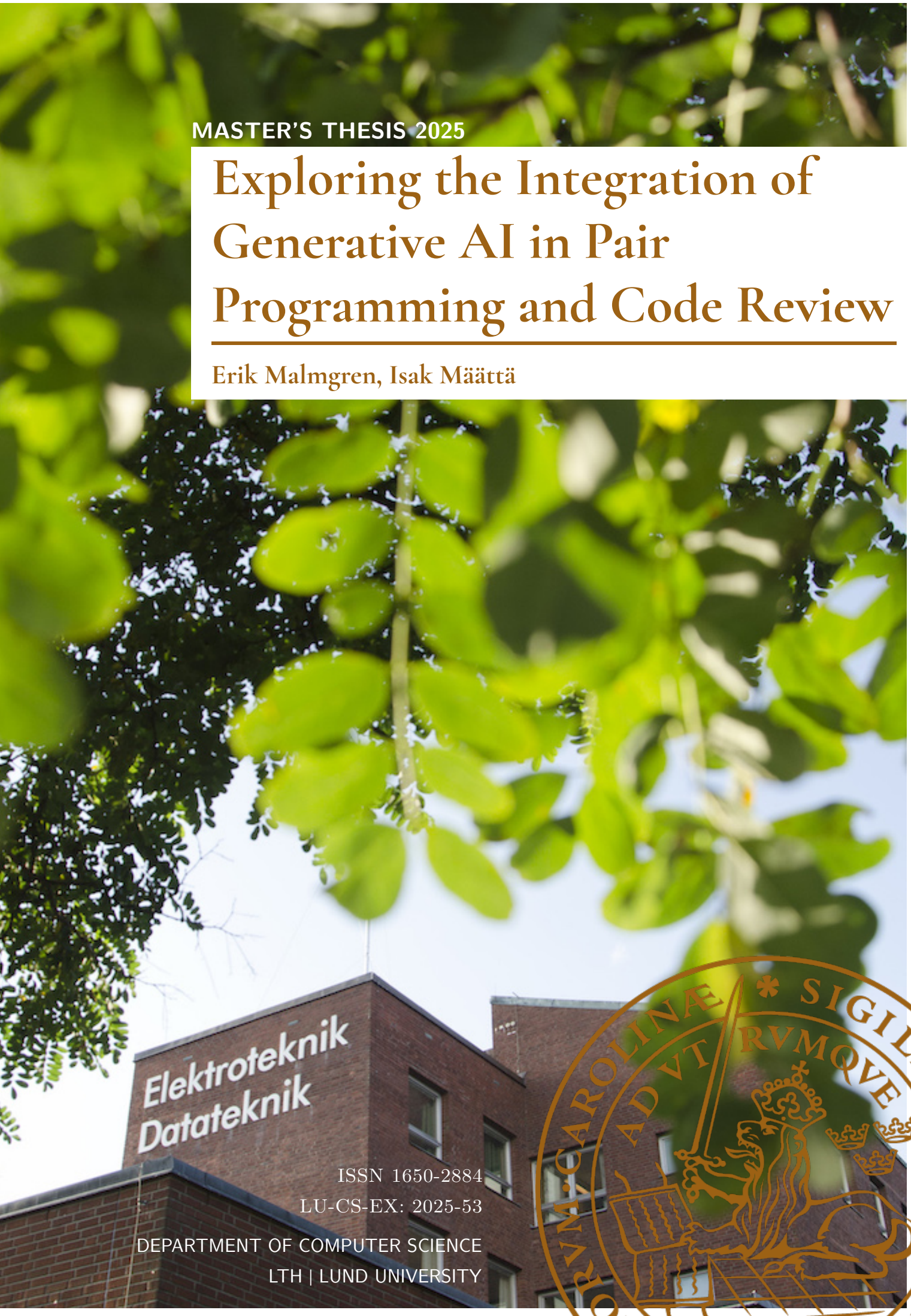

MASTER'S THESIS 2025

Exploring the Integration of Generative AI in Pair Programming and Code Review

Erik Malmgren, Isak Määttä

ISSN 1650-2884
LU-CS-EX: 2025-53

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2025-53

Exploring the Integration of Generative AI in Pair Programming and Code Review

Utforskning av integrationen av generativ AI i parprogrammering och kodgranskning

Erik Malmgren, Isak Määttä

Exploring the Integration of Generative AI in Pair Programming and Code Review

Erik Malmgren
erik@malmgren.dev

Isak Määttä
isak@maatta.se

September 2, 2025

Master's thesis work carried out at E.ON.

Supervisors: Christer Friberg, christer.friberg@eon.se
Lars Bendix, lars.bendix@cs.lth.se

Examiner: Per Andersson, per.andersson@cs.lth.se

Abstract

Software development teams rely on collaborative practices such as pair programming and code review to ensure quality, share knowledge, and maintain productivity. While effective, these practices are time-consuming and often involve repetitive or low-level tasks that divert attention from higher-order problem solving. The rise of generative AI (GAI) tools such as GitHub Copilot introduces opportunities to streamline collaboration, yet their role in real-world workflows remains underexplored. Existing research offers limited guidance on how GAI can complement, rather than disrupt, team dynamics, especially in industrial settings where domain knowledge is essential.

This thesis investigates how GAI can be effectively integrated into pair programming and code review within E.ON's development teams. A mixed-methods approach was applied, beginning with a literature review, developer interviews, and an observational study to inspire and inform two experimental iterations. GitHub Copilot was chosen as the GAI tool, supported by structured prompt guides tailored to specific tasks.

Results show that in pair programming, GAI enhanced productivity in code generation, refactoring, and documentation, while also supporting complex problem decomposition through conversational workflows. In code review, GAI effectively surfaced syntactic and stylistic issues, enabling human reviewers to focus on architectural concerns. Findings suggest GAI adds greatest value as a complement to human judgment, emphasizing the need for structured prompting and thoughtful workflow integration.

Keywords: Generative AI, Pair Programming, Code Review, Workflow Integration, Collaborative Development

Acknowledgements

First, we want to thank our academic supervisor Lars for his extensive feedback, never allowing a moment of potential learning to pass unnoticed, and his persistent encouragement of the practice of sustained espresso consumption.

Moreover, we also want to thank our industrial supervisor Christer, for his input, guidance and inspiring curiosity.

Furthermore, we want to extend our thanks to the members of the Data Analytics team for generously giving their time to participate in our experiments, and for sharing those delightful fika sessions with us.

Finally, we wish to express our profound gratitude to fellow thesis workers Jacob and Alexander for joining us in Pingis Eon Nationella Idrottssällskap, where spilled tea outnumbered great table tennis matches.

Contents

Chapter 1

Introduction

The interest in Generative Artificial Intelligence (GAI) has been substantial and widespread, with its integration into software development practices attracting significant attention and investment. Organizations increasingly recognize GAI tools such as ChatGPT and Github Copilot, subsequently referred to as Copilot, as a powerful assets, with opportunity to enhance productivity and development workflows. Despite the eagerness for GAI adoption, challenges remain in identifying strategies for optimal GAI implementation that address specific development needs.

This master thesis aims to address these implementation challenges by exploring the integration of GAI into two foundational software development practices: pair programming and code review. These practices were chosen because they are central to software development and have proven effective in facilitating knowledge sharing, detecting defects and improving code quality. Although prior research has broadly examined AI-driven code generation, there has been little focus on how these technologies can enhance collaborative development within established teams with diverse technical expertise.

Our research builds upon the practices of pair programming and code review, addressing the primary questions of how the practices are transformed with GAI. The first research question investigates whether GAI can be used as a pair programming partner, examining the implications of GAI assuming this collaborative role. The second research question concerns code review, and how effectively GAI can carry out or assist developers with reviewing code. By focusing on these questions, we aim to provide insights into how organizations can integrate AI in their development workflows.

This thesis was conducted at E.ON, an energy company currently undertaking several machine learning projects to improve its operational processes and energy solutions. The company's development teams offers a research environment shaped by their diverse makeup, combining professionals with expertise in both the energy sector and specialized software development. By focusing on one representative team for in-depth analysis while ensuring insights remain applicable across E.ON's departments, the goal is to develop adaptable AI implementation strategies that benefit not only teams at E.ON but also at companies.

The study follows a mixed-methods approach, focusing on evaluating practical experiments to develop guidelines for integrating GAI into collaborative development. To lay the groundwork, a literature review, observational study, and interviews will be conducted. These initial findings will inform and inspire the second phase of the study, which involves hands-on experiments with GAI integration in the development process.

The study aims to provide guidelines grounded in findings and observations from practical experiments for integrating GAI into collaborative software development. The guidelines will cover its role in pair programming and code review, highlighting both efficiency gains and potential pitfalls. Additionally, we expect to identify factors influencing GAI effectiveness, such as team experience and task complexity. These findings will offer actionable insights for organizations looking to integrate GAI into their development workflows in a structured and informed manner.

This thesis is structured into seven chapters. Following this introduction, Chapter 2 provides the background and theoretical foundations, framing the study. Chapter 3 presents the information gathering phase, which consists of a literature review, developer interviews at E.ON, and an observational study conducted at the same company. Chapters 4 and 5 then present the results from our experimental iterations, each focused on one of the research questions. Chapter 6 contains a broader discussion, including related work, threats to validity, and the generalizability of our findings, as well as suggestions for future research. Finally, Chapter 7 concludes the thesis by summarizing the key contributions and insights.

Chapter 2

Background

In this chapter, we establish the essential context for our investigation into GAI's role in software development practices. By presenting the specific challenges that motivated this work, the context of the work, and the theoretical foundation, we create a shared foundation for the thesis. Key concepts and working definitions are introduced, shaping our approach throughout the thesis, providing clarity in terminology and scope.

We begin by introducing E.ON and its specific challenges, followed by an analysis of the initiating problem. We then explain how our research questions emerged from organizational needs, research gaps, and practical constraints. Next, we outline our methodological approach and justify its suitability for addressing these questions. Finally, we establish the theoretical foundation that connects our study to existing research and positions our work within the broader academic context.

2.1 E.ON

In this section, we provide context for understanding how E.ON's internal processes relate to the study's background. This context is essential for many because E.ON's specific organizational structure and development practices directly influence how GAI can be integrated. Understanding this specific environment help us base our research upon real world factors that lead to practical recommendations. The section will outline how E.ON operates, highlighting factors that influence the integrations of GAI in their development workflows. By examining these aspects, we establish the necessary foundation for later analysis and ensure that the proposed guidelines align with E.ON's organizational structure, challenges and needs.

In this section, we provide context on E.ON's internal processes and organizational structure, which influence how GAI can be integrated into their development workflows. Understanding

this specific environment helps us ground our research in real-world factors, leading to practical recommendations tailored to E.ON's actual challenges and needs. We examine E.ON's operations and development practices to establish the foundation for our analysis and ensure the proposed guidelines align with their organizational realities.

E.ON is a leading energy company focused on sustainable and digitalized energy solutions. As one of the main actors in the powergrid system, E.ON operates extensive infrastructure and relies on a broad range of software to manage energy distributions, optimize efficiency and enhance customer services. Given the critical nature of its operations falling under government regulation, the company must ensure that its digital solutions are secure, reliable and compliant with regulations.

The primary team which the study is mainly conducted in operates in digital development, with about 12 people which will be our main source of information from developers. Most team members work as data scientists, with varying academic backgrounds, along with some machine learning engineers also in the team. The team combines conventional software development practices, utilizing a modified version of scrum as its development methodology, with data science workflows that together create an intersection resembling MLOps, an approach centered on streamlining deployment and maintenance of machine learning models in production environments.

Code review practices at E.ON vary across projects and teams. Some projects prioritize shorter reviews focused on knowledge sharing and maintainability, while others adopt a more comprehensive review process. These differences in priorities and project constraints are important to note, as they influence how GAI can be effectively integrated into the review workflow.

While code review is a well-established practice within the team, pair programming occurs less frequently and is not formally mandated. Instead, developers engage in pair programming spontaneously when they determine it would be beneficial for a particular task. It is primarily employed to bridge the gap between data science concepts and their software implementation. Additionally, these collaborative sessions serve as valuable knowledge transfer opportunities among colleagues.

2.2 Problem Statement

In this section, we define the core problem that motivated this study and explains why it is important for E.ON. By understanding the challenges and opportunities associated with GAI adoption, we establish the foundation for the research questions and ensure the study addresses the relevant concerns. We also describe the driving forces behind E.ON's interest in GAI integration, the specific obstacles it faces and the broader industry context.

There is a growing consensus in the industry that GAI tools are here to stay and will become an integral part of every developers toolkit in the near future. The rapid advancements in GAI have significantly influenced various industries, including software development. GAI tools like ChatGPT saw widespread usage when first introduced, with developers using them for development practices such as pair programming and AI-driven code review, which is showing promising results [1], [2]. These tools have the potential to enhance productivity, improve

code quality and reduce time spent on repetitive tasks, ultimately enabling developers to focus more on the difficult part of their work.

Many companies, including E.ON, recognize the transformative potential of AI in software development, and believe that AI tools could be here to stay and may aid all developers in the future. The introduction of GAI presents new opportunities to enhance software development processes, streamline workflows and improve collaboration within development teams. Because companies that effectively adopt AI-driven solutions can gain a competitive advantage by accelerating the development process, E.ON does not want to miss out on the presented opportunity and end up behind the competition, and therefore seeks enlightenment in how large scale enterprises can integrate AI in their organization. Conversely, organizations that hesitate to embrace AI risk falling behind competitors, missing out on the potential gains from using AI.

Despite the possible benefits of introducing GAI, E.ON will face certain challenges when adopting AI-driven tools, particularly those involving external parties. Concerns around data security, GDPR compliance and legal constraints make the company hesitant to share code or sensitive information with untrusted external AI services. These challenges must be carefully considered when exploring AI integration within E.ON's development practices.

Considering the challenges discussed above, there is a need for E.ON to carefully evaluate how GAI can be integrated into its software development processes while addressing concerns related to security, regulatory compliance and organizational constraints. The purpose of this master's thesis is therefore to investigate how E.ON can adopt GAI-driven tools in the software development process while maintaining alignment with operational requirements, ensuring security and compliance, and preserving control over its infrastructure.

2.3 Research Questions

This section defines the research questions that guide the study, ensuring a focused investigation into the role of GAI in software development. It explains why pair programming and code review were chosen as focal points, how they relate to E.ON's needs and what specific aspects of GAI integration will be explored. By distinguishing between primary and secondary research questions, this section clarifies the study's scope while outlining additional areas of interest that may provide valuable insights if time permits.

Given the problem of uncertainty in adopting GAI-driven tools in the software development process, this thesis aims to investigate how GAI can be effectively implemented while addressing these concerns. GAI introduces many possibilities to augment software developers, targeting different parts of the development process. The ecosystem ranges from tools that generate tests and documentation, real-time code completion assistants, and refactoring aids, to systems that explain complex code, automated pull request reviewers, and interfaces that convert natural language requirements into functional code.

Exploring all the previously mentioned opportunities would be far to exhaustive for the scope of this thesis, so together with E.ON, we narrowed our focus to pair programming and code review based on our first-hand experiences with GAI tools. Our experience with tools like Copilot and ChatGPT revealed particularly promising potential in these collaborative

development practices. From our previous testing, we observed that GAI could offer code suggestions that mimicked aspects of human pair programming dynamics, while also demonstrating an ability to analyze code with attention to detail that could complement human code review.

The decision to focus specifically on pair programming and code review was driven by several compelling factors. First, these foundational activities directly impact code quality, knowledge sharing and team efficiency, critical aspects for E.ON and software teams. Second, they involve human-to-human interactions that could be influenced by GAI augmentation, raising interesting questions about changing dynamics. Third, both practices are widely adopted with metrics for effectiveness. Fourth, they play a key role in facilitating knowledge sharing across teams, ensuring that expertise and best practices are distributed effectively. Finally, they represent different modes of collaboration: pair programming being synchronous, while code review is asynchronous, providing a comprehensive view of how GAI might integrate into software development workflows.

The thesis has been divided into primary and secondary research goals. This was done because we identified the key interesting topics, primary research questions, and side activities which might be required in order to explore the primary research questions, which has been labeled secondary research questions. While the primary goal of this thesis is to explore pair programming and code review the secondary will be at a minimum be kept in mind during the research if time does not permit the exploration of those.

- **RQ1: Can generative AI be used as a pair programming partner?**
- **RQ2: How effectively can AI carry out or assist developers with code review?**
- **RQS1: How do different AI models compare to each other when used for pair programming or code review?**
- **RQS2: How can a company such as E.ON run an LLM on premise?**

2.4 Methodology

In this section, we discuss the methodology used in this thesis and motivate why the specific methodology was chosen. The overall work process is illustrated in Figure 2.1

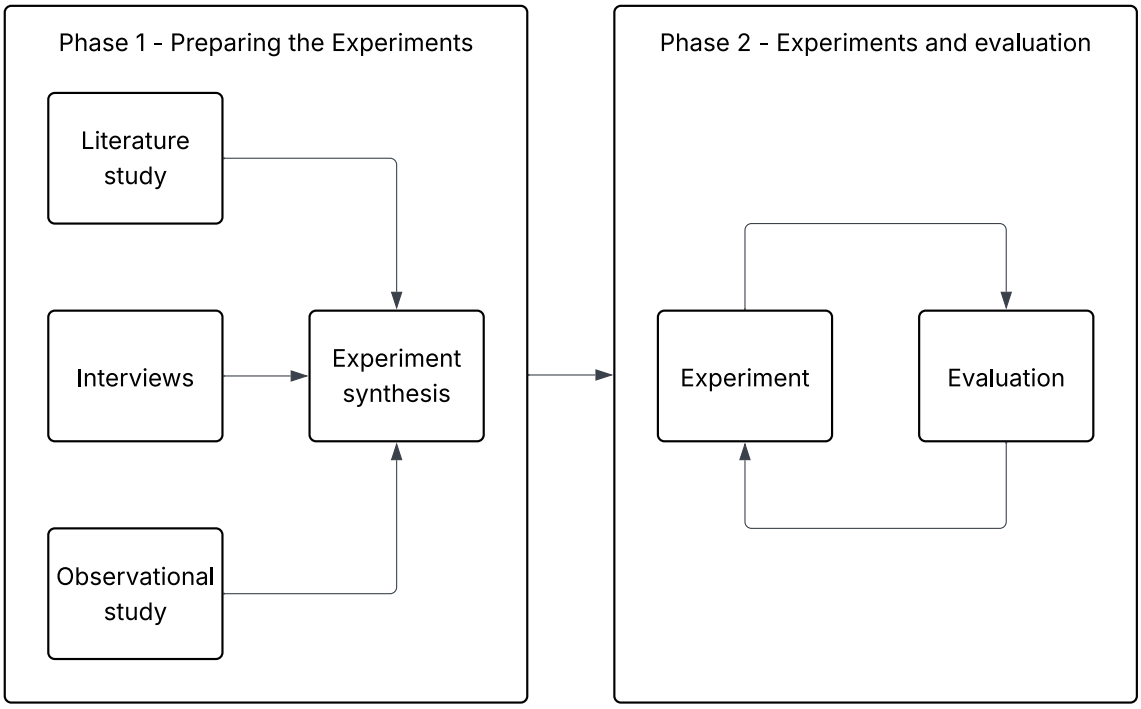

Figure 2.1: Method

The study is structured in two distinct phases. Phase 1 focused on gathering information and drawing inspiration related to the research topic. Insights from this phase served as a foundation for Phase 2, which involved conducting experiments. This two-step approach allowed us to gain an understanding of the current state of GAI usage and to familiarize ourselves with the software development practices at E.ON.

2.4.1 Phase 1 - Preparing the Experiments

To ensure that the experimental phase of this study is grounded in real world needs and current practices, the initial information-gathering phase was designed to provide both inspiration and a solid foundation. The goal was to capture a broad and nuanced understanding of how GAI is currently being adopted, including the challenges and opportunities it presents in software development contexts. To achieve this, we drew on three complementary sources: a literature study, qualitative interviews and an observational study. By diversifying our sources, we aimed to bridge the gap between theoretical potential and practical implementation, ultimately grounding the design and relevance of the experimental phase.

Literature Study

As shown in Figure 2.1, the literature study represents one of three inputs in phase 1. It follows a dual-track approach to gather comprehensive insights from both academic and practitioner sources. Academic studies benefit from the rigor and credibility of scholarly methods, including peer review and established research standards. However, this area of research is still in its infancy. The available academic literature is limited and often based on GAIs built on earlier generations of LLMs (Large Language Models), which may not reflect the capabilities of current systems. To address this gap, we also include discussions, blog posts, and reports from practitioners, who tend to adopt and experiment with new models and practices more rapidly.

For the academic dimension, the literature study will begin by focusing on two key practices in software engineering: pair programming and code review. These activities play a crucial role in addressing challenges related to knowledge sharing, code quality, and team learning. By exploring existing research on these practices, we aim to identify best practices and investigate how GAI could enhance or support them. In addition, we will review current literature on the use of GAI in software development regarding pair programming and code review to gain insight into the academic landscape.

For the practitioner dimension, we will explore discussions and blog posts from developer communities such as Stack Overflow and Hacker News. These sources offer valuable insights into current AI development practices, real world experiences, practical challenges, and experimental use cases, many of which have yet to be documented in academic literature.

Interviews

The interviews are our second source of input for phase 1, as can be seen in Figure 2.1. To motivate the design of our experiments and ensure their relevance to the organizational context, we conducted qualitative interviews with developers and project managers at E.ON. The purpose of these interviews was to gain insights into current software development practices in general at E.ON, as well as the challenges, expectations and perceived opportunities related to the adoption of GAI.

The goal was to include participants from a range of roles and with varying levels of experience, so we aimed to capture diverse perspectives that reflect different priorities and concerns across the organization. This variety allows us to identify role-specific barriers and opportunities, and to better understand how GAI might be integrated in ways that address the distinct needs of individuals and teams at different levels.

Observational Study

The observational study is the third and final source of input into phase 1 shown in Figure 2.1. To gain a more grounded and nuanced understanding of how developers actually work, we complement the interviews with an observational study. While interviews provide valuable insights into participants' reflections and intentions, they may not fully capture the complexities of real world practices. Observations allow us to study development activities as they happen in their everyday environment, revealing tacit behaviors, communication patterns,

and workflows that participants might overlook or take for granted.

This study focuses specifically on how developers engage in collaborative practices such as pair programming and code review. By observing these activities in their natural setting, we aim to identify practical challenges, adaptive strategies, and opportunities where GAI could provide meaningful support. These observations play a crucial role in shaping the design of our experimental phase by grounding it in authentic, day-to-day development work.

Experiment Synthesis

The synthesis shown in Figure 2.1 is the culmination of the first phase. The experiments in phase 2 are not predefined in detail from the outset but will shaped based on the insights gathered during phase 1. Rather than designing the experiments in isolation, we use the findings from from the literature study, interviews and observational study as a foundation for identifying meaningful scenarios to explore. The interviews and observational study ensures that the experimental tasks reflect real world challenges and opportunities within the organization. The literature study complements this by providing theoretical grounding and highlighting current academic perspectives and knowledge gaps related to the use of GAI in software development.

During the synthesis process, we will engage in collaborative discussions and analytical reflection, where patterns and themes from the qualitative data are translated into possible experimental setups. This includes identifying pain points or tasks where GAI could provide support, selecting representative workflows, and considering organizational constraints or needs revealed during data collection. The aim is to ground the experiments in actual development context, thereby increasing both their validity and the relevance of the results for practice.

2.4.2 Phase 2 - Experiments and Evaluation

In the second phase of the study, see Figure 2.1, we have chosen to conduct experiments as we believe this is the most suitable approach for addressing our research questions. These questions are closely connected to developers' actual workflows, and we therefore consider it essential to evaluate potential uses of GAI in realistic development settings. By embedding the experiments within developers' day-to-day work processes, we aim to generate insights that are both relevant and practically grounded.

The experimental phase is designed to be iterative and exploratory. Rather than assuming that a single, static setup will yield optimal results, we recognize that early iterations will likely need refinement. After each round, we will evaluate outcomes, refine existing hypotheses, and potentially formulate new ones. This adaptive approach allows us to respond to unexpected challenges and continuously improve the experimental setup based on observations and participant feedback.

Evaluation will initially focus on qualitative user feedback, which provides a valuable perspective on perceived utility, usability, and potential friction points. We also anticipate that additional factors may emerge organically during the experiments, elements that are difficult to predict in advance but important to consider in assessing the feasibility and impact of GAI

integration. These emergent insights will be incorporated into the ongoing evaluation and interpretation of the results.

Expected Results

Through this experimental phase, we expect to generate both practical and conceptual insights into pair programming and code review in regards to GAI in software development. The primary outcome will be a set of guidelines describing how GAI can be effectively integrated into modern development workflows. These guidelines will address not only where GAI adds value but also how it can be used in a way that complements existing team practices.

For example, one guideline might read: "When using GAI for pair programming with junior developers, set up a three-stage workflow where: First, the developer first explains the programming task to the GAI assistant. Second Both collaborate on writing a test case that defines success criteria. Third, the developer and GAI alternately suggest code solutions, with the GAI providing explanations for each suggestion. This approach maximizes learning opportunities while maintaining the developer's agency in the coding process."

Secondary outcomes could address our supplementary research questions (RQS1 and RQS2), providing both a comparative analysis of different GAI models' performance in software development contexts and a technical framework for on-premise LLM deployment at organizations like E.ON. This might include practical considerations for model selection, resource requirements, and implementation strategies.

A key consideration in our approach is to ensure that the results are not tied to the capabilities or limitations of any single GAI implementation. Instead, we aim for our findings to remain relevant even as new generations of models are released. By focusing on interaction patterns, task suitability, and integration strategies, we hope to produce results that are transferable across tools and resilient to rapid advancements in the underlying technology.

2.5 Theoretical Foundation

The theoretical foundation section provides a foundational understanding of the problem domain, bridging knowledge between industry practice and academic research providing unique context for this thesis. By establishing a shared understanding of agile methodologies, pair programming, code review, and GAI concepts, we create a common vocabulary particularly interesting for readers with varying levels of familiarity with either software development practices or GAI.

2.5.1 Agile Development Practices

Agile methodologies and Extreme Programming (XP) marked a turning point in software development, by moving to flexible iterable approaches that centered on adaptability and collaboration. Together, agile and XP form the foundation of modern software development, largely due to their quality focused and collaborative practices. The core principle of Agile frameworks accentuate delivering working software in short iterations, by prioritizing

individuals and interactions over comprehensive documentation, and embracing changing requirements.

XP is a specific agile methodology, consisting of a set of practices including pair programming and collective code ownership. Code review occupies a distinct but complementary position to pair programming within agile methodologies. While pair programming is carried out synchronously, code review is operated asynchronously involving other perspectives beyond the orignal authors.

Pair Programming

Pair programming is a fundamental collaborative technique in contemporary software development methodologies. In this practice, two programmers work in tandem, with one person, the driver, writing code, while the other, the navigator, reviewing code as it is typed, thinking ahead about potential issues and the direction of the work. Role switching happens frequently to maintain engagement and take advantage of both programmers perspectives.

Studies have demonstrated many benefits of pair programming [3], [4]. They reveal that pair programming has been shown to catch real time mistakes and produce a lower end defect count thanks to continuous code review. Additionally, pair programming leads to better designs with shorter code length, accelerated problem-solving, and improved learning and knowledge, both of the system and of software development, all while increasing work satisfaction.

The practice of regularly rotating pairs facilitates collective code ownership within development teams. Through organized rotation, team members gain familiarity with diverse sections of the codebase, fostering collective responsibility that ultimately increases code quality [5]. The mechanic of of rotating pairs enables developers to share techniques and best practices, effectively reducing knowledge silos that typically occur when information is not distributed among other individuals.

Knowledge transfer constitutes another benefit of pair programming in teams [6]. New team members can quickly learn development practices and deepen their understanding of the codebase by pairing with experienced developers. This coach-like setting not only speeds up technical learning, but also strengthens team relationships and improves communication skills beyond the current programming task.

Code Review

Code review is a systematic process where code is examined by developers to identify bugs, improve quality, and overall adherence to code standards before it is integrated into the codebase [7]. Similar to pair programming, code review promotes collective ownership, facilitates knowledge sharing and supports bug detection. However, unlike pair programming, it is typically carried out asynchronously and involves reviewers other than the original author.

Modern code review has evolved from formal inspections to lightweight, tool-assisted approaches that adapts to distributed teams and fast-paced development cycles [8]. In this context, code review serves as a mean for documenting design decisions and creating valuable learning opportunities for team members [9]. Moreover, it enables input from stakeholders

with specialized expertise who may note be available for real-time collaboration. For example, a security or performance specialist who cannot participate in pair programming can still contribute by reviewing code asynchronously.

Code review practices vary in depth and intensity. At one end of the spectrum are comprehensive, quality-driven reviews that may last longer. On the other end are quick, lightweight reviews, often just 2–3 minutes long, that serve as sanity checks to confirm that the code meets essential standards and can be understood and maintained by others. It is not just the depth and intensity of a code review that affects the reviewer, the size of the code under review also plays a significant role, as larger review sizes are associated with decreased review usefulness [10].

The range of approaches to code review reflects differing priorities and constraints across teams and organizations. Successful code review practices balance efficiency and comprehensiveness, adapting to specific project and contributor needs. As we explore the potential of generative AI in the context of code review, the diversity in review intensity and purpose opens up distinct opportunities for integration.

2.5.2 Introduction to Generative AI

In this thesis, GAI is examined in the context of software development with particular focus on how it can assist in collaborative tasks such as pair programming and code review. Rather than exploring the internal workings of generative models, this work focuses on the practical integration into development workflows and the implications of their use. Understanding the general properties and limitations of GAI provides essential context for evaluating its role and effectiveness in these settings.

GAI refers to a category of artificial intelligence systems designed to create new content based on patterns learned from existing data [11]. Unlike traditional AI systems that primarily classify, predict or recognize inputs, GAI models are capable of producing original outputs such as text, images and code. These outputs are generated by learning statistical relationship and structures within large datasets, enabling the system to simulate human-like creativity. Importantly, the generation process in probabilistic, meaning the same input can produce different results depending on the context and randomness inherent in the model [12]. This ability to generate novel and diverse outputs distinguishes GAI from other forms of AI.

A defining feature of GAI is its prompt-driven interaction model. Users provide input that serve as instructions or questions, and the model generates relevant outputs in response. This interaction style makes GAI accessible to both technical and non-technical users, enabling flexible applications across domains. This accessibility has led to adoption in various implementations, from conversational assistants like ChatGPT to specialized coding tools such as Copilot that integrates directly into the IDE to provide contextual suggestions and assist with programming tasks.

GAI is particularly well-suited for the collaborative software development tasks explored in this thesis due to several key characteristics. Its ability to understand and generate human-like text enables natural communication with developers, while its code generation capabilities allow it to contribute meaningfully to programming activities. The context-awareness of

modern GAI systems helps them maintain coherence across extended interactions, essential for both pair programming sessions and iterative code reviews. Furthermore, GAI's availability as an always-present collaborator addresses challenges of scheduling and resource constraints often encountered in traditional human-to-human collaboration. These properties position GAI as a potential complement to human expertise in software development workflows, potentially enhancing knowledge sharing and code quality while reducing cognitive load, precisely the aspects this research aims to investigate through the lens of pair programming and code review practices.

Despite its impressive capabilities, GAI has notable limitations. One issue is hallucination, where the model generates fluent but factually incorrect or nonsensical content. Additionally, since these systems learn from large, often uncurated datasets, they reproduce and even amplify existing biases [13]. Generative models also lack true understanding or reasoning abilities, as they rely on pattern matching rather than comprehension, which can limit their usefulness in complex or high-stakes scenarios.

Chapter 3

Results 1 – Preparing the Experiments

Pair programming and code review encompass many facets and approaches in software development. To evaluate how GAI can enhance these practices, we need to develop meaningful and effective experiments. In order to design conducive experiments, we must first identify relevant practices, challenges, and opportunities where GAI can reasonably *augment* pair programming and code review at E.ON. To discover the most promising potential applications and use cases, we conducted a comprehensive information gathering phase. By triangulating data from diverse sources, we aimed to identify themes that would form the foundation for well-grounded experiments.

In this chapter, we present the results from our initial information gathering phase. First, we start by presenting insights from our literature study, which examines both academic sources as well as experiences from the developer community. Second, we present our results from the interviews with E.ON developers, focusing on way of working and attitudes towards GAI adoption. Third, we discuss our observational study of developers, where we take note of workflow patterns and possibilities for GAI integration. Finally, the chapter is concluded by a synthesis of important key themes.

3.1 Literature Study Findings

The literature study revealed valuable insights into GAI in software developments, with practitioner experiences contributing to more inspiration than the academic sources. This was partly expected, as the rapid pace of tool development and adoption often outpaces academic publishing, making blog posts, talks, and online documentation more reflective of current practice and experimentation in real world settings. By examining peer-reviewed publications alongside developer community discussions and blog postings, we identified

emerging patterns, potential applications, and common challenges for AI integration in pair programming and code review.

3.1.1 Academic Literature

To review the state of academic research, we conducted a literature search focused on GAI in pair programming and code review. The search emphasized recent work, reflecting the rapid evolution of LLM-based tools and practices.

Search Strategy

Databases The literature search was carried out using a combination of databases.

- **LUBsearch**, this search engine include a selection of other databases. For our research, the primary results are usually from IEEE and arXiv.
- **Google Scholar**, for broad academic indexing.

Many studies on human-AI collaboration label the activity as pair programming, but often diverge from the original concept introduced by Kent Beck. Instead of a structured driver-navigator dynamic, the interaction typically involves a single developer using an AI assistant more as a tool than a true programming partner. This shift complicates comparisons with traditional pair programming and limits the transferability of earlier findings, making it more challenging to conduct a clear and effective information search in the academic literature.

These databases were selected to provide both reliable access to peer-reviewed technical literature and broad coverage of emerging research, ensuring a comprehensive view of the current academic literature regarding GAI in relation to pair programming and code review.

Keywords and Search Strings Search queries we developed iteratively based on initial readings and refined as new terms and concepts emerged. The goal was to capture both established terminology and emerging phrases related to the use of GAI in the software development practices pair programming and code review. The following search strings gave the best results.

```
"generative ai AND pair programming"
"ai AND pair programming"
"code review generative ai"
"code review large language model"
"generative ai AND ("developer productivity" OR "developer experience")"
"ai cognitive load"
"human-ai collaboration"
```

Time frame The searches were performed during end of March and early April of 2025. As the field of GAI is rapidly evolving, attention was given to the most recent literature, particularly from 2022 onward. In total, we identified and studied nine different relevant papers.

Current State of Pair Programming Research

GAI demonstrates particular strength in suggesting code, especially in unfamiliar domains, and in rapidly generating boilerplate or template code [1]. A notable benefit is its ability to assist with navigating and using software library APIs. However, a frequently discussed challenge in the literature is the issue of "hallucinations," where the GAI proposes non-existent functions, parameters, or configuration options [1]. This can lead to code that compiles and runs but produces incorrect results, particularly when more complex algorithmic reasoning is required. Additionally, some suggestions may involve deprecated functions, likely due to outdated examples present in the training data.

Recent research on human-AI pair programming reveals both potential and uncertainty. While perceived productivity is often high, frequently correlating with how often GAI suggestions are accepted, comprehensive evaluation metrics are still lacking [14]. Compared to human-human collaboration, working with an GAI may lack meaningful communication, suggesting this is a distinct form of teamwork. Factors like task complexity, programmer expertise, and the need for debugging influence outcomes. The literature emphasizes the need for realistic field studies, especially involving developers, and calls for better-designed GAI tools that can adapt to human needs, support effective collaboration, and encourage learning [14].

GAI tools, such as Copilot, are transforming software development and reshaping how pair programming is understood [1], [15]. These tools act as developer assistants, capable of generating code and providing interactive support during development. Research shows that developers often report feeling more productive when using GAI in pair programming contexts [16].

Early user studies on tools like Copilot indicate that developers often accept AI suggestions for the sake of efficiency, potentially at the cost of reduced autonomy and control over the codebase [16]. In response to these concerns, recent research has focused on improving the quality of training data, for instance, through curation pipelines designed to enhance the clarity, conciseness, and civility of review comments used in model training [17].

Another aspect gaining attention in recent research is the role of GAI in managing cognitive load during development tasks. Studies suggest that these tools can help reduce mental effort by presenting information in ways that align with cognitive learning principles, thereby supporting more efficient processing and decision-making [18]. However, the effectiveness of this support depends on how easily users can interpret and engage with the AI's outputs. When interaction becomes cognitively demanding or the AI's behavior is difficult to understand, user engagement may decline [19].

Code Review Practices and Challenges

GAI is increasingly being integrated into the code review process, particularly in its early stages. Tools powered by GAI can offer preliminary feedback by identifying issues related to code formatting, adherence to best practices, and potential defects [20]. This type of automated assistance can serve as a first-pass review, helping to surface routine concerns before the code reaches a human reviewer.

While GAI shows promise in assisting parts of the code review process, several challenges

persist. A key issue is trust. Developers may be hesitant to rely on AI-generated feedback, especially when it comes to nuanced design decisions or context-specific practices. There is also concern that AI-generated suggestions might lack the depth or contextual sensitivity required for complex codebases, leading to superficial or misleading recommendations. As such, the role of GAI in code review is still emerging, with ongoing debate around when and how it adds the most value.

Practitioners have also begun exploring the use of GAI to support self-review, where developers receive AI-generated feedback on their code prior to formal submission [21]. In this context, GAI acts as a virtual reviewer, offering suggestions and highlighting areas for improvement, which may ultimately reduce the cognitive load on human reviewers and improve the overall quality of submissions.

In addition to identifying issues, GAI tools are also being developed to suggest actionable improvements, further expanding their role within the review process [21]. These capabilities position GAI not only as a support mechanism but as a potential contributor to faster, more iterative review cycles.

To improve the effectiveness of such systems, researchers are curating high-quality code review datasets to fine-tune large language models. This work emphasizes enhancing the clarity, conciseness, and civility of review comments, which in turn enables models to generate more useful and contextually appropriate feedback [17].

Where Could GAI Be Helpful?

Taken together, current research illustrates that GAI is largely understood as a productivity tool in software development, especially effective in generating boilerplate code and other less critical supporting structures. Developers often report a perceived increase in output when working with GAI tools, an effect that has been quantified in some studies, though often using older model versions such as GPT-3.

In pair programming contexts, GAI is most commonly used as a supportive assistant, rather than as a strict implementation of the driver/navigator model from XP. Its role typically resembles that of a background collaborator that can offload routine tasks or assist with unfamiliar APIs, rather than engaging in co-creative dialogue or architectural decision-making. While GAI can reduce friction in early development steps, its support is less reliable when navigating complex logic or architectural choices, where incorrect suggestions or hallucinations may go unnoticed unless the developer is sufficiently experienced.

For code review, the literature suggests that GAI is well-suited as an aid in either pre-review or live review contexts. Its capacity to identify formatting issues, flag common problems, and suggest improvements enables faster iteration cycles and can ease the burden on human reviewers. There is a growing recognition that GAI can support self-review practices by providing developers with preliminary feedback before code is submitted for formal review. However, effective use of these tools still depends on a developer's ability to interpret, critique, and selectively accept GAI suggestions.

Overall, GAI tools show considerable promise across both pair programming and code review scenarios. Their success, however, hinges on thoughtful integration, clear communication of

model limitations, and continued refinement of training data to improve feedback accuracy and reduce the risk of over-reliance.

3.1.2 Developer Community Insights

To complement our academic literature review, we conducted a exploration of the developer community, looking at discussions, blog posts and other experience reports. Given the rapid evolution of GAI practices and tools, this approach helped us discover practitioner experiences preceding academic studies. The platforms we searched include Hacker News, Stack Overflow and Medium. The terms used in the search were related to "AI pair programming", "AI code review", "AI/LLM coding workflow" and "generative AI software development".

Our initial search revealed that many posters described practices that effectively constituted AI-assisted pair programming without using the pair programming terminology. In contrast, code review as a descriptor was more commonly used. This lead us to broaden the search criteria from the previous terms, aiming to capture developer practices with AI, regardless of what developers labeled them.

Emerging AI-Assisted Workflows

A consistent theme in developer discussions was the structured workflow using GAI as a collaborative partner during the development process. One poster described their approach as "*Brainstorm spec, then plan a plan, then execute using LLM code gen. Discrete loops. Then magic.*" [22]. This workflow contains three distinct steps that reflect aspects of traditional pair programming.

First, in the "idea honing stage", developers engage with the AI to refine specifications iteratively, which resembles how requirements are discussed before coding in pair programming. The AI is used to help clarify developer thoughts and identify problems beforehand. The jointly produced specification is later used to guide the implementation.

Second, during the "planning" stage, the specification is broken down into smaller, manageable, implementation steps by prompting the AI. This decomposition of the specification is similar to the strategic thinking the navigator performs in traditional pair programming, where the navigator considers the overall approach while the driver focuses on the coding task.

Finally, in the "execution stage", the solution is implemented by following the plan established in the second step, using AI to assist with generating code, and finding errors or suggesting improvements. Here, the driver and navigator roles become more fluid as needed, showing a dynamic relationship rather than a predetermined role assignment. The workflow is adapted to work on existing codebases, where the importance of providing context to the AI is emphasized.

The matter of knowledge sharing is also discussed. One developer extends its prompts by prompting the AI to ask additional questions to further its understanding of what the developer is trying to achieve [23]. This two-way communication appear to mimic how a junior developer might learn by asking a senior developer during pair programming.

Code Review Perspectives

Conversations among developers about AI-assisted code review highlighted a mix of optimism and careful consideration. Many noted that they use AI tools to check their code prior to human review, but there was agreement that, for now, AI is most effective as a supplement to human reviewers rather than a substitute.

One post suggested that considerable contextual information is required for effective AI code review [24], with examples given including diff tools, title and description of pull request, access to the code base and historical pull requests and comments, along with the ability to search through the developer teams communication channels. The requisite context marks the difficulty in integrating an AI code review tool that takes context from different domains into account.

Furthermore, it is pointed out that AI still doesn't grasp the more intangible parts of software development. The post puts it this way: *"It probably doesn't know how your product roadmap shifted after a big meeting with The Customer. It won't capture your team's subjective bias toward composition over inheritance. It can't weigh personal or strategic factors that might dictate an otherwise suboptimal code pattern. Real code review demands domain expertise and forward-thinking alignment"*, underscoring the need for human judgment that draws on organizational context.

AI code review is instead positioned as most valuable for handling minor issues, like style guide concerns or performance traps, especially when integrated within continuous integration pipelines, offering quick feedback.

Developer Mindset and Approaches

By broadening the search criteria, we found more general discussions about the use of GAI by practitioners, which indirectly influence how GAI can be used as a pair programming partner and carry out or assist developers with code review. This approach helped point ut practices and experiences that affect similar functions in the development workflow.

Community discussions revealed a mindset when working effectively with GAI [25]. The importance of maintaining responsibility and agency when using AI tools in the development process was emphasized. Also, it was suggested that developers approach AI assistance as an iterative process which involves refinement and adaptation rather than expecting the perfect result on the first attempt. The importance of clear and specific communication was noted, with a concrete implication that developers should provide detailed context concerning the technology stack, coding conventions, and architectural goals, to enhance the relevance and usefulness of AI-generated suggestions. The principles of clear communication, shared responsibility and mutual learning parallel the best practices used for effective human-human pair programming.

Community Sentiment Analysis

Our examination of developer community insights provided valuable context for addressing our research questions about GAI in software development, and by analyzing these practices and real world experiences, we can better understand how GAI might be effectively integrated into E.ON's development workflow.

Regarding pair programming, our findings suggest that a more flexible approach is required for effective GAI collaboration, compared to traditional human-human pair programming. Instead of determining role assignment beforehand, developers utilize dynamic workflows where the roles are based on GAI capabilities and the specific task. When organizing GAI-human collaboration, the three steps of idea honing, planning, and execution, provides a promising structure for collaborating in a manner that maximizes the strengths of both the developer and GAI. Furthermore, encouraging GAI to ask clarifying questions creates a two-way flow of knowledge, offering an interesting representation of the learning advantages traditionally associated with pair programming.

For code review (RQ2), the community discussions indicate that GAI tools currently are most effective when used as preliminary reviewers or assistants, aiding but not replacing human judgment. Using GAI to identify stylistic issues and smaller performance traps was deemed successful, effectively forming a structured, multi-stage review process. However, the community pointed out that successful implementation depends on integrating GAI with current development tools and giving it access to organizational knowledge, ensuring it has the context needed to be effective.

These insights from the developer community, alongside findings from academic literature and organizational interviews, offer a well-rounded basis for designing experiments tailored to E.ON's specific development challenges. The trends emerging in AI-assisted development point toward valuable opportunities to improve both pair programming and code review workflows. At the same time, they underscore critical factors such as the need for clear context, rigorous quality checks, and thoughtful distribution of responsibilities between human developers and AI tools.

3.2 Interview Results

Our discussions with E.ON developers provided context for understanding how GAI might fit into existing development workflows. We conducted interviews with team members representing different experience levels and technical backgrounds within the development process. The interviews helped bridge the theoretical possibilities identified in the literature review with the practical realities of E.ON's development environment. By building an understanding of the organization's specific context through the interviews, we were able to uncover workflows, cultural factors, and challenges, which ultimately helped design experiments that were more meaningful and led to actionable results rather than generic findings.

3.2.1 Participants, Expertise and Attitudes Toward GAI Adoption

We conducted semi-structures interviews with two E.ON developers, one junior and one senior, which aligned well with the exploratory nature of our research, offering a balance between structure and flexibility, which ensured consistency while allowing us to act on unexpected insight or ideas. The interview guide, included in appendix A, was developed based on preliminary findings and included specific question paths for participants with varying levels of prior GAI experience.

The diversity in participant selection, though limited in number, enabled us to capture a broad spectrum of perspectives on current practices, challenges and attitudes toward AI adoption. This variety is important, as the adoption and effectiveness of GAI may vary across different career stages and technical specializations. Senior developers could evaluate AI in relation to its alignment with established practices and its potential to enhance existing workflows, while junior developers might focus more on how it can help solve specific technical tasks.

The decision to limit interviews to just two developers was based on two main reasons. Primarily, our research used a triangulation approach, where interviews were only one of three data sources alongside the literature study and observational study. Their main role was to offer contextual insights that complemented the other data, so they didn't need to be comprehensive on their own. Additionally, the exploratory nature of our research meant that the goal of the interviews was to identify promising starting points for experimentation rather than gathering all possible viewpoints. While time and resource limitations further narrowed the scope of the interviews, deliberately choosing participants with differing levels of experience proved well-suited to our objectives.

Our interviews revealed notable differences in attitudes toward GAI adoption based on experience level. The senior developer demonstrated more enthusiasm, already integrating AI tools into their daily workflow, while the junior developer showed more cautious interest but remained open to experimentation. This insight was relevant to our research, as developer attitudes impact how effectively GAI can be integrated into processes like pair programming and code review. Most significantly, the interviews identified concrete workflow challenges and improvement opportunities that, when integrated with insights from the literature review and observational data, provided sufficient context to guide the design of our experiments.

3.2.2 Current Development Practices at E.ON

We asked participants about their current development practices to gain a clear understanding of E.ON's existing workflows before introducing AI interventions. By mapping these practices, we could identify inefficiencies, collaboration patterns, and quality assurance methods that could be improved through GAI integration. We also identified certain process challenges that, while beyond the scope of GAI solutions in this research, merit future investigation.

Our interviews revealed several characteristics of practices. The team operates withing a culture that does not expect immediate responses to communications, where response times from one to four hours are acceptable. Test practices follow a "test-after" approach rather than test-driven development, with participants noting inconsistencies across projects in the implementation of integration tests. Communication between different project teams was identified as insufficient. This contributes to barriers in collaboration and occasional uncertainty in decision-making processes. The code review workflow often includes large pull requests that are reviewed asynchronously, with less real-time collaboration during development. Overall, we found that development work is largely done individually, with pair programming and other synchronous collaboration practices happening rarely and on an ad-hoc basis, rather than being part of a consistent workflow.

While these findings suggest that GAI-assisted pair programming could offer benefits, such as providing immediate feedback when human collaborators are unavailable, we recognized that

AI alone wouldn't resolve deeper issues related to organizational communication structures. Similarly, for code review, GAI could help manage large pull requests by offering initial feedback and flag common issues, but it may not address the underlying causes that lead to oversized pull requests in the first place. These insights could lead us to design experiments that would use GAI to address specific pain points, while also acknowledging its limitations in solving organizational challenges.

3.2.3 Specific Integration Ideas from Practitioners

Our interviews revealed several concrete experiment opportunities for GAI integration that address the workflow within E.ON's development practices. These opportunities emerged from our discussions with developers, connecting to specific issues that emerged from the developer discussions.

The first opportunity stemmed from our discovery that unit testing represents a workflow challenge. Developers highlighted that, although testing is crucial for ensuring quality, it frequently acts as a bottleneck, delaying release cycles. This reflects the team's tendency toward a "test-after" model, where testing follows development rather than running in parallel. To mitigate this issue, we identified the potential of GAI to automatically generate unit tests from requirements specifications. This capability could shift testing into the best practice of test-driven development.

The second opportunity relates to the complexity of creating mock objects for testing. GAI could potentially simplify this process by automatically creating suitable mock objects, helping ease developers' mental effort and help standardize the inconsistent testing practices seen across projects.

The third opportunity concerns code review specifically, as interest in GAI being used to enhance code comprehension was expressed. Using GAI to produce contextual explanations and documentation for complex code could help reviewers more quickly grasp unfamiliar sections, especially when evaluating code from other teams.

3.3 Observational Study Findings

The observational study aimed to observe how developers at E.ON work in their natural environment and to identify opportunities for GAI support that might not have been surfaced during interviews. We invited ourselves into a developer's regular workspace and asked them to prepare a piece of functionality to implement, allowing us to observe active coding. Our approach was to intervene minimally, only asking clarifying questions when necessary. While the study did not proceed exactly as planned, the limited observations we gathered still provided insights that complemented the findings from the interviews, helping to confirm that there are opportunities for AI integration in the development workflow.

3.3.1 Challenges and Outcomes of the Observation

The observational study did not unfold as we had initially planned. The developer appeared to be influenced by our presence, and the session became more of a demonstration of the steps leading up to production-ready code rather than a natural coding workflow.

We identified several factors that contributed to the observational study challenges. Our inexperience with conducting observational studies led to insufficient time allocated for the actual observation, a lack of clear instructions provided to the developer, and scheduling challenges due to limited time availability. Despite these limitations, we still gained valuable insights into certain workflow aspects, particularly concerning strategies related to Git usage, which complemented the information gathered through our interviews.

After evaluating the initial findings and considering the trade-off between potential insights and the resources needed, we concluded that additional observation sessions would likely offer diminishing returns. Although more sessions could have yielded some extra insights, the existing data already offered enough guidance for designing experiments, and limited team availability made further scheduling difficult.

3.3.2 Identified Opportunities for AI Integration

During the observational study, we identified a potential opportunity for GAI integration with in the existing CI/CD pipeline. The pipeline currently includes multiple steps, such as running integration tests before code reaches production. We observed that there could be room to incorporate a GAI code analysis step. Similar to a linter, but with the purpose of doing an automated code review, analyzing code quality, detecting potential bugs, or suggesting improvements before human code review.

3.4 Synthesis and Experiment Direction

In this section, we distill our findings into actionable insights to guide the experimental exploration of GAI in software development at E.ON. By combining insights from the literature, interviews, and observational data, we identify recurring themes and opportunities where GAI could add meaningful value to pair programming and code review. This synthesis connects the

information gathering phase and experimental phase, anchoring the research in theory and the specific context of E.ON. In doing so, it lays the foundation for the experimental designs presented in the next chapter and concludes our investigation on how GAI can support pair programming and enhance code review.

We begin by presenting key themes that emerged from our data sources. These themes form a conceptual foundation for understanding how collaborative development might be shaped by GAI. We then outline targeted research areas and questions tied to GAI's role in pair programming and code review. Finally, we present the results as specific areas where GAI could meaningfully support software development, including when to use GAI, its potential as pair programming partner, how it compares to human review and the importance of effective prompting strategies.

3.4.1 Key Themes Across Gathered Information

In this section, we identify key themes that emerged from our information gathering phase, which connect to the initiating problem of introducing GAI in the software development process. By examining patterns across the literature findings, developer interviews and observational data, we highlight factors influencing the integration of GAI, chiefly with pair programming and code review.

Knowledge Transfer

Both the interviews and the literature review highlighted challenges in knowledge transfer among team members, particularly across projects. To address this issue, developers expressed interest in using GAI to enhance code comprehension during reviews, noting that it could help clarify unfamiliar code. However, concerns were also raised about the potential downsides of relying too heavily on GAI in this role. In particular, there is a risk that junior developers may become overly dependent on AI-generated suggestions, which could reduce their engagement in problem-solving and limit opportunities for learning. When GAI is used as a pair programming partner, this risk may increase if the human developer becomes passive, missing valuable chances to build foundational skills and a deeper understanding.

Expertise Dependency and Critical Assessment

The developer community suggests that the effectiveness of GAI depend strongly on the user's level of expertise. Developers with more experience are generally better at providing the necessary context for GAI to perform well, crafting precise prompts, and evaluating the quality of the generated outputs. These developers are also more likely to detect subtle errors or hallucinations. Because less experienced developers may struggle with these tasks, the performance gap can widen without targeted support. This suggests that introducing GAI into teams with mixed skill levels requires strategies for support and guidance, such as reusable prompt templates or curated examples. Not surprisingly, critical assessment remains an essential skill, regardless of experience, since trusting GAI suggestions without verification can introduce serious quality or security risks.

Lowered Barriers to Advanced Techniques

The literature study noted that GAI has the potential to make advanced programming techniques more accessible to a broader group of developers. By generating example code or explaining unfamiliar concepts, GAI can encourage experimentation and learning. This theme was particularly relevant for developers exploring new tools, libraries, or programming languages. GAI was seen as a valuable aid in projects where fast prototyping or learning on the fly is required.

Productivity Enhancement

GAI was frequently praised in the developer community for improving productivity, especially in routine or repetitive tasks. Developers in the community reported using GAI to handle boilerplate code, documentation, and syntax corrections, which allowed them to focus on more complex aspects of their work. In pair programming or code review scenarios, GAI helped by offloading lower-level work, enabling the human developer to concentrate on design decisions, architectural thinking, or deeper code analysis. For organizations such as E.ON, the potential for improved productivity is a central reason for exploring GAI adoption. However, further study is needed to measure the long-term effects on productivity across different types of development tasks.

Context Importance

The quality of GAI output is heavily influenced by the context it receives. Especially in the literature study, there was a clear emphasis on the need to provide relevant, specific, and persistent context in order to get useful responses from GAI tools. Without proper context and task description, GAI suggestions may be off-target or misleading. This finding highlights the importance of prompt engineering and the development of context-sharing practices, such as including structured documentation, in-line comments, or prompt templates tailored to specific projects.

Balance Between Automation and Human Judgment

A consistent theme across the literature and interviews was that GAI works best when used to assist rather than replace the human developer. Developers who actively guided the interaction and used GAI suggestions as a basis for further reasoning reported better outcomes than those who relied on GAI outputs without scrutiny. This balance is particularly important for preserving accountability and ensuring that critical thinking remains part of the development process. Maintaining human oversight is essential, especially in collaborative practices like code review, where decisions can have long-term consequences for code quality and maintainability.

3.4.2 Areas of Interest and Key Questions

In this section, we translate our research insights into focused questions that highlight promising opportunities for experimentation. Each area of interest marks a point of convergence between findings, indicating where integrating GAI could offer meaningful impact.

When to use GAI

GAI can be a valuable aid throughout the software development process, but it is not always the most appropriate tool for every situation. If used indiscriminately, there's a risk that every problem begins to resemble one that GAI is designed to solve, potentially narrowing developers' problem-solving approaches. In some cases, it may be more effective to work without GAI.

There are, however, several stages in the development process where GAI has proven to be particularly useful. Both the literature and our interviews highlight that GAI supports early-phase activities such as brainstorming, refining ideas, and exploring alternative approaches.

During implementation, GAI can assist with repetitive or cognitively draining tasks. The literature emphasizes GAI's effectiveness in generating boilerplate code and handling tasks that resemble those of a helpful assistant, thereby reducing developers' cognitive load. Interview participants also noted that GAI could help with complex and tedious activities, such as constructing mock objects.

GAI can also play a role in unit testing as part of pair programming. Interviews revealed that although developers recognize the importance of tests for ensuring that future changes do not break existing functionality, many find writing tests to be an unappealing task. GAI can lower this barrier by generating initial test cases, suggesting test structures, and assisting with mocking.

Another important aspect is GAI's ability to maintain useful context over the course of a session. While some tools can carry limited stat across interactions, they typically struggle to retain broader context. This limitation can become particularly noticeable in long term projects where details provided some time ago might be dropped.

Prompt Engineering

Effective prompting strategies are also essential. Developers need to learn how to communicate clearly with GAI tools to get relevant and useful output. This includes specifying intent, constraints, and expected behavior, often in natural language. Onboarding developers to work effectively with GAI involves teaching these prompting skills, as well as helping them understand what GAI can and cannot do.

Well-structured prompts tend to yield more accurate, targeted and context-aware responses. This may involve including background information, clarifying the stage of development or even formatting the prompt in ways that align with how the tool best interprets the input. Over time, developers may also build an intuition for how to iterate on prompts to refine results, an important part of working efficiently with GAI.

Developers may benefit from lightweight guidelines or examples that illustrate effective prompts in different contexts, such as debugging, generating boilerplate, or asking for design alternatives. Rather than treating prompting as an ad hoc task, teams can approach it as a skill that improves with practice. As developers become more familiar with how GAI responds to different types of input, they can begin to use it more intentionally and efficiently across a range of tasks.

GAI as Pair Programming Partner

Using GAI as a pair programming partner introduces new dynamics into the development workflow. One key consideration is the role GAI should play. Should it act as the driver, generating code directly, or as the navigator, offering feedback, alternatives and high-level guidance? Or a third different role? Sentiment among developers suggest that maybe it should shift continuously, depending on the task and the developers preferences.

Unlike a human colleague, GAI is always available, does not require scheduling and can be engaged on demand. This makes it an attractive option for solo developing which from the interviews seems to be quite widespread at E.ON. The constant availability allows for spontaneous collaboration and rapid iterations without needing to wait for another person.

Finally, while GAI can be a helpful collaborator, it has limitations. It lacks awareness of team norms, informal context, and often cannot assist with high-level architectural decisions that depend on domain-specific trade-offs. Developers need to be aware of these boundaries and use GAI as a complement to, rather than a replacement for, thoughtful human decision-making.

GAI Code Review

In the context of code review, GAI can provide support in understanding unfamiliar code, identifying potential issues, and generating improvement suggestions. Insights from both the literature and interviews indicate that while GAI should not replace human judgment, it can enhance the reviewer's understanding and efficiency.

GAI-based code review differs in nature from human review. It excels at providing consistent and immediate feedback on syntax, code style, and common programming patterns. Developers appreciate its ability to surface minor issues and offer rewrite suggestions, especially in unfamiliar parts of the codebase. This can accelerate the review cycle and reduce the burden of repetitive comments for human reviewers.

However, GAI often lacks the deeper contextual awareness that human reviewers bring. Human feedback is shaped not just by the code itself but also by an understanding of the broader system architecture, team conventions, and the rationale behind design decisions. These dimensions are difficult for GAI to capture, especially when relevant information is distributed across discussion threads, previous commits, or informal team knowledge.

GAI also lacks the social and collaborative elements that often play a role in human review. Reviews are not just about correctness but also about communication, alignment, and learning. While GAI can support the technical aspects of review, it does not actively participate in conversations or adapt to interpersonal nuances. As a result, GAI may serve as a complement to human review, helping identify surface-level issues and potentially allowing humans to focus more on nuanced aspects of software quality.

Chapter 4

Pair Programming Experiments

The integration of GAI into pair programming presents unique challenges around role distribution and workflow dynamics. Traditional pair programming is based on clearly defined driver and navigator roles, but GAI's capabilities blur these boundaries. Our preliminary research revealed that developers expect GAI's role to be fluid: sometimes leading code generation, sometimes providing guidance, and sometimes taking a more passive supportive stance depending on the specific task and developer needs. To investigate GAI's role in pair programming scenarios, we designed a series of experiments exploring GAI-developer interactions.

In this chapter, we present and discuss the results from two experimental iterations investigating human–GAI interaction through the use of structured prompt guides. First, we discuss two different approaches to introducing GAI in pair programming. Second, we describe and discuss the outcomes of Iteration 1, which explored how a prompt guide could support routine development tasks such as refactoring, documentation, and test generation. We examine how developers adopted the guide, integrated it into their workflows, and what benefits and limitations they experienced. Third, we present and discuss the results from Iteration 2, which focused on introducing a more complex, conversation-driven workflow aimed at creative and strategic tasks such as problem decomposition, brainstorming, and architectural planning. In both cases, we analyze guide usage, interaction patterns, strengths, and challenges, as well as shifts in how participants perceived GAI's role.

4.1 Ideas for Iteration 1

In this section, we explore possible ways to integrate GAI into software development workflows. The goal is to map out realistic alternatives that could enhance existing practices without requiring major structural changes. We focus on two prominent patterns. The first involves human-human pair programming supported by GAI, inspired by traditional models of collaborative development. The second centers on a single developer working together with GAI, where the system takes on either the driver or navigator role.

4.1.1 Integration Approach Options

Our assessment of integration approaches focused on two distinct models. Each reflects different assumptions about how GAI can support development work and what kind of collaboration it enables. Defining these models serve as a starting point for discussing how GAI might participate in programming tasks and what kinds of collaboration each approach enables.

Human-human-GAI Approach

In the human-human-GAI approach, two developers engage in traditional pair programming while incorporating GAI assistance into their workflow. Several design considerations emerge in this setup. One key question is how the GAI fits into the driver-navigator dynamic. Should it function as a third voice, contributing suggestions independently? Or should it be directed by one of the developers, perhaps used by the driver to generate code or by the navigator to provide feedback and verification?

It is also possible for both developers to interact with the GAI directly, treating it as a shared resource that supports their joint reasoning. This arrangement allows for flexible and adaptive collaboration but also introduces coordination challenges. If one developer relies too heavily on the GAI, it may affect the depth of discussion and reduce the collaborative value of the session. The overall success of this model depends on how well the GAI integrates into the communication flow and whether it contributes constructively to shared understanding without creating friction.

Human-GAI Approach

The human-GAI approach involves a single developer working in tandem with a GAI system, treating it as a full programming partner. This model reflects how GAI is increasingly used in industrial settings, including in our collaboration with E.ON. The developer may alternate between acting as driver and navigator, while assigning complementary roles to the GAI. For example, the AI may lead in generating code while the human provides strategic guidance. In other cases, the developer may take the lead and prompt the GAI for review and suggestions.

This approach offers a high degree of flexibility and is well suited for individual workflows. Because it removes the need for human-human coordination, it can be easily adopted in settings where developers work independently. At the same time, it raises important questions. Without a second human in the loop, how does the GAI affect the developer's critical thinking

and decision-making? Can it provide meaningful feedback or strategic direction in a way that supports high-quality outcomes?

4.1.2 Approach Discussion

To better understand the practical implications of each approach, we evaluated the two approaches presented in the previous section: human-human pair programming enhanced with GAI, and human-GAI pairing. This evaluation considers both technical and organizational factors, including cultural fit at E.ON, implementation complexity, and potential impact on collaboration and productivity.

Evaluation of Human-Human-GAI Approach

One of the main advantages of combining GAI with human-human pair programming is that it builds on an established model of collaborative development. When two developers work closely together, they are able to exchange ideas, review each other's thinking in real time, and reason through complex decisions as a team. Introducing GAI into this dynamic could strengthen the collaboration by adding an additional source of suggestions, explanations, or examples. The AI can function as a supplementary resource, potentially increasing the efficiency or range of the discussion during a session.

However, this approach also presents several challenges. First, coordinating between three contributors, two humans and one AI, can be complex. There is a risk that the presence of GAI might disrupt the natural rhythm of pair programming rather than support it, especially if developers have different expectations about how and when to use the AI. Additionally, while pair programming is a known technique, it is not widely practiced at E.ON. Adopting this model would require a shift in collaboration habits and potentially a broader cultural change. Without a strong pairing culture already in place, the introduction of GAI could become more of a distraction than a benefit.

Evaluation of Human-GAI Approach

In contrast, pairing a single developer with GAI offers more direct access to GAI capabilities and avoids the coordination complexity that comes with involving multiple developers. The Human-GAI model is also more closely aligned with current development practices at E.ON, where most developers work independently. It is also easier to implement, as it does not depend on introducing or reinforcing human pairing practices. From an experimental perspective, the human-GAI model provides a clear and focused setup, making it easier to study the interaction and isolate the effects of GAI support.

Nonetheless, this approach has limitations as well. By removing the second human, it loses the collaborative benefits typically associated with pair programming. Developers working in this model must adapt to a new kind of interaction where feedback, discussion, and validation come from an AI rather than a colleague. There is also a risk that important aspects of team knowledge sharing and mutual learning may be reduced or overlooked when the pairing is between a human and GAI.

Determining the Starting Point

Given these considerations, we chose to proceed with the human-GAI pairing approach for our first iteration. This model aligns well with current development practices at E.ON, where most developers work independently. It allows us to integrate GAI without requiring the introduction of new collaboration routines, which could add complexity and resistance. By focusing on a setup that is easy to implement and minimally disruptive to existing workflows, we can more effectively study the impact of GAI on individual development practices. Furthermore, insights from this setup may translate to human-human-GAI collaboration as well.

4.1.3 Possible Experiments

Based on the human-GAI pairing model, we identified two main directions for experimentation. Both are designed to fit E.ON's solo development culture while offering insight into how developers interact with GAI in practice. Each approach has different strengths and limitations, which are considered below.

The first approach is to observe how developers use GAI without any structured guidance. This kind of open-ended setup could capture natural usage patterns and show how developers explore and integrate GAI into their workflow on their own terms. It may reveal differences in prompting strategies, highlight what works well and what does not, and provide a sense of how proficient developers are with the tool. At the same time, this approach brings several challenges. Without a shared structure, it may be difficult to compare experiences or draw consistent conclusions. In practice, this type of work often occurs outside formal team environments, which limits opportunities to coordinate efforts and observe developers during the experiment.

The alternative approach involves introducing a basic prompt guide to support developers as they start using GAI. The guide could include example prompts and relevant use cases to help users understand when and how GAI can be useful. This approach reduces the need for trial and error, especially for those unfamiliar with prompting, and offers a shared reference point that can support discussion and reflection. Developers with more experience may also use the guide to provide informed feedback on how it could be improved. While this experiment offers a more consistent and structured way to evaluate GAI usage, it may also influence how participants interact with the tool and limit the range of exploration.

Balancing these trade-offs, we chose to explore the human-GAI pairing model using the prompt guide approach, as it provided the structure necessary for systematic evaluation while remaining accessible to developers with varying GAI experience.

4.2 Iteration 1

In this section, we discuss the first experimental iteration, which investigates how a structured prompt guide affects developers' collaboration with GAI during routine development tasks. The aim is to explore whether providing developers with ready-made prompt templates can support more effective use of GAI, reduce friction, and help integrate AI assistance into daily workflows.

The guide focused on tasks that were identified as frequent and repetitive during earlier interviews and observations, prioritizing practical impact over broad coverage. Developers used the guide in real work settings, and we collected insights through diaries and follow-up interviews. These data allowed us to evaluate how the guide was used in practice, which situations it supported well, and what should be refined in future iterations.

4.2.1 Experiment setup

The experiment was designed to evaluate the usefulness and limitations of a structured prompt guide in a real world setting. The guide consisted of template-based prompts for four task categories: planning, implementation, debugging, and refactoring. Each template included a contextual structure with placeholders for relevant information, allowing developers to quickly adapt the prompt to their current task. The prompts were designed to replicate key benefits of pair programming, such as real-time feedback and shared reasoning, but in a form suitable for solo work.

Participants and Methodology

Three developers with varying backgrounds and GAI experience were given access to the guide and used it over a ten-day period as part of their normal workflow. Prompts could be pasted directly into their development environment, such as Copilot in VS Code or GAI tools in PyCharm, and were intended to be adapted freely. The guide was provided as an optional resource, enabling us to observe when and how developers chose to use it without imposing artificial constraints.

Throughout the experiment, participants maintained brief diaries where they documented how they used the prompts, what kinds of tasks they applied them to, and what outcomes they observed. They also noted what aspects of the prompts worked well, where they fell short, and in which situations they chose not to use them. This gave us a clearer picture of both the strengths and limitations of the guide in practice.

Evaluation

After the developers had used the guide, we conducted semi-structured interviews, included in appendix C, to gather more detailed feedback on their experience. These interviews focused on how the guide was used, how easily it fit into existing workflows, and whether the prompts were perceived as helpful or lacking. Developers were also asked to comment on how their level of experience may have influenced their interaction with the prompts and to suggest possible improvements or new prompt types.

The evaluation was designed to explore several key areas: how developers responded to different types of prompts, how easily the guide was adopted, and whether it introduced any friction in daily work. We also aimed to understand whether the prompts aligned with developers' real needs, how they influenced interaction with GAI tools, and what gaps or missed opportunities might exist for future versions of the guide.

4.2.2 Results

All three participants reported a generally positive experience using both GAI and the guide. It was described as a helpful starting point for engaging with GAI, especially for generating ideas, improving existing code, and getting unstuck during implementation. One developer remarked that "it opens up new possibilities" and appreciated how the guide helped them focus less on phrasing the perfect question and more on the outcome they wanted to achieve.

Workflow and Adoption

GAI along with the guide generally integrated well into developers' existing workflows. Since the templates were optional and could be adapted, most developers used the guide selectively, some keeping it open on a second screen or referencing it when needed. One participant described the experience as "smooth" and said the guide helped them "stay in flow" during tasks like refactoring or documentation.

Despite varying levels of prior experience with GAI tools, each developer found the guide easy to get started with. However, one participant noted that the guide could benefit from more onboarding support, such as clearer instructions for activating Copilot in VS Code or better integration tips for PyCharm. Another mentioned that while the templates were useful, they occasionally required iteration to tailor to specific project needs.

Prompting and the Guide

The prompt templates were widely seen as well-structured and clearly written. Developers especially appreciated the use of tags specific to Copilot like `/explain` and `@workspace`, which made it easier to access context-aware help. These templates inspired many participants to modify prompts or create new ones that better matched their tasks.

Several developers noted that the effectiveness of the prompts depended on specificity. For instance, when asking GAI to refactor a large function or generate tests, the quality of the response improved when the request was detailed and focused. As one participant put it, "functions are often very specific, and you can tell when the response is too generic."

The prompts also encouraged developers to reflect more critically on GAI outputs. One developer said they often asked follow-up questions or asked GAI to simplify the response until they felt confident in the result. Another noted, "I never just copy-paste. I need to understand what the AI gives me before I trust it in production." Two developers mentioned the importance of understanding the output, particularly in projects with production-level constraints. "If something is too advanced or unclear," one explained, "I ask it to simplify the output so I can understand and verify it myself."

Benefits and Use Cases

Several developers felt that using GAI helped improve the structure and readability of their code. The GAI suggestions were often seen as "a second pair of eyes," particularly valuable for things like naming variables, writing docstrings, or spotting edge cases. One developer commented that GAI "helped me clean up and modularize my code," especially when working with functions that had grown too large.

GAI was also seen as beneficial for learning. Developers said it helped them iterate faster and spend less time stuck on minor issues, which freed up time to focus on design decisions. One participant noted that they felt "more effective overall" because they could reach working solutions faster and review them more critically.

The most effective use cases included generating boilerplate code, writing unit tests, producing documentation, and suggesting improvements to existing code. The guide was especially helpful for initial brainstorming, creating rough drafts, and offering formatting suggestions.

Limitations and Suggested Improvements

Use cases that involved broader context, such as full project understanding or large-scale refactoring, proved more difficult. In these cases, developers had to modify the prompts heavily or supplement them with additional context.

Several participants suggested expanding the guide to include more project-specific or role-specific examples. One idea was to tailor prompts for junior and senior developers or to include common E.ON patterns, such as style conventions, formatting standards, or project scaffolds.

Others requested better integration with their preferred tools. For example, adding PyCharm-specific tips or providing a clearer example of a full prompt exchange (including follow-up questions) would help set expectations for how to work with the GAI interactively.

Some developers also suggested extending the guide with recommendations on which model or prompt format is better suited for particular tasks. One participant remarked, "It would be useful to know what prompt or model works best for which language or framework."

4.2.3 Discussion of Results

The results suggest that the prompt guide was a useful tool for helping developers initiate and structure their interactions with GAI, particularly for routine coding tasks. By providing clear templates and prompting strategies, the guide lowered the threshold for effective use and offered support similar to that of a helpful peer during early-stage development. Developers reported feeling more confident when they had a starting point to work from, and several noted that the prompts helped them stay productive when they might otherwise get stuck.

Active Use of GAI in Workflows

A recurring theme across the interviews was the value of GAI as a "second pair of eyes." Participants used the tool to review code, suggest improvements, and offer alternatives,

reflecting the kind of lightweight review and feedback often found in pair programming. This connection highlights the potential of GAI to support some aspects of the pair programming model, especially when it comes to improving clarity, surfacing edge cases, or suggesting names and structural improvements.

However, this form of support was not passive. Developers frequently engaged in iterative workflows, refining prompts and asking follow-up questions to clarify or improve the responses they received. This back-and-forth interaction shows that effective use of GAI depends not only on the prompts but also on the developer's ability to actively steer the conversation. The process of questioning, adjusting, and validating reflects a dynamic workflow that resembles how experienced developers reason through problems in conversation with a peer.

At the same time, the results point to what is currently missing from GAI-supported workflows. Developers noted the absence of an active partner who could monitor ongoing decisions, challenge assumptions, or introduce relevant domain knowledge without being prompted. The AI responded only when asked and had limited understanding of the surrounding context, including project structure, coding standards, or team-specific practices. One developer commented that GAI was helpful for tasks like refactoring or test generation, but less so when broader architectural reasoning or domain-specific insight was required.

Knowledge Transfer and Learning

Some participants used the guide for tasks involving legacy systems or unfamiliar codebases. In these cases, GAI served as a tool for knowledge transfer, helping developers form an initial mental model of the system. This mirrors the kind of onboarding or situational understanding that a senior developer might provide during a pair programming session. Although the AI lacked deep contextual awareness, it was still able to scaffold the developer's understanding through structured summaries and targeted explanations.

Skill development also emerged as a theme. Developers emphasized the importance of evaluating responses critically, and some expressed concern about over-reliance. While GAI improved speed and fluency, there is a risk that frequent use may reduce opportunities for deeper reasoning or hands-on problem solving. One participant described themselves as reluctant to copy and paste code they did not fully understand, especially in production environments. This cautious attitude reflects a responsible approach, but also highlights a tension between efficiency and learning.

4.2.4 Ideas for Next Iteration

One direction involves expanding and refining the existing prompt guide. This could include improving the clarity and specificity of individual prompts, as well as extending the guide to cover additional types of development tasks. While this approach represents a relatively low-risk, incremental improvement, it may also offer diminishing returns. The value of the guide appears to lie less in perfecting the templates and more in how it helps developers get started. For experienced users, the guide may function more as reference point than as a driver of deeper collaboration. As such, expanding the guide further may yield less insight into GAI's broader potential.

A second idea is to explore the use of pre-prompting or configuration files to establish working context more effectively. This could involve providing developers with a structured instruction file that defines conventions, task framing, or preferred tooling. By reducing cognitive overhead and aligning GAI with the developer's expectations, this setup may support smoother and more consistent interactions. However, the challenge lies in adapting these configurations to different developers, teams, and projects. As such, this idea may be better suited for a longer-term exploration or a follow-up iteration with more tailored implementation.

A third direction shifts focus from well-defined and repetitive tasks toward more strategic or creative work. Developers frequently rely on judgment and abstraction when making design decisions, refining requirements, or reasoning about trade-offs. These moments are harder to formalize but central to experienced software work. Supporting this kind of thinking may require a move beyond simple prompting to more conversational and reflective interactions, where the developer and GAI explore open-ended questions together. This aligns with the concept of pair programming, not just in surface structure but in the quality of thought: understanding the problem, brainstorming solutions, and collaboratively planning how to proceed.

A fourth direction worth exploring is the integration of GAI into test-driven workflows. For example, experiments could focus on how GAI contributes when generating or reviewing tests, and whether it functions more effectively as the one writing tests or as a critical reviewer of existing ones. This would provide insight into the potential for GAI to contribute to quality assurance and early validation, both key components of professional software development.

Taking these considerations into account, we concluded that focusing on more strategic and creative tasks was the most promising direction for increasing our understanding of GAI's collaborative potential in software development. This focus enables an examination of how GAI can assist with higher-order thinking and decision-making processes that are present in pair programming practices and central to software development.

4.3 Iteration 2

In this section, we discuss the second experimental iteration, which explores how GAI can support developers in the more creative and strategic aspects of software development. The focus shifts from routine tasks to activities that require deeper dialogue, open-ended reasoning, and collaborative ideation.

With this iteration, we aim to investigate how GAI can contribute meaningfully to tasks such as problem decomposition, brainstorming, and early-stage design decisions. These situations are often less defined and more exploratory, resembling the kinds of discussions that take place during pair programming. The experiment builds on insights from the first iteration and examines how developers engage with GAI when the goal is not just execution, but also reflection, judgment, and shared understanding.

4.3.1 Experiment Setup

To explore how GAI could support more creative and strategic aspects of software development, we once again used a structured prompt guide as the main approach. Based on the positive reception of the guide in the first iteration, we concluded that this format offered a lightweight and accessible way to introduce new interaction patterns to developers.

We developed a new guide for this iteration, organized into four areas, each reflecting a different stage of exploratory or early-stage development work. These included: understanding the problem, exploring potential solutions, planning the implementation, and refining the resulting code. The prompts were designed to support more open-ended conversations with GAI, aiming to explore deeper reasoning, back-and-forth refinement, and ideation beyond strictly defined tasks.

The methodology closely followed the structure used in the previous iteration. Over the course of ten days, two developers participated in the study. The developers were given access to the prompt guide and used it in VS Code with GPT-4.1 and Claude 4.0. They maintained short diaries documenting their use of the guide, including when and how prompts were used, what worked well or poorly, and any situations where the guide was not used. After the experiment, unstructured interviews were conducted in which participants reflected on their experiences and discussed the contents of their diaries in more detail, supporting a richer analysis of how the guide influenced their work.

4.3.2 Results

Both developers engaged with GAI in a conversational workflow, moving away from single-prompt, single-response interactions workflows. Neither developer followed the guide rigidly, instead treating it as a set of training wheels and a reminder of practices. The senior developer primarily used the guide as a source of inspiration and as a structural aid for their workflow, while the junior developer adopted a more exploratory approach, experimenting with the agent-based mode of interaction.

For the senior developer, the guide was especially valuable for inspiring pre-coding activities,

such as preparing for architectural decisions. The junior developer viewed it as a good introduction to more structured GAI interactions, helping to frame their approach to prompting.

Interaction Patterns and Evolution of GAI Perception

The senior developer preferred minimal prompting, only adding more context and iterations if an answer was unclear. Both reported engaging in multi-step conversations, often lasting three to five iterations, with some instances leading them down unproductive rabbit holes. The potential for branching conversations emerged as a notable aspect of the workflow, where one participant expressed interest in exploring multiple solution paths from a single conversation starting point.

The senior developer's view of GAI shifted from seeing it primarily as an advanced code completion tool to treating it more like a colleague for brainstorming. The junior developer was particularly impressed by the GAI's ability to create complex, functional codebases. Both reported that their ability to work effectively with GAI improved over time, suggesting a learning curve in mastering the conversational workflow.

Particular Strengths and Challenges

The GAI tools proved effective in several areas: supporting code refactoring, breaking down complex problems into manageable steps, generating boilerplate code, handling tedious tasks, and facilitating architectural discussions. These strengths aligned well with the conversational workflow design of this iteration.

Several challenges emerged during use. Both developers noted that the volume of information produced by the GAI could be overwhelming, and hallucinations occasionally undermined trust in the output. The high output sometimes reduced the signal-to-noise ratio, making it harder to identify valuable contributions. Additionally, the constant need to validate the GAI's work placed a sustained burden on the developers.

4.3.3 Discussion of Results

The findings reveal how developers integrated the prompt guide into their workflows and how it shaped their use of GAI. Two key themes emerged: the presence of pair programming–like elements in GAI interactions, and the adoption of guide categories based on alignment with existing development practices.

Pair Programming Elements

Several elements of pair programming were reflected in the developers' use of GAI. Conversational problem-solving emerged as a central practice, with the GAI serving as an interactive partner in exploring solutions. Code refactoring and refinement were common, with GAI providing suggestions for cleaner, more maintainable code. Developers also reported learning new approaches to problems through these interactions, similar to knowledge transfer in traditional pair programming.

Some aspects of pair programming were less present in the GAI interactions. Contextual awareness remained limited, requiring developers to supply significant domain and project-specific knowledge to keep the GAI on track. Unsurprisingly, the natural flow of knowledge sharing that occurs between human pairs working in the same environment was absent.

Workflow Fit and Adoption Pattern

The interviews revealed a clear adoption pattern across the four guide categories, with refine implementation being most immediately valuable and widely adopted, followed by plan implementation, which showed good appreciation with observable workflow changes. Explore solutions was recognized as valuable but remained underutilized, while understand problem received the least attention and may require further scrutiny.

This pattern suggests that categories most closely aligned with existing developer practices, particularly code review and implementation planning, achieved the strongest adoption, while the more exploratory, early-stage categories required significant workflow shifts that developers had not yet fully embraced. The guide appears to have been most effective at enhancing familiar development activities rather than introducing entirely new interaction patterns.

One reason for the earlier activities not receiving as much attention may relate to the nature of current development tasks. When developers already understand the problem and its possible solutions, GAI interaction becomes less necessary compared to situations involving more difficult or unfamiliar challenges.

4.4 Future Work

If the experimental sequence were to be continued, several directions stand out as promising opportunities to deepen our understanding of GAI-assisted pair programming. One area is developing strategies to combat conversational rabbit holes, where the interaction drifts away from productive work. Future experiments could trial interventions such as periodic summarization prompts, automated checks for task alignment, or explicit refocusing cues to see how they affect efficiency and quality of outcomes.

A second direction is to explore GAI as a stepping stone for introducing Extreme Programming–style pair programming in environments where it is not yet a standard practice. By offering developers a collaborative partner that mimics some aspects of human pairing, GAI could help familiarize teams with the dynamics and benefits of continuous dialogue, shared reasoning, and mutual review, potentially lowering the cultural and logistical barriers to full human–human pairing.

Finally, the integration of GAI into test-driven development workflows represents a compelling avenue for study. Here, the key question is whether GAI is more effective in generating initial test cases, reviewing and refining human-written tests, or working in both roles. Understanding its strengths and weaknesses in TDD could provide valuable insight into how GAI might contribute to quality assurance, early defect detection, and improved developer confidence across the software lifecycle.

Chapter 5

Code Review (RQ2) Experiments

The integration of GAI into code review raises questions about its role, scope, and value within established workflows. Traditional code review blends technical scrutiny with team-specific judgment, but GAI's strengths lie in pattern recognition, consistency, availability, and speed. This creates opportunities for early defect detection and improved code hygiene, while also exposing limitations in contextual understanding and domain knowledge. Our preliminary exploration suggested that GAI may fit best as a pre-review filter, catching low-level issues before human reviewers focus on higher-level design and domain concerns.

In this chapter, we present and discuss the results from two experimental iterations examining GAI's role in code review. Iteration 1 established a baseline of GAI's review capabilities by applying Copilot to previously reviewed code and comparing its feedback to that of human reviewers. We examine the types of issues Copilot identified, the overlap with human findings, and where its coverage fell short. Iteration 2 built on these insights by introducing a structured prompt guide for using Copilot as a pre-review filter in live development work. We analyze how developers incorporated Copilot into their workflows, the criteria they used to assess its suggestions, and the balance between helpful insights and unnecessary noise. In both iterations, we highlight interaction patterns, strengths, and limitations, as well as factors that shaped developers' trust in and adoption of GAI-assisted review.

5.1 Ideas for Iteration 1

To design a relevant first experiment, we used insights from our information gathering phase to identify concrete scenarios where GAI could realistically provide value within E.ON's existing review workflows. Based on findings from chapter 3, we identified three candidate patterns: GAI-only, GAI-assisted, and GAI-as-a-filter. Each presents different trade-offs in terms of review quality and developer involvement. We aim to investigate where GAI can meaningfully contribute, as a first-pass reviewer, a supportive assistant, or a full replacement for human reviewers.

GAI-only Review

GAI-only review delegates the entire review task to the generative AI, producing comments, suggestions, and code quality assessments without human involvement. The primary appeal of this approach lies in its efficiency: it could significantly reduce the time developers spend performing and waiting for code review, and standardize review quality by consistently applying guidelines and identifying routine issues. However, this model removes the collaborative and communicative aspects of the review process, including peer learning and contextual judgment based on team norms or project-specific conventions. It also assumes a level of trust in GAI output that may not be warranted, especially in edge cases or complex architectural decisions.

GAI-assisted Review

GAI-assisted review integrates GAI as a real-time co-reviewer during the traditional human review process. In this setup, GAI might generate contextual explanations of code changes, point reviewers to related files or documentation, and highlight issues based on static analysis or learned patterns. This pattern has the potential to improve comprehension and reduce the cognitive load on reviewers, especially for large or unfamiliar codebases. It may also improve consistency and catch low-level issues early. However, GAI suggestions might not always align with the reviewer's intent or priorities, requiring reviewers to filter out irrelevant or redundant comments. Additionally, integrating GAI-generated suggestions into existing review platforms can be technically challenging.

GAI-as-a-filter

GAI-as-a-filter places GAI earlier in the development workflow by having it analyze code before a merge request is submitted. In this model, developers receive GAI-generated suggestions and explanations while finalizing their code. This approach aligns well with existing developer tooling and encourages authors to reflect on and incorporate feedback prior to peer review. In addition to potentially improving code quality, it may reduce the burden on human reviewers by resolving routine issues beforehand and allow review conversations to focus on broader, higher-level concerns. However, developers may overlook or ignore suggestions if they do not perceive them as relevant or well-targeted, and there is a risk of suggestion fatigue if the system produces too much feedback.

Determining the Starting Point

Considering the respective advantages and challenges of the integration patterns, we identified GAI-as-a-filter as the most suitable starting point for our first experimental iteration. Positioning GAI early in the workflow ensures that the generated feedback reaches developers while they are still actively working on the code, increasing the likelihood that the suggestions are actionable and relevant. The filter model also integrates naturally with common developer IDEs, lowering the barrier to adoption and minimizing the need for new review processes or training, also aligning well with the existing development practices at E.ON.

5.2 Iteration 1

To establish a baseline of GAI code review capabilities, we decided that the first step is to find out how GAI can be used for code review and its strengths and weaknesses. Through this initial analysis of GAI code review, we aim to gain insight into three main areas. The first is the relevance and practicality of GAI suggestions not raised by human reviewers. The second is GAI's ability to identify issues that human reviewers flagged. The third is the types of issues GAI detects compared to the issues it tends to miss. By isolating GAI's performance, we establish a baseline for evaluating the strengths and limitations of GAI. These findings are provide input for the potential design of our GAI-as-a-filter approach, ensuring that its integration fits a workflow that maximizes value for developers and minimizes unnecessary friction in the review process.

In order to create the baseline of comparison, we generated automated reviews of code changes that had already been through human review. This allowed us to evaluate GAI-generated feedback in a realistic context without influencing the development process. By applying two different Copilot tools to the same merge requests, we created parallel reviews and compared their outputs to the original human reviewer comments as well as to each other. This comparison enabled us to assess the relevance, completeness, and overlap of GAI feedback in relation to established human standards.

5.2.1 Experiment Setup

The experiment was conducted using real world data from an active software project, specifically focusing on merge requests that had undergone human code review. Ten merge requests were randomly selected for analysis, each containing reviewer comments that served as a reference point for evaluating the GAI generated feedback. The code in the repository was written in Python 3. To generate automated reviews, Copilot was used, since it is the primary GAI-assisted coding tool available and supported within E.ON's development environment.

Tools Used in the Experiment

In this study, we used two Copilot tools: the code review feature in Visual Studio Code and the Copilot Chat interface. Both were applied to each merge request. The code review feature generated feedback on staged changes, simulating a human-style review and representing the automated feedback phase in a typical workflow. At the same time, Copilot Chat was

prompted to review the same changes, focusing only on the relevant files and edits due to context size limits. Using both tools in parallel allowed us to compare their outputs and explore whether teams might benefit more from a single AI assistant or from combining multiple tools. This setup helped us see the range of review capabilities available and better understand how each tool might fit into a development process.

Ensuring Comparable Review Contexts

To ensure a fair comparison between Copilot and human reviewers, we made sure Copilot only saw the same code changes that the human had access to, without any additional context or future changes. This was done by setting up the Git repository so that Copilot reviewed exactly the same diff. Specifically, for the Copilot review feature, we configured the repository by setting the `HEAD` to the last commit before the new feature was implemented. We then checked out the files from that commit without moving `HEAD`, ensuring that the diff Copilot analyzed matched the one seen by the human reviewer. For the Copilot Chat setup, the repository was similarly checked out to the final commit preceding the human review. The modified files were included in the chat context, and the prompt "Perform a code review" was used to initiate the review.

Evaluation Approach

The evaluation focused on comparing Copilot's feedback to the actual reviewer comments as well as between the tools. The analysis involved identifying common themes or issues raised by Copilot and assessing their relevance and completeness by comparing them against a set of established code review criteria. This comparative analysis aimed to highlight strengths and limitations of Copilot's review capabilities in relation to human reviewers.

Experiment Details

The experiment was conducted in early June 2025. At the time of testing, Copilot version 1.335.0 was used within Visual Studio Code version 1.100.3. Github does not provide the information regarding which LLM is used by the code review feature. The chat features were powered by the GPT-4.1 model by OpenAI.

5.2.2 Results

The output from applying Copilot's code review and chat feature to the selected merge requests revealed several notable patterns in the nature, quality and relevance of the feedback generated. In general, the Chat interface produced a broader and more detailed set of suggestions compared to the review feature, which tended to be more conservative in its outputs. This discrepancy suggests that the chat based approach may be better suited for exploratory or investigative code reviews, whereas the review feature potentially serves better as a final check before code is committed.

Categorization of Feedback

To facilitate a structured analysis of Copilot's output, we categorized the feedback into three tiers: high value, medium value, and low value, which can be seen in appendix D. This approach was chosen because the generated suggestions were often extensive, and we aimed to capture the overarching structure and intent of the feedback rather than focusing on the exact wording. Each suggestion was evaluated based on its practical relevance, technical soundness, and alignment with the kind of review input we believe is valued within E.ON's development context.

Higher Value Suggestions

Across both interfaces, several types of high value suggestions were identified. Copilot was effective at catching concrete issues such as spelling mistakes and inconsistencies in naming conventions. It also frequently flagged type safety problems, many of which aligned with those noted by human review. In terms of code hygiene, Copilot successfully identified unused imports, stray debug statements and outdated or inaccurate docstrings. Beyond surface level issues, Copilot occasionally offered deeper suggestions, such as refactoring functions with accompanying rationale, proposing performance optimizations and recommending architectural improvements.

Medium Value Suggestions

Several suggestions was classified as medium in value. These included improvement to error handling, such as replacing assertions with explicit error exception handling and the addition of type annotations throughout the codebase. Copilot also flagged the use of unexplained numerical values recommended their replacement with named constants. In the areas of documentation and testing, it identified missing docstrings and suggested improvements for clarity and consistency. It also proposed the inclusion of additional tests, especially for edge cases, although these were not always reflected in the human comments.

Lower Value Suggestions

Some feedback was considered lower in value, primarily due to lack of relevance or contextual awareness. Examples include extensive recommendations for input validation or error handling that were not mentioned in the human review and often seemed disproportionate to the actual complexity of the code. In some cases, Copilot addressed the same issues as the human reviewer but proposed alternative solutions, for instance, recommending dynamic handling of unexplained numerical value while the reviewer simply asked for it to be explained or justified.

Limitations in Contextual Understanding

While Copilot surfaced many useful suggestions, there were notable gaps in its coverage. Most importantly, it struggled to identify issues related to business logic, such as incorrect assumptions about domain-specific workflows or violations of implicit requirements that are well understood by the development team. Similarly, it did not consistently enforce

project-specific coding standards or architectural patterns, which are typically agreed upon within the team but not codified in the code itself.

5.2.3 Discussion of Results

A clear distinction emerged in the nature of feedback provided by Copilot and the human reviewers. Copilot's focus remained largely technical and localized to specific code constructs. In contrast, the human reviewers tended to provide higher-level insights related to design choices, domain-specific knowledge, and team conventions. This difference points to a complementary relationship rather than direct equivalence. For example, several patterns emerged in the chat interface where Copilot proposed unit test additions, refactorings, or performance tweaks that were plausible and technically sound, but not always aligned with the priorities reflected in the actual reviewer comments.

Differences between Copilot and Human Reviews

It seems that these differences stem from the respective strengths and limitations of each reviewer type. Copilot operates mainly on patterns learned from a large collection of public code and may not have awareness of the specific context, architecture, or practices of the target codebase. This makes it seem well suited to catching general issues related to syntax, type safety, and code hygiene, while being less reliable for reasoning about business logic or understanding the informal norms that guide a particular team's development process. Human reviewers, on the other hand, seem to draw on a shared understanding of the system's goals, historical decisions, and team-wide expectations, which can enable feedback that is more contextually grounded and strategically aligned.

Copilot Position in Review Workflow

The Copilot reviews are well suited for use as a pre-review filter. By running Copilot before submitting a merge request, developers can surface and resolve low-level or technical issues early, such as unused imports, missing type annotations, or inconsistent naming. This helps reduce the burden on human reviewers and allows them to focus on higher-level concerns such as design choices and domain-specific logic. Copilot's ability to quickly identify common issues makes it a valuable tool for improving code hygiene and ensuring consistency across a codebase.

Managing Feedback Volume Through Prompting

One challenge observed was the large volume of suggestions produced. Many of these were valid but varied in importance, which could risk overwhelming the developer. This effect can be mitigated by refining how prompts are structured when interacting with Copilot. By clearly scoping the request or narrowing the focus of the review, developers can guide Copilot to provide more targeted and relevant feedback. With appropriate prompting, Copilot becomes a practical and efficient support tool in the early stages of the review process.

5.2.4 Developer Validation

Following the initial evaluation of the GAI-as-a-filter approach using a selection of real merge requests at E.ON, we conducted interviews with developers to validate the results and gain further insights based on their practical experience. These interviews aimed not only to confirm the relevance of our findings but also to uncover additional perspectives on the tool's strengths and limitations in a real world context.

Two experienced developers familiar with the codebase were interviewed using a semi-structured format to validate the categorization framework and gather further insights. The interviews explored their views on code review goals, the relevance of the value categories, and the role of GAI in current workflows. Topics included framework validation, workflow integration, signal-to-noise tolerance, domain-specific constraints, and comparisons with existing tools.

Overall, the developer feedback aligned closely with the outcomes of our testing, our three categories high-, medium- and low value. Participants confirmed that Copilot was effective as a first-pass reviewer, capable of identifying issues early in the review process. Its ability to catch clear mistakes and suggest meaningful refactorings was particularly appreciated in situations involving overly complex or suboptimal code. Copilot was generally seen as a helpful filter that could address lower-level concerns before human reviewers became involved, allowing their attention to be directed toward more complex issues such as architectural decisions or domain-specific logic. This perspective reinforced our initial impression that the GAI-as-a-filter approach offers practical value when integrated thoughtfully into existing workflows.

5.2.5 Ideas for Next Iteration

Following our initial analysis of live merge requests, we identified several potential ways to further investigate Copilot's role in the code review process. Copilot's ability to identify technical issues, code hygiene problems, and opportunities for improvement before human review takes place indicates that Copilot could be useful as a pre-review filter. This approach could allow human reviewers to concentrate more on higher-level aspects such as business logic and architectural decisions.

Another direction we considered was shifting the focus from supporting the author to supporting the reviewer. In this scenario, Copilot would assist reviewers as they examine submitted code. However, we found this to be less suitable for the current context. It is generally easier to provide concise and relevant prompts to the AI when working with small, isolated functions than when trying to summarize entire repositories or large sets of diffs. At E.ON, code reviews are typically conducted by comparing the differences directly within the GitLab merge request interface, rather than by downloading and navigating the full repository. Although some participants have expressed interest in more comprehensive reviews that consider broader context, this is not the standard practice at present.

Given these considerations, we chose to continue focusing on the GAI-as-a-filter approach, which is better aligned with existing workflows and provides clearer boundaries for prompt construction and evaluation. To support this direction, we developed a guide to help developers

incorporate Copilot review into their workflow. This approach serves as a tool for us to evaluate Copilot's practical usefulness in a professional setting.

5.3 Iteration 2

The second iteration was designed to investigate developers' perceptions of Copilot's code review features, with the design being aided by the limitations and opportunities identified in the initial evaluation. While Copilot frequently generated technically sound suggestions, particularly when used as a filter, the relevance and focus of its feedback varied considerably. To investigate developers' perceptions of Copilot's code review features and the feedback it produces, we introduced a prompt guide aimed at developers using Copilot as a screening mechanism. The guide was designed not only to support the effective use of Copilot during the review process, but also to function as a data collection instrument. It incorporated the value categorization from the previous iteration to help developers reflect on the relevance and usefulness of Copilot's suggestions.

5.3.1 Experiment Setup

We designed this iteration to explore several questions regarding the GAI-as-a-filter approach, which can be seen in appendix E. Our focus was on how developers incorporate Copilot's feedback into their pre-submission workflows, the decision-making criteria they apply when accepting, modifying, or discarding suggestions, and whether Copilot enhances code quality or developer satisfaction without interfering with established practices. Additionally, we examined which categories of suggestions developers considered particularly helpful or less relevant.

Methodology and Participants

To investigate these questions, we adopted a methodology using a prompt guide similarly to the first pair programming experiment. The same two experienced developers from the first iteration participated in this phase, providing continuity and enabling cross-experiment comparison. Over a 10–14 day period, they used Copilot for code review during their regular development work, recording brief reflections after each Copilot-assisted review. These reflections were followed by unstructured interviews that explored their experiences in greater depth, focusing on workflow impact, decision-making processes, and the perceived value of different types of suggestions. Through this approach, we aimed to assess both the practical integration of GAI into everyday development and the subjective experiences of developers interacting with GAI-generated feedback.

Workflow Integration and Data Collection

In this iteration, the experiment was integrated into E.ON's existing development practices. Developers were asked to use Copilot on their local changes immediately before submitting a merge request for peer review. The prompt guide provided instructions for two complementary modes of use: Copilot's built-in review feature and chat-based prompting. Developers were

encouraged to use both, which allowed us to examine the strengths of each approach and how they might reinforce one another. After each review session, participants documented their impressions. They noted what was helpful and what introduced unnecessary noise. These reflections formed the primary basis for our analysis.

5.3.2 Results

Copilot yielded value as a pre-review filter in E.ON's development workflow. When developers prompted it after finishing a function or other logical unit of work, obvious defects, type inconsistencies and minor style violations surfaced immediately. This early feedback enabled developers to correct hygiene issues on the spot and ultimately generated cleaner merge requests.

Adoption Patterns

Developers invoked Copilot opportunistically on files they considered "finished enough," which shortened the interval between coding and defect discovery. Instead of running it on every commit, they relied on personal judgment to decide when automatic feedback would help, applying Copilot to individual functions as well as complete files. The resulting "review-as-you-go" habit complemented continuous integration practices.

Performance of Built-in Review

Copilot's built-in review reliably flagged syntactic or structural issues but missed subtle logic errors and project specific conventions. Because the interface presents only diffs, participants sometimes struggled to maintain broader context, particularly when assessing an entire merge request. Using Git commands such as `git checkout` followed by a no commit merge in Visual Studio Code improved diff quality, yet the narrow focus remained. Feedback volume stayed manageable, a handful of pointed comments, but the developers wanted deeper analysis in complex modules.

Performance of Chat Review

Chat-based review sessions provided wider context and richer, higher order suggestions, often recommending refactorings, parameterizing hard-coded values or reducing duplication, albeit with higher noise levels than the built-in feature. In some cases Copilot pushed for extensive validation or exception handling that conflicted with E.ON's logging-oriented strategy and its lack of domain knowledge limited the usefulness of certain recommendations.

Observed Limitations

Excessive verbosity, inconsistent git proficiency and limited domain awareness emerged as the chief limitations. One participant worried that low-confidence suggestions diluted attention, while the other noted that the workflow required to run a full file review could be cumbersome for less experienced colleagues. Copilot also sometimes failed to recognize when an exception should propagate versus being logged.

5.3.3 Discussion of Results

The study shows that GAI sits somewhere between linting tools and human insight. Copilot caught many syntactic errors and style issues before formal review, letting human reviewers move on to design and domain questions. This observation supports earlier work that argues automation works best when it clears low-level clutter and frees people for higher-order thinking.

Differences in review mode help explain when Copilot feels useful and when it feels heavy. The built-in review feature produced short, precise comments that were easy to scan, but it lacked the wider context needed to detect subtle logic flaws. The chat interface supplied that context and often suggested larger refactorings, yet it also generated more text and, at times, irrelevant advice.

Domain Knowledge and Context Challenges

Domain knowledge remains Copilot's weakest point. It sometimes proposed defensive checks that conflicted with the team's established logging practices, or it missed trade-offs that experienced reviewers handle instinctively. Trust in the tool grew as developers learned when to accept its advice and when to rely on colleagues, underscoring the need for prompt templates that embed local conventions.

Prompt quality and code clarity turned out to be decisive. Files with clear names, tidy structure, and good docstrings led to higher-value suggestions. Poorly documented code drew longer, less helpful output. This pattern suggests that projects with weak documentation could see a widening gap in review quality unless they invest in clearer naming and documentation.

Managing Noise and Friction

Noise and verbosity still keep adoption from spreading faster. Participants handled these issues by tightening their prompts and calling Copilot only when they sensed it would help. While effective, this strategy demands extra effort, suggesting that better defaults and clearer organizational guidelines would lower the barrier to entry. Git-related friction also surfaced: running a whole-file review required command-line skill that not all team members had, pointing to the importance of seamless IDE integration.

Copilot's overlap with existing static analysis drew mixed reactions. One developer liked the plain-language explanations that accompanied issues already flagged by linters, contrary opinions voiced that repetition added clutter unless a new angle or fix was provided.

Complement, Not a Substitute

Taken together, the results point to a complementary pairing. Copilot is strong at early defect detection and code hygiene, occasionally offers design-level insights through chat, and is always available when peers are not. Its real value, however, hinges on good documentation, careful prompting, smooth tool integration, and a developer's judgment about when to lean on automation and when to think for themselves.

Future improvements in model capabilities, domain knowledge integration, and contextual understanding could potentially expand GAI's role, but based on current observations, human reviewers remain essential for business logic, architectural decisions, and domain-specific considerations.

5.4 Ideas for Future Work

Due to time limitations, we were unable to conduct a third iteration, but if the experimental sequence were to continue, two promising directions emerged.

First, examining GAI as a continuous development companion rather than a pre-submission filter would allow us to explore its integration with pair programming approaches. Our results showed that developers adopted a "review-as-you-go" approach, suggesting potential value in examining the continuous review approach. This direction would investigate how GAI can support ongoing code evolution and refactoring as an active development partner throughout the coding process.

Second, addressing GAI's lack of domain-specific context and E.ON conventions through targeted training approaches would be valuable. Both iterations consistently revealed that GAI's weakest point was its lack of domain knowledge and understanding of E.ON's conventions. This direction could involve developing custom prompt templates, exploring fine-tuning methodologies, implementing retrieval-augmented generation (RAG) solutions, or leveraging the model context protocol. A comparative analysis of domain-informed versus generic GAI feedback would help quantify the impact of contextual knowledge on review quality.

Chapter 6

Discussion and Related Work

This chapter examines the outcomes of our study in relation to its goals, methodology, and position within existing research. We reflect on the methodology, how it influenced the results, and what could have been approached differently. Key limitations and threats to validity are identified, along with a discussion of the generalizability of our findings. We then connect our results to related research on GAI in software development. The chapter concludes by outlining future directions based on our observations and emerging questions.

6.1 Reflection on Methodology

This section revisits our experimental choices and methodological decisions, including the design of the prompt guides, participant involvement, and data collection. We discuss what worked well, what could have been done differently, and what lessons we draw from the process. The reflection is structured around the activities we carried out during the thesis, followed by a more general assessment of the overall methodology.

6.1.1 Overall Reflection

E.ON's starting point was a clear but unresolved challenge: the company wanted to explore how GAI could be used in software development, yet lacked concrete knowledge of where and how such tools would fit into their workflows. Our thesis set out to address this gap. Looking back, several general points stand out.

First, our methodological choices often reflected a trade-off between breadth and depth. We prioritized covering multiple perspectives through different methods and iterations, but this came at the cost of deeper analysis in some areas. Second, the qualitative emphasis of our study

allowed us to capture rich insights into workflows and perceptions, yet incorporating more quantitative measures could have strengthened our claims about productivity and impact. Third, while our ambition to conduct three iterations of experiments in two different contexts was high, this ambition helped us explore a wide space and learn what directions are worth pursuing further, as well as what directions appear less promising. Finally, we recognize that with more participants and longer-term engagement, our data would have been more robust. Despite these limitations, the methodology provided us with valuable insights and a clearer sense of where future research on GAI in software development should focus.

6.1.2 Phase 1

The first phase of the thesis focused on building an understanding of the research domain and the company context. We approached this through a combination of a literature study, interviews, and an observational study. The literature study allowed us to situate our work within the broader field of generative AI in software development, even though the body of academic work was limited. The interviews provided valuable insight into E.ON's workflows and organizational culture, while the observational study represented an early attempt to capture developer practices in situ. Together, these activities formed the foundation for the experimental design that followed, highlighting both opportunities and challenges for incorporating GAI into collaborative development tasks.

Literature study

Conducting a literature study on such a rapidly evolving field proved challenging. The emerging nature of research on GAI in software development meant that academic publications were limited, often focusing on isolated case studies or conceptual frameworks. A broader and more systematic choice of keywords might have helped us identify a wider range of relevant work. At the same time, complementing academic literature with developer forums and informal discussions proved to be a valuable decision. While not academically rigorous sources, these provided inspiration and insights into ongoing debates and practical experiences that informed our own study.

Interviews

The interviews with developers at E.ON offered a useful introduction to the company's workflows, agile practices, and overall culture. They helped us contextualize our experiments and identify areas where GAI might meaningfully integrate into existing processes. However, the number of interviews was limited. With more time, conducting additional interviews or follow-up sessions could have deepened our understanding, especially in areas where the initial conversations left gaps or raised new questions. Such an approach might have enriched our data and allowed for more targeted experimental design.

Observational study

Our attempt at an observational study was not successful. The lack of a clear purpose and direction meant that the activity risked influencing participants negatively rather than captur-

ing authentic practices. In hindsight, we recognize that observational studies in this context require careful planning, defined objectives, and participants who are well prepared for the format. While the method is inherently promising, its execution in our case was poor. Repeating it under better conditions or with different participants might have produced more useful results.If a future study were to successfully conduct an observational study we believe that it could be a great way to both understand the workflow at a company, but also reveal the true nature of GAI usage pros and cons when developers are actively using GAI.

6.1.3 Phase 2

In the second phase, we moved from exploration toward targeted experimentation. Based on the insights from Phase 1, we designed small-scale experiments focused on pair programming and code review, two collaborative practices where GAI seemed particularly promising. These experiments allowed us to probe specific interaction patterns between developers and GAI, iteratively refining our approach across multiple iterations. Given the time constraints of the project, this shift toward concrete experiments was an effective way to generate actionable findings. At the same time, it required us to make trade-offs in scope, prioritizing depth in selected scenarios over breadth across many possible use cases.

Pair Programming Experiments

The iterative structure of the pair programming experiments was appropriate, but the outcomes highlight the importance of long-term assessment. The use of a structured prompt guide in the first iteration was well aligned with E.ON's workflow and proved to be a reasonable entry point, enabling focused discussions with developers. Building on this, the second iteration aimed to explore a larger shift in workflow and GAI integration. However, this may have been too disruptive for participants, as the changes both to workflow and to the role of GAI required more time to adapt to than our ten-day period allowed. A more gradual incorporation or an alternative experimental design could have yielded richer insights. The experience raises the question of whether the challenges we observed reflect limitations of GAI for this type of task, or rather the difficulty of introducing new work practices within such a short timeframe.

Code Review Experiments

The code review experiments provided valuable contrast between traditional human review practices and GAI-augmented review. The first iteration introduced us to GAI's potential in this area and highlighted the differences compared to E.ON's established practices. In retrospect, it is debatable whether this initial iteration was strictly necessary, as we could arguably have begun with the more advanced setup of the second iteration. At the same time, understanding the foundational capabilities of GAI in this context provided a baseline that informed subsequent design choices. The planned third iteration was abandoned due to time constraints and the vacation period, which limited participant availability. This would have been an opportunity to refine our experimental setup further and explore GAI's integration in more depth. Despite these limitations, the two iterations we carried out

successfully demonstrated both the opportunities and challenges of incorporating GAI into review workflows.

6.1.4 Preliminary RQS'

Early in the thesis we outlined preliminary research questions beyond our main focus. These remained in the background throughout the project but were not actively pursued due to time and resource constraints. Concentrating our efforts on the primary research questions was ultimately the right choice, as the scope of the project was already ambitious. Nonetheless, future work could revisit these preliminary directions.

6.2 Discussion of Results

The application of GAI in software development represents a vast and rapidly evolving domain. This study explores a specific subset of this landscape, focusing on pair programming and code review workflows within a particular organizational context. While our exploration has significantly expanded our understanding of GAI's capabilities and limitations in professional development settings, this foundational knowledge helps identify promising avenues for future work while also revealing approaches that may warrant caution or different strategies.

In this section, we assess the validity and broader relevance of our findings, recognizing the inherent limitations of our exploratory approach and identifying which insights may translate to other contexts.

6.2.1 Threats to Validity

In this section, we identify and discuss internal and external factors that may have affected the outcomes of our study. We examine limitations in sample size and selection, methodological considerations, and potential sources of bias in data collection and analysis. We also consider how observational effects and measurement choices may have influenced our findings.

Sample Size and Participant Selection

We consider the general sample size of participants to be the most limiting factor to our claims. The findings may not represent the broader developer population, which makes it more difficult to draw conclusions applying to different backgrounds or working contexts. This limited diversity of participants may not have captured the full range of developer approaches or strategies, which influences our understanding of how different developers interact with GAI.

However, the exploratory nature of our study makes the sample size deemed less critical. The aim was not to establish statistical significance but to investigate emerging patterns in GAI usage for pair programming and code review, while identifying directions for promising future research. The qualitative approach enabled us to capture detailed developer experiences and establish foundational insights that inform more targeted investigations with larger samples in future research.

Scope and Methodological Considerations

The wide exploratory scope of our study was valuable for identifying diverse usage patterns and potential applications across multiple development activities, but it also produced less detailed evidence for individual findings compares to what a more narrowly focused investigation would have produced. While this breadth was necessary to map the landscape of GAI integration possibilities, it inherently limits the depth of evidence for any specific claim.

The study also involved a learning effect, as participants improved proficiency with GAI tools through the study period, potentially influencing observed usage patterns. In addition, our reliance on subjective outcome measures such as self-reported productivity gains and qualitative assessments, introduced potential bias in the evaluation of GAI's effectiveness. Furthermore, our evaluation of GAI may not have captured all relevant aspects, such as long-term code maintainability or architectural decisions, due to the limited time frame of the study preventing observations of these long-term aspects.

Selection bias was also present, since participants decided for themselves when and how to engage with GAI tools. This means that our observations may place greater emphasis on successful use cases while overlooking cases where developers avoided using GAI because of perceived limitations. Finally, the overlapping nature of code review and pair programming practices may have produced reinforcing effects, with positive experiences in one area shaping perceptions of the other, and potentially amplifying reported benefits across the practices.

6.2.2 Generalizability

This section explores how our findings might apply beyond the immediate context of the study. We reflect on whether similar results could be expected in other teams and organizations. The focus is on transferability rather than broad general claims.

Contextual and Organizational Limitations

The selection of participants from a single organization raises questions about wheter similar results would be observed in organizations with different cultures, technical environments, or development practices. Since our findings are closely connected to the collaborative practices of code review and pair programming, companies that emphasize alternative methodologies may see different outcomes. The study was conducted within an agile development environment, so its applicability to waterfall or other methodologies with different collaboration and feedback dynamics may be limited. Additionally, the mix of experience levels, programming languages, and technical backgrounds may differ to developer populations of other organizations, potentially affecting the generalizability.

Universal Aspects of GAI Integration

However, several aspects of our findings suggest broader applicability. The fundamental tasks we studied, such as code generation, refactoring, and debugging, are universal across software development regardless of methodology or organizational context. Similarly, the limitations we identified, such as hallucinations, information overload and validation burden,

are inherent to current GAI capabilities rather than organizational factors, suggesting that these challenges would emerge across different contexts.

Culture-Specific Factors

The culture of individual work and collaboration norms at E.ON may have influenced how developers engaged with GAI. This contrasts with organizations with stronger forms pair programming traditions, such as those following Kent Beck's established pairing norms, where GAI integration might follow different patterns.

6.3 Related Work

In this section, we situate our thesis within the emerging research on GAI in software engineering. Our aim is not only to summarize prior studies but also to compare them with our own findings, highlighting where they strengthen, contrast, or leave open questions that our work addresses. We focus on this literature because it provides both conceptual visions and early empirical evidence of how GAI can be integrated into development workflows, which directly relates to our investigation of pair programming and code review.

6.3.1 Scope and Selection of Related Work

Research on GAI in software engineering is still in its early stages, and the availability of rigorous, high-quality studies is limited. Many published works provide exploratory perspectives or conceptual frameworks rather than detailed empirical evidence. This makes it challenging to identify related work that can be directly compared with our own experiments. To address this, we selected four representative papers that, despite differing in scope and method, each shed light on key aspects relevant to our focus on pair programming and code review.

The related works were identified through searches in Google Scholar and LUBSearch using terms related to generative AI, software engineering, pair programming, and code review, with a focus on studies that either aligned with or contrasted our own findings, prioritizing recent publications. The chosen works cover a spectrum of perspectives: Heander et al. propose an AI-OS architecture for supporting code review [26]; the StackSpot AI case study analyzes lessons learned from building a contextualized coding assistant [27]; Ulfsnes et al. investigate collaboration and workflow changes when developers adopt GAI tools [28]; and Banh et al. examine industry perceptions of GAI's potential through interview-based analysis [29].

Together, these studies provide complementary insights into how GAI is envisioned, implemented, and experienced in software engineering practice. While some emphasize architectural visions or technical challenges, others capture developer perceptions and reported benefits. By discussing each in relation to our own results, we highlight both the emerging consensus that GAI should complement rather than replace human judgment, and the practical gaps our work addresses through embedded experiments in an industrial setting.

6.3.2 Support, not automation: towards AI-supported code review for code quality and beyond

Similar to our research direction, Heander et al. argue that artificial intelligence should enhance rather than replace code review activities [26]. They propose an AI-OS architectural design for a code review platform that maintains human involvement while leveraging AI capabilities. This perspective aligns with our thesis, as we investigate practical approaches to integrating AI-assisted code review into existing development workflows. Both works recognize the value of preserving the human element in code review while exploring how AI can address efficiency challenges.

Summary

There comes many benefits for adopting code review in the software development process, including more quality focused aspects such as defect finding and code improvement, but also more collaboration heavy factors including knowledge transfer and team awareness. The authors demonstrate that code review is time consuming, which means that optimizing reviews is a key factor in improving team productivity. Fully automated reviews by AI tools can help in maintaining quality, but the authors imply that the collaborative aspects is at risk.

Therefore, the authors describe their vision of an AI review platform which can support developers during code review, rather than replacing the activity completely, striving to boost all positive effects of code review. Their vision is illustrated by an architectural design, which basically integrates many AI agents for the different parts of context and domain a developer interfaces with during the review process. A few examples of areas the agents connect to are the version control system, issue tracker and CI/CD pipeline. They argue that this system could provide contextual help during the review, aiding developers in understanding the rationale of change, connecting it to related work, reading code changes, defect finding, writing constructive comments, and assisting in making a decision on integration or rejection for adjustments.

For their vision to succeed, the authors name a couple of research activities important for success. The understanding of different user needs and experience during code review warrant further research, as well as measuring the more intangible collaborative benefits of code review. The authors also describe technical challenges in how to build the AI pipeline, but suggest implementing a minimal viable use case first and iterating on a smaller version of the system with fewer AI agents.

Discussion

Both works advocate for GAI as a complement, not a replacement for human developers in code review. The shared emphasis on preserving developer judgement and maintaining the collaborative aspects of code review, supports the validity of the GAI-as-a-filter approach. The challenges of domain knowledge and context during GAI review might be alleviated with newer and better models, but the authors proposed architectural design integrating multiple agents drawing from multiple sources of context and domain knowledge represents a promising future direction. Whereas their work presents a theoretical vision of such systems,

our contribution provides empirical evidence from a real developer environment. In this sense, their vision of a comprehensive multi-agent system is complemented by our work, which focuses on incremental, targeted implementations with immediate value.

6.3.3 Lessons from Building StackSpot AI: A Contextualized AI Coding Assistant

This paper presents a case study of StackSpot AI, a contextualized coding assistant built with GPT-3.5 and retrieval-augmented generation, aiming to overcome the generic answers and lack of organizational awareness often found in tools like Copilot and ChatGPT [27]. The study is relevant to both of our research questions. For pair programming, it speaks to how GAI can take on a more knowledgeable and context-sensitive role when assisting developers in code generation and problem solving. For code review, the focus on retrieval and integration of organizational knowledge addresses a core challenge we also observed: the usefulness of AI feedback depends heavily on its ability to incorporate project-specific context.

Summary

The paper [27] addresses limitations in current LLM-based coding tools like ChatGPT and CoPilot, which despite their potential, often produce generic or incorrect answers and require significant prompt engineering. These tools also struggle to incorporate domain specific or contextual information relevant to a particular organization. The authors were motivated by the need for a more specialized coding assistant that could deliver accurate, context aware support tailored to real world software development environments.

The main contribution is an in-depth case study of the development of StackSpot AI, a contextualized LLM-based coding assistant. The paper presents 13 lessons learned from this process, grouped into three categories: LLM-specific, user-related, and technical. The LLM-specific lessons include handling token limitations, structuring knowledge input, and managing contextual retrieval. These illustrate practical challenges that occur when adapting general-purpose language models to the specific demands of software development.

The StackSpot AI system was developed over a few months by a software development team. It integrates OpenAI's GPT-3.5 with retrieval augmented generation architecture, allowing it to incorporate external knowledge sources into its prompts. The researchers collected data through engagement with the development team, interviews and repository mining. Developer responses were analyzed using qualitative coding techniques to identify recurring challenges and solutions.

The study highlights key technical, user-experience, and model-related challenges in building LLM applications, providing valuable insights for practitioners and researchers. It emphasizes that close collaboration between engineers and researchers is essential, and that the identified LLM-specific challenges deserve further attention in future work.

Discussion

While the study provides practical insights, some lessons are strongly shaped by the limitations of GPT-3.5's 4k token window, raising questions about generalizability to newer models with larger contexts. In addition, several challenges—such as document chunking and semantic retrieval—are described only at a high level, without detail on the technical reasoning behind attempted solutions. This reduces the transferability of the results for practitioners facing similar problems.

Nevertheless, the focus on context management and domain-specific retrieval aligns closely with our findings. In our experiments at E.ON, effective prompting and the availability of relevant project context were decisive for whether GAI produced valuable output. The StackSpot lessons therefore reinforce our observation that integration details, not just model capability, determine success. Where StackSpot identifies technical gaps such as chunking strategies or RAG architectures, our work shows the same issue from the workflow perspective: GAI output was most effective when developers supplied precise prompts or contextual cues, and least effective when such context was missing. Together, the two studies highlight the central role of context in GAI-assisted development and point toward complementary future directions: improving retrieval techniques on the system side and refining prompt guides and workflow integration on the user side.

6.3.4 Transforming Software Development with Generative AI: Empirical Insights on Collaboration and Workflow

In this paper, Ulfsnes et al. examine how software developers use GAI and investigate its impact on collaboration and teamwork in software development, with some findings indirectly connected to collaborative practices such as pair programming and code review [28].

Summary

The authors argue that while effective tools like Copilot are valuable, they are not sufficient on their own. For successful GAI integration, they emphasize the need for further research into its implications on team collaboration. Given the rapidly evolving and unpredictable nature of GAI, the authors conducted a multi-case study based on multiple interviews with GAI users to investigate how GAI is used in software development, and why developers chose to interact with it.

Regarding the activities that GAI is used for, the main activities include asking for assistance when stuck, learning new things, creation of boilerplate code and working with existing code, such as refactoring, new features, debugging, tests. In terms of developer motivations, the authors distinguish two main styles of interaction: simple dialogue and advanced dialogue extended with prompt engineering. The choice of style is depending on the type of problem to be solved and the surrounding context.

Reported benefits include improved productivity during manual and repetitive tasks, leading to higher motivation, enjoyment to work, and more time for creative and complex tasks.

When compared to Google, GAI dialogue was described as faster and offering more freedom, with no need for correct phrasing or politeness. The quality of prompt engineering was found to be important, and the study also observed the emergence of "pair prompt engineering," similar to pair programming. GAI was reported to facilitate learning. For example, enabling data scientists to code more effectively or helping front-end developers dealing with back-end tasks.

Challenges identified include concerns about data confidentiality, policy restrictions, the cut-off date for training data leading to out of date information, and potential cultural biases influenced by how more hierarchical organizations conduct their work. The findings indicate that developers are increasingly turning to AI instead of human colleagues for assistance, which could impact how knowledge is shared within teams.

The paper's core contribution lies in documenting GAI benefits. GAI helps automate routine work and allows developers to concentrate on more strategic thinking, while improving both efficiency and code standards. Participants consistently noted enhanced productivity and greater mental capacity available for tackling challenging problems.

Discussion

The paper recognizes many of the same productivity benefits that we identified in our study, though their focus on more general GAI usage contrasts with our specific investigation on collaborative practices. While both studies acknowledge GAI's advantages for individual efficiency, Ulfsnes et al. raise important concerns about the long term perspective on GAI integration, particularly the potential tension between individual productivity and declining team interaction.

Their perspective on the reduced human to human interaction warrants monitoring, given the critical role of knowledge transfer between colleagues during pairing and reviews. However, our findings indicate that instead of replacing collaborative practices, GAI can enhance team interactions by handling routine tasks, freeing developers to focus on higher-level, more strategic discussions. The disruption to collaborative workflows was also found to be minimal, indicating that individual GAI productivity can successfully coexist with collaborative or agile development environments.

This contrast in findings show valuable directions for future research, especially for long-term studies examining team adaption patterns, bridging the gap between concerns for diminishing interaction and collaborative focus.

6.3.5 Copiloting the future: How generative AI transforms Software Engineering

This paper by L. Banh et al. explores the perceptions of GAI in the software engineering industry through short phone interviews with practitioners of varying experience levels [29]. The study is mainly relevant to pair programming, as it discusses how developers perceive GAI as a collaborative partner that can assist with coding, debugging, and problem-solving.

Summary

The paper examines the introduction of GAI tools, such as Copilot and ChatGPT, into software development. It is set against an increase of pressure on developers to improve productivity, with GAI offering potential value in tasks like testing, code generation, error detection, and discussing implementation concepts. The study aims to understand how GAI affects software development, the challenges that need to be addressed, and how these tools can be effectively integrated into workflows.

The authors propose a framework outlining both the opportunities and challenges of adopting GAI in software development. The benefits identified include reducing developer time and improving code quality, while key challenges involve reliability issues such as hallucinations and inconsistent outputs, underestimated overhead where prompting can take longer than completing the task manually, limited integration into existing workflows, and privacy concerns. The framework is intended as a set of guidelines for organizations planning to introduce GAI.

To build this framework, the researchers conducted eighteen semi-structured interviews with industry professionals from varied backgrounds. All interviews were held virtually, recorded, transcribed, and translated into English. The resulting data was systematically coded to identify recurring ideas and patterns.

The findings show that GAI can support developers throughout the software lifecycle, including ideation and planning, coding, debugging, and prototyping. It can accelerate development, improve code quality, and enhance learning and problem-solving. The authors conclude that GAI should be seen as a collaborative partner rather than a replacement for human developers and highlight the need for future research into developer behavioural changes, quality control, and cognitive effects.

Discussion

The study highlights opportunities and challenges of GAI adoption but leaves important gaps. It does not address how tools were integrated into workflows or the role of prompting strategies, both of which strongly shape effectiveness. Reported inefficiencies or inconsistencies may stem from weak prompting, while some benefits could reflect well-crafted prompts. Moreover, the reliance on short, self-reported interviews limits the strength of claims about productivity, as perceptions may not reflect actual outcomes.

Our thesis addresses these gaps by combining interviews with direct observation and iterative experiments within E.ON's development teams. By embedding GAI into real workflows, we were able to study integration practices, prompting strategies, and workflow fit in detail. The use of structured prompt guides showed how developer practices influence outcomes, providing concrete evidence beyond perceptions. While both studies agree that GAI works best as a complement to human judgment, our results demonstrate how this complementarity materializes in pair programming and code review, including benefits such as faster refactoring and pre-review filtering, as well as challenges like conversational rabbit holes and validation burden.

In this way, our work both strengthens and concretizes the claims of Banh et al., showing how

abstract opportunities translate into observed practices and highlighting integration details that are decisive for realizing GAI's value.

6.4 Future Work

Building on the findings and limitations identified in this thesis, we identify promising directions for continued research that could advance our understanding of GAI integration in software development.

The short timeframe of our study limited our ability to observe GAI integration over extended periods, which means future research could examine adaption patterns and skill development over time. Another direction would be to establish quantitative metrics to strengthen the understanding of productivity impacts, including code quality metrics or review time reductions.

Both pair programming and code review experiments consistently revealed limited GAI capabilities in understanding domain specific knowledge. Future research could explore approaches to address this gap, possibly through custom prompt templates, retrieval-augmented generation or the model context protocol.

Our experiments revealed interesting collaboration dynamics between developers and GAI. Future research could investigate more collaboration patterns, including driver-navigator dynamics, three-way collaboration with human-human-GAI, or GAI's potential role in facilitating a shift to traditional pair programming. Another promising area is inserting GAI into test driven development. Studies could examine GAI effectiveness in generating initial test cases, refining tests written by developers, or its role in iterative test cycles, such as red-green-refactor.

Future research could investigate tighter integration with development infrastructure such as continuous integration and deployment pipelines and version control systems. Understanding how GAI can be embedded within existing development ecosystems, rather than treating it as an external tool, has the potential to enhance developer adoption and effectiveness.

While our study focused on GAI as a complement to human developers, future work could investigate alternative paradigms, such as agentic approaches in which GAI systems function with greater autonomy. Exploring the trade-offs between complementary and autonomous integration strategies could help organizations balance GAI responsibility and human oversight.

In addition to these directions, we also outlined secondary research questions at the outset of the thesis that remain unexplored. These include comparing the performance of different AI models in pair programming and code review (RQS1), as well as investigating how an industrial actor such as E.ON could operate an LLM within its own infrastructure (RQS2). Addressing these questions would provide valuable insights into both the comparative capabilities of current models and the practical considerations of enterprise deployment.

Chapter 7

Conclusion

This thesis set out to explore how GAI can be integrated into collaborative practices in software development, focusing specifically on pair programming and code review. Through a series of experimental iterations with industry developers, we examined both the opportunities and challenges of introducing GAI into these contexts.

Our findings show that GAI can serve as a valuable partner in pair programming by supporting productivity in tasks such as code generation, refactoring, and documentation. Developers perceived the AI's role as fluid, shifting between driver, navigator, and supportive assistant depending on the situation. At the same time, successful collaboration required clear contextual framing and active steering by the human partner to avoid unproductive detours.

In code review, positioning GAI as an early-stage filter proved to be an effective approach. The AI was able to identify syntactic and stylistic issues, enabling human reviewers to focus on higher-level concerns such as architecture and domain-specific reasoning. However, its effectiveness was strongly dependent on the quality of prompts and contextual information, underscoring the need for careful integration into existing workflows.

Across both settings, our results indicate that GAI is most effective when applied as a complement rather than a substitute for human judgment. While it can accelerate certain activities, it also introduces challenges related to reliability, validation burden, and information overload.

The contributions of this thesis lie in providing empirical evidence from real world development contexts and in offering practical insights into how GAI can be integrated into collaborative workflows. Future research should expand the scope by involving larger teams, longer study periods, and more diverse development environments. Ultimately, the findings suggest that with thoughtful integration, GAI can enhance collaborative practices while preserving the critical role of human expertise.

References

[1] D. Spinellis, "Pair programming with generative ai", *IEEE Software*, vol. 41, no. 3, pp. 16–18, 2024. DOI: 10.1109/MS.2024.3363848.

[2] M. S. S. Chowdhury et al., "Ai-powered code reviews: Leveraging large language models", in *2024 International Conference on Signal Processing and Advance Research in Computing (SPARC)*, vol. 1, 2024, pp. 1–6. DOI: 10.1109/SPARC61891.2024.10829223.

[3] A. Cockburn and L. Williams, "The costs and benefits of pair programming", in *Extreme Programming Examined*. USA: Addison-Wesley Longman Publishing Co., Inc., 2001, pp. 223–243, ISBN: 0201710404.

[4] L. Williams et al., "Strengthening the case for pair programming", *IEEE Software*, vol. 17, no. 4, pp. 19–25, 2000. DOI: 10.1109/52.854064.

[5] K. Beck and C. Andres, *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley Professional, 2004, ISBN: 0321278658.

[6] L. Plonka et al., "Knowledge transfer in pair programming: An in-depth analysis", *International Journal of Human-Computer Studies*, vol. 73, pp. 66–78, 2015, ISSN: 1071-5819. DOI: https://doi.org/10.1016/j.ijhcs.2014.09.001. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1071581914001207.

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

[8] P. C. Rigby and C. Bird, "Convergent contemporary software peer review practices", in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013, Saint Petersburg, Russia: Association for Computing Machinery, 2013, pp. 202–212, ISBN: 9781450322379. DOI: 10.1145/2491411.2491444. [Online]. Available: https://doi-org.ludwig.lub.lu.se/10.1145/2491411.2491444.

[9] C. Sadowski et al., "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, Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 181–190, ISBN: 9781450356596. DOI: 10.1145/3183519.3183525. [Online]. Available: https://doi.org/10.1145/3183519.3183525.

[10] 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*, vol. 2, 2015, pp. 27–28. DOI: 10.1109/ICSE.2015.131.

[11] W. H. L. Pinaya et al., *Generative ai for medical imaging: Extending the monai framework*, 2023. arXiv: 2307.15208 [eess.IV]. [Online]. Available: https://arxiv.org/abs/2307.15208.

[12] T. B. Brown et al., "Language models are few-shot learners", in *Proceedings of the 34th International Conference on Neural Information Processing Systems*, ser. NIPS '20, Vancouver, BC, Canada: Curran Associates Inc., 2020, ISBN: 9781713829546.

[13] M. Hall et al., *A systematic study of bias amplification*, 2022. arXiv: 2201.11706 [cs.LG]. [Online]. Available: https://arxiv.org/abs/2201.11706.

[14] Q. Ma, T. Wu, and K. Koedinger, *Is ai the better programming partner? human-human pair programming vs. human-ai pair programming*, 2023. arXiv: 2306.05153 [cs.HC]. [Online]. Available: https://arxiv.org/abs/2306.05153.

[15] M. T. Çaldağ, "Ai pair programming acceptance: A value-based approach with ahp analysis", in *2024 10th International Conference on Control, Decision and Information Technologies (CoDIT)*, 2024, pp. 556–561. DOI: 10.1109/CoDIT62066.2024.10708135.

[16] C. Bird et al., "Taking flight with copilot", *Commun. ACM*, vol. 66, no. 6, pp. 56–62, May 2023, ISSN: 0001-0782. DOI: 10.1145/3589996. [Online]. Available: https://doi.org/10.1145/3589996.

[17] O. B. Sghaier, M. Weyssow, and H. Sahraoui, *Harnessing large language models for curated code reviews*, 2025. arXiv: 2502.03425 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2502.03425.

[18] X. Hong and L. Guo, "Effects of ai-enhanced multi-display language teaching systems on learning motivation, cognitive load management, and learner autonomy", *Education and Information Technologies*, vol. 30, pp. 17 155–17 189, Mar. 2025. DOI: 10.1007/s10639-025-13472-1.

[19] M. Puerta-Beldarrain et al., "A multifaceted vision of the human-ai collaboration: A comprehensive review", *IEEE Access*, vol. 13, pp. 29 375–29 405, 2025. DOI: 10.1109/ACCESS.2025.3536095.

[20] C. Adapa et al., "Ai-powered code review assistant for streamlining pull request merging", in *2024 IEEE International Conference for Women in Innovation, Technology & Entrepreneurship (ICWITE)*, 2024, pp. 323–327. DOI: 10.1109/ICWITE59797.2024.10503540.

[21] N. Davila, J. Melegati, and I. Wiese, "Tales from the trenches: Expectations and challenges from practice for code review in the generative ai era", *IEEE Software*, vol. 41, no. 6, pp. 38–45, 2024. DOI: 10.1109/MS.2024.3428439.

[22] Harper, *My llm codegen workflow atm*, Feb. 2025. [Online]. Available: https://harper.blog/2025/02/16/my-llm-codegen-workflow-atm/.

[23] danphilibin, *Comment on "llm codegen workflow"*, Hacker News Comment, Feb. 2025. [Online]. Available: https://news.ycombinator.com/item?id=43094006.

[24] Graphite Dev Team, *Ai won't replace human code review*, 2024. [Online]. Available: https://graphite.dev/blog/ai-wont-replace-human-code-review.

[25] A. Davjekar, *Ai-assisted software development: A comprehensive guide with practical prompts (part 1/3)*, 2024. [Online]. Available: https://aalapdavjekar.medium.com/ai-assisted-software-development-a-comprehensive-guide-with-practical-prompts-part-1-3-989a529908e0.

[26] L. Heander, E. Söderberg, and C. Rydenfält, "Support, not automation: Towards ai-supported code review for code quality and beyond", in *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*, ser. FSE Companion '25, New York, NY, USA: Association for Computing Machinery, 2025, pp. 591–595, ISBN: 9798400712760. DOI: 10.1145/3696630.3728505. [Online]. Available: https://doi.org/10.1145/3696630.3728505.

[27] G. Pinto et al., "Lessons from building stackspot ai: A contextualized ai coding assistant", *2024 IEEE/ACM 46th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), Software Engineering: Software Engineering in Practice (ICSE-SEIP), 2024 IEEE/ACM 46th International Conference on, ICSE-SEIP*, 2024. arXiv: 2311.18450 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2311.18450.

[28] R. Ulfsnes et al., *Transforming software development with generative ai: Empirical insights on collaboration and workflow*, 2024. arXiv: 2405.01543 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2405.01543.

[29] L. Banh, F. Holldack, and G. Strobel, "Copiloting the future: How generative ai transforms software engineering", *Information and Software Technology*, vol. 183, p. 107 751, 2025, ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2025.107751. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584925000904.

Appendices

Appendix A

Phase 1 Interview Questions

Introductory questions

1. Can you tell us a bit about your role in the company?
2. How long have you worked here and in software development generally?
3. Does the development work in the team follow any particular processes, such as scrum, agile or XP?
4. What type of projects do you usually work on?
5. What type of tasks do you typically have?
6. What do you do if you need help with something?
7. What is the general attitude toward generative AI in your team? Is that also your perception?
8. What is your sentiment? What have you heard, and are you skeptical or positive?

Experience using AI

9. For what types of tasks would you like help from GAI? Why, and in what way?
10. For which tasks do you not feel a need for help and why?
11. Do you have any boring tasks you'd rather leave to someone else, e.g, GAI?
12. Do you have any fun tasks you'd prefer not to leave to GAI?

For the Experienced Developer:

E1. How did you start using AI?

E2. How often and in what way do you use AI in your work? (Personal use too?)

E3. What does AI do well/poorly?

E4. Can you give a concrete example?

E5. Which AI tools do you use?

E6. How has AI changed your way of working?

E7. How has it affected code review?

E8. How has it affected pair programming?

E9. What are the biggest advantages of using AI in your work?

E10. What challenges or limitations have you encountered?

E11. What do you wish AI did better?

For the Inexperienced Developer:

I1. Do you usually collaborate when writing code? How do you review code?

I2. Have you had the opportunity to test AI in your work? Why/why not?

I3. What resources would be required to start using AI?

I4. What help/training would be needed?

I5. Is there any reason you don't use AI?

I6. How do you view AI's role in software development?

I7. What would you like to use AI for?

I8. Would you or team members use it if we determined it was beneficial?

I9. Are you concerned that you or others will lose jobs due to generative AI?

Benefits and Challenges (For Both):

13. What do you see as potential benefits of AI in software development?

14. What are potential disadvantages or risks?

AI at E.ON in the future

15. What do you see as potential benefits of AI in software development?

16. What are potential disadvantages or risks?

Appendix B

Pair Programming Prompt Guide Iteration 1

B.1 GitHub Copilot Developer Guide

A comprehensive guide for developers to maximize productivity with GitHub Copilot through effective prompting.

B.1.1 Purpose

We are conducting an evaluation of Generative AI (GAI) within E.ON's workflows to explore potential benefits and limitations. If you don't already have access to Github Copilot, please request it as soon as possible. There are many ways Copilot can be used, such as line completion, but we are focusing on the chat feature.

To support your use of GAI, we have created an initial version of a guide/wiki that includes example prompts. We encourage you to use this guide whenever you feel the need to search on Google or ask a coworker for help, or simply use it as you would any other AI tool if you've used them before.

As you use the guide/wiki, please take some notes on your experience, what worked well, what didn't, and any suggestions for improvement. If you use Copilot in other scenarios, we are curious to know about those as well. These observations are valuable for refining our approach and this guide.

We will schedule a follow-up interview to discuss your notes!

If features seems like they are missing in VSCODE, install the latest version of from the website rather the the company portal

B.1.2 Table of Contents

B.1.3 Quick Reference

Essential Prompts Template

```
Context: [Describe your situation]
Goal: [What you want to achieve]
Constraints: [Any limitations or requirements]
Examples: [Provide input/output examples if relevant]
```

Effective Follow-ups

- "Explain why you chose this approach"
- "What are the potential downsides?"
- "How would this handle [specific edge case]?"
- "Can you make this more [specific quality]?"
- "Show me an alternative approach"

Quality Checkpoints

Before accepting any generated code: 1. Do I understand what this code does? 2. Does it handle edge cases appropriately? 3. Is it testable and maintainable? 4. Does it follow project conventions?

B.1.4 Getting Started

What Copilot Chat Excels At

- **Natural Language Code Interaction**: Ask questions about code in plain English
- **Iterative Development**: Generate code, then refine it through conversation
- **Context-Aware Assistance**: Understand your codebase and provide relevant suggestions

Fundamental Prompting Principles

- **Break Down Complex Tasks**: Split large problems into smaller, manageable pieces

- **Be Specific**: Provide clear requirements and constraints
- **Avoid Ambiguity**: Use precise language and concrete examples
- **Provide Context**: Include relevant code, error messages, and background information
- **Iterate and Refine**: Use follow-up prompts to improve responses
- **Maintain Relevant History**: Keep conversations focused on the current task

B.1.5 Chat Features & Context

There are a few ways to use Copilot more effectively, including chat participants and slash commands. CoPilot uses the files open in the IDE as context, so keep the files you want CoPilot to consider open.

Chat Participants

These participants function like domain experts, assisting you with their particular area of expertise. They can be used to provide better context.

- `@azure` - Azure-specific development questions (Not currently available)
- `@vscode` - Functionality and extensions VS Code, like keybinds
- `@terminal` - Shell and command line operations
- `@workspace` - Project-wide analysis and operations

Slash Commands

These commands are faster ways to execute certain prompts:

- `/clear` - Clear chat history
- `/explain` - Explain selected code
- `/fix` - Fix issues in selected code

B.1.6 Prompts

The prompts are divided into different activities during the coding session. Some of the prompts are designed to work as-is, while other prompts need modification, acting more as inspiration of what Copilot could be used for. Your mileage may vary, so some customization is most probably needed.

Planning & Problem Definition

These prompts can be used to bounce some ideas with Copilot before implementing any code.

Requirements Analysis

```
Given the criteria, list implementation options and tradeoffs for
each option.

What edge cases should I consider for this feature?

Break down this user story into implementable tasks with clear
dependencies.
```

Solution Design

```
Review my proposed solution before I implement: [describe approach]

Suggest 2 different approaches.
```

Code Generation

These are the prompts you want to use when writing code, or when you want Copilot to write it for you.

Initial Implementation

```
Write a Python function to process CSV files with error handling.

Show me the python syntax for list comprehension with conditionals.

Implement a binary search algorithm.
```

Iterative Refinement

```
Make this function more performant for large datasets.


Add error handling for network timeouts and connection failures.
```

Code Comprehension & Navigation

These prompts can be used to explain code or projects.

High-Level Overview

```
@workspace Create a high-level overview of this application's
architecture.

Explain how main.py and data.py interact and their responsibilities.

Generate a sequence diagram showing the data flow through this
system.
```

Code Analysis

```
Explain this algorithm step-by-step with examples.

What design patterns are used in this codebase?

How does this function handle edge cases?

Walk me through the execution flow of this complex method.
```

Refactoring & Code Improvement

These prompts can be used to modify any existing code.

Structure & Organization

```
How could this method be refactored for better maintainability?

Split this class into separate concerns using appropriate design
patterns.

Extract common functionality into reusable utility functions.
```

Code Quality

```
Improve the variable names in this function to be more descriptive.

Simplify this code and eliminate nested if/else chains.

Optimize this database query for better performance.

Make this code more readable without changing functionality.
```

Testing & Quality Assurance

These prompts can be used for testing and debugging.

Test Generation

```
Write property-based tests for this sorting algorithm.

Generate mock objects for testing this payment processing module.
```

Debugging & Troubleshooting

```
@workspace Analyze test failures in #file:user_test.py and suggest
fixes.
```

```
Debug this memory leak in the background service.

Explain why this async function isn't working as expected.

Help me trace through this recursive algorithm that's causing a stack
overflow.
```

Code Quality

```
Use PEP8 standards to fix the linting errors in this Python file.

Suggest improvements for better error handling and logging.
```

Documentation

These prompts can be used for writing and updating documentation.

Code Documentation

```
Add comprehensive documentation comments to this file.

Update the existing documentation for the getUserById function to
reflect current implementation.
```

Appendix C

Interview guide Pair Programming iteration 1

1. Experience with software development / Generative AI (GAI)
2. How do you use GAI? In what situations?
3. How easy was it to get started with the guide?
4. Did the guide affect your workflow positively or negatively? In what way?
5. Describe your general experience with the guide.
6. Name one thing that worked well – when have you been most satisfied with GAI?
7. Name one thing that worked poorly – when have you been most dissatisfied with GAI?
8. If experienced with GAI:
   (a) What did you miss most from your usual GAI process?
   (b) Did you feel more or less productive with the guide compared to your usual GAI usage?
   (c) How did the prompts in our guide differ from your usual GAI process?
9. How do you determine whether a GAI response is reliable?
10. How does this experience differ from other AI tools you have used?
11. Did you notice any difference in the quality of the code generated?
12. Did the guide affect your code understanding or learning?

13. Do you think your colleagues would benefit from the prompt guide? Why or why not?
14. Do you think the guide could be improved by including more specific E.ON context, for example repetitive tasks that often occur?
15. Would you prefer different guides for different roles (junior/senior)?
16. Which types of tasks worked best? (e.g., boilerplate, refactoring)
17. Which types of tasks/languages/frameworks worked worst?
18. Did you modify any prompts? Would you like to?
19. Were there any prompts you did not use? Why not?
20. Do you have suggestions for tasks or areas that could be added, modified, or removed?
21. Which feature or improvement would make the biggest difference for you?
22. Would you continue to use the guide?

Appendix D

Code Review Prompt Guide Iteration 2

D.1 Welcome to the Copilot Code Review Guide

This guide outlines two methods, a code review feature and code review via chat, for using GitHub Copilot to support your code review process. We recommend using both approaches in combination.

The guide is structured in three parts:

1. We share our purpose for creating this guide, and what feedback we request of you
2. We walk you through the two main methods for using Copilot in code review (the built-in feature and chat-based approach),
3. We share our categorization of suggestion types to help you recognize which feedback we believe tends to be most valuable in practice.

D.1.1 Purpose

This guide aims to help you use copilot for code review, and to help us explore whether GitHub Copilot can serve as an effective "pre-review filter" in E.ON's development workflow. Based on our initial analysis of live merge requests, we believe Copilot can help developers catch technical issues, hygiene problems, and potential improvements before human code review, allowing reviewers to focus on higher-level concerns like business logic and architecture decisions.

We're now seeking real-world validation from E.ON developers to understand which Copilot suggestions provide genuine value versus those that create noise, and whether this GAI-assisted approach is helpful and fits naturally into your existing development process. After each Copilot-assisted review session, please take 30 seconds to note down any thoughts, problems, or observations. We want to understand both Copilot's strengths and blind spots — what it identifies well and what it overlooks entirely. Basically, we want to know if you find copilot code review helpful.

We have explored copilot review on full merge requests, but if you have any experience of using copilot to review code continuously, such as after writing a line of code or completing a function, we would want to be made privy of that too!

D.1.2 Copilot Code Review Feature

Within the *Source Control* tab, there is a built-in code review feature for changes.
We are interested in whether you find this feature useful, but keep in mind that its feedback may be limited.

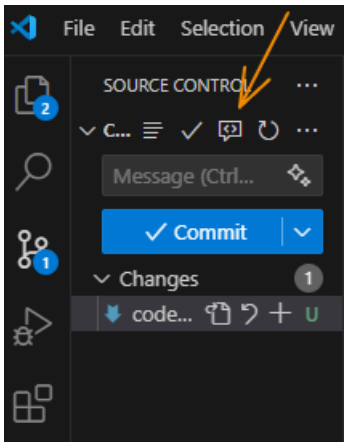

Figure D.1: Copilot CR Feature Button

D.1.3 Code Review via Chat

Using Copilot's chat functionality for code review has shown the most promise in our pre-study.
There are several ways to phrase your questions and provide context. Below is how we approached it during our study:

- Make sure to highlight the relevant code, in our case, we highlighted lines 1–243 in `merge_energies.py` for review.
- You can ask follow-up questions during the session. Check out the Prompting Guide if you need inspiration.
- If you're unable to open the chat, press `Ctrl+Shift+P` and search for `Chat`.

- Even if you disagree with issues Copilot identifies, these highlights may still indicate a need for better documentation, improved code clarity, or additional comments.

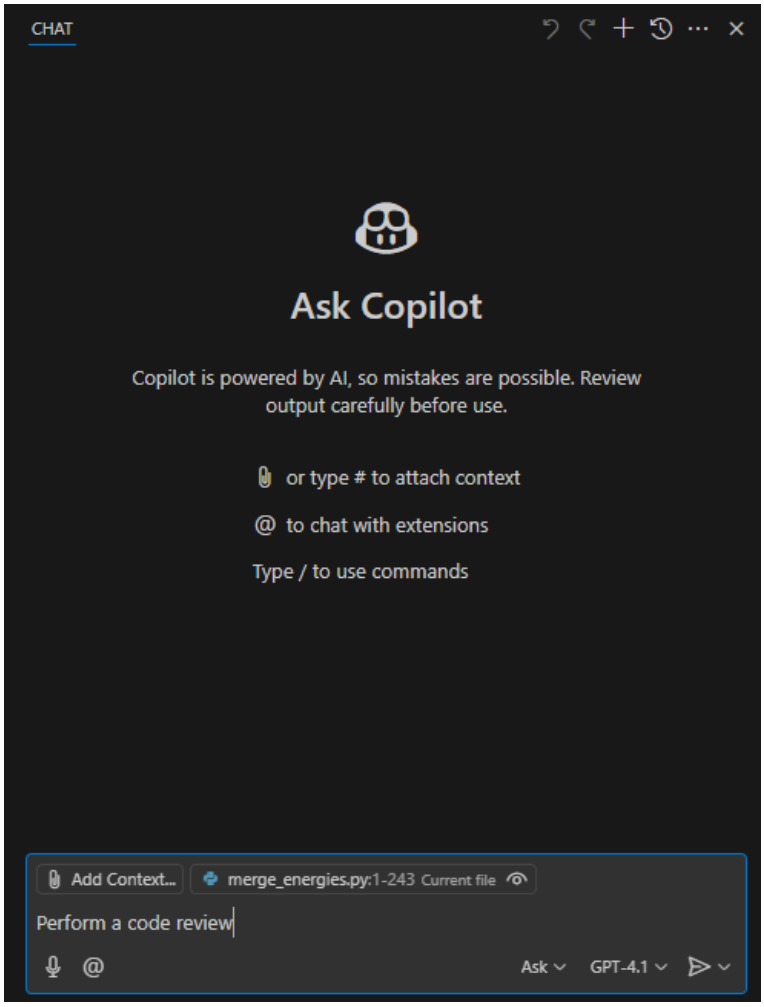

Figure D.2: Copilot CR in Chat

D.1.4 Code Review Value Categories

To tackle the broad character of the suggestions from copilot's code review, we have tried to generalize issues that copilot can catch and the value of them, and we are interested to know if you share our opinion. Specifically, we want to know which issues suggested by copilot are the most helpful, and which suggestions that are distracting.

High Value

Catches Issues

- Catches spelling/consistency errors
- Type safety issues, matching developer concerns
- Hygiene issues, as unused imports, stray print(), and outdated docstrings

Provides Suggestions For

- Refactoring function with rationale
- Performance optimization suggestions
- Architecture improvements (dependency injection, unified approaches)

Medium Value

Checks for Code Quality

- Error handling improvements (try/catch, explicit exceptions vs assertions)
- Type annotation suggestions throughout
- Magic number identification

Testing and Docs

- Missing docstring identifications
- Suggests edge case tests
- Suggest doc improvement for consistency

Lower Value

Generic Suggestions

- Extensive validation and error handling recommendations (not really mentioned in human reviews)
- Edge case handling – might not be relevant practically
- Backwards compatibility for python versions

Appendix E

Pair Programming Prompt Guide Iteration 2

E.1 Copilot Guide: Iteration 2

This guide builds on the initial prompt guide, which included more task-specific one-way prompts, providing prompts and examples of interaction that shift copilot to a collaborative partner. The purpose of the guide is to help developers with the creative/strategic parts of development, drawing inspiration from the discussions between developers emerging during pair programming sessions.

We have divided the guide into four areas we believe represent the different creative/strategic parts of programming, with potential follow-up questions to support a productive dialogue.

E.1.1 Context

These examples aim to enhance interaction with Copilot by enabling deeper dialogue and exploring creative areas where AI assistance can be most valuable during development. This experiment evaluates Copilot's capacity for brainstorming and collaborative ideation, including the functions that are typically fulfilled by colleagues. Our primary focus is assessing Copilot's performance in creative and strategic domains and measuring its practical value in these contexts.

The kind of feedback that we're looking for: - Which of these areas are most useful? And the most useful subcategories? - Does this workflow feel natural to use? - How well did the prompts work? - How well did the followup prompts work? - Basically, we're interested in how well Copilot works when using it for creative and strategic problems

Please take 30 seconds after a Copilot session to write down problems, thoughts or other observations in a diary

E.1.2 When to Use This Guide

Use this guide when you're working on tasks that involve **creative thinking**, **strategic planning**, or **design decisions**, not just writing code. It's most helpful in moments where you're unsure how to start, want to explore multiple options, or need a second opinion on architecture, implementation, or trade-offs.

This guide is designed to help you treat Copilot as a **collaborative partner** rather than just a code generator. Whether you're brainstorming a new feature, breaking down a complex problem, or refactoring legacy code, the prompts and examples here are meant to spark meaningful dialogue and sharpen your thinking.

Refer to this guide whenever you find yourself thinking: - "I'm not sure what the best approach is here." - "I wish I had someone to bounce ideas off." - "This problem feels vague or open-ended."

By shifting the conversation with Copilot toward exploration and reasoning, you'll get more value out of the tool—especially during the early and ambiguous phases of development.

E.1.3 Table of Contents

  - 4. Refine the Implementation
    * 4.1 Performance Optimizaiton
    * 4.2 Debugging Dialogue
    * 4.3 Refactoring Partner
    * 4.4 Refactoring Strategy (Legacy Code)

E.2 Quick Reference

Instead of treating Copilot as a code generator, position it as an active collaborator that questions, challenges, and refines ideas. Use these conversation starters:

1. Understand the Problem

- Clarify the problem and assumptions.
- Example: "Before we start coding, help me clarify what we're really trying to solve."
- Example: "What questions should I be asking that I'm not asking?"

2. Explore Solutions

- Brainstorm approaches and discuss trade-offs.
- Example: "What are three different ways to approach this?"
- Example: "Challenge my assumptions about [approach]"
- Example: "Play devil's advocate on this design decision…"

3. Plan Implementation

- Break down the solution into steps and anticipate risks.
- Example: "How should we break this down into implementable steps?"
- Example: "Which parts of this implementation are likely to take the most time?"

4. Refine the Implementation

- Improve the solution and check for edge cases.
- Example: "How can we improve what we just wrote?"

Meta Prompting

Description: Refine your prompts for better results by letting Copilot help clarify your intent.

Example Prompts: - "I'm trying to get better results from prompts for generating unit tests. Can you help me refine my thinking by asking questions about what I want first?"

E.3 Example of Copilot interaction

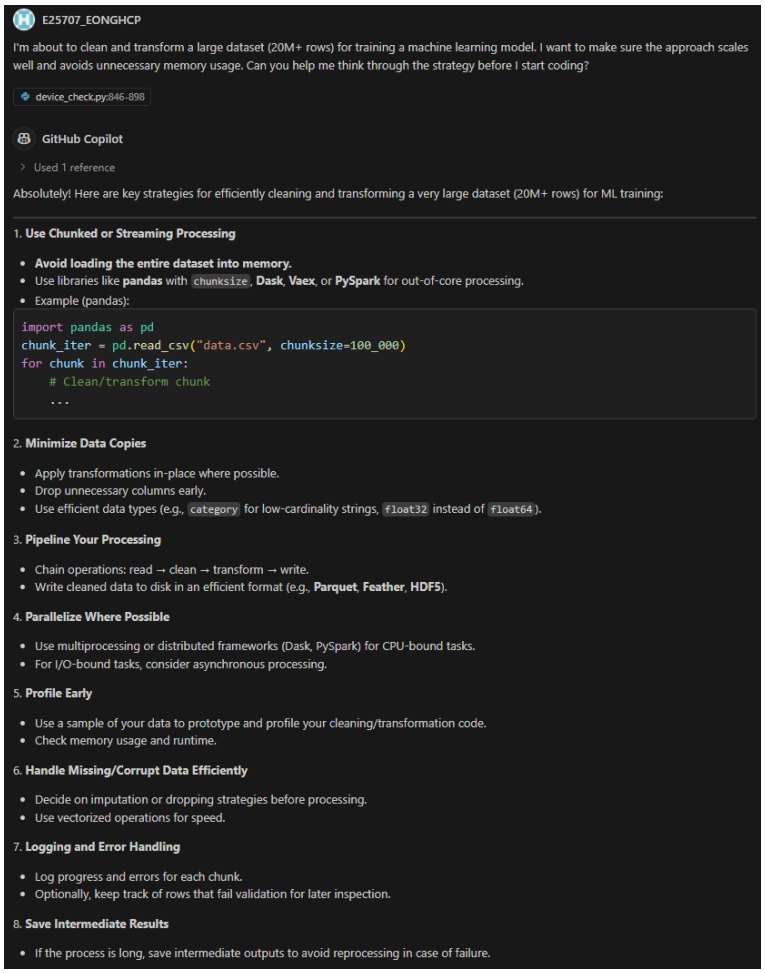

Figure E.1: Image of Copilot interaction

E.4 Collaboration framework

E.4.1 1. Understand the Problem

1. Feature Design Collaboration

Description: Use Copilot as a design partner, let it clarify requirements before suggesting code or architecture.

Example Prompts: - "I want help designing a new feature for my program. Let's figure it out together, please ask clarifying questions before suggesting code or architecture."

- "I'm designing [system/feature]. What are the key architectural decisions I should consider? Challenge my assumptions about [specific aspect]."
- "Help me think through the scalability implications. What questions would an experienced architect ask about this design?"

Follow-up refinement:

- "Given those concerns you raised, how would you modify this approach"
- "What additional constraints should influence this decision?"
- "What would change your recommendation if we had [different constraint]?"
- "Walk me through how [technology] works at a conceptual level before we dive into implementation."

2. Requirements Clarification

Description: Clarify requirements and edge cases before implementation.

Example Prompts: - "Before we implement this feature, help me identify edge cases and clarify requirements." - "What questions would an experienced developer ask about this story?" - "What could go wrong if we don't address this upfront?"

Follow-up refinement:

- "What happens in the failure scenarios you mentioned?"
- "How do those edge cases change our implementation approach?"
- "What additional business rules should we consider?"

3. Creative Exploration

Description: Explore vague ideas or possible implementations through a back-and-forth conversation.

Example Prompts: - "I have a vague idea and want to explore possible implementations. Let's have a back-and-forth conversation — start by asking what you need to know."

E.4.2 2. Explore Solutions

1. Solution Exploration

Description: Brainstorm multiple approaches and discuss their pros and cons.

Example Prompts: - "I need to implement feature X. Walk me through 3 different approaches and discuss the pros and cons of each. Which one would you lean toward and why?" - "Compare these two approaches: [approach A] vs [approach B]. What factors should influence my decision?" - "What implementation challenges would we face with [approach]?

Follow-up refinement: - "What's the maintainability of each approach?" - "How do these approaches handle future requirements changes?" - "Which approach would you choose and why?" - "How would you test this approach?"

2. Architectural Trade-Offs

Description: Consider key architectural decisions and their trade-offs.

Example Prompts: - "I'm designing [feature description]. What key architectural decisions should I consider?" - "Given those trade-offs, what would you recommend for a team of our size and scope?"

Follow-up refinement: - "Where are the potential failure points?"

E.4.3 3. Plan Implementation

1. Problem Decomposition

Description: Break down complex problems into manageable steps with Copilot as a co-navigator.

Example Prompts: - "What happens step by step when X occurs?" - "How could we break this feature down into smaller parts? What would be your step-by-step approach?" - "What should we implement first to validate our approach?"

Follow-up refinement: - "How would you sequence these tasks?" - "What would you do if [specific step] proves more complex than expected?" - "Which assumptions should we validate first?"

2. Resource and Effort Estimation

Description: Estimate the relative complexity, effort, and time required for parts of the plan. Use Copilot to identify likely bottlenecks, quick wins, or areas that require special attention.

Example Prompts: - "Which parts of this implementation are likely to take the most time?" - "How long would it take to build a basic version of this?" - "What could be implemented quickly to demonstrate value early?" - "Which parts would you consider low-risk, high-effort vs high-risk, low-effort?"

Follow-up refinement: - "What dependencies or blockers could delay progress?" - "Are there parts of this we could parallelize to save time?"

E.4.4 4. Refine the Implementation

1. Performance Optimization

Description: Use Copilot to analyze code for potential performance suggestions and identify optimization opportunities

Example Prompts: - “Analyze [function] for performance bottlenecks and suggest optimizations”

Follow-up refinement: - “Why is that suggestion better than the current implementation?”

2. Debugging Dialogue

Description: Work with Copilot to diagnose errors by narrowing down the issue through questions.

Example Prompts: - “My code is throwing an error, but I don’t know why. Instead of jumping to conclusions, could you ask me a few questions to narrow down the issue together?”

3. Refactoring Partner

Description: Engage Copilot to help you refactor code by first understanding your goals and context.

Example Prompts: - “I want to refactor this function, but I’m not sure how to best improve it. Can you ask me some questions first to understand the context and my goals for the refactor?”

4. Refactoring Strategy (Legacy Code)

Description: Discuss maintainability and technical debt, especially when refactoring legacy code.

Example Prompts: - “This class has too many responsibilities, but if we split it, we’ll need to update 20 other files. What’s a good migration strategy?” - “Let’s isolate this logic in case the requirements change next quarter.”

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD 2025-08-27

EXAMENSARBETE Exploring the Integration of Generative AI in Pair Programming and Code Review
STUDENTER Erik Malmgren, Isak Määttä
HANDLEDARE Lars Bendix (LTH), Christer Friberg (E.ON)
EXAMINATOR Per Andersson (LTH)

När AI kliver in i utvecklarteamet

POPULÄRVETENSKAPLIG SAMMANFATTNING **Erik Malmgren, Isak Määttä**

Föreställ dig att du aldrig mer behöver sitta fast i timmar med att jaga småfel i koden, eller att du på sekunder kan få förslag på hur du ska lösa en komplex programmeringsuppgift. Det låter för bra för att vara sant — men det är precis vad som händer när generativ AI kliver in i mjukvaruutvecklarens värld.

Mjukvaruutvecklare gör mycket mer än att bara skriva ny kod. En stor del av tiden går åt till att samarbeta, granska och förbättra varandras arbete. Parprogrammering, där två personer kodar tillsammans framför samma skärm, och kodgranskning, där en kollega går igenom en annan kollegas kod, är etablerade sätt att säkra kvalitet och sprida kunskap. Men allt är inte guld och gröna skogar: metoderna är tidskrävande, och mycket av arbetet stjäl fokus från det riktigt spännande – det kreativa problemlösandet.

Tillsammans med E.ON undersökte vi hur generativ AI kan vävas in i utvecklarnas vardag. Resultaten? AI fungerar som en outtröttlig "kodkompis" som snabbt föreslår lösningar, skriver om kodstycken och hjälper till med dokumentation, alla timmar på dygnet. Utvecklare upplevde att AI-verktyg snabbt föreslår lösningar på knepiga problem, skriver om komplicerad kod till något mer läsbart, och ökar produktiviteten märkbart.

Intressant nog visade resultaten också att AI kan stötta i mer avancerade situationer. Den kunde bryta ned stora problem i mindre delar och genom dialog med utvecklaren resonera fram alternativa vägar framåt. Ungefär som att ha en erfaren kollega tillgänglig à la minute!

När det gäller kodgranskning visade sig AI fungera bäst som ett första filter. Snabbare än ögat kan blinka fångade den upp enklare fel, som stilbrott, glömda parametrar eller uppenbara buggar. Det frigjorde tid för människan att fokusera på det de gör bäst: arkitektur och domänspecifika frågor där AI ännu inte har samma kompetens. Resultatet? Granskningar blev effektivare utan att den mänskliga expertisen gick förlorad.

En central observation är att AI fungerar bäst när den används som ett komplement, inte som en ersättare. För att lyckas krävs tydliga arbetsmetoder, välformulerade instruktioner till AI:n, och en medvetenhet om verktygens begränsningar. Felaktiga förslag eller så kallade "hallucinationer" förekommer. Erfarna utvecklare som behåller kontrollen och kritiskt granskar AI:ns förslag får ut mest av samarbetet. Tricket är att integrera AI i befintliga arbetsflöden, snarare än att bygga helt nya rutiner runt verktygen.

Spännande frågor ligger runt hörnet: En fråga är hur långvarig användning påverkar utvecklares arbetssätt och lärande? En annan är hur olika AI-modeller skiljer sig åt i praktiken? Och hur kan företag köra AI-verktyg i sina egna utvecklingsmiljöer?

AI är alltså inte på väg att ersätta utvecklare, den blir istället den perfekta kollegan som tar hand om det repetitiva och frigör tid för det människor gör bäst: kreativt problemlösande.