with two internal developers who were not part of the main participant pool. The pilot involved a walkthrough of the tool, the review task, and the interaction modes, followed by an informal test of the tool using the selected pull requests. The pilot was used to assess the suitability of the selected PRs, detect any major usability or technical issues in the tool, and evaluate whether the introduction and guidance provided to participants were sufficiently clear and effective. Each participant in the actual experiment then reviewed both PRs, using Mode A for one PR and Mode B for the

other. The assignment of modes to PRs was rotated across participants to mitigate ordering effects. By having all participants conduct the same two code reviews with alternating modes, this approach allowed for a more controlled comparison of interaction styles while ensuring the tasks remained realistic and relevant to actual code review practices.

To investigate how varying levels of codebase familiarity may influence the use of the AI assistant, both developers belonging to the team that owns the selected PRs and developers from unrelated teams were invited to participate. Phase 1 interviews indicated that limited contextual knowledge can have negative effects on the review quality, and therefore, measuring differences in AI reliance across these two participant groups was expected to produce further insights.

Before starting the code review sessions, each participant was given a short onboarding briefing to explain the tool’s features as well as the structure and setup of both the study and the experiment. The two different interaction modes were introduced, along with guidance on how to effectively prompt the AI assistant to obtain useful and relevant responses. The participant then completed the two review sessions consecutively. While no in-depth feedback or direct assistance was provided during the sessions, limited guidance was offered when needed, for example, if participants inquired about specific ways of interacting with the assistant. Participants were also encouraged to think aloud during the experiment, often verbalizing their reasoning, confirming the AI’s suggestions, or commenting on its usefulness. If the assistant’s response did not meet expectations, participants were occasionally guided to try rephrasing or retrying the query. Researchers were present to observe the sessions, record notes, and perform the post-session data collection. Following the review sessions, short semi-structured interviews were conducted with the participant to reflect on their experience across the two modes and in comparison to their regular code review workflow.

4.5.3 Artifact Implementation

The software artifact developed for this study was a web-based chat interface designed to explore and evaluate different interaction styles in AI-assisted code reviews. Its primary purpose was to enable structured experimentation by allowing researchers to observe how developers interact with AI assistance in different contexts. The source code for the artifact is publicly available on GitHub: frontend⁴ and backend⁵.

The artifact consisted of a chat interface connected to a backend system that integrated OpenAI’s `o4-mini`⁶ language model via API. The artifact was also supported by a Retrieval Augmented Generation (RAG) infrastructure built with LlamaIndex. This setup enabled the AI assistant to produce more informed and context-aware responses by using project data such as code diffs, related source code files, and

⁴<https://github.com/BearPays/code-review-assistant-ui>

⁵<https://github.com/BearPays/code-review-assistant-back>

⁶<https://platform.openai.com/docs/models/o4-mini>

associated feature requirements (Jira tickets). The RAG index had to be manually prepared and indexed before experiments, ensuring full control over what data was available to the model in each experimental session.

As highlighted both by the literature and the Phase 1 interview results, lacking a broader repository context can lead to superficial AI feedback that overlooks critical design or architectural concerns. This RAG setup ensured that the LLM can reference deeper project-level information on demand. This setup not only enhanced the AI assistant’s capacity to generate context-aware suggestions but also directly targeted the gap in existing tools and recent studies on the subject.

The artifact implemented the two distinct interaction modes, described in Section 4.5.2, to allow controlled comparison of AI initiative.

While not intended to be a production-level system, the artifact was designed to be realistic and usable enough to engage developers meaningfully. It supported live interaction through a chat interface, maintained session-specific context, and allowed researchers to configure session parameters such as interaction mode and PR data source.

A full specification of the functional requirements that guided the implementation is listed in Appendix A.3. These requirements informed the technical architecture and provided a framework to validate the implementation.

4.5.3.1 System Architecture and Technology Choices

The artifact was composed of three main components: a web-based frontend, a Python-based backend, and a RAG infrastructure. This architecture was chosen to balance flexibility, modularity, and experimental control.

The **web-based frontend** was implemented using Next.js⁷ with React components. The interface enables users to:

- Select a review mode (Mode A or Mode B).
- Choose the pull request (PR) dataset to be used for context.
- Interact with the AI assistant in a live chat format.

This design aimed to emulate real-world developer tooling while preserving experimental control. Screenshot of the user interface is shown in Figure 4.2.

⁷<https://nextjs.org/>

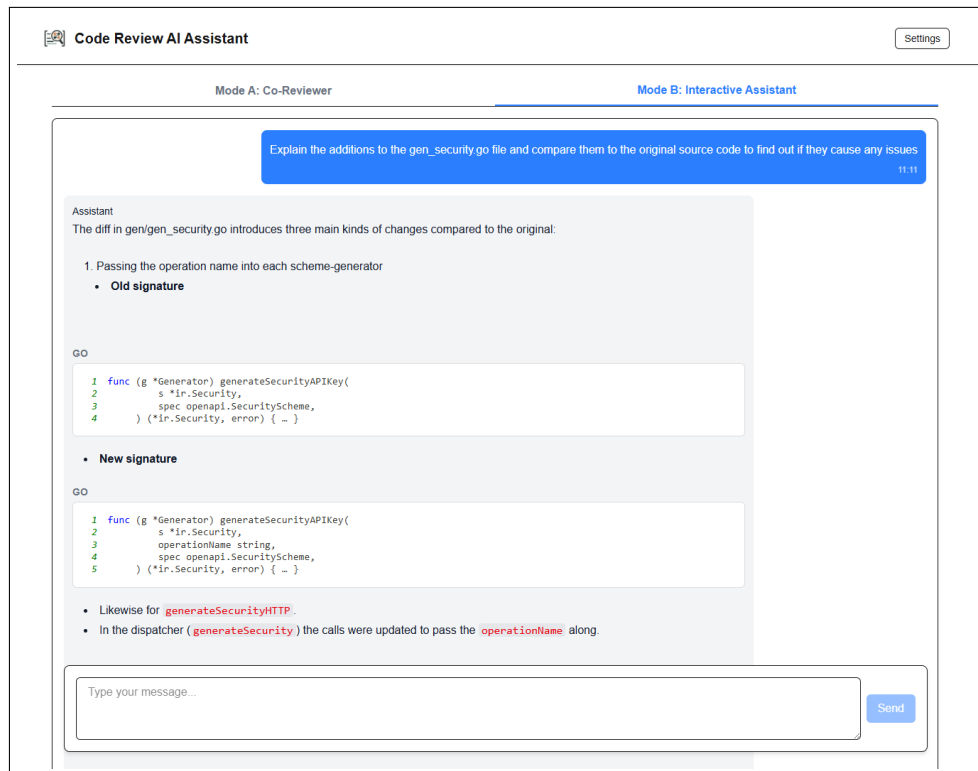


Figure 4.2: Screenshot of the chat interface in Mode B, reviewing a pull request from an open-source project at <https://github.com/ogen-go/ogen/pull/1440>.

The **backend** was implemented using Python and the FastAPI⁸ library. That offered a lightweight and modular web server for managing requests, sessions, and interactions with the AI assistant. The assistant was implemented using OpenAI’s o4-mini language model, accessed via API. The decision to use a cloud-hosted model instead of deploying a local model with frameworks like Ollama was based on the cloud models’ superior performance and the hardware limitations that made achieving similar performance locally impossible. Additionally, deploying local models would introduce additional complexities in terms of infrastructure, maintenance, and compliance. OpenAI was selected as the provider since it is the chosen provider of WirelessCar for this type of AI service.

The model o4-mini was selected due to its performance and efficiency. The decision to use a reasoning model was made so that the assistant could make informed decisions on where to look for information and extra context when performing code reviews and answering users’ queries. OpenAI’s cost-efficient reasoning model, o4-mini delivers impressive results, particularly in tasks involving mathematics, coding, and visual analytics. It achieves good performance for its size and cost, making it suitable for applications requiring fast and affordable reasoning capabilities [34]. To support embedding and retrieval, the artifact also used OpenAI’s text-embedding-3-small⁹ embedding model. This model is optimized for gener-

⁸<https://fastapi.tiangolo.com/>

⁹<https://platform.openai.com/docs/models/text-embedding-3-small>

ating semantically rich vector representations of contextual documents. At the time of writing, it ranks among the best-performing embedding models offered by OpenAI, while maintaining a balanced cost-performance ratio [35], making it a suitable choice for the current implementation.

The backend employed a RAG infrastructure built with LlamaIndex to enable context-aware AI behavior. Project-specific data was manually gathered, indexed, and organized into semantically distinct collections. These collections are queried by the AI assistant using dedicated tools managed by LlamaIndex’s ReActAgent¹⁰ framework, utilizing query engine tools.

This modular agentic architecture was chosen to support the study’s focus on AI initiative and context-awareness. It enables an agentic setup in which the assistant engages in a reasoning process, dynamically deciding what contextual data to query and how to use it based on the evolving dialogue.

4.5.3.2 Data Preparation and Indexing

To enable context-aware AI behavior, the system relied on pre-built vector indexes constructed from project-specific review data. For each pull request under review, the researchers manually gathered three types of context documents: PR metadata and diffs, associated source code, and the feature requirement (in the current research environment, a Jira ticket). These were organized under a dedicated folder structure and indexed using a custom script¹¹ built with LlamaIndex and ChromaDB¹² to create and store the vector databases. Each context type was indexed into a separate vector collection, allowing the assistant to reason over semantically distinct sources through dedicated retrieval tools. Code documents were processed using language-specific splitting to preserve syntactic and semantic boundaries, improving retrieval accuracy. Additional metadata, such as file names, paths, and language, were stored with each chunk, enabling more precise and explainable retrieval. The resulting indexes were persisted locally and loaded at runtime by the backend, ensuring consistent and reproducible context across experimental sessions.

4.5.3.3 Agentic Structure and Prompt Engineering

This section describes how the AI assistant was structured as an agentic system and how prompting strategies and tool configurations were used to shape the assistant’s behavior. The aim was to enable the assistant to behave in contextually appropriate ways under two experimental modes, while maintaining control for research purposes. This design was informed by the conceptual foundation of agentic LLMs outlined in Section 2.6, particularly the emphasis on dynamic reasoning, tool-augmented behavior, and structured control flow. To implement these capabilities

¹⁰https://docs.llamaindex.ai/en/stable/examples/agent/react_agent_with_query_engine/

¹¹https://github.com/BearPays/code-review-assistant-back/blob/main/scripts/index_data.py

¹²<https://www.trychroma.com/>

in practice, the assistant used LlamaIndex’s ReActAgent, which operationalizes the ReAct framework introduced in Section 2.6.2. This choice provided a structured reasoning loop in which the model could not only decide when and how to use tools, but also generate well-justified responses by reasoning over user queries in combination with retrieved context and intermediate findings.

Tooling Design

The assistant interacts with three core semantic tools:

- `search_pr`: accesses PR diffs and metadata.
- `search_code`: provides the full, unmodified source code of the repository.
- `search_requirements`: contains the feature requirement (the Jira ticket) motivating the PR.

In Mode A (Co-Reviewer), a fourth tool, `start_review`, was added. This tool contained a sub-agent that was designed to perform an initial, structured code review based on the full PR data and guided by a detailed review-specific prompt. Unlike the main agent, this sub-agent did not use `search_pr`, as all PR data was injected into its initial context via a prompt. This ensured that the agent considers everything in the PR data and examines each file change. By retrieving the PR data via a query engine, the agent might not consider all the data as required when generating a complete code review of all changes. The agentic structure for this setup is shown in Figure 4.3. The simpler tool structure for Mode B (Interactive Assistant) is shown in Figure 4.4.

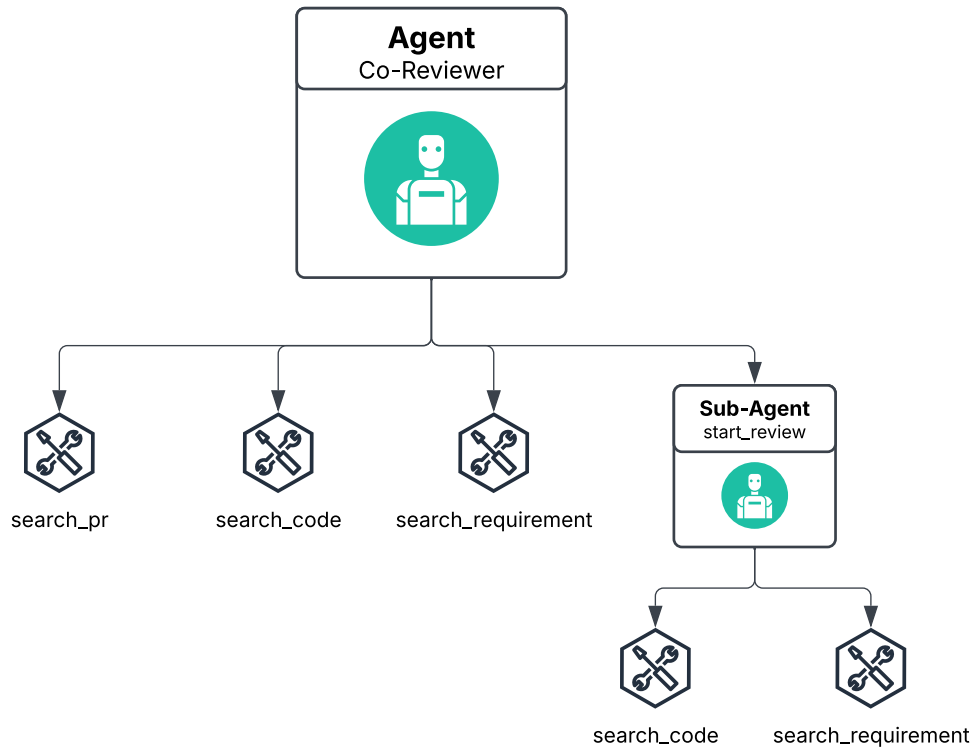


Figure 4.3: Agentic tool structure in Co-Reviewer mode.

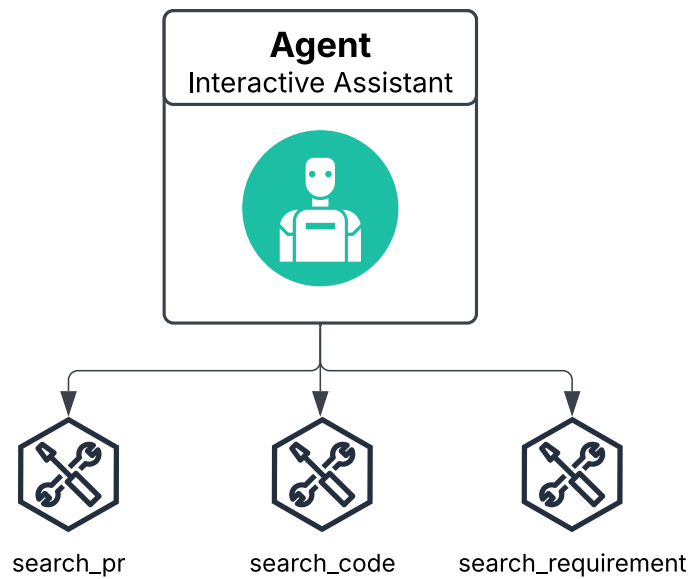


Figure 4.4: Agentic tool structure in Interactive Assistant mode.

Query Processing Flow

Regardless of interaction mode, all tool call queries from the agent were routed through a semantic search pipeline where related data chunks are fetched. The extracted context was then appended to the query and sent to an LLM, where it synthesized an answer to send back to the agent. This process is shown in Figure 4.5.

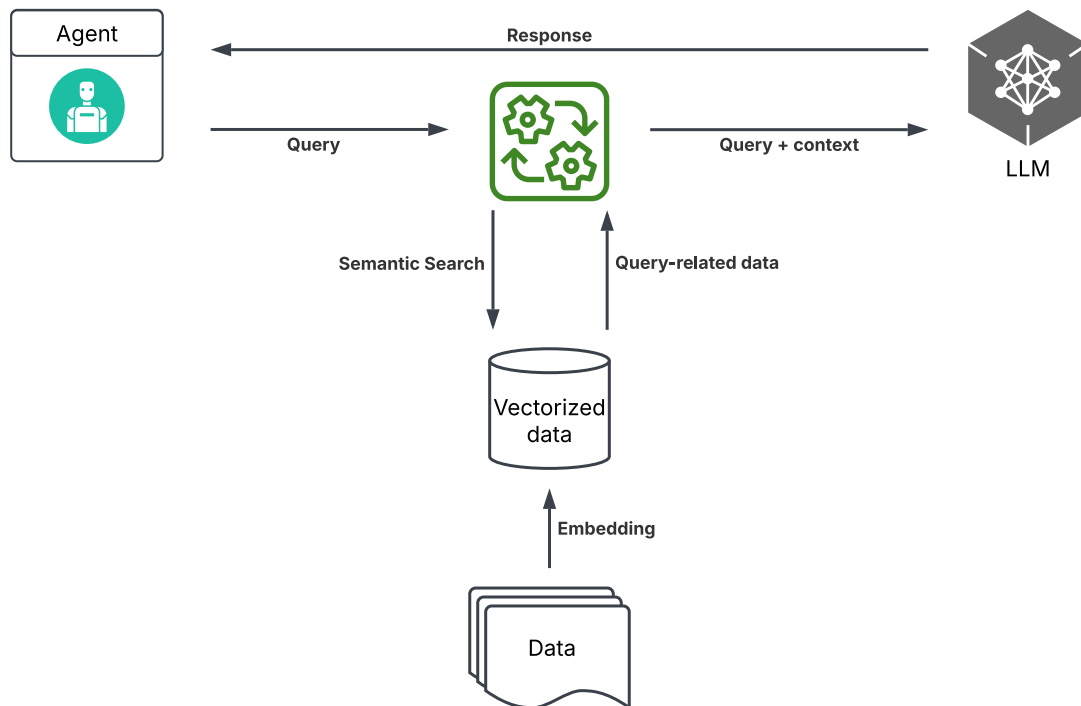


Figure 4.5: Overview of the RAG query processing pipeline.

Interaction Modes

The assistant was also designed to behave differently between the two interaction modes:

- **Mode A (Co-Reviewer):** When a session starts and an initial review is requested, the system invokes the `start_review` tool. This produces an initial high-level review of the PR, covering requirement alignment, file-by-file assessments, and cross-cutting concerns. Subsequent user queries are handled by the main agent using the three core tools (the `start_review` tool should not be invoked in subsequent queries). The assistant was instructed to lead the review and suggest next steps for the human reviewer.
- **Mode B (Interactive Assistant):** The assistant remains passive and only responds when queried by the user. It is instructed to be reactive and not to initiate reviews or propose next steps unless explicitly asked. The assistant always has access to the three general-purpose tools.

This separation allowed for a controlled investigation of AI initiative and whether an AI that takes the lead shapes developer experience differently than one that waits for instructions.

Prompt Design and Reasoning Control

Each agent and sub-agent was initialized with a detailed system prompt that defined its role, objectives, tone, and tool usage strategy. These prompts were tailored to

the goals of each mode:

- The **Co-Reviewer** prompt instructed the agent to lead the review, proactively suggest insights, and conclude each response with next steps.
- The **Interactive Assistant** prompt instructed the agent to take on a reactive role where it responds only when queried and avoids unsolicited recommendations.
- The **Review Tool** prompt (used only in Mode A’s initial code review) provided detailed instructions on how to perform the code review, what to focus on, and how to format its response.

These prompts shape how the agent interprets user input, chooses tools, synthesizes responses, and frames its output. An example of the agent’s thought process, including tool decisions and reasoning flow, can be found in Appendix A.4. This control was designed to maintain experimental validity by minimizing randomness and ensuring that any differences in user experience would arise primarily from the interaction design rather than unpredictable variations in AI behavior. To further reduce variability, the LLM was configured with a temperature of 0.0. This setting minimizes randomness in the model’s responses, ensuring consistent and deterministic outputs across sessions and participants. This was especially beneficial in the current use case, where the assistant was expected to perform logic-based reasoning tasks that demand consistent, factual, and reproducible output.

An excerpt from the Co-Reviewer system prompt is shown below:

System Prompt

You are an expert AI code reviewer tasked with guiding a human developer through a specific Pull Request (PR) review.

You lead the review—proactively identifying critical changes, pointing out risks or inconsistencies, and helping the developer understand the deeper implications of each modification.

Your main objectives are:

- Generate a thorough, structured summary of the PR when prompted using the `start_review` tool.
- Provide authoritative, detail-rich responses to follow-up questions.
- Offer specific, actionable recommendations that help move the review forward. These recommendations should be purely focused on the next steps that the reviewer should take regarding the review itself and not on what future changes the developer should make to the code.
- Clarify how and why the changes impact the system, its architecture, or its goals.

Additionally, below is an excerpt from the `search_pr` tool instructions:

Tool Description Prompt

Provides detailed information about file diffs, changes, modifications, and other information about the PR being reviewed. **This is THE ONLY tool that can answer questions about code changes in the PR.**

When to use:

- Queries about specific diffs or code changes
- Investigating additions, removals, or logic modifications
- Questions about what files have changed in a PR

4.5.3.4 Summary

The implemented artifact fulfilled the functional requirements defined in Appendix A.3, offering a controlled, reproducible environment for comparing interaction styles in AI-assisted code review. While the tool was not optimized for scalability or production use, it was explicitly designed to support structured experimentation and interpretability. The reliance on manual context indexing and preconfigured session data reflects the prioritization of experimental control over automation. Overall, the artifact served as a purpose-built research instrument that enables investigation of developer-AI collaboration.

The majority of the implementation work was conducted with the assistance of AI coding tools such as GitHub Copilot and Cursor¹³. Those tools were extensively used to generate boilerplate code, suggest completions, assist with debugging, and improve documentation and code style. While the final architecture and logic were designed by the authors, AI tools contributed significantly to the development efficiency and iterative design process.

4.5.4 Data Collection

To explore developer experience with the AI assistant during code reviews, two data collection methods were employed to capture user perspectives and behavioral interaction patterns. The primary data source consisted of post-interaction interviews with each participant. These were supported by a secondary data source consisting of researcher observation notes recorded during review sessions.

Participants were recruited through convenience sampling [32], following the same approach as used in Phase 1. All individuals who participated in the earlier phase of the study were invited to return for Phase 2. However, only five were available and agreed to participate again. To reach a total of ten participants, five additional individuals were recruited via internal Slack channels, where the study's purpose and structure were briefly described. Of the ten participants, four belonged to the team responsible for the pull requests used in the experiment, while the remaining six were from other teams. Table 4.3 provides an overview of the participants, including their roles and team affiliations. The code used in the experiment originated from Team A, which is the team associated with the familiar participants.

¹³<https://www.cursor.com/>

Table 4.3: Overview of Participants for Phase 1 & 2

ID	Role	Team	Phase 1	Phase 2
P1	Quality Assurance Specialist	Team A	✓	✓
P2	Application Developer	Team A	✓	✓
P3	Software Engineer	Team A	✓	✓
P4	Software Engineer	Team A	✓	
P5	Security Engineer	Team B	✓	✓
P6	Software Engineer	Team C	✓	
P7	Software Engineer	Teams D & B	✓	✓
P8	Software Engineer	Team E		✓
P9	Software Engineer	Team A		✓
P10	Software Engineer	Team E		✓
P11	Software Engineer	Team F		✓
P12	Software Engineer	Team G		✓

The participant group varied in experience levels, time spent at the company, and on their current teams, as well as the sizes of the teams they belonged to. This diversity provided a broad view of how developers with varying backgrounds and organizational contexts perceive and interact with AI assistance in code reviews. As noted in Phase 1, prior research suggests that thematic saturation typically occurs within 6–12 interviews in qualitative studies [33]. Given that this study focuses on evaluating interaction patterns and perceived usefulness of an AI assistant in a well-defined organizational context, a sample size of ten was deemed sufficient to bring out common themes and contrasting perspectives between participant groups.

The primary data collection method consisted of semi-structured interviews conducted directly after each participant completed the experimental sessions. The interviews were held in English and lasted approximately 10-20 minutes. This format allowed for consistent coverage of core questions while also enabling follow-ups on each participant’s unique experience. The interview guide was designed to extract information about how participants experienced the AI tool, how it compared to their traditional code review process, their opinions on the two different interaction modes, perceived risks or limitations, and improvement suggestions for the tool. The questions were open-ended to promote reflection and discussions. The interview questions can be seen in Appendix A.2.1. The use of interviews as the primary data source is grounded in established user experience evaluation practices. As reported in a recent study, interviews are one of the most frequently used methods for user experience (UX) evaluation. The study highlights that interviews are both practical and impactful, with statistically significant associations with positive UX outcomes [36].

Before each session, participants were informed that the study focused on evaluating the AI assistant, not individual performance, and were encouraged to respond honestly. They were told there were no right or wrong answers and were assured of anonymity in both observation notes and interview transcripts. During each session, the researchers unobtrusively observed participant behavior and recorded informal notes. These notes focused on notable behaviors during the session, such as moments of hesitation, visible frustration, or engagement. While not systematically structured, the notes served as reflective memos to support the interpretation of the interview data. As with interviews, observational methods are also highly regarded in UX research and are identified as one of the top five most used UX evaluation methods [36].

4.5.5 Data Analysis

Data collected in Phase 2, including semi-structured interviews and researcher observation notes, was analyzed using the same thematic analysis methodology as described in Section 4.4.3, following the six-phase process of thematic analysis [30]. Both interview transcripts and observation notes were treated as components of a shared analytic data set. Initially, the two data sources were coded separately to preserve the unique context of verbal and behavioral data. As in Phase 1, coding was conducted at a semantic level [30], focusing on the explicit meaning and experience expressed in interview transcripts and demonstrated through observed behaviors. The analysis also adopted a realist stance, assuming that participants' words and actions reflected their actual experiences and interactions with the AI assistant. It took a primarily inductive orientation, allowing themes to emerge from the data without being constrained by a pre-defined coding framework. However, it was also informed by the research focus areas, such as exploring differences between interaction modes and identifying potential design implications.

Interview transcripts were produced using the same procedure as in Phase 1. Audio recordings were transcribed with Otter AI, then manually reviewed and corrected for accuracy and readability. Familiarization with the data was performed through multiple close readings of each transcript, where initial observations and code ideas were noted.

Observation notes were collected by researchers during each review session in the form of short notes. These notes were informal and unstructured, but were treated as qualitative data complementary to the interview transcripts. They were first reviewed independently from the interview data and then coded using the same semantic thematic approach. An example of how a quote was coded and connected to a theme is shown in Table 4.4.

Once the initial codes had been generated from both data sources, they were integrated into a single analytic space. Themes were then developed across this combined dataset. To ensure analytic precision, two researchers collaboratively conducted all coding and theme development through multiple iterations of review and refinement as in Phase 1. Candidate themes were assessed for internal con-

sistency, distinctiveness from one another, and relevance to the study's objectives. Themes that were conceptually overlapping or poorly supported were merged, split, or discarded as appropriate. Final theme names and descriptions were defined to accurately reflect their scope and the content they represent, and supporting quotes or behavioral descriptions were selected to illustrate each theme in Section 5.2.

Table 4.4: Example Coding of a Quote from Phase 2 Transcripts

Quote	Code	Theme
<i>"I think it has the potential to speed up and make the review process more efficient and effective, like improving the results [...] if the tool works."</i> [P5]	AI speeds up code review	Efficiency and Thoroughness

5

Results

This chapter presents the results of the two-phase empirical study conducted. Phase 1 involved a field study aimed at understanding current code review practices and identifying opportunities for AI integration. Phase 2 consisted of a field experiment evaluating developers' experiences with an AI-assisted code review tool. The results from each phase are organized into thematic sections, supported by representative quotes and analysis.

5.1 Results from Phase 1

Interviews that were conducted with several developers across several teams at WirelessCar gave good insight into their code review process. The interview results highlight key challenges that developers face during code reviews, while at the same time revealing potential opportunities for possibly improving the process with AI assistance. The following will state the results that emerged from the thematic analysis of phase 1 (the field study). Six themes emerged from the thematic analysis of the qualitative data and can be seen in Table 5.1. Additionally, all themes with their associated codes that were identified during the thematic analysis can be seen in Appendix A.1.2

The following sections will discuss the themes and the results that emerged systematically. Section 5.1.1 will discuss the first three themes to describe the typical code review process at WirelessCar, along with how developers collaborate, evaluate code, and share knowledge during reviews. Section 5.1.2 discusses the code review challenges identified, Section 5.1.3 the current AI adoption, and finally, Section 5.1.4 the possible AI adoption in code reviews mentioned by the interviewees.

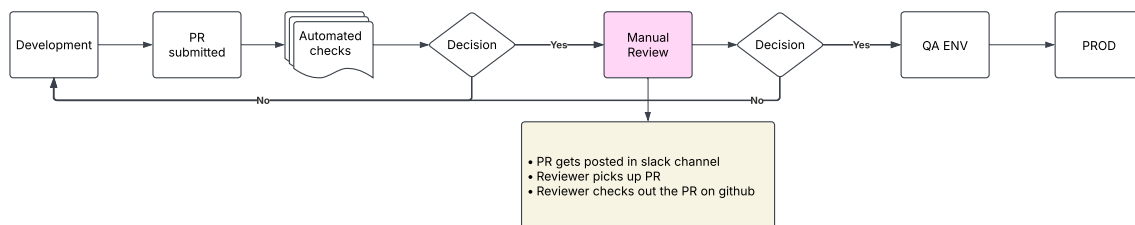
5.1.1 Observed Code Review Process

When reviewing the *informal review process at WirelessCar and their practices*, it appears to be similar to the typical MCR process previously described. Interviews with multiple developers reveal that most reviews are initiated when a developer creates a pull request (PR) in GitHub. Those PRs are also commonly posted in a dedicated Slack channel (their preferred tool of internal communication) to notify team members that the code is up for review. This aligns with the previously dis-

Table 5.1: Identified themes and their descriptions from analysis of interview data from Phase 1.

Theme	Description
Informal Review Process and Practices	Describes how teams coordinate and manage code reviews in practice, including informal communication, tool use, and the absence of structured processes or metrics.
Review Strategies and Evaluation Focus	Captures what developers focus on during the actual code review process.
Learning, Knowledge, and Review Expertise	Explores how code reviews serve as opportunities for learning and knowledge sharing within teams. It also captures the role of reviewer expertise in conducting effective reviews and the challenges that arise when reviewers lack sufficient understanding of the code-base or architecture.
Code Review Challenges	Identifies recurring challenges and inefficiencies encountered in the code review process.
Current AI adoption	Describes the current state of AI tool usage in development and code review.
Possible AI adoption in code reviews	Describes developers' expectations, suggestions, and concerns regarding future AI assistance in code reviews.

cussed asynchronous, tool-supported nature of MCR practices. Figure 5.1 illustrates WirelessCar's typical code review workflow.

**Figure 5.1:** Code review workflow at WirelessCar

In most teams, there is no strict reviewer assignment system. Reviews are often “up for grabs”, allowing any available team member to take on the review. Some developers, however, tend to ping specific team members to accelerate the review process, especially when dealing with urgent changes or large PRs. This highlights

how direct communication is preferred to ensure quick reviews. As one developer explained:

“I usually ping someone on Slack to have a look, because that’s quicker... You tend to drown in those automatic notifications” [P7]

Before the manual review, the code usually undergoes static analysis, which includes linting and formatting checks. Static analysis tools, like CodeQL¹, are employed to detect vulnerabilities and ensure coding standards. Teams rely on such tools to handle basic quality issues and identify them before the code reaches manual review. Reviewers then tend to rely on their experience and intuition when reviewing and evaluating code quality. They do not tend to follow a formalized checklist when conducting the review, but this also tends to vary a lot between development teams:

“I know that some teams have a to-do list that’s pre-populated in their code reviews with check marks. But we don’t, and we don’t have any predefined thing to check, but it’s also always the same.” [P5]

Some teams implement a two-reviewer rule for larger or critical changes, while smaller PRs usually only require one reviewer’s approval. Reviewers then confirm their approval directly in Slack or by marking the PR as approved in GitHub. Once approved, PRs are merged, and the deployment process often initiates automatically. Some teams also rely on an internal tool to run integration tests in staging environments, where successful results from these tests indicate a stable release, which may then proceed to production. No formal metrics are collected regarding code reviews that track their outcomes, such as defect rates, reviewer satisfaction with the workflow, or the success rate of code reviews. Some teams rely on automated test results, but there are no formal KPIs for assessing the review effectiveness.

When the *review strategies and evaluation focus* of the reviewers are investigated, it is apparent that most reviewers prioritize reviewing the core logic of the code and sometimes pay less attention to infrastructure changes or tests unless they appear critical. Some interviewees, however, expressed that they review tests, file structure, and coding standards as well. One specialist mentioned:

“I’m a quality assurance guy. I also tend to look into it and see, oh, this is not how it is supposed to look like or this is a coding practice we should have followed.” [P1]

In addition, developers also mentioned checking the acceptance criteria during code reviews to ensure that the changes meet the stated requirements.

When diving into the *learning, knowledge, and review expertise* when it comes to code reviews, it is clear that the interviewees value the code reviews. Five participants state that their way of introducing their codebase to a new team member is via hands-on experience, as one developer put it:

¹<https://codeql.github.com>

“I would say we mainly try to push the person to work with everything in the system to get this understanding. So I would say the experience is the way to go [...] But as a developer, I would say you need to work with the codebase to really get the proper understanding of it anyway.” [P3]

Code reviews support this way of learning since the interviewees mention the code review to be a valuable tool for knowledge sharing, one developer even states that as the main purpose of code reviews:

“I mean, that’s not really why we’re doing the code review, not to find the bugs, because that’s impossible [...] In reality, what you do is that you spread knowledge about changes. You also spread knowledge about the repositories [...] You probably get to learn about things that you don’t really know about.” [P7]

Some developers also stated that they invest time in code reviews and value thoroughness over speed when reviewing. Additionally, they expressed that having a broader architectural knowledge of the system or codebase improves the quality of the review, where the majority of bugs are often traced back to a lack of understanding of the whole architecture.

5.1.2 Common Challenges in Code Reviews

One of the most frequently mentioned challenges across the developer interviews was the issue of delayed reviews. Situations were described where PRs remained unreviewed for extended periods, often requiring repeated reminders for a reviewer to take action. One interviewee described it as:

“Sometimes you need to ping people more often, and sometimes the PR is very big, so people don’t dare to pick it up.” [P2]

Since PRs are not assigned to a specific reviewer, it can become a problem if no one takes the initiative and “picks up” the PR.

Four interviewees reported difficulties regarding reviewing large or complex PRs. PRs that combine new features, refactoring, and changes to infrastructure often become overwhelming. They note that this can result in more superficial reviews or longer delays.

Context switching was also noted as another major challenge. Three mentioned that the cognitive burden of pausing ongoing work to review code, not at all related to their current work, was challenging. This requires additional time to regain focus, and one interviewee highlighted the time lost cause of this:

“As soon as you need to context switch, even if it’s just a three-minute thing, it’s 20 minutes of lost time.” [P7]

Four interviewees explained that they sometimes lack sufficient context when re-

viewing the code. Sometimes, important details about why the change was made or its expected impact are missing from the PR description. This increases the time it takes for the reviewer to comprehend the PR and effectively point out defects or problems.

5.1.3 Current Use of AI in Software Development

The interviews revealed that common AI tools like GitHub Copilot and ChatGPT are commonly used for development tasks. Interviewees mentioned tasks like writing boilerplate code, assisting with syntax, and quickly generating documentation. One example mentioned was:

“[...] as to help to create the boilerplate stuff, it’s outstanding, right? I mean, you do it in 30 seconds instead of a couple of hours. So I try to use it as much as possible during the development process.” [P7]

However, not all teams are allowed to use AI-generated code or share information (such as source code) with AI tools. Furthermore, the teams that are allowed to use AI tools only utilize them via an enterprise subscription, where the data provided is not used for training the models utilized by the tools. This ensures that no data is leaked from the organization via the use of AI tools.

All interviewees who use AI tools reported positive experiences in their development work. They noted that these tools improved the efficiency of their development processes. However, some also pointed out drawbacks, such as AI-generated code sometimes failing to meet the team’s defined standards. One interviewee shared the following experiences regarding both the advantages and drawbacks of AI-assisted development:

“Yeah, I find AI to be great. When you’re writing code, it can provide examples or suggest, ‘Hey, do you want to write this?’” [P5]

“I feel like AI is ruining my colleagues’ unit tests. It suggests stupid unit tests in ways that my colleagues don’t understand [...] That’s a very common thing that AI suggests weird unit tests that don’t really make sense or don’t really follow the pattern that we want it to follow.” [P5]

None of the interviewees reported using AI tools as part of the formal code review process. One interviewee did mention that some reviewers might occasionally paste code into tools like ChatGPT for clarification or explanation during reviews. However, beyond such informal use, AI has not been formally integrated into the code review workflow in any of the interviewed teams.

5.1.4 Potential AI Use Cases in Code Review

The interviewees expressed interest in potential AI integrations with the code review processes. They mentioned features like summarizing PR changes and accompanying

descriptions, where AI could generate a concise summary and help reviewers to quickly understand the intent of a code change. One interviewee mentioned:

“When you create the pull request, an AI bot could say, ‘Hey, you’re trying to achieve this, do you want this as your summary or description?’”
[P5]

Interviewees also mention possibilities such as AI assisting in validating whether code meets stated requirements. Additionally, interviewees highlighted the potential for AI to detect hidden bugs or vulnerabilities, such as race conditions, dependency issues, or other subtle defects that human reviewers might overlook. As one developer put it:

“An AI would probably be able to identify a race condition, for instance, which, as I said, is an example that’s borderline impossible to catch on the fly. In three minutes, you’re never going to find that.” [P7]

Finally, interviewees mentioned potential drawbacks of integrating AI into the code review process. These were mostly concerns about security risks and false positives, which could reduce the reviewer’s trust in the AI assistant and divert their attention from real issues. As one interviewee put it:

“The problem with those kinds of checks is that, if they’re not good enough, you stop reading them. We see that all the time, you get flooded with false positives, and then you miss the real issues because you start ignoring the feedback.” [P7]

5.1.5 Summary

The key findings from Phase 1 are summarized below.

- The analysis in Section 5.1.1 describes the typical PR lifecycle at WirelessCar, including the use of GitHub, Slack, static analysis tools, informal reviewer selection, and deployment automation.
- Section 5.1.2 identifies major pain points such as unassigned PRs, delays, large review scope, context-switching overhead, and lack of clear review ownership.
- The results revealed an absence of formal metrics for review success and no standardized KPIs across teams.
- Developers emphasized the value of architectural understanding for effective reviews and highlighted code review as a learning and knowledge-sharing process. This was especially evident in how new team members were onboarded.
- The field study detailed current uses of AI tools like Copilot and ChatGPT in development, while also capturing developer perceptions of their potential for code review.

- The results documented the tools, practices, communication channels (Slack), and CI/CD pipeline integration that shape the context in which reviews occur.

The identified challenges and opportunities served as the empirical basis for designing the LLM integration strategies evaluated in Phase 2.

5.2 Results from Phase 2

This section presents the results from the second phase of the study, which involved a field experiment evaluating developer experiences with an AI-assisted code review tool. Semi-structured interviews were conducted with developers after they completed code reviews using two different interaction modes with the AI assistant: Mode A (Co-Reviewer) and Mode B (Interactive Assistant). Observation notes were also collected during each review session to complement the interview data.

As in Phase 1, the interview transcripts were analyzed using reflexive thematic analysis [30]. Table 5.2 provides an overview of the themes along with descriptions of the themes.

Table 5.2: Identified themes and their descriptions from analysis of data from Phase 2.

Theme	Description
Accuracy, Reliability, and Trust	Focuses on the perceived correctness of AI-generated feedback, concerns about over-reliance, and varying levels of trust in the assistant’s recommendations.
Efficiency and Thoroughness	Captures how the assistant affects review speed, cognitive load, issue detection, and the overall thoroughness of the code review process.
Integration Expectations and Limitations	Highlights developer expectations for seamless integration, responsive design, and context-aware suggestions, while also surfacing frustrations related to current UX and tooling limitations.
Usage Contexts and Interaction Patterns	Describes how interaction with the assistant varied based on review context, including preferences for different modes, alternative usage strategies, and team-specific practices.

This section presents one subsection for each theme. Each theme captures a distinct pattern across participants’ experiences and perceptions of the AI-assisted review process. Within each theme, it is examined how it manifested across the two interaction modes, supported by quotes and observational insights. The analysis highlights the opportunities and challenges of integrating AI into code review workflows.

5.2.1 Accuracy, Reliability, and Trust

Participants frequently commented on the accuracy of the assistant’s feedback and how this influenced their trust in the tool. Several interviewees described the assistant as generally accurate and capable of identifying relevant issues. For example, one participant stated:

“As I see it, they were quite accurate [...] It was quite nice, not all of them, but a lot of them.” [P9]

Another participant noted that the assistant surfaced an issue he had already planned to comment on himself:

“It was good that it reminded me to write tests that didn’t exist [...] I would normally write this comment myself anyway.” [P11]

In multiple cases, the assistant’s output was described as confirming the developer’s own thoughts or surfacing something they might not have otherwise caught. Observation notes also reflect this, with one session noting that “the user said the AI caught exactly what he was looking for in a certain file.” In another session, the reviewer remarked that “the summary gave something that he would not have seen.”

Several participants also reported instances where the assistant produced incorrect or unclear suggestions. One participant questioned whether the tool was even “doing what it was asked,” while another described the assistant incorrectly flagging a missing import. In some sessions, the assistant hallucinated requirement lists that were not grounded in the requirement data or flagged nonexistent issues based on incomplete context. Observation notes also recorded a case where the assistant’s explanation referenced something the participant had not stated, such as replying “Exactly as you observed...” to the user’s question, despite the user having made no such observation. This could confuse participants, and one noted:

“Sometimes it says some slightly strange things.” [P8]

When asked about concerns, participants expressed different perspectives. Several participants warned of the risk of over-relying on the assistant, especially in Mode A, where the assistant led the review:

“It feels like I might get a bit colored by getting the improvements from the LLM [...] I feel like maybe I could miss something else, because I would focus on those improvements a lot.” [P3]

While others mentioned more general risks of using AI:

“I mean, it’s always risky when you rely on AI tools.” [P5]

“Of course, if you trust it and it’s wrong [...] I mean, the number of times I’ve ended up talking to ChatGPT about a problem — I’ve had to say, ‘You’re wrong, it doesn’t work like that,’ and then it changes its

mind.” [P7]

However, not all participants saw the assistant as risky. Some viewed it as a low-stakes addition to the workflow, especially when its use remained optional:

“As long as there’s an opt-out option, there’s no real harm in it.” [P5]

Finally, some participants saw the assistant as offering a net benefit even if it wasn’t always accurate. One reflected:

“If we miss 10 [issues] today, we might miss two with a tool like this.”
[P9]

The role of trust emerged as a key factor in shaping how participants viewed the tool. A few emphasized that for the tool to be useful or even used in general, it must be trusted, but not blindly. One participant said this:

“So I think that worked really well, especially with larger [pull requests] [...] I mean, you can at least use it if you trust it.” [P7]

Another warned that misplaced trust in the tool could lead to wasted effort if the tool isn’t accurate:

“I could go on for hours, just to realize I can never do this. Then I’ve just lost a few hours trying to pursue something that wasn’t possible.”
[P5]

Observation notes indicated that the assistant occasionally failed to retrieve sufficient context before responding, either by prematurely concluding its reasoning process or by fetching only partial content from relevant code files. In some cases, this meant that critical parts of a file were omitted, limiting the assistant’s ability to generate accurate answers. In at least one instance, a tool reported that a file did not exist when it should have been accessible. These inconsistencies were rare but noticeable, and in a few cases, the assistant returned internal reasoning steps or speculative reflections not intended for user display. Such behaviors occasionally led to participant confusion and possibly undermined their confidence in the assistant’s reliability.

5.2.2 Efficiency and Thoroughness

A strong theme was the AI’s impact on code review efficiency and the thoroughness or quality of reviews. Developers reported that the AI assistant could speed up the review process, reduce reviewers’ workload on tedious tasks, and potentially catch more issues, although some participants remarked that it sometimes focuses on low-priority findings, introducing noise.

Multiple participants described the assistant as helping to speed up reviews. One participant explained:

“I think it has the potential to speed up and make the review process more efficient and effective, like improving the results [...] if the tool works.”
[P5]

Others framed the benefit more directly in terms of time savings:

“It would save a lot of time.” [P9]

“It has a lot of potential [...] it makes a lot of things easier and faster.”
[P8]

Beyond efficiency, participants also pointed to improvements in review quality. Some noted that the assistant could identify issues that might otherwise go unnoticed:

“There are probably findings you get in the report that you don’t find when you do it manually.” [P10]

Additionally, as previously mentioned, participants sometimes confirmed during the session that the assistant had identified a problem that they acknowledged they likely would not have caught themselves, thereby contributing to their review thoroughness.

Others felt that the assistant would be particularly helpful on large pull requests where it would be difficult for a human reviewer to catch everything:

“It’s taken someone two weeks to write it [...] giving it 15 minutes, you won’t have a chance to understand it, at least not well enough to find the hard stuff. I would imagine that the tool would actually raise a flag for a potential deadlock or race condition as well.” [P7]

“I find a review not very accurate [...] people will approve the PR, but like one month later we realize that it was not correct [...] In this situation, these tools, I think, will help us a lot.” [P2]

For some participants, the assistant reduced the effort involved in reviewing by removing the need to search through the codebase or external documentation:

“I like that it helped me not need to go into the actual source code or to my IDE and look at the code or follow references so I understand it, because I could ask a question and it kind of follows the references and then gives me a report of it, and that will help.” [P10]

“But I really like the integration with the requirements part, because if I open up [a PR] and I don’t know what it’s about. [...] The first thing I do every time is I open the [Jira] ticket anyway, because I need to see what is supposed to have been achieved here. So I think that’s a really nice functionality to have” [P3]

In some sessions, participants expressed that the assistant could help them with

large or complex reviews that they might otherwise have skipped or postponed. For example, one participant reflected that during a sprint involving a pull request with nearly 100 changed files, no one had reviewed it until the final day. They noted that if the assistant had provided an early summary, they could have started reviewing earlier and contributed comments more proactively.

The tool was also seen by some as a way to make the review process more approachable:

“It’s not the most fun thing to do. To review other people’s work, so it’s nice to have it all laid out.” [P9]

However, some participants noted that the assistant occasionally surfaced low-priority or unclear findings. One participant described the difficulty of sorting through non-essential comments in large pull requests:

“A lot of this feedback might never be looked at anyway. [...] it brings up things that might not really matter.” [P10]

This was also observed during review sessions, where participants sometimes expressed difficulty distinguishing important findings from minor ones, especially in lengthy summaries generated by the assistant.

5.2.3 Design Expectations and Limitations

Many participants shared expectations about how an AI assistant for code review should behave and be integrated. A recurring theme was the desire for seamless integration into existing workflows and tools. Rather than switching to a new interface, several participants expressed that it would be preferable to access the assistant directly from familiar environments like GitHub, Slack, or their IDEs. As one participant put it:

“I think most of the developers don’t want to use something new, like a [new] UI, but rather have an integration to what exists.” [P11]

Building on this, another participant described a preference for having the assistant’s comments embedded directly into GitHub’s interface, with expandable in-line comment boxes, while also having the ability to further ask questions in a chat interface.

One participant described how having it in the pipeline through a Slack bot could be useful.

“Maybe a Slack auto bot could even be triggered on each message [...] and a full review could be dropped as a message under that thread.” [P11]

Beyond integration, participants also critiqued how the assistant’s output was presented. Several interviewees felt that the feedback was overly long or difficult to

scan. One participant noted:

“Maybe it could be structured a bit differently, it was a bit much, maybe.”
[P3]

One participant noted how the answer to his relatively simple query was at the bottom of a long response from the assistant, while another emphasized the importance of the output being specific and concise:

“I mean, what’s important is that it very clearly lists the file. I’d rather have it list the file and the line number and be very specific, in a short way.” [P5]

Response time was another point of friction. While some delays were tolerated, long wait times were cited as a major barrier to adopting the tool in real development workflows:

“I think the speed and accuracy are mainly what need to be improved. [...] I wouldn’t use this if it took, I don’t know, how many minutes it took for it to respond.” [P3]

“I mean, of course, asking it, then I’m waiting three minutes. It’s not a really smooth process for 550 changed files.” [P7]

One participant reflected on whether the quality of interaction also depended on their own ability to ask good questions:

“Maybe my way of asking questions was also wrong. I did not always feel like I got the response that I was asking for. So, I might need to learn how to be more detailed in my questions.” [P10]

Some participants also described instances where the assistant’s suggestions were unclear or lacked justification:

“With the import [not being a correct suggestion] and it not being able to explain itself very well.” [P3]

“I thought the description they suggested was a bit weird as well.” [P8]

Many limitations were traced to a lack of access to broader context, such as architectural documentation, internal conventions, or metadata:

“Ideally, you want to inject as much relevant information as possible [...] like the JIRA ticket, relevant [documentation] pages, the codebase itself, the README, and any similarly named repositories that might be connected to the same service.” [P5]

“If you only look at the PR, and the code is programmed in a way that doesn’t fit the rest of the code, you would miss that, because the PR is

good.” [P9]

Some participants also highlighted that the assistant’s usefulness depended in part on how good the documentation and PR descriptions are to begin with. During the testing, one reviewer reflected that the tool could help keep the documentation up to date by suggesting updates that align with PR changes.

“Of course, it cannot cover every case, because of the cases that are not covered by the requirements.” [P2]

“I think the hard part is doing a good setup in JIRA. ‘What is it that I’m supposed to do?’ And how to give that back to the [AI]?” [P12]

5.2.4 Usage Contexts and Interaction Patterns

Participants expressed a range of preferences, strategies, and situational factors that shaped how they interacted with the AI assistant, often shaped by the review context. These patterns included both the predefined interaction modes: Mode A (Co-Reviewer) and Mode B (Interactive Assistant), as well as emergent workflows that blended or extended beyond them.

Out of the ten participants, six explicitly stated a preference for Mode A, typically citing its ability to provide an immediate, high-level summary that helped them orient themselves within the pull request. Two participants preferred Mode B, with one of them still noting situations where Mode A would be more suitable. The remaining two participants were unable to express a clear preference, instead highlighting that the optimal mode depended on the specific task or context.

Participants found Mode A especially helpful for getting oriented in a pull request. They described the high-level summaries and suggestions provided at the beginning of the review as useful for gaining quick context, particularly in unfamiliar or complex codebases:

“I prefer this one [Mode A] where you actually get the overview directly [...] it had a lot of good pointers, that it already found.” [P12]

“The first engine [Mode A] that gave me a breakdown of everything [...] that was quite clever, and I would gladly use that.” [P7]

“The first part of the AI assistant that you built helped me to understand what this PR is all about, without me going through all the features, theoretical set commands, and status.” [P11]

“Yeah, I think the like, Mode A, or even both modes would help you get faster into the code, and help me understand the PR faster, especially if you can just read the overview.” [P8]

Mode A was also described as particularly useful for low-risk PRs, or when partici-

pants were unfamiliar with the codebase or the team:

“Let’s say the change is relatively small and it’s not causing any risk, then I would definitely go with the first one [Mode A], where I let AI do most of the work.” [P11]

“In this specific instance, because I didn’t have any domain knowledge. Then it was really good to have it, just out of the box, summarize everything and be like, ‘Hey, you should look at this.’” [P5]

Many participants saw the assistant, especially Mode A, as valuable for newcomers:

“I think if I were in a new team, and I am unsure what is happening, then it could be really good to start with a summary.” [P8]

“If I were sent into a new team and met a situation like this, it would be very beneficial. [...] optimally, I would be able to also talk to the person [code author], but perhaps I would have a better understanding before talking to that person, if I could read through the AI summary.” [P5]

“Yeah, if you can write questions like ‘What is this?’ or ‘What is this really about?’, it could also be a very good tool to get to know the codebase and to learn as a new guy.” [P9]

Additionally, some saw mode A as useful in teams where code review standards are lower or in teams that prefer other methods for reviewing code, such as pair programming. In such cases, the assistant would serve as a fallback mechanism.

“I think it would be great for those who usually just skim through and say, ‘It looks good to me’. [...] I think the biggest effect would be for those developers, I guess, and those teams.” [P5]

“Sometimes people are just, like ‘Ah, review PR, approve, whatever.’ So then, something would be better than nothing.” [P8]

As stated, Mode B was less preferred in general, but some participants remarked that in cases where they are already familiar with the codebase, for higher risk PRs, or if they wanted to maintain full control over the review process, they would prefer Mode B:

“I prefer the second one, mostly because I like to read code, and I’m good at it. [...] I feel like I can have more control when I look at the code.” [P10]

“But if it’s in some codebase I already know, some codebase where we have a lot of experience and have worked in it a lot. It could probably be nice to have [Mode B].” [P8]

It was also observed during the sessions that some participants felt inclined to ask

Mode B to provide similar summaries as Mode A.

In some cases, participants preferred a combination of both modes or expressed that the preferred interaction mode depended on the situation:

“I think I’m 50/50 [...] both are useful. One is on demand, the other one is on its own.” [P1]

Several participants described additional usage patterns not strictly defined by the study design. For instance, some saw mode A as useful for the author before submitting the PR, rather than during review:

“I feel it might not be as much of a review help. I think it might be a pre-review help.” [P10]

“but some smaller issues [...] It should already be feedback to the programmer, not in a PR.” [P9]

Others described an approach where they conducted a human-led review first and then used the assistant to validate or catch anything they might have missed:

“You start out with it just to sum up what the code is doing. Then I look for issues, and then I can ask, ‘Are there any further issues?’” [P3]

“Sometimes it should maybe just say, ‘Do it yourself this time’ [...] and when you’ve brought your report, I’ll bring mine.” [P9]

Participants also frequently noted that Mode A was especially helpful for large PRs:

“Especially for large PRs, it’s nice to get the breakdown on what’s happening [...] because usually, you always have to do that sort of manually anyway.” [P3]

“The bigger the PR, the more relevant the AI tool is, I guess, because humans are lazy, tired, and we don’t have the time or the energy to go through all those lines of code.” [P5]

However, there was also some uncertainty about how effective the assistant would be at scale. One participant expressed concerns about the assistant’s ability to handle very large codebases or complex business logic:

“I think it’s going to be a bottleneck for such things, because there will be so many moving parts in it, so much business logic going around.” [P1]

5.2.5 Summary

The following summary synthesizes the key insights across the themes identified in Phase 2.

- Participants generally perceived the assistant’s feedback as accurate, though occasional errors or inaccuracies led to hesitation or reduced trust in its suggestions.
- Many developers found the assistant useful when it comes to accelerating the review process and increasing thoroughness, but some reported that low-priority or unclear findings introduced unnecessary noise.
- Future expectations centered on tight integration with existing tools (e.g., GitHub, Slack) and readable, concise output. Delays in response time and vague or verbose suggestions were common frustrations.
- Although in general, Mode A was preferred by most. The usefulness of each interaction mode was dependent on context. Mode A was preferred in unfamiliar or large PRs, while Mode B was favored when reviewers knew the code base well or preferred full control.
- Beyond predefined modes, participants described emergent usage patterns, such as using the assistant for author pre-review, follow-up validation, or in teams with weaker code review standards.

6

Discussion

This chapter interprets and contextualizes the findings presented in Chapter 5, focusing on how they answer the study’s two research questions and what they reveal about the integration of LLM-based assistants into real-world code review workflows. Drawing on insights from both phases of the study, the discussion reflects on key user experiences, examines the performance of the implemented artifact, and considers broader implications for the use of AI in code reviews. The chapter concludes with a discussion of the study’s limitations and suggestions for future work.

6.1 Addressing the Research Questions

This section revisits the two research questions introduced in Section 1.3, providing a synthesis of findings from both phases of the study.

6.1.1 RQ1

What practices, challenges, and expectations characterize modern code review processes, and where do developers see potential for AI-based assistance?

The research question aimed to understand how software developers conduct code reviews in practice, the challenges they encounter, and their expectations for future AI support. The Phase 1 field study revealed not only how developers perform code reviews in practice but also how they experience recurring breakdowns in coordination, context comprehension, and tool support. These factors shape both the limitations of the current workflow and their expectations for future AI assistance.

The findings revealed a code review culture that varies by teams, is collaborative and team-oriented, but largely informal and lacks consistent structure and performance tracking. The teams largely adopted what has been referred to as Modern Code Review (MCR), a lightweight, asynchronous practice typically supported by platforms like GitHub and Slack [7]. While this reflects a shift from formal inspections toward more agile, developer-driven workflows, the findings revealed that review responsibility was often unclear and informally decided, with no consistent assignment of reviewers. This flexible, ad hoc approach allowed adaptability and was often described as effective despite a limited reliance on formal guidelines. However, this also

caused friction, such as pull requests remaining unreviewed or developers needing to remind others via Slack to take action. This reliance on informal coordination is consistent with prior observations that modern teams often lean on interpersonal dynamics and tool-supported collaboration [10]. Furthermore, the findings from Phase 1 highlight the absence of formal review metrics and quality indicators. While this may reflect a healthy culture of trust, it also raises questions about how review effectiveness is monitored and improved in practice. This lack of formal structure may further pose challenges when integrating AI assistance, as AI systems often depend on structured signals, predefined workflows, or task boundaries to perform reliably and interpretably.

When asked about review strategy, developers typically reported a focus on the core logic of the code. Changes such as configuration files or test scaffolding were often given lower priority unless they were seen as directly impacting correctness. This prioritization seems pragmatic but might also introduce the risk that the important non-functional or architectural aspects may receive insufficient scrutiny, particularly in large or mixed-scope PRs. This reliance on personal judgment over structured checklists aligns with what is commonly found in MCR practices, where the code review is often shaped more by team culture and informal practices than by formal process controls [7]. Several interviewees also described conducting reviews based on a “gut feeling”, which reflects an expertise-driven approach rather than a standardized one. This is likely to be the most effective method for experienced reviewers operating within their familiar codebase. Still, it may, on the other hand, prove fragile when scaling to unfamiliar teams, larger organizations, or more regulated environments where traceability and consistency are necessary.

Prior research has emphasized the social and cognitive dimensions of MCR [10]. The Phase 1 findings confirm this perspective since many developers saw code review as a knowledge-sharing mechanism rather than only a mechanism for enforcing code quality standards. This was especially apparent in how the developers framed code reviews as an important tool when it comes to onboarding new members. One interviewee even emphasized that the primary purpose of the code review is not bug detection, but rather knowledge sharing. These attitudes show that code reviews help teams share and build knowledge, especially in teams where architectural understanding and domain expertise are critical. This also reinforces the idea that context switching during reviews is cognitively demanding and potentially disruptive, a challenge that cannot be easily mitigated by automation alone. These insights further support the argument that AI-based tools should not aim to replace human reviewers but rather augment their capabilities. Any AI integration must preserve, or ideally enhance, the collaborative learning that takes place during reviews. Designing AI assistants that support rather than bypass human sense-making is therefore essential to maintaining the educational and communicative value of the code review process.

While the culture of trust and shared responsibility was evident in Phase 1 findings, developers also reported challenges they encountered in their code review process. One of the most frequently mentioned issues was delays due to unclear ownership.

Since no specific reviewers are assigned to review PRs, they could linger for long periods, where no one picks them up and reviews them. This lack of accountability often required developers to prompt colleagues manually. This could cause frustration and communication overhead. Another issue involved large PRs, particularly those that combined new feature implementations with refactoring changes. Developers described those as overwhelming and prone to only partial or superficial inspections, particularly when time was limited. Additionally, cognitive friction from context switching emerged as a notable challenge. Developers reported that switching from a development task to reviewing unfamiliar code disrupted their mental flow, and they had to spend time switching context and familiarizing themselves with the PR under review. This aligns with previous findings that highlight the cost of context switching in software engineering tasks like code reviews [1]. Finally, many reviewers reported struggling with insufficient context in PR descriptions, especially when the rationale for changes was missing or poorly documented. In such cases, reviewers had to reconstruct the intention behind the code, leading to wasted time and inconsistent interpretations. These challenges reinforce prior findings that a key barrier to effective code review is the difficulty of achieving sufficient context and change understanding [7]. Reviewers often lack familiarity with the codebase, operate without shared ownership of the system, and are constrained by tool interfaces that offer limited support for deep comprehension. This frequently results in slower reviews and shallower comments, particularly when subtle or conceptual defects are involved.

Interestingly, while developers frequently used AI tools like GitHub Copilot and ChatGPT in development tasks, none reported formal AI use in the code review process. However, informal use cases (like pasting code into ChatGPT for explanation) were noted. Despite this informal status, developers generally expressed a high degree of interest in AI integration when it comes to the code review workflow, particularly as a way to address some of the challenges they face. Several suggested that an AI could automatically summarize the pull request, which can be seen as a direct response to the problem of insufficient PR context. Others proposed that AI could help reviewers validate whether code changes align with the stated requirements or architectural constraints, which could potentially mitigate issues related to partial reviews of large or complex PRs. Others proposed that AI could assist with assigning reviewers and even saw potential for AI in surfacing less obvious risks, such as edge-case bugs or race conditions, scenarios where human attention might slip due to time pressure or fatigue. These expectations further strengthen and suggest that developers are not seeking to outsource the entire process to AI, but rather augment their capabilities in areas that are cognitively demanding or poorly supported by current tools. At the same time, developers were cautious with concerns regarding trust, reliability, and the balance between helpful feedback and unnecessary noise. This highlights how fragile trust in AI tools can be. Once they are perceived as noisy or unreliable, developers may not just ignore them, but actively resist their use.

These findings collectively offer a detailed view of modern code review workflows and the forms of AI assistance that developers would consider valuable. Taken together,

the field study revealed a review culture that is collaborative, context-sensitive, and grounded in trust. However, it also reflects a flexible and informally structured approach to review coordination and performance monitoring, which may limit consistency and make systematic improvement more difficult. Code reviews were shown to serve multiple purposes, including quality assurance, onboarding, and architectural knowledge sharing. Developers are open to AI integration, provided such tools are reliable, transparent, and aligned with their existing workflows. Importantly, they do not seek to replace human reviewers but to enhance their efficiency and comprehension in areas that are cognitively demanding. These insights directly informed the design of Phase 2, where the AI assistant was evaluated in both proactive and reactive interaction modes to investigate under what conditions such support could be meaningfully integrated into the review process.

6.1.2 RQ2

How do developers perceive LLM-assisted code review tools, and what is the preferred interaction?

This research question aimed to understand how developers perceive LLM-assisted code review tools and what interaction they prefer. The Phase 2 field experiment revealed not only how developers engage with such AI tools in the review process but also how their perceptions are shaped by factors such as trust, efficiency, usability, and the level of control they have over the interaction. These factors influenced both the acceptance of the AI assistant and their preferences between proactive and on-demand interaction modes.

A key factor in the developers' perception of the introduced assistant was trust. Participants generally found the assistant helpful, particularly when it surfaced issues they might have missed or confirmed their observations. However, they also encountered incorrect suggestions and suggestions that did not provide much value, which made them cautious. The participants reviewed the AI output critically rather than accepting it blindly, in line with prior studies on human-AI collaboration where participants emphasized the need to trust but verify the AI's output [28]. This form of measured trust was especially evident in how reviewers responded to Mode A's upfront AI suggestions, which some felt risked introducing initial bias or skewing their attention. This underscores the importance of preserving human oversight even when the AI proves helpful.

A strong theme in the developers' feedback was the AI assistant's potential to improve code review efficiency and thoroughness. Many reported that receiving a summary and suggestions before they started reading the code sped up their review process. They felt that this might save a lot of time by, for example, reducing the need to skim large diffs or search code files for certain information. This has the potential to reduce tedious tasks and allow reviewers to focus on more complex or critical issues. Participants also believed the AI could enhance review effectiveness by catching bugs or code smells they might overlook. This optimism among participants was, however, accompanied by concerns regarding noise in the AI's output.

The developers found it hard to distinguish between the critical findings from the nitpicks, especially when provided with lengthy responses with multiple suggestions. In those cases, reviewers had to spend extra effort parsing which items truly required action, which could distract them from the key issues, potentially hurting efficiency rather than helping it. This confirms findings from recent studies where automatically generated reviews often flag a greater number of total issues (especially minor ones) but do not significantly increase the detection of critical bugs. Moreover, those studies found little to no net time saved in the review process with AI assistance, because developers still had to read and verify the AI's suggestions one by one [27]. Participants in this study perceived somewhat greater efficiency, whereas participants felt faster and more thorough with the AI. But it is worth noting that perceived efficiency may differ from actual efficiency once the overhead of false positives and verification is accounted for. The Assistant was generally seen as increasing thoroughness, and one person, for example, stated that an imperfect AI that catches some issues is still better than nothing. This resonates with the idea that even if the tool does not catch every problem, it can act as a safety net to reduce the chance of serious issues going unnoticed. This was seen as especially relevant for large or complex PRs where the assistant was viewed as a useful aid to thoroughness since it could persistently scan for issues across a large set of code modifications in a way that a human reviewer, prone to cognitive overload, might not. This could increase quality where manual inspection may be inconsistent. Participants therefore largely agreed that the AI assistant augmented their review capabilities by making them faster and more confident that no obvious issues had been missed. Still, they were aware that this efficiency gain depended on the AI's suggestions being accurate and relevant, as any decrease in precision would quickly undermine the time saved. This clearly shows that the effectiveness of AI assistance is tightly linked to its precision and the effort required to use its outputs. When done right, it promises faster, more thorough reviews, but when done poorly, it could simply add more work.

In addition to the content of the AI's feedback, participants discussed how the assistant should fit into their tools and workflow. They often discussed a desire for a seamless integration where the AI could be directly accessed within their existing review interfaces (e.g., GitHub or their IDE) rather than through a separate application or unfamiliar UI, as was the case in this experiment. Some described a smoother experience where the assistant's comments would appear inline on the PR changes. This suggests that if the AI assistant becomes just another feature of the code review platform, developers are more likely to adopt it and use it more frequently. The results also show that it is crucial for the assistant to produce usable output since the developers emphasise well-structured and concise responses that prioritise the most important information. Several remarked that current outputs were too long and verbose, making it hard to grasp the main points. This supports observations in prior work that LLM-generated reviews can be overwhelming in length and detail, making it hard to identify the actionable suggestions from other noise [28]. Participants suggested improvements like conciseness and more clearly listing the file name and line number relevant to each finding. Such changes could possibly address the cognitive load issue, whereas research has noted that AI feed-

back lacking conciseness and context requires additional time and effort for humans to parse [28]. Those comments and suggestions from the participants are essentially a call to reduce cognitive overhead. The tool should streamline the review, not complicate it. Another usability factor discussed was the assistant’s response time. Participants were aware that the tool was only a prototype used as an experimental instrument, but noted that to be adopted in real workflows, the AI assistant must deliver results in a timeframe that keeps pace with normal human review speed. Minor delays were acceptable, but anything that breaks the reviewer’s flow is seen as a blocker for everyday use. This insight, while straightforward, is important for further development and adoption of such an assistant.

Participants also emphasized the importance of the assistant having access to sufficient and relevant context. While many appreciated that the assistant could draw on information such as requirements from the JIRA ticket and parts of the codebase before the PR, they often felt that the assistant still lacked a full understanding of the broader context. In some cases, the assistant did not query the codebase in cases where it would have been helpful to do so, and even when it did, it often yielded limited or superficial results (see Section 6.2). These limitations can partly be attributed to how the participants prompted the assistant; users had to be quite explicit to trigger deeper context retrieval, especially when looking up in the codebase. This can also be attributed to the performance and design of the assistant’s search ability, which could be inconsistent and slow. This reflects a broader insight that even though the assistant had the technical ability to access contextual data, its usage of the capability was not always effective. Participants viewed context awareness as essential to improving the assistant’s usefulness and reliability, which suggests context awareness to be particularly important for reviews where understanding the intent and implications of code changes requires connecting information across multiple sources. These reflections suggest that future designs of such tools should more proactively and systematically retrieve relevant context, and that more advanced approaches to gathering and incorporating contextual data are likely necessary for the assistant to provide more complete and coherent feedback. This confirms concerns about the role of context availability in LLM review accuracy [28], and the findings suggest that the implemented retrieval-enhanced LLM agent can mitigate, but not fully overcome, these limitations.

A key focus in Phase 2 was the comparison of the two interaction modes of the AI assistant. Mode A (Co-Reviewer), where the AI proactively provided an upfront summary and initial comments, was favored by the majority of participants for most scenarios. They described Mode A as being especially useful to get an overview of a PR. With the AI giving a high-level summary of the changes and pointing out potential issues, reviewers could more quickly orient themselves, which has been identified as often the hardest part of tackling a new review. Participants stressed the significant advantage that would come with this when dealing with large or unfamiliar code changes, situations where comprehending the scope and intent of the PR is time-consuming. This was seen as time-saving, where Mode A acts as an assistant reviewer who briefs you on what is going on. Several participants cited examples of Mode A, highlighting elements of the code that they might have over-

looked initially, thereby contributing to a more thorough review. This experience ties back to the challenge identified in Phase 1, that reviewers often struggle with understanding big PRs or those outside their domain. By preemptively supplying relevant contextual information, Mode A directly addresses that challenge, and the results from Phase 2 show that developers responded very positively to it. Some even noted that the summary of the PR (its purpose and what has changed) was even more useful than the suggestions provided alongside it. This shows that when an AI can summarize and outline changes, it reduces the effort required to make sense of the review; however, only if the output is easily parsed by the developer, as mentioned above. This stands in contrast to earlier findings where LLM-generated reviews failed to improve reviewers' comprehension [27]. Phase 2 findings show that the assistant's high-level summaries were repeatedly cited as aiding understanding, particularly for reviewers unfamiliar with the codebase. This may be explained by the assistant's access to broader context and the ability in both modes to ask follow-up questions, which likely supported deeper comprehension. Importantly, Mode A was also seen as a great learning tool for less experienced reviewers or newcomers to a codebase. By reading the AI's summary and suggestions, a newcomer could more quickly grasp what the code changes are about and what to pay attention to. This reflects an educational benefit also noted in prior literature where AI-assisted code review tools can help deepen developers' knowledge and understanding of best practices [25]. In essence, Mode A was seen as a strong default mode or a way to ensure that a baseline review (with major issues and summary) is always provided, even if a human reviewer is unfamiliar with the code. This was even further reinforced by developers, who noted that the AI co-reviewer would be particularly valuable in teams where code reviews are less strict or not the preferred way of reviewing code. In such cases, having the AI co-reviewer could catch things missed by a superficial or non-existent review. This further highlights how Mode A could raise the minimum quality of reviews in practice.

On the other hand, Mode B (Interactive Assistant), where the AI remains passive unless asked, received a more mixed reaction. A subset of participants appreciated this mode for changes to code they are already relatively familiar with. Some expressed that they did not necessarily want the AI's full summary or list of suggestions provided by Mode A when they already had a good grasp of the code. In those cases, they felt it was more useful to ask targeted queries and get answers on demand. Some participants expressed a preference for Mode B because they felt confident in their review process and saw the assistant more as a backup or a specialized tool. Additionally, they preferred Mode B in order to maintain control and avoid any bias that might be introduced by the AI's unprompted suggestions. This perspective confirms earlier findings that AI highlights could skew human judgement [27]. However, aside from these specific scenarios, Mode B was generally seen as less helpful. The main critique was that it missed the opportunity to proactively assist. Some participants were also inclined to ask Mode B to provide similar summaries as Mode A. This suggests that unless the reviewer knows exactly what to ask, they might not leverage the AI's potential, and some participants also wondered if they were "asking the wrong things" when the AI's answers did not seem helpful. In effect, Mode

B shifted the burden onto the user to drive the interaction, introducing a learning curve for effective use. This result suggests that while on-demand AI support can be helpful in many scenarios, it might not be as immediately impactful for users who are new to such tools. It puts more cognitive load on the reviewer to identify when the AI could help, while in contrast, Mode A’s approach of offering help unasked ensures that the AI’s value is presented every time, without having to actively seek it out. Nonetheless, Mode B was valued in cases where reviewers wanted precision and control, showing that its flexibility makes it a useful complement in AI-assisted code reviewing.

Given the strengths and weaknesses of each mode, an interesting insight is that context and reviewer preference seem to dictate the interaction style. Some participants envisioned an adaptive or hybrid approach of both interaction modes, depending on the PR under review. The ability to opt out of AI suggestions when not needed was also considered important, as one interviewee noted, having an easy opt-out means there is not “real harm” in the tool being present. This points to control being an important factor when it comes to introducing AI to the code review workflow, developers want to be in charge of how and when the AI is used. The results do not declare one mode universally superior, instead, they highlight that Mode A is preferred by default for its time-saving and guidance benefits, but Mode B has its place in scenarios requiring human-led analysis.

Overall, the Phase 2 findings show that developers are receptive to LLM-assisted review tools and see them as empowering if integrated correctly. Participants perceived the AI as a competent collaborator that can improve review speed and thoroughness. At the same time, they maintain a critical stance, treating the AI’s output as suggestions to be evaluated rather than absolute truths. This suggests they treat AI output with the same level of criticism as they would treat output from a human peer, aligning with previous findings [28]. As stated, the preferred interaction mode leans towards an AI co-reviewer that provides upfront assistance, but with the crucial ability for the human to override, guide, or limit the AI’s involvement as needed. In practice, developers want the AI to augment their capabilities, not overshadow them, which confirms the importance of the correct balance between automation and human control in software engineering tools [27]. In conclusion, the ideal LLM-assisted code review experience, as indicated by the findings, is one of a partnership. The AI contributes speed, thorough analysis, and coverage, while the human provides oversight, contextual understanding, and final judgment. When this balance is met, developers seem open to embracing LLM-assisted review tools as a natural extension integrated into their workflow.

6.2 Reflections on Assistant Behavior and Experimental Fidelity

The developed artifact (AI-Assistant) served as the central experimental tool in this study. While participant perceptions and experiences are reported in Section 6.1.2,

this section provides an independent reflection on the assistant’s behaviour during the experiment sessions from a system perspective. It considers how the artifact’s design choices, technical performance, and retrieval mechanisms may have shaped developer interactions, expectations, and trust. The goal is to contextualize the experimental outcomes by examining where limitations in the assistant’s behavior may have influenced participant experience.

While the distinction between the two interaction modes was technically implemented (see Section 4.5.3) by using different prompting strategies and agent structure, participants largely identified the primary difference as the presence or absence of the initial summary that Mode A provided. Participants generally considered these summaries useful, but researchers observed inconsistencies in the assistant’s risk evaluations across sessions, even for the same PRs. Additionally, some participants suggested that Mode A essentially “contained” Mode B, reducing the perceived utility of the reactive mode. Although Mode A was designed to take a more leading role and Mode B a more passive one, this distinction was not always apparent to users.

The ReAct-based agent structure enabled the assistant to reason about which tools to use before generating a response. While this design generally functioned as intended, it also introduced behavioral inconsistencies in how context was assessed and retrieved. In Mode B, the assistant had no prior knowledge of the PR and needed to retrieve all relevant information through tool usage, often resulting in longer response times compared to Mode A, which had PR data pre-loaded into the initial prompt. A recurring issue was that the agent frequently decided it had sufficient context and prematurely returned answers without querying additional tools. This could result in less accurate responses when deeper code understanding or architectural context was required. On the other hand, it was also noted that several participants rarely issued prompts that would require the assistant to fetch new information, and general knowledge or data already loaded into context were sufficient for an acceptable answer. This suggests a possible mismatch between the assistant’s retrieval potential and how it was actually used.

As observed, the assistant also encountered issues when it came to handling its internal reasoning. For example, the assistant sometimes responded in ways that appeared to reflect its own internal thoughts, such as replying “Exactly as you observed. . .” to a user query. This likely occurred when the agent confused its own internal observation with the user’s prompt and answered according to that. Similarly, when internal reasoning steps or partially processed thoughts were returned prematurely, participants could become confused or uncertain about how to proceed. While these instances were rare and usually recoverable with follow-up prompts instructing the assistant to try again, they highlight a mismatch between how the assistant reasons internally and how that reasoning should be surfaced to the user. When internal thoughts or assumptions are shown prematurely, they can be misinterpreted as final conclusions, possibly causing reduced clarity and user trust.

The query engine tools (RAG-based retrieval) introduced further challenges that

influenced assistant reliability. As observed in some sessions, the assistant occasionally returned incomplete data when fetching code files or project artifacts. These breakdowns likely stem from limitations in how the query tools parsed or indexed large files. Additionally, indirect LLM involvement in retrieval sometimes led to subtle hallucinations. For instance, when the `search_requirement` tool returned generated requirement lists that over-interpreted the brief requirement description in the source, resulting in non-existent or speculative requirements. Another example is when tools returned narrative explanations of source code files rather than raw code. In those cases, the assistant might reconstruct the source code based on the summary, leading to fabricated or incorrect code. These examples indicate that even indirect LLM involvement in retrieval can subtly distort factual grounding.

The assistant’s behavior demonstrated mixed relevance and reliability, particularly in how it grounded its responses. Mode B occasionally produced hallucinated responses when it decided not to fetch any data, suggesting a failure to trigger the retrieval process when needed. Similarly, Mode A sometimes flagged non-existent issues, such as missing imports, likely due to its reliance on file diffs without validating against the full codebase. These limitations reflect gaps in factual grounding that can reduce user trust, especially when surfaced without clear context or supporting evidence.

Additional observations emphasized the importance of both prompt engineering and the structural organization of the assistant’s accessible context. During development, it became clear that small errors or ambiguities in the system prompt could lead to drastically different outcomes. This could cause the assistant to fail silently or to perform tasks in a way it was not supposed to. Significant debugging effort was invested into refining prompt phrasing, and further work is needed to stabilize and optimize prompt design. Beyond prompting, the performance was also influenced by where contextual data was made available. When key data was embedded directly into the initial prompt (as in Mode A), the assistant was able to get a more holistic view of the data and evaluate all relevant information before returning an answer. In contrast, when the same information had to be retrieved dynamically via the RAG pipeline (as in Mode B), the assistant had to issue a series of semantic searches to reconstruct the basic context. This resulted in longer reasoning chains, more tool calls, and occasional breakdowns where the system failed to assemble a complete picture. These observations suggest that data that is predictable, static, or structurally central to the task (e.g., the PR metadata) should be injected into the initial prompt, while the RAG layer should be reserved for more specific, context-sensitive queries for less frequently required data. Further development of the artifact should consider this division when balancing context-window constraints against retrieval latency and agent performance.

6.3 Implications

The findings of this study advance theoretical understanding of human–AI collaboration in software engineering. They reinforce the view of AI systems not merely

as automation tools, but as cognitive partners that must adapt to human workflows, preferences, and contextual knowledge. The observed variation in interaction preferences and emphasis on trust and control highlight the importance of flexible, context-aware systems that support, rather than replace human decision-making. By empirically grounding these insights, the study contributes to software engineering research on AI-augmented development practices and informs the design of future tools that embed AI seamlessly into workflows while preserving human oversight. These implications extend beyond code review, offering guidance for responsible and effective AI integration across expert domains.

The findings of this study also highlight several practical implications for the design and adoption of LLM-assisted code review. First, developers are open to AI assistance when it improves review efficiency and thoroughness, but they emphasize precision and clarity. For AI suggestions to be useful, they must be concise, structured, and targeted, otherwise, the cognitive overhead can outweigh the benefits. This aligns with prior work showing, in their case, that LLM-generated reviews did not reduce review time, as developers still had to interpret and verify suggestions [27]. Tools that aim to reduce reviewer workload must therefore minimize irrelevant output and surface only actionable insights.

Secondly, seamless workflow integration is crucial. Developers expressed a strong preference for AI features embedded directly into familiar platforms, with inline feedback and minimal friction. Poor interface design or latency can disrupt the review process, making responsiveness and usability central to adoption.

Thirdly, the assistant’s context awareness emerged as a critical factor. Participants appreciated the tool’s ability to access requirements and codebase data, but also noted limitations in how effectively it used that context. This reflects prior findings that LLMs struggle to provide meaningful insights when not grounded in broader project or architectural information [28]. Future implementations should aim to proactively retrieve and incorporate richer contextual capabilities to support deeper understanding and relevant feedback.

Finally, while proactive assistance (Mode A) was preferred in general for its guidance and coverage, preferences varied in cases where participants wanted more control or were more familiar with the code. This highlights the need for flexible interaction modes. Tools should allow for customization depending on individual reviewers and review contexts.

6.4 Threats to Validity

The following section discusses limitations to the study’s validity, structured across construct validity, internal validity, external validity, and reliability.

6.4.1 Construct Validity

A potential threat to construct validity lies in the measurement of effectiveness and developer experience when integrating LLM into code reviews. The evaluation relied on qualitative perceptions rather than quantitative metrics. Perceptions of effectiveness or usefulness may not accurately reflect actual improvements in code review performance. Additionally, participant responses may have been shaped by prior experiences with LLM tools or individual preferences, introducing bias in participants' perceptions and responses.

Thematic analysis of interview data introduces subjectivity. The identification of themes and interpretations of developer experience was done by researchers and could be influenced by their expectations or prior beliefs. There is a risk that the themes reflect the researchers' perspective as much as the participants' actual experiences. To address this, two researchers conducted all coding collaboratively, reviewing transcripts line-by-line in joint sessions where codes and themes were discussed and applied by consensus. Although no inter-rater reliability metrics were calculated, the use of joint coding sessions allowed for negotiated agreement and ongoing clarification throughout the process, reducing the likelihood of unexamined researcher bias.

Another construct-related concern is that the participants knew they were being observed by the researchers and might have been inclined to give positive or expected answers about the AI tool. This could skew the measurement of true opinions of effectiveness. While participants were assured of anonymity and encouraged to give honest feedback, their responses may still have been influenced by wanting to appear accommodating.

6.4.2 Internal Validity

A key limitation arises from the use of convenience sampling. Participants for both phases were selected based on availability and willingness rather than randomization, which introduces bias and may skew the internal results. The observed effects might partly reflect the pre-existing attitudes of this self-selected group rather than the intervention alone.

The study did not include a control group that performed code reviews without AI, making it difficult to directly attribute observed effects to the introduction of AI. There was no baseline measurement of performance or behaviour under identical conditions without the tool. Instead, the comparison relied on participants' subjective reflection based on their experience across the two conditions and experience performing code reviews.

Since each experiment participant completed two code review tasks, one for each interaction mode, there is a risk that the order in which they experienced the tasks influenced their performance or feedback. For example, participants might have become familiar with the general task or the AI tool during the first review, making

the second feel easier regardless of the mode. Alternatively, they may have become tired, or their impression of the second mode may have been shaped by their experience with the first, making it harder to isolate the effect of the interaction mode itself. To reduce this risk, the assignment of AI modes to specific PRs was rotated across participants. However, with the current sample size, some learning or order effects may still have impacted the results.

The presence of researchers during the session may have influenced participants' behaviour, potentially leading them to act in ways they assumed were expected, rather than responding naturally. This observer effect could lead to behaviour and responses aligned with expectations rather than natural usage. This represents a threat to internal validity because the cause of certain behaviors or feedback could be the study context itself rather than the AI assistant.

The experiment tool (AI assistant) exhibited inconsistencies, including hallucinations and inaccurate or irrelevant outputs. A participant who experienced the assistant giving a false or irrelevant suggestion might become distrustful or use the tool differently thereafter, compared to a participant who only saw accurate, helpful outputs. As such, participants' experiences varied based on how reliably the tool functioned during the session. This inconsistency could also have affected the perceived effectiveness of each interaction mode.

6.4.3 External Validity

The study was conducted exclusively at one specific software development company, WirelessCar, within a specific organizational and technological context. Practices, tools, and team dynamics differ from other companies, which limits the generalizability of the findings to other organizational settings.

Because of convenience sampling, the participant group is not representative of the broader developer population. Their experience levels, roles, or attitudes towards AI assistance may not reflect those in other settings. Additionally, the sample size of 10 in the field experiment might constrain generalization. The study aimed for a diverse mix within the company (participants varied in roles, team affiliation, and familiarity with the codebase), which helped capture varied perspectives. While this diversity does not guarantee statistical representativeness, it increases ecological validity by reflecting a range of real-world conditions encountered in professional software teams. The participants were embedded in active development teams within a modern organization, and their workflows, tool usage, and exposure to industry-standard practices suggest alignment with broader trends in software engineering. Additionally, the participants engaged in a standardized code review process that closely resembles common industry practices, suggesting that the findings may be relevant to a broader population of developers. However, the limited number of participants still restricts the breadth of experiences represented, limiting the ability to generalize findings beyond the studied context.

All experiment tasks involved the testing of exactly two different interaction modes

on two pre-selected PRs of moderate size and complexity. Other types of software engineering activities (like coding, testing, or debugging) were outside the scope, as were other possible AI support paradigms in code review. The tasks were chosen to be realistic yet manageable and therefore do not encompass the full spectrum of code review scenarios that exist in industry. As a result, the findings may not generalize to all forms of AI-assisted reviewing, such as reviews of very large, highly critical, or technically different PRs.

The experiment tool relied on a specific LLM (OpenAI model via API) combined with a custom retrieval-augmentation setup. The capabilities and limitations observed are partly a function of that specific technology, but other AI tools or future versions may behave differently. Furthermore, the assistant's performance was dependent on the quality and relevance of the retrieved context, which may differ across projects. Generalizing to other tools or contexts should therefore be done with caution. This study has effectively evaluated one implementation of AI-assisted code review, and results could vary with a different tool setup.

External validity over time is also a concern, given the rapid development of LLM technology. The findings capture a snapshot with a particular version of the technology that may become obsolete or significantly improved in a short span. Hence, generalizability is limited, especially to future settings or different AI tools.

6.4.4 Reliability

As noted under internal validity, the AI tool's output could be inconsistent. If someone attempted to replicate the experiment, they might not get the exact same assistant responses or interactions that the observed participants did. The implementation and configuration of the AI assistant were documented thoroughly, and the same version of the LLM model and prompts were used for all sessions to maximize consistency. Nonetheless, slight differences in retrieved context could result in different outputs.

To enhance reliability, all sessions and interviews were conducted with two researchers present to reduce individual bias in note-taking and interpretation. The coding of interviews and development of themes was similarly carried out collaboratively by two researchers, who iteratively discussed and refined the codes. Codes and their descriptions were systematically recorded in a shared digital workspace throughout the analysis process. While no formal codebook with fixed definitions was constructed beforehand, this shared code list functioned as an evolving codebook and supported consistent code application across data sources and time. These measures aimed to improve consistency and reduce the risk of individual bias, though some variation in interpretation would still be expected if the study were replicated by different analysts.

6.5 Future Work

While this study provided qualitative insights into how developers interact with LLM-assisted review tools, several directions remain for future exploration. As this study was qualitative and exploratory, future research could complement these findings with a quantitative evaluation. Studies might examine review time, issue detection rates, reviewer agreement, or cognitive load across assisted and non-assisted conditions. Such data would help clarify the practical impact of AI-assisted reviews and support a broader understanding of when and how these tools provide the most value.

Another area relates to tool performance and the potential of more advanced implementations. For instance, future work might explore how a more comprehensive version of the co-reviewer could operate in a CI/CD pipeline, where response time is less critical. Such an agent might systematically investigate all changed files, traverse the broader codebase, past pull requests, and incorporate linked requirements or acceptance criteria to generate a highly comprehensive review. This could potentially improve the assistant's depth and consistency, as well as provide the assistant with high-quality up-front context.

In addition to these workflow changes, future implementations might also explore the value of injecting more structured metadata directly into the assistant's context. This could include the list of changed files, file structure, coding conventions, and architectural documentation. Having this context available from the outset might reduce redundant querying, improve the accuracy of direct questions, and reduce response time.

Future work may consider how interaction modes can remain flexible. While the co-reviewer mode showed strong potential and was generally well-received, an interactive assistant or even alternative modes, such as pre-review for code-authors, could be offered as toggleable settings. Ideally, both would operate on the same comprehensive context but would differ in their responses or time of activation.

7

Conclusion

This thesis presents a field study and field experiment conducted at WirelessCar to explore the integration of Large Language Models (LLMs) into real-world code review workflows. The study surfaces persistent challenges in current review practices, such as context switching, reviewer fatigue, inconsistent review depth, and developer perceptions of how LLMs can augment the process. By evaluating two interaction modes (AI-led reviews and on-demand assistance), it was found that developers generally value AI-generated summaries and contextual clarifications, particularly in large or unfamiliar pull requests. However, concerns around trust, false positives, response latency, and integration friction remain. While most participants preferred the AI-led mode in unfamiliar or low-risk scenarios, preferences were context-dependent, with some favoring human-led reviews when code familiarity or criticality increased.

This study contributes practical insights into how LLMs can complement human reviewers, rather than replace them. The assistant was most appreciated when it helped reduce the cognitive overhead of understanding large changes or navigating unfamiliar code. The findings also highlight that trust in the AI system hinges on response relevance, clarity, and access to meaningful context. Techniques such as retrieval-augmented generation, which allow LLMs to draw from the broader codebase and project documentation, are promising directions to improve review relevance and accuracy. However, effective integration depends not only on what the assistant provides, but when and how. Flexibility emerged as a key requirement. Some developers preferred a fully AI-led review experience, while others wanted to engage with the assistant only when needed. This reinforces the importance of adaptive interaction modes.

Ultimately, the results suggest that the most impactful role for LLMs in code review is as an AI-developer partnership, supporting human judgment rather than substituting for it. When integrated thoughtfully, LLMs can streamline repetitive or cognitively demanding review tasks, enhance understanding, and enable reviewers to focus their attention on higher-level design and reasoning.

Bibliography

- [1] J. Czerwonka, M. Greiler, and J. Tilford, “Code reviews do not find bugs. how the current code review best practice slows us down,” in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. IEEE, 5 2015, pp. 27–28.
- [2] H. Naveed, A. U. Khan, S. Qiu, M. Saqib, S. Anwar, M. Usman, N. Akhtar, N. Barnes, and A. Mian, “A comprehensive overview of large language models,” 2023. [Online]. Available: <https://arxiv.org/abs/2307.06435>
- [3] M. Vijayvergiya *et al.*, “Ai-assisted assessment of coding practices in modern code review,” in *Proc. 1st ACM Int. Conf. on AI-Powered Software*. Association for Computing Machinery, 2024, pp. 85–93. [Online]. Available: <https://doi.org/10.1145/3664646.3665664>
- [4] J. Sarker, A. K. Turzo, M. Dong, and A. Bosu, “Automated identification of toxic code reviews using toxicr,” *ACM Trans. Softw. Eng. Methodol.*, vol. 32, 7 2023. [Online]. Available: <https://doi.org/10.1145/3583562>
- [5] H. Y. Lin, P. Thongtanunam, C. Treude, and W. Charoenwet, “Improving automated code reviews: Learning from experience,” in *Proceedings of the 21st International Conference on Mining Software Repositories*. ACM, 4 2024, pp. 278–283.
- [6] M. Hamza, D. Siemon, M. A. Akbar, and T. Rahman, “Human-ai collaboration in software engineering: Lessons learned from a hands-on workshop,” *2024 IEEE/ACM International Workshop on Software-Intensive Business (IWSiB)*, pp. 7–14, 2023, proposal:Used in The collaboration of humans and AI in SE. [Online]. Available: <https://api.semanticscholar.org/CorpusID:266348401>
- [7] A. Bacchelli and C. Bird, “Expectations, outcomes, and challenges of modern code review,” in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 5 2013, pp. 712–721.
- [8] A. Kamal Turzo and A. Bosu, “What Makes a Code Review Useful to OpenDev Developers? An Empirical Investigation,” *arXiv e-prints*, p. arXiv:2302.11686, Feb. 2023.

- [9] 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, p. 181–190. [Online]. Available: <https://doi.org/10.1145/3183519.3183525>
- [10] D. Badampudi, M. Unterkalmsteiner, and R. Britto, “Modern code reviews—survey of literature and practice,” *ACM Trans. Softw. Eng. Methodol.*, vol. 32, 5 2023. [Online]. Available: <https://doi.org/10.1145/3585004>
- [11] S. Chatterjee, C. L. Liu, G. Rowland, and T. Hogarth, “The Impact of AI Tool on Engineering at ANZ Bank An Empirical Study on GitHub Copilot within Corporate Environment,” *Software Engineering*, 2024. [Online]. Available: <https://arxiv.org/abs/2402.05636>
- [12] K. Li, A. Zhu, W. Zhou, P. Zhao, J. Song, and J. Liu, “Utilizing Deep Learning to Optimize Software Development Processes,” *Southern United Academy of Sciences*, apr 27 2024. [Online]. Available: <https://zenodo.org/doi/10.5281/zenodo.11004006>
- [13] V. Garousi, N. Joy, A. B. Keleş, S. Değirmenci, E. Özdemir, and R. Zarringhalami, “Ai-powered test automation tools: A systematic review and empirical evaluation,” *arXiv.org*, 2024. [Online]. Available: <https://arxiv.org/abs/2409.00411>
- [14] Mitul Dilip Bhai Modi, “Transforming Software Development Through Generative AI : A Systematic Analysis of Automated Development Practices,” *International Journal of Scientific Research in Computer Science, Engineering and Information Technology*, vol. 10, no. 6, pp. 536–547, nov 13 2024. [Online]. Available: <http://dx.doi.org/10.32628/cseit24106197>
- [15] S. Oh, K. Lee, S. Park, D. Kim, and H. Kim, “Poisoned ChatGPT Finds Work for Idle Hands: Exploring Developers’ Coding Practices with Insecure Suggestions from Poisoned AI Models,” in *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, may 19 2024, pp. 1141–1159. [Online]. Available: <http://dx.doi.org/10.1109/SP54263.2024.00046>
- [16] E. Syyrilä and J. Kasurinen, “Developing Smarter Software Engineering Tools by Utilizing AI Assistance,” in *2024 47th MIPRO ICT and Electronics Convention (MIPRO)*. IEEE, may 20 2024, pp. 2013–2018. [Online]. Available: <http://dx.doi.org/10.1109/MIPRO60963.2024.10569365>
- [17] Y. Fu *et al.*, “Serverlessllm: Low-latency serverless inference for large language models,” 2024. [Online]. Available: <https://arxiv.org/abs/2401.14351>
- [18] S. Laskaridis, K. Katevas, L. Minto, and H. Haddadi, “Melting point: Mobile evaluation of language transformers,” 2024. [Online]. Available: <https://arxiv.org/abs/2403.12844>

-
- [19] Y. Gao *et al.*, “Retrieval-augmented generation for large language models: A survey,” *arXiv preprint arXiv:2312.10997*, vol. 2, 2023.
- [20] X. Du *et al.*, “Vul-rag: Enhancing llm-based vulnerability detection via knowledge-level rag,” 2024. [Online]. Available: <https://arxiv.org/abs/2406.11147>
- [21] A. Plaat, M. van Duijn, N. van Stein, M. Preuss, P. van der Putten, and K. J. Batenburg, “Agentic large language models, a survey,” 2025. [Online]. Available: <https://arxiv.org/abs/2503.23037>
- [22] C. Sypherd and V. Belle, “Practical considerations for agentic llm systems,” 2024. [Online]. Available: <https://arxiv.org/abs/2412.04093>
- [23] S. Yao *et al.*, “React: Synergizing reasoning and acting in language models,” 2023. [Online]. Available: <https://arxiv.org/abs/2210.03629>
- [24] R. Tufano, L. Pascarella, M. Tufano, D. Poshyvanyk, and G. Bavota, “Towards automating code review activities,” in *Proceedings of the 43rd International Conference on Software Engineering*, ser. ICSE ’21. IEEE Press, 2021, p. 163–174. [Online]. Available: <https://doi.org/10.1109/ICSE43902.2021.00027>
- [25] Z. Rasheed *et al.*, “Ai-powered code review with llms: Early results,” 4 2024. [Online]. Available: <https://arxiv.org/abs/2404.18496>
- [26] R. I. T. Jensen, V. Tawosi, and S. Alamir, “Software vulnerability and functionality assessment using llms,” *2024 IEEE/ACM International Workshop on Natural Language-Based Software Engineering (NLBSE)*, pp. 25–28, 2024. [Online]. Available: <https://api.semanticscholar.org/CorpusID:268379288>
- [27] R. Tufano, A. Martin-Lopez, A. Tayeb, O. Dabi’c, S. Haiduc, and G. Bavota, “Deep learning-based code reviews: A paradigm shift or a double-edged sword?” *ArXiv*, vol. abs/2411.11401, 2024. [Online]. Available: <https://api.semanticscholar.org/CorpusID:274131620>
- [28] A. Alami and N. A. Ernst, “Human and machine: How software engineers perceive and engage with ai-assisted code reviews compared to their peers,” *ArXiv*, vol. abs/2501.02092, 2025. [Online]. Available: <https://api.semanticscholar.org/CorpusID:275336938>
- [29] K.-J. Stol and B. Fitzgerald, “The abc of software engineering research,” *ACM Trans. Softw. Eng. Methodol.*, vol. 27, no. 3, Sep. 2018. [Online]. Available: <https://doi.org/10.1145/3241743>
- [30] V. Braun and V. Clarke, “Using thematic analysis in psychology,” *Qualitative Research in Psychology*, vol. 3, no. 2, pp. 77–101, 2006. [Online]. Available: <https://www.tandfonline.com/doi/abs/10.1191/1478088706qp063oa>

- [31] O. A. Adeoye-Olatunde and N. L. Olenik, “Research and scholarly methods: semi-structured interviews,” *J. Am. Coll. Clin. Pharm.*, vol. 4, no. 10, pp. 1358–1367, 2021.
- [32] I. Etikan, S. A. Musa, and R. S. Alkassim, “Comparison of convenience sampling and purposive sampling,” *American journal of theoretical and applied statistics*, vol. 5, no. 1, pp. 1–4, 2016.
- [33] G. Guest, A. Bunce, and L. Johnson, “How many interviews are enough?: An experiment with data saturation and variability,” *Field Methods*, vol. 18, no. 1, pp. 59–82, 2006. [Online]. Available: <https://doi.org/10.1177/1525822X05279903>
- [34] OpenAI, “Introducing openai o3 and o4-mini,” <https://openai.com/index/introducing-o3-and-o4-mini/>, April 2025, accessed: 2025-05-03.
- [35] OpenAI, “New embedding models and api updates,” <https://openai.com/index/new-embedding-models-and-api-updates/>, January 2024, accessed: 2025-05-03.
- [36] R. Alves, P. Valente, and N. J. Nunes, “The state of user experience evaluation practice,” in *Proceedings of the NordiCHI 2014: The 8th Nordic Conference on Human-Computer Interaction: Fun, Fast, Foundational*. Association for Computing Machinery, 10 2014, pp. 93–102.

A

Appendix

A.1 Phase 1 - Data

A.1.1 Interview Question List

Below are the questions used during the developer interviews:

1. Can you walk me through the step-by-step process your team follows for a typical code review? For example, how does a review start, what tools are used, who is involved, and how is feedback implemented?
 - (a) Are there steps that sometimes get skipped/modified? Why?
 - (b) Are any steps in the process automated? Which tools?
 - (c) Do you follow any predefined methodology?
 - (d) Does your team have any documentation regarding how code reviews should be executed or any guidelines one should follow when reviewing code?
 - (e) Do you have a personal checklist that you go through?
2. What would you say are the pain points in this process? Such as the time-consuming or frustrating parts of the process, and why?
3. How do you measure the success of a code review?
 - (a) Does your team keep any records of your review metrics or statistics?
4. How important is understanding the broader codebase or project architecture when reviewing code?
 - (a) How do you currently gain that context?
5. How do you balance thoroughness with speed/efficiency when performing code reviews?

6. Have you used any AI tools before in the code review process or other development processes? If so, how and what worked well and what worked badly?
7. Are there specific tasks in code reviews that you think would benefit from AI assistance?
 - (a) Any specific tools in mind?
 - (b) Do you see any limitation to why that has not been implemented or why it should not be implemented?

A.1.2 Identified Themes and Codes from Thematic Analysis: Phase 1

The following section presents the results from the thematic analysis [30] conducted on the qualitative data collected during Phase 1 interviews. Six distinct themes were identified, each accompanied by a set of codes that encapsulate recurring concepts and observations across participants.

Informal Review Process and Practices Describes how teams coordinate and manage code reviews in practice, including informal communication, tool use, and the absence of structured processes or metrics.

- No code review metrics
- No strict reviewer assignment
- Use Slack for requesting reviewers
- PRs up for grabs
- Manually pinging to get reviews done
- Not much use of standardized checklists or documentation
- Multiple reviewers for big PRs
- Static analysis before code is reviewed

Review Strategies and Evaluation Focus Captures what developers focus on during the actual code review process.

- Focus on logic
- Look at tests during review
- Reviews coding standards
- Reviews file structure

- Compare code to requirements during review

Learning, Knowledge and Review Expertise Explores how code reviews serve as opportunities for learning and knowledge sharing within teams. Also captures the role of reviewer expertise in conducting effective reviews and the challenges that arise when reviewers lack sufficient understanding of the codebase or architecture.

- Use code reviews for knowledge sharing
- Learning via hands-on exposure
- Invests time on code reviews
- Broader architecture knowledge improves review quality

Code Review Challenges Identifies recurring challenges and inefficiencies encountered in the code review process.

- Overwhelming size of PR
- Delays in code review feedback
- Mixed-type PRs (feature + refactoring)
- Context switching interrupts productivity
- Manually finding vulnerabilities in dependencies

Current AI Adoption Describes the current state of AI tool usage in development and code review.

- Use of GitHub for code reviews
- No AI tools used in code reviews
- AI not used in development
- AI used in development
- Use of Copilot in development
- Use of ChatGPT in development
- Skepticism about AI-generated code quality
- Use AI to generate boilerplate code
- Use AI for generating simple documentation
- Feels AI is useful in development

Possible AI Adoption in Code Reviews Describes developers' expectations, suggestions, and concerns regarding future AI assistance in code reviews.

- Possible to use AI to summarize PRs
- Possible for AI to detect subtle bugs or architectural issues
- Possible to use AI to explain code
- Possible to use AI to check if code fulfills requirements
- Positive towards AI in code reviews
- Possible to get AI help with dependency versions and vulnerabilities
- Possible to use AI to enforce code standards
- AI in code reviews might suggest false positives
- AI in code reviews might reduce interaction between team members
- Possible to use AI to verify the relevance of commit comments

A.2 Phase 2 - Data

A.2.1 Post-Experiment Interview Question List

Below are the questions used to guide the Phase 2 interviews exploring developer experiences with an AI assistant during code review:

1. Can you describe your experience using the AI assistant during the code reviews?
2. Did the AI assistant change the way you approached the code review task? If so, how?
3. Between the two interaction modes (Mode A: Co-Reviewer and Mode B: Interactive Assistant), which one did you prefer and why?
4. Did the AI assistant help you get enough context to understand the pull request (PR) faster during the review?
5. Do you think this tool, or a more advanced version of it, would be helpful when reviewing large or complex PRs?
6. In what ways did the AI assistant help you during the review? In what ways, if any, did it hinder you?
7. Do you see any risks or concerns with using AI tools like this in code reviews?

A.2.2 Identified Themes and Codes from Thematic Analysis: Phase 2

The following section presents the results from the thematic analysis [30] conducted on the qualitative data collected during Phase 2 (field experiment). Four distinct themes were identified, each accompanied by a set of codes that reflect recurring insights and participant observations.

Accuracy, Reliability, and Trust Focuses on the perceived correctness of AI-generated feedback, concerns about over-reliance, and varying levels of trust in the assistant's recommendations.

- Positive general perception of AI
- Positive perception of AI-generated code suggestions
- Trust in the tool is important
- Tool inaccuracy reduces trust
- AI suggestion confirmed by reviewer
- Negative perception of accuracy
- Positive perception of accuracy
- Inaccurate suggestions can cause wasted effort
- Concerns over AI
- Mode A could cause over-reliance
- Not much risk perceived
- Net gain in issue detection despite risk of over-reliance

Efficiency and Thoroughness Captures how the assistant affects review speed, cognitive load, issue detection, and the overall thoroughness of the code review process.

- AI reduces need to access external sources
- AI speeds up code review
- AI improves review quality
- AI reduces tedious effort
- AI reduced need to check code

- AI reduces cognitive or motivational burden of code review
- Tool did not hinder the review
- AI made reviewing code easier
- AI support influenced review approach
- AI support did not influenced review approach
- AI support reinforces human judgment
- AI suggestions as inspiration not automation
- AI expands solution space
- AI brings up low-priority or irrelevant issues

Integration Expectations and Limitations Highlights developer expectations for seamless integration, responsive design, and context-aware suggestions, while also surfacing frustrations related to current UX and tooling limitations.

- Positive towards more reliable version
- Desire for integration into existing tools
- Criticism of AI feedback formatting or structure
- Tool response time not ideal
- Perceived need to improve prompting skills
- AI suggestions/feedback sometimes unclear
- AI usefulness depends on documentation and metadata access
- Limitations due to lack of context awareness
- AI lacks internal project/tool knowledge

Usage Contexts and Interaction Patterns Describes how interaction with the assistant varied based on review context, including preferences for different modes, alternative usage strategies, and team-specific practices.

- Facilitates learning
- Tool useful for gaining context
- Tool more useful for newcomers

- AI especially helpful when reviewing code by new team members
- Perceived benefit in low-review-culture teams
- AI improves less-preferred review processes
- AI feedback facilitates author-reviewer negotiation
- Mode A good for small/low risk PRs
- Mode A providing context
- Mode B good for risky PRs
- Mode B as a extra pair of eyes
- Prefers Mode A
- Mode A for large PRs
- Mode A for unfamiliar PRs
- Mode A for newcomers
- Mode B if familiar with codebase
- Prefers Mode B
- Mode preference depends on context
- AI-generated summary as pre-review aid for authors
- Developer-driven review followed by AI critique
- AI assists in large PR reviews
- Uncertainty about AI efficiency in large PRs
- Degree of AI integration depends on review purpose
- Tool helpful for remote working teams

A.3 Functional Requirements for AI assistant

This appendix outlines the functional requirements that guided the design and implementation of the AI assistant artifact. These requirements served both as design constraints and as validation criteria for the completed system.

- **FR1: Core Chat Interface**

- Provide a web-based user interface displaying a chat window.
 - Allow the user (experiment participant) to input text queries.
 - Display responses from the AI assistant.
 - Maintain conversation history within a single session.
- **FR2: LLM Integration**
 - Uses ReAct Agent with the OpenAI o4-mini model.
 - Supports tool use via function-calling, including access to PR-specific RAG search tools.
 - Includes a dedicated sub-agent that can be called as a tool, responsible for producing the initial review summary at the start of Mode A sessions.
 - **FR3: RAG Infrastructure**
 - Integrate with LlamaIndex to manage the Retrieval Augmented Generation (RAG) process.
 - Use an agentic RAG pipeline to augment prompts sent to the LLM with relevant contextual data using chain-of-thought prompting.
 - Maintain separate RAG databases—one for each PR dataset.
 - Each RAG database must consist of three collections:
 - * PR data and metadata
 - * Source code
 - * Feature requirement
 - **FR3: RAG Infrastructure**
 - Uses LlamaIndex to build and query Retrieval-Augmented Generation (RAG) indexes.
 - Each PR has its own RAG database, composed of three distinct indexed collections:
 - * Pull request content and metadata
 - * Source code files
 - * Feature requirements

- The main and sub-agents can call these tools with a query.
- **FR4: Context Data Loading and Indexing**
 - The researcher must manually gather context data for each PR.
 - The researcher must run the indexing script to build one RAG database per PR from the collected data.
 - The backend must be capable of loading and accessing all pre-built RAG databases during runtime.
- **FR5: Mode A – Co-Reviewer**
 - Upon session start, the system must display a button labeled “Generate Initial Code Review.”
 - When the user clicks this button, the AI assistant must call the `start_review` tool, which then generates a full review of the PR and returns it to the agent.
 - After this summary is presented, the user must be able to ask follow-up questions via the chat interface, which the assistant answers using the same contextual resources.
- **FR6: Mode B – Interactive Assistant**
 - The system must not proactively display any summaries or suggestions.
 - The AI assistant remains passive until explicitly queried by the user.
 - When queried, the assistant must generate a response using the LLM and RAG context.
- **FR7: Mode Selection and Configuration**
 - The frontend UI must include two clearly labeled tabs or equivalent elements to allow researchers to select the desired interaction mode:
 - * Mode A: Co-Reviewer
 - * Mode B: Interactive Assistant
- **FR8: Session Management**
 - The system must manage sessions corresponding to individual participants reviewing a specific PR.
 - Context data and mode configuration must be session-specific.

- **FR9: User PR Selection**

- The frontend must provide a settings interface where researchers can select the PR dataset and participant ID to be used for the current session.

- **FR10: Basic Logging**

- The system must log user-AI interactions (e.g., queries, responses, timestamps).

A.4 Example of AI Assistant Thought Process

This appendix shows an excerpt from an interaction with the AI assistant running in Mode B (Interactive Assistant) during a review of an open-source pull request from a project available on GitHub¹. The user prompt was: “*Explain the additions to the `gen_security.go` file and compare them to the original source code file to find out if they cause any issues*”. The exchange demonstrates the assistant’s tool usage and reasoning loop. This specific example shows how it utilizes both the `search_pr` and `search_code` tools to look up information.

To save space, some responses have been partially omitted, marked with `{...truncated...}` to indicate that output was shortened without modification.

```
> Running step 1449d64c-e03f-41f6-a17c-13e714150f72. Step
  input: Explain the additions to the gen_security.go
  file and compare them to the original source code file
  to find out if they cause any issues
Thought: I need to fetch the diff of gen_security.go from
  the PR to see what was added, then compare it to the
  original source. First I'll retrieve the PR diff for
  that file.
Action: search_pr
Action Input: {'input': "Show changes to file '
  gen_security.go'"}
Observation: Here's a consolidated unified diff showing
  all of the changes in gen/gen_security.go:
diff
diff --git a/gen/gen_security.go b/gen/gen_security.go
index 943aef82..7e06a155 100644
--- a/gen/gen_security.go
+++ b/gen/gen_security.go
@@ -1,6 +1,8 @@
  import (
      "errors"
      "strings"
```

¹The AI assistant was tested using data from the `ogen-go/ogen` project (Pull Request: <https://github.com/ogen-go/ogen/pull/1440>), which is licensed under the Apache License, Version 2.0. For license terms, see <http://www.apache.org/licenses/LICENSE-2.0>.

```

+     "github.com/ogen-go/ogen/openapi"
)
{... truncated ...}

> Running step 87618ed0-0ac2-4a50-8220-02d3ad27b446. Step
input: None
Thought: I need to see the original contents of gen/
gen_security.go to compare the signatures and behavior
before and after the PR.
Action: search_code
Action Input: {'input': 'Show me the original file gen/
gen_security.go'}
Observation:
go
// File: ogen-main/gen/gen_security.go
package gen

import (
    "fmt"
{... truncated ...}

> Running step e3b14638-62bb-485c-9acf-d92698d4ea93. Step
input: None
Thought: I can answer without using any more tools. I'll
use the user's language to answer
Answer: Here's what the PR adds to gen/gen_security.go
and the main points where it diverges from the
original implementation-and where you should watch out
for regressions:

1. New operationName parameter on the two helper methods
- Old signatures
func (g *Generator) generateSecurityAPIKey(s *ir.
Security, spec openapi.SecurityScheme) .
func (g *Generator) generateSecurityHTTP( s *ir.
Security, spec openapi.SecurityScheme) .
- New signatures
func (g *Generator) generateSecurityAPIKey( s *ir.
Security, operationName string, spec openapi.
SecurityScheme) .
func (g *Generator) generateSecurityHTTP( s *ir.
Security, operationName string, spec openapi.
SecurityScheme) .
{... truncated ...}

```