Free University of Berlin
Department of Mathematics and Computer Science
Institute of Computer Science

Pair Programming with AI: Analyzing the challenges and limitations of the new form of programming for professional and novice programmers

Bachelor's Thesis

1st Reviewer: Prof. Dr. Lutz Prechelt
2nd Reviewer: Prof. Dr. Claudia Müller-Birn

Thesis adviser: M.Sc. Linus Ververs

Author: David Vadim Friedmann

Matriculation number: 5337038
david.friedmann@fu-berlin.de

May 2024

Declaration of Authorship

I certify under penalty of perjury, that the present work has been produced by me independently, all tools and sources are used as indicated and the work has not been submitted to any other institution for consideration.

Berlin, Mai 10, 2024  David Vadim Friedmann

Abstract

This bachelor's thesis explores the emerging phenomenon of pair programming with AI-powered coding assistants (a human + AI tool), focusing on the challenges and limitations experienced by both professional and novice programmers.

The rapid advancement of AI technology has led to the development of tools like GitHub Copilot, which aim to streamline the coding process by providing intelligent code suggestions and automating repetitive tasks. However, the impact of these tools on pair programming dynamics and the potential drawbacks of relying on AI assistance remain largely unexplored. Through a qualitative analysis using grounded theory methodology (and some complementary elements of conversational analysis), this research investigates the experiences of programmers collaborating with GitHub Copilot.

The findings reveal that while AI-assisted pair programming offers benefits such as increased productivity, it also introduces new challenges related to over-reliance on AI suggestions. This thesis concludes by highlighting the need for a balanced approach to AI-assisted programming, emphasizing the importance of human oversight, critical thinking, and continuous learning to ensure the effective integration of AI tools into the software development process.

Introduction

The release of ChatGPT in late 2022 marked a significant milestone in the field of artificial intelligence, captivating the attention of professionals across various industries, including software development. This powerful language model, developed by OpenAI, demonstrated remarkable capabilities for understanding and generating human-like responses, sparking widespread interest in the potential applications of AI technology. As a result, AI-powered tools have gained unprecedented attention, with developers eagerly exploring how these advancements can be leveraged to enhance their workflows and streamline the software development process.

One such tool that has garnered significant interest among programmers is GitHub Copilot, an AI-powered coding assistant that aims to revolutionize the way developers write code. By providing intelligent code suggestions, automating repetitive tasks, and offering context-aware assistance, Copilot promises to boost productivity and improve code quality. However, the integration of AI into the traditionally human-centric practice of pair programming raises important questions about the challenges and limitations that may arise from this collaboration.

By examining the experiences of both professional and novice programmers who collaborate with Copilot, this bachelor's thesis aims to shed light on the challenges and limitations that arise from this new form of programming partnership. The insights gained from this study will contribute to a better understanding of how AI can be effectively integrated into the software development process, while addressing potential drawbacks and ensuring the continued growth and skill development of programmers.

In their paper "Explaining Pair Programming Session Dynamics from Knowledge Gaps", F. Zieris and L. Prechelt [1] introduced the concept of system-specific (S) and generic (G) knowledge to analyze knowledge gaps in pair programming. Their work highlights the influence of these knowledge gaps on session dynamics. They found that gaps in S knowledge were more detrimental than G knowledge gaps, and pairs with complementary knowledge were particularly effective.

While the S and G knowledge framework provides valuable insights, it may not be directly applicable to this bachelor's thesis due to the participant pool. Experienced programmers possess a high level of both S and G knowledge, as they have extensive experience with the software system and strong programming fundamentals. Conversely, novice programmers have limited knowledge in both areas due to their early career stage and lack of exposure to the specific system. Consequently, the knowledge gap analysis based on S and G knowledge might not be the most suitable approach for understanding the dynamics of the programming sessions in this thesis, as the participants fall into more distinct categories of overall knowledge level.

Structure of this thesis

To conduct the study for this bachelor's thesis, I observed four programmers (two professionals and two novices) working with GitHub Copilot. The programming sessions were recorded and analyzed using grounded theory methodology. My observations and analysis reveal that programmer experience and goals significantly influence the nature and effectiveness of collaboration with Copilot during real-world coding sessions. Thus, programmers with different levels of expertise might leverage Copilot for distinct purposes, such as accelerating code completion or exploring alternative functionalities.

Each programmer worked on their own chosen tasks; there was no standardized assignment for the participants. This is one of the main differences between my study and existing qualitative research on GitHub Copilot. The majority of the papers I found were based on studies where programmers received a specific task to implement with Copilot. This approach – observing programmers without a premade assignment – allowed me to gain insights into how programmers integrate Copilot into their typical workflows. However, it is important to acknowledge that the lack of standardized tasks made it more challenging to directly compare this thesis' findings with studies that employed standardized assignments.

This thesis is structured into five main chapters, each focusing on a different aspect of pair programming with AI-powered coding assistants.

1. The first chapter provides an introduction to the research topic, highlighting the background and significance of AI in software development. Additionally, the chapter provides a historical perspective on pair programming and a detailed overview of GitHub Copilot, the AI-powered coding assistant that forms the core of this study.

2. The second chapter reviews existing research on GitHub Copilot and its impact on programmers who utilize the tool.

3. The third chapter delves into the methodology employed in this research. It explains the qualitative approach using grounded theory methodology, justifying its suitability for examining the experiences of programmers collaborating with GitHub Copilot.

4. The fourth chapter outlines the data collection process, including the participant selection and the tools used to gather and analyze data.

5. The fifth chapter is the main chapter of this thesis, where I present all the findings of the grounded theory methodology analysis on programmer's interactions with Copilot. It also contains a subchapter with complementary quantitative analysis, which helped to reveal additional insights and to identify usage patterns and support qualitative observations.

6. The sixth chapter discusses the implications of the findings, situating them within the broader context of AI integration in software development. It explores the potential long-term effects of AI-assisted pair programming on the skills and knowledge acquisition of programmers.

Acknowledgments

I'm incredibly grateful to my partner, Etel, whose unwavering support made this thesis possible. She sacrificed countless days, nights, and weekends, understanding the demands of this project. Her encouragement and belief in me kept me going, especially during challenging times.

I also owe a deep debt of gratitude to my thesis advisor, Linus Ververs. His invaluable guidance and insightful feedback significantly improved my thesis. His expert advice helped me navigate the research process and refine my work to a much higher standard.

Special thanks to my little daughter, Astrid, who provided excellent moral support in the form of peaceful naps and nights.

Table of content

1. Laying the Foundation: AI in Software Development and Pair Programming

1.1. Pair programming

What is pair programming? Despite the inclusion of "programming" in its name, it does not necessarily involve solely the creation of program code. In some cases, it can also refer to the collaborative conceptualization of an algorithm or conducting a test together.

The “classical” approach refers to pair programming (thereafter referred to as PP) as a process in which two individuals work together at the same computer on a shared design, algorithm, code, or test, as defined by Laurie Williams [2, p. 1].

The person in control of the keyboard and mouse is commonly referred to as the Driver. Various role names have been devised for the other person, but probably the most common term in scientific papers is Navigator [3, p. 1], [4, pp. 14, 61]. Other possible options are Non-Driver, Observer, and Reviewer.

In line with these traditional designations, the typical responsibilities of each role are defined as follows: the Driver performs the actual work, while the Navigator maintains an overview, considers strategic consequences, and highlights weaknesses. After some time, the programmers will usually switch roles, allowing each to benefit from the other's perspective.

The following research sees this model as simplistic, as it does not always reflect the reality of PP and might be misleading for beginners. S. Salinger, F. Zieris and L. Prechelt [4] used grounded theory methodology to analyze video recordings of real PP sessions. They identified various roles that programmers take on during these sessions, along with the actions associated with those roles. For example, a "*watchman*" might identify potential problems and suggest solutions, while a "*task expert*" might share their knowledge about the task at hand.

The findings of this study suggest that PP is more complex than previously thought. Programmers can shift between different roles throughout a session, and they can even hold multiple roles simultaneously [5, p. 4]. This complexity highlights and goes beyond the traditional driver/navigator model. I keep both approaches in mind for the further analysis.

There is evidence of similar to pair programming collaborative coding approaches being used, at least in the 1970s [6, pp. 17–18]. Bolboacă even claims that pair programming is basically as old as programming itself: “*Pair programming has been around for a long time in the history of software development, even from the early days, when programming meant plugging in some wires into a control dashboard or inserting punch cards into a reader in order to compile your program. It's just that people who used to program back in the day didn't call it pair programming*” [4, p. 4].

Since 1999, when the concept of pair programming was formalized by Kent Beck [7, pp. 42–43], pair programming gained popularity. Dozens of scientists conducted ample research, highlighting its advantages for both productivity and learning. These are some of them: improved code quality, enhanced problem-solving [2, p. 5-6], increased coding knowledge exchange, increased developer confidence in their code, and a more enjoyable coding process [8, p. 95].

However, having two developers at one computer incurs double the cost of a single developer, prompting the question of cost-effectiveness: do the described benefits outweigh the doubled work hours? Since the 1990s, numerous studies have been conducted to address this question, yielding mixed results. Clear-cut conclusions, crucial for economic decision-makers, have proven elusive.

However, the economic benefits of pair programming become less central to the decision-making process when one of the programmers is replaced by an artificial intelligence tool. This shift also raises another question: can an AI tool fully substitute a human programming partner?

1.2. Artificial Intelligence and Pair programming

The development of modern AI programming tools has taken a long time, with roots stretching back much further than one might think.

In the late 1980s, two artificial intelligence researchers from MIT, Charles Rich and Richard C. Waters, envisioned a groundbreaking concept in their seminal book, "The Programmer's Apprentice" [9]. Recognizing the promising potential of artificial intelligence in aiding programmers, the authors explored the development of a programming assistant aimed at radically transforming the collaborative process by including AI in it. Their vision extended beyond conventional programming paradigms, delving into the realm of natural language processing and problem-solving, laying the foundation for an innovative approach to computer-assisted programming.

Published in 1990, the book revealed the authors' visionary insight into the transformative potential of AI, foreseeing a future where intelligent systems actively contribute to the programming process.

Fast-forward 35 years, with enhanced computational power, the availability of vast datasets, and advancements in the pre-training and architecture of neural networks, a perfect convergence for natural language processing tasks has emerged in recent years.

Since their introduction in a 2017 paper written by eight scientists working at Google [10], transformer models have become the dominant architecture for large language models (LLMs), paving the way for services like ChatGPT. Nearly every popular chatbot or AI assistant in recent years utilizes transformer-based LLMs, including:

- OpenAI's ChatGPT (the "T" in GPT stands for "transformer" [11])

- GitHub Copilot (details in the subsection "Short history of GitHub Copilot" below)
- Microsoft's Bing Chat and the family of other Copilots (based on OpenAI's GPT models, DALL-E models, and Prometheus technology combining Bing Search and GPT models [12-14])
- Google's Bard [15-16] and Gemini (which combines transformer and a Sparsely-Gated Mixture-of-Experts layer architecture, more details on [17])
- Meta's CodeLlama [18]
- Salesforce's CodeT5 [19, p. 6]
- Replit's GhostWriter [20]
- AskCodi (powered by external LLMs, e.g., OpenAI's GPT, Meta's CodeLlama, etc. [21])
- Tabnine [22]
- Replete AI [23]
- Accubit's Polycoder [24]
- Snyk's DeepCode AI [25]

These LLMs typically involve modifications and different approaches to training and fine-tuning.

Several companies, including Amazon (CodeWhisperer), Anthropic (Claude family of models), Apple(MM1), and Isotropic (CodeWP), remain tight-lipped regarding the specific architecture underlying their AI-powered coding tools. While they often mention using "large language models", "large multimodal models", "multimodal large language Models" or "custom AI models" (see, e.g., [26-28]) they do not disclose the specific details of the architecture employed. Nevertheless, the lack of transparency does not necessarily imply that these companies are not utilizing the transformer architecture. E.g., while Apple does not explicitly name the architecture used in their model, the company's research paper mentioned that its approach "*based on a decoder-only architecture, akin to Kosmos-1*" [29, p. 4], which is transformer-based [30]. Therefore, the limited information available does not definitively rule out the use of transformer architectures by these companies.

This thesis focuses on the collaborative dynamic between programmers and a single AI coding assistant tool: GitHub Copilot. This choice is driven by Copilot's widespread adoption within the developer community, providing a strong foundation for exploring the impact of AI collaboration on programmers. While the analysis concentrates on Copilot, I believe the findings can be extended to similar AI tools. This approach is a common practice in scientific research, exemplified by an MIT and Microsoft Research paper [31, p.2] by H. Mozannar et al. that employed the pseudonym "*CodeRec*" to emphasize the generalizability of their findings beyond Copilot.

Furthermore, Copilot was designed specifically for developers, aligning with GitHub's marketing slogan from the outset: "Your AI pair programmer." This reinforces its suitability as a representative tool for studying programmer-AI collaboration.

1.3. Short history of GitHub Copilot

Since it was created by GitHub in partnership with OpenAI, Copilot is closely related to the main OpenAI's product: ChatGPT.

ChatGPT gained 1 million users just 5 days after launching on November 30th, 2022 [32]. Two months later, in January 2023, the service already had 100 million users. According to media reports (e.g., [33]), ChatGPT's growth was the fastest ever recorded for a consumer internet application.

At the moment of public launch, ChatGPT was based on OpenAI's 3rd version of general pre-trained transformer, also known as GPT-3. While it was generally possible to generate programming code with ChatGPT, it was not its main feature.

Specifically with software engineers assistance in mind, OpenAI – in collaboration with Microsoft – created another LLM, named Codex. This model was also based on GPT-3, but essentially trained on a dataset of publicly available code repositories on GitHub, allowing it to understand code structure, syntax, and functionalities related to various programming languages. Codex was powering the GitHub Copilot after its official release in October 2021 in a limited beta status [34]. GitHub officially communicated at least once, in February 2023, about improvements, but stated that Copilot is still primarily based on Codex [35].

In March 2023, GitHub announced that its Research and Development Team was working on Copilot X [36] – a new generation of AI tool based on the brand-new OpenAI GPT-4 model, currently with the best performance.

Interestingly, in July 2023 GitHub announced that they are updating the model behind Copilot: "*The improved AI model behind GitHub Copilot goes beyond the previous OpenAI Codex model, offering even faster code suggestions to developers. It was developed through a collaboration between OpenAI, Microsoft Azure AI, and GitHub, and offers a 13% latency improvement over the previous model*" [37].

The announcement did not explicitly say the new model's name. The Copilot X still was not officially released, either. Up to May 2024, when this thesis was finalized, neither GitHub nor OpenAI ever said if the Copilot was currently based on GPT-4.

Solely in November 2023, GitHub informed the public that the new *Copilot Chat* extension is now powered by ChatGPT-4 model [38].

While I can only assume Copilot currently uses GPT-4, I lack direct confirmation. Indirect confirmation of this theory is that the multiple links on the GitHub official website with the official presentation of the Copilot X[1] currently (as of March-May 2024) forward to the Copilots main page[2].

[1] E.g., https://github.com/features/preview/copilot-x
[2] https://github.com/features/copilot

However, I cannot entirely rule out the possibility of Copilot using a different, custom-built model. If this is true, OpenAI and GitHub would have done so secretly, without announcements or press releases.

This brief explanation is crucial. While the studies reviewed in the next chapter focus on GitHub Copilot, the underlying models might differ. These variations, including version-specific features or bugs, could influence the studies' conclusions.

PART 1

2. Related work

The introduction of GitHub Copilot in 2021 marked a significant step forward for professional programmers using AI-powered code completion tools. While earlier offerings like Tabnine (released in 2018 [39]) pre-date Copilot, all these options are relatively new in scientific terms. Research on them is, therefore, an ongoing process.

In this chapter, I review existing papers regarding (pair) programming with Copilot. In general, the majority of publications about Copilot can be divided into two categories: (1) how programmers are using Copilot and (2) how effective Copilot and its generated code is. Due to the topic of this thesis, I am focusing on the first group of studies.

While this thesis focuses on the concept of pair programming with AI-powered coding assistants, it is important to acknowledge that not all researchers consider programming with AI to be a form of pair programming. Some argue that the unique dynamics and challenges introduced by AI-assisted programming warrant its classification as a separate category altogether. Researchers who view AI-assisted programming as distinct from pair programming point to the fundamental differences in the nature of the collaboration between human programmers and AI-powered tools. E.g., Sarkar et al. states that programming with AI "*ought to be viewed as a new way of programming with its own distinct properties and challenges"* [40, p.1].

However, it is crucial to mention that the discussion surrounding the categorization of programming with AI as either pair programming or a separate form of AI-assisted programming is not the primary focus of this thesis. While acknowledging the existence of this debate, the current study aims to explore the challenges and limitations experienced by programmers collaborating with GitHub Copilot, regardless of the specific categorization of this collaboration.

2.1. Is AI the better programming partner?

To find relevant papers for this review, I utilized search engines, scientific databases, and surveys by Ma et al. [41] and Ani et al. [42]. However, I conducted my own analysis of the original studies.

2.1.1. Changing programmer behaviors and interactions with Copilot

A recent study performed by MIT and Microsoft Research (Mozannar et al., [31]) sheds light on how programmers collaborate with AI tools. The authors considered GitHub Copilot only but assumed that their conclusions were valid for similar code completion instruments. These tools promise increased programmer productivity, but the researchers were also

curious about potential downsides. Their primary focus was on understanding h*ow* programmers interact with these AI assistants during the coding process.

To achieve this, the researchers developed a classification system named Coding User Activity with Programming Suggestions (CUPS). CUPS categorizes different programmer actions while using AI code completion tools. Here are all of them:

- *Prompt crafting*
- *Writing Documentation*
- *Writing New Functionality*
- *Editing Written Code*
- *Editing Last Suggestion*
- *Debugging/ Testing Code*
- *Waiting For Suggestion*
- *Thinking*
- *Verifying Suggestion*
- *Deferring thought for later*
- *Not Thinking*
- *Thinking about New Code to write*
- *Looking up Documentation*

The researchers then conducted a study involving 21 professional developers who completed assigned coding tasks with Copilot's assistance. By analyzing recordings of these sessions through the lens of CUPS, they were able to categorize the programmers' specific interactions with the AI tool.

The analysis revealed a change of programmer behaviors. Programmers spend a significant portion of their time (34.3%) on activities related to Copilot suggestions. This includes double-checking, editing, and waiting for suggestions to be generated. Interestingly, some software engineers might accept suggestions without immediate verification, intending to review them later. While some participants felt more productive with Copilot, objective data on task completion times was not conclusive.

However, it is important to mention that Mozannar et al. do not consider human-AI interaction of programmer and Copilot during their experiments as any form of pair programming.

In contrast, scientists from Microsoft Research (C. Bird et al.) view programmer-Copilot interaction through the lens of pair programming. They are explicitly talking about a shift in PP: Copilot represents a new approach to the established technique, where the AI acts as a “navigator” suggesting code, while the human developer remains in control (“driver”) reviewing and editing the suggestions([43, pp.1-2). This frees developers from writing every line of code themselves, potentially increasing productivity.

The researchers used three methods to understand how developers interact with Copilot:

1. Analyzing discussions from a Copilot forum to understand user experiences and challenges.

2. Conducting a case study with new Copilot users to observe their engagement with the tool for various tasks.
3. Launching a large-scale survey to measure Copilot's impact on developer productivity.

Bird et al. discovered that programmers leverage Copilot for various tasks beyond basic code completion, including learning new languages and unit test writing. Their study suggests a shift in developer roles, with more time spent reviewing and comprehending AI-generated code compared to writing it independently.

While the survey indicates improved productivity for many developers, challenges remain. These include potential security vulnerabilities and a lack of transparency in code suggestions. The researchers compiled the gathered insights (positive, negative, and mixed) into a small table, which is included here without modification (Table 1).

THE GOOD	THE BAD	THE MIXED
users reported productivity improvements	users report Copilot gets into loops of suggesting the same thing	less coding but more reviewing
users compared Copilot to a human	Copilot doesn't write "defensive code" e.g., check null pointers	less time on Stack Overflow, but now less understanding of how/why the code works
"Copilot removes the ceiling on creativity"	Copilot sometimes suggests inappropriate text	API discoverability is supported but doesn't provide enough information to select best solution
some said Copilot suggested something they would "ordinarily overlook"	Copilot at times leaks PII in header files	

Table 1. *Early experiences with Copilot according to Bird et al. [43, p.9].*

As Table 1 demonstrates, while the researchers identified some limitations in Copilot's suggestions (e.g., potential security vulnerabilities), they also noted multiple positive aspects. Perhaps the most surprising finding was the comparison of Copilot to a human collaborator, and the idea that the tool can extend developers' creative boundaries.

While Microsoft Research might have a vested interest in Copilot's positive evaluation (Copilot is a product created by Microsoft's subsidiary GitHub, and OpenAI, for which Microsoft is a strategic partner and one of the biggest investors [44]), other researchers are also exploring its effectiveness as an AI pair programmer.

2.1.2. Copilot as a pair programming partner and its impact on productivity

Another empirical study investigates pair programming with GitHub Copilot in comparison to human-human pair programming. S. Imai from Colby College performed an experiment with 21 participants who were programming with Copilot (first phase) and then in pairs with another human as Driver (second phase) and as Navigator (third phase). The codes generated from all the trials were analyzed to determine how many lines of code, on average, were added in each condition and how many lines of code, on average, were removed in the subsequent stage. While research by S. Imai suggests that Copilot can help generate more lines of code compared to traditional human pair programming within the same timeframe, the study also found that the quality of the code generated by Copilot appeared to be lower [45, pp. 1-2]. It is important to consider that Copilot's underlying models have been updated multiple times since the spring of 2022, when the paper was published, so the quality of generated code may have also improved.

One more empirical study of GitHub Copilot was performed by researchers from Harvard and Purdue Universities [46]. They investigated the real-world effectiveness of AI-powered code generation tools.

The scientists conducted a user study with a “within-subjects” design where programmers (undergraduate, master, and Ph.D. students and one professional software engineer; only one participant with less than two years of programming experience) were tasked with completing coding activities. The participants used both Copilot and a default code autocompletion tool (IntelliSense) in Visual Studio Code for two different tasks. By comparing task completion times, success rates, and user feedback, the study aimed to bridge the gap between the theoretical promise and practical experience of using LLMs for code generation.

The findings revealed an interesting disconnect. While Copilot did not necessarily improve task completion times or success rates compared to the traditional tool, participants generally preferred using Copilot. This suggests that Copilot's value might lie beyond simply accelerating coding. It could potentially enhance the development experience by offering creative suggestions or prompting new approaches to problem-solving. However, the study also highlights that Copilot could be more useful with more support for understanding and validating the generated code [46, p.6].

2.1.3. Interaction patterns and experiences with Copilot

A particularly relevant study for my chosen approach comes from researchers at UC San Diego, USA. Barke et al. utilized grounded theory methodology, a research approach that builds understanding through close observation of the phenomenon of interest [47].

They recruited 20 participants with varying levels of Copilot experience and observed them working on diverse, relatively small *given* programming tasks across four different languages. Only three of 20 were real-world software engineers, the rest came from academia (students, postdoctoral researchers, and professors). To understand programmer-Copilot interaction, the researchers analyzed video recordings of the programming sessions, focusing on the participant’s verbal and non-verbal actions.

The core finding of the study revolves around two distinct modes of programmer-Copilot interaction:

Acceleration Mode: In this mode, the programmer already knows the desired outcome and uses Copilot to expedite the coding process. Copilot helps them write code faster by suggesting completions or generating specific code snippets.

Exploration Mode: When a programmer is unsure about the best approach, they enter exploration mode. Here, Copilot is used to explore different coding possibilities. The programmer might use Copilot's suggestions as a springboard for brainstorming solutions or experimenting with alternative functionalities.

This bimodal interaction pattern highlights how Copilot can be a versatile tool, adapting to the programmer's current needs during a coding session. The study also emphasizes the importance of programmer vigilance and the need for careful evaluation of AI-generated code.

Prather et al. [48] reached similar conclusions after their empirical study. This international group of researchers specifically investigated how novice programmers interact with GitHub Copilot. Their research aimed to understand the usability and unique interaction patterns that emerge when beginners leverage this AI tool.

The researchers recruited 20 participants with "*little to no prior programming experience*" [48, p. 4]. Each participant completed a series of programming tasks while using Copilot. The researchers observed and recorded these sessions, focusing on how the participants interacted with Copilot verbally and non-verbally.

The study revealed two key findings: Firstly, while Copilot can accelerate coding for experienced programmers, novices exhibited a broader range of interactions. In addition to using Copilot to write code faster, they also employed it for exploration purposes. For instance, they might use Copilot's suggestions to brainstorm solutions or experiment with different functionalities.

Secondly, Prather et al. identified two distinct interaction patterns specific to novices: "*shepherding*" and "*drifting*". Shepherding involved participants spending significant time crafting prompts to guide Copilot towards the desired outcome. Drifting, on the other hand, involved passively accepting Copilot's suggestions without fully understanding the generated code. These patterns highlight the challenges novices face in effectively utilizing Copilot and the importance of fostering critical thinking and code evaluation skills alongside AI assistance.

This study by Prather et al. suggests that Copilot can be a double-edged sword for beginners. While it offers assistance and can accelerate learning, novices might require additional guidance to leverage its functionalities effectively and avoid potential pitfalls associated with overreliance on AI-generated code.

The final paper I want to mention in this review is the study by Sarkar et al. [40]. The Microsoft researchers explored the concept of programming with large language models (LLMs) like those used in GitHub Copilot. Their goal was to understand how this collaboration between programmers and AI assistants differs from traditional programming methods.

The scientists did not directly involve human participants. Instead, they analyzed publicly available reports of experiences with LLM-assisted programming. Additionally, they drew upon existing usability studies and design principles related to programmer assistance tools.

Through this analysis, Sarkar et al. identified key distinctions between LLM-assisted programming and prior approaches. Unlike traditional compilers that check code syntax or search tools that help find existing code snippets, LLM-based assistants can understand natural language descriptions and generate code. While Copilot offers suggestions and corrections similar to a pair programming partner, the programmer ultimately decides whether to incorporate them. This collaborative aspect combined with the AI's ability to generate code sets LLM-assisted programming apart, as was mentioned previously.

The study by Sarkar et al. also highlights the potential of LLM-based tools to transform programming into a more collaborative endeavor.

In conclusion, the first part of this thesis has provided a comprehensive overview of pair programming and the emergence of AI-powered coding assistants, focusing on GitHub Copilot as a prime example. The historical context of pair programming, dating back to the early days of computing, highlights the long-standing recognition of the benefits of collaborative coding practices. The formalization of pair programming in the late 1990s and its subsequent popularity as a core practice in agile software development methodologies underscore its significance in the field.

However, the introduction of AI-powered coding assistants, such as GitHub Copilot, has introduced a new dimension to the concept of pair programming. By leveraging the power of large language models and machine learning, these tools offer developers intelligent code suggestions, automate repetitive tasks, and provide context-aware assistance. The potential benefits of AI-assisted programming, such as increased productivity and improved code quality, are undeniable. Nevertheless, it is crucial to acknowledge the limitations and potential drawbacks of relying on AI-powered coding assistants, which will be explored in greater detail in the following part of this thesis through an empirical qualitative study.

PART 2

3. Grounded theory methodology

Grounded theory (GT) is both a research method and a research methodology. Emerging from the field of social research, grounded theory methodology (GTM) initially found application in healthcare and education (e.g., [49]). In the last few decades, it has also become a common approach in software engineering research [50, p.1], including research in the Institute of Computer Science at the Free University of Berlin.

GTM is not a monolithic methodology. Two distinct theoretical streams emerged following an ideological rift between Barney Glaser and Anselm Strauss, the co-creators of GTM. This intellectual disagreement arose after their initial collaboration in 1967, when “The Discovery of Grounded Theory: Strategies for Qualitative Research” was originally published [51]. The controversy led Glaser and Strauss to pursue separate research paths, resulting in the two distinct currents of GTM observed today.

The first, the Glaserian approach, developed by Barney Glaser, prioritizes inductive reasoning and the emergence of theory directly from the unadulterated analysis of data. Here, researchers strive for minimal preconceptions, minimizing the influence of existing literature to avoid imposing pre-existing frameworks. This allows for the discovery of novel patterns and concepts directly grounded in the raw data [51].

In contrast, the Straussian approach, developed by Anselm Strauss and Juliet Corbin, acknowledges the value of prior research and existing knowledge. This method incorporates a more systematic approach, integrating theoretical concepts alongside data analysis. This allows for a more iterative process, where emerging findings are constantly evaluated against established knowledge and refined accordingly. The emphasis here lies on achieving theoretical saturation, ensuring a comprehensive understanding of the phenomenon under study, while maintaining scientific rigor and verifiability [52].

In this thesis project, I leverage the latter technique to analyze video recordings of programming sessions, since this approach acknowledges the value of existing research while allowing for the emergence of new insights from the data itself. This iterative process, where findings are constantly evaluated against established knowledge, ensures a comprehensive analysis grounded in both theory and the video evidence.

From my perspective, this focus on established knowledge while allowing for new insights makes the Straussian approach particularly well-suited for a bachelor’s thesis project.

Open Coding: Fragmenting Data into Meaningful Concepts

Open coding forms the foundation of GTM analysis. In this initial work mode, researchers meticulously fragment the data into discrete segments, assigning labels (codes) that capture the essence of the phenomena observed. This initial sweep involves coding all data elements to gain a comprehensive understanding of core themes and concepts.

As new codes emerge, they are constantly compared against existing ones. This iterative process allows researchers to refine their coding scheme by merging similar concepts, revising labels, or creating new categories. A crucial aspect of open coding is recognizing the multi-dimensionality of concepts. Each concept can manifest differently depending on the context. By identifying these variations, researchers gain a deeper understanding of the data's nuances.

Through this analysis, researchers may refine their coding scheme, merging similar concepts, revising existing labels, or creating entirely new categories to enhance the organization and clarity of their analysis [52, pp. 50-51].

Axial Coding: Unveiling Relationships

The second work mode, axial coding, moves beyond identifying concepts. It explores their connections using a paradigmatic model. This model positions the identified phenomena as central events influenced by causal conditions and shaped by the surrounding context. Axial coding also incorporates intervening conditions, factors that can limit or modify the phenomenon's consequences. This analysis fosters a more profound understanding of the “why” behind the “what”, building a foundation for a more explanatory framework in later stages [52, p.78].

Selective coding: Refining the Narrative

Selective coding marks the final mode of GTM analysis. With the core category identified, researchers refine their focus, prioritizing data that illuminates its connections to other concepts [52, pp. 94-95]. Less relevant phenomena are de-emphasized. The goal becomes building a cohesive theory. Selective coding fosters intricate relationships between categories, capturing the essence of the phenomena at a more abstract level. This refined analysis culminates in a data-driven “story” centered on the core category, explained within the context of supporting themes.

Theoretical Sampling: A Focused Data Selection Strategy

Unlike aiming for a statistically representative sample, theoretical sampling prioritizes data that illuminates a specific category of concepts identified earlier in the analysis process [52, p. 148]. This targeted approach becomes particularly valuable during selective coding, the work mode where in researchers focus on refining their understanding of the core category.

By strategically selecting data rich in insights relevant to the core category, researchers can streamline the analysis and avoid expending effort on data that holds limited value for their developing theory. The goal of theoretical sampling lies not in achieving statistical generalizability, but in fostering a deep understanding of the processes under investigation. Through this deliberate selection process, researchers ensure the data they analyze directly contributes to building a robust theory grounded in the specifics of their study.

4. Analysis of Programming Sessions: Preparation

4.1. Participants, Tools, and Recording Procedures

All the recordings of the programming sessions were organized by the author of this bachelor's thesis. The recordings took place in January and February 2024.

Due to the nature of this research, focusing on programmer interaction with Copilot in their real-world workflows, participants were recruited through the author's professional network. All research procedures adhered to ethical guidelines regarding informed consent and participant anonymity. Personally identifiable information was not collected at any stage.

For general information about participants and their programming experience, please refer to Table 2. Participants P1, P2, and P4 received a small compensation (20-40 Euros per person, depending on request) as a token of appreciation for their time commitment. Participant P3 opted to decline a compensation.

During the recordings, all the participants worked in their home offices and used their personal computers' various operating systems (Ubuntu Linux, Windows and 2x MacOS) and a Visual Studio Code IDE [53]. The GitHub Copilot was installed as an extension within the VS Code. All the participants (except P2) also had the extension GitHub Copilot Chat installed.

To comply with European data protection regulations and minimize burden on participants, the study focused on their work on personal, non-commercial programming projects, eliminating the need for employer permissions.

ID	Age	Gender	Occupation	Computer Science Education	Programming Language	Experience in Programming, years	Prior Experience with GitHub Copilot
P1	23	m	Freelance programmer, English teacher	no	Python	1	no
P2	41	m	Software Developer	yes	C#	20	yes
P3	24	m	Software Developer	yes	Golang	6	yes
P4	26	f	Data Analyst	yes	Python	2	no

Table 2. *Participants of the programming sessions.*

All sessions were conducted remotely using audio and video conferencing software with built-in content sharing capabilities (Webex Meeting in most cases; Skype was used for participant P4 due to initial technical difficulties with Webex). Participants shared their screens throughout the sessions, allowing the recordings to capture both the coding environment (VS Code IDE) and a picture-in-picture video stream of the programmers themselves. The audio and video recording was handled directly by the meeting software (Webex/Skype).

The author of this thesis asked standardized pre- and post-session questions (located in Appendix 1-2) but did not participate in the actual programming. Participants were encouraged to “think aloud” during the sessions, verbalizing their thought processes and decision-making while interacting with GitHub Copilot. This practice aimed to capture not only written and non-verbal communication with the AI tool, but also their internal thought patterns while using Copilot.

Throughout the recording of all programming sessions, it became readily apparent, even prior to the application of GTM for analysis, that participants exhibited distinct utilization patterns of Copilot. Novice programmers, P1 and P4, appeared to encounter similar challenges with general design of the code. Conversely, the seasoned programmers, P2 and P3, demonstrated a more streamlined approach. While the subsequent GTM analysis served to solidify these initial observations, a foundational understanding of participant behavior was established during the observation phase.

4.2. Short description of the programming sessions

Session 1

Participant P1, a freelance programmer who self-taught Python, took part in this session. His overall programming skills can be characterized as moderate. The session's objective centered on connecting a previously written web scraper with a simple website. The goal was to dynamically display the collected data in a format resembling a self-updating with scrolling Facebook feed. Despite possessing a basic understanding of HTML and web development concepts, P1's knowledge was limited. This necessitated extensive natural-language interaction with Copilot throughout the session. P1's inquiries began with fundamental concepts and architecture, continuing through each subsequent step in the development process. The length of the session was 2 hours and 48 minutes.

Session 2

Participant P2 was a highly experienced programmer (20+ years) with a passion for a specific online game (Anarchy Online). He previously led a complex C# hobby-project involving over 300 classes to emulate players in this game. In contrast to P1, P2's interaction with Copilot was minimalistic. Possessing a clear understanding of his goals, he used Copilot during code refactoring and minor adjustments. P2 acknowledged that Copilot's

suggestions were not always optimal, but he still believed the tool saved him time, despite needing careful evaluation to reject unsuitable proposals. The length of the session was 2 hours and 52 minutes.

Session 3

Participant P3, a seasoned software engineer with over six years of experience, focused on a personal project: a German-language roleplay server for the game GTA Online. Due to confidentiality constraints, the recorded session showcased his work on a specific task – crafting a custom URL shortener which was supposed to be part of the GTA RP server. Notably, P3 is the sole participant who utilized Copilot daily for all non-work-related projects. This session clocked in at 1 hour and 29 minutes.

Session 4

Another novice programmer, participant P4, volunteered her time and expertise to develop a web scraper for an NGO assisting prisoners. P4 is a computer science graduate currently working as a data analyst. Even though she possessed a foundational understanding of her goal and core programming principles, similar to P1, she heavily relied on natural language interaction with Copilot during the entire development process. Lasting 2 hours and 3 minutes, this session exhibited multiple similarities to Session 1.

5. Analysis of the Programming Sessions

Due to limitations of the MAXQDA24 (a software program for qualitative data analysis) provided by the Free University of Berlin for students, which lacks automatic transcription functionality, the audio recordings were transcribed using YouTube's automatic transcription feature. The resulting transcripts and corresponding videos were then imported into MAXQDA24 for further analysis. These videos remained private and unpublished on YouTube.

5.1. Complementary Quantitative Insights

Since the coding was done with MAXQDA24, the software automatically yielded some statistics, which was helpful to identify a number of general insights. I want to emphasize that I am aware these quantitative aspects are not part of the grounded theory methodology. Nevertheless, I preferred to keep them, as the insights provide a complementary perspective on the data and enhance the richness of the overall analysis. However, it is important to acknowledge that the core focus remains on understanding the underlying processes and experiences through qualitative coding and interpretation.

Interestingly, almost all the codes I found could be grouped in the 13 categories, as shown in Table 3. See Appendix 3 for the bar chart illustrating the data presented in the Table 3.

36 initial codes like *Expressing excitement after solving the subtask, Installing/Updating tools, Starting a VPN, Restarting the IDE, Interrupted by a phone call* etc. were omitted since they were not leveraged for the analysis.

CUPS	***CUPS adjusted***	***P1***	***P4***	***P2***	***P3***
Prompt crafting	Prompt crafting	48	34	3	3
Writing Documentation	Writing Commentary	8	0	4	5
Writing New Functionality	Writing New Functionality	26	28	100+	100+
Editing Written Code	Editing Written Code	41	31	100+	100+
Editing Last Suggestion	Editing Last Suggestion	6	7	19	76
Debugging/ Testing Code	Debugging/ Testing Code	37	25	0	28
Waiting For Suggestion	** not applicable **	N/A	N/A	N/A	N/A
Thinking	Thinking (also loudly)	53	26	16	37
Verifying Suggestion	Reading the Suggestion	79	29	100+	100+
** not applicable **	Accepting Suggestion (in Editor)	6	14	100+	100+
** not applicable **	Copy-Pasting Code to/from the Chat	13	16	0	0
Deferring thought for later	** not applicable **	N/A	N/A	N/A	N/A
Not Thinking	** not applicable **	N/A	N/A	N/A	N/A
Thinking about New Code to write	Thinking about New Code to write	0	0	7	17
Looking up Documentation	Looking up Documentation/other sources	4	5	2	6
** not applicable **	Being explicitly confused	64	16	0	0

Table 3. *Frequency of Copilot User Actions in different Sessions.* ***CUPS*** *(Coding User Activity with Programming Suggestions) is the concept by Mozannar et al. [31].* ***CUPS adjusted*** *reflects adapted categories specific to this study (details in text). Please find the bar chart visual made with this data in Appendix 3.*

To define the categories, I built upon the concept of Coding User Activity with Programming Suggestions (CUPS) introduced by Mozannar et al. [31]. I adapted the CUPS categories to

better reflect the specific interactions observed in data I was working with. For example, categories like *Waiting for Suggestion* were not applicable due to current low latency with Copilot responses. Please find the categories I used in the column ***CUPS adjusted*** of Table 3.

5.2. Seasoned programmers' sessions: differences and similarities

In analyzing sessions P2 and P3, I discontinued marking (and correspondingly counting) specific actions once their frequency surpassed 100 instances within the first hour. Given the focus on qualitative analysis within grounded theory methodology, the exact number holds less significance compared to identifying the underlying tendencies.

As one can see in Table 3, participants P2 and P3 were writing and editing the overwhelming majority of the code all by themselves. They were also reading and accepting a substantial number of Copilot suggestions in the text editor. Both of them sent only a very few natural language prompts to the AI tool, and similarly, they did not utilize Copilot Chat during their sessions at all. They typed their requests as comments and received a response directly in the text editor, saving time on switching to another pane in the IDE and back.

Through the iterative process of open, axial and selective coding, constant comparison, and theoretical sampling, several common trends emerged from the analysis of seasoned programmers' sessions with GitHub Copilot.

Similarly, P2 and P3 only used natural language prompts in two scenarios:

1) Generating large code blocks: They aimed to save time by prompting Copilot to generate substantial code chunks.

```
Example 1. P2 complained that Copilot wasn't providing useful
suggestions for the current refactoring task. He then decided to
explicitly ask Copilot for help:

"Create a static class that contains the database of all characters and
objects in the game"
```

```
Example 2. To create a new function, P3 began by writing a comment
describing his desired functionality and letting Copilot generate the
entire code.

"Redirect to the original URL"
```

```
Example 3. As a potential time-saving measure, P2 wanted to check if
Copilot can finish the task instead of him.
```

```
"Skip the rest of the packet until I peek that the next 4 byte are equal to 53019"
```

2) When the programmers were unsure about the exact function, algorithm, or syntax, they employed natural language prompts.

```
Example 4. To implement a specific functionality, P3 didn't know which built-in Golang function to choose. He utilized a natural language prompt, asking Copilot to write the code for him.

"Send response code 204 with no content"
```

```
Example 5. Similarly, when P3 needed to decide on an algorithm to implement another function, he sought assistance through a natural language prompt:

"Generate random string with length n"
```

```
Example 6. Like the P3, P2 also used a natural language prompt when he needed a simple and efficient way to achieve his goal and lacked the specific syntax:

"Reverse the byte array"
```

If the AI answer was not satisfactory, the seasoned programmers did not prefer to rephrase the prompt and ask Copilot again (that was typical reaction to a disappointing Copilot's response for P4 and especially for P1), but to google the same question and get an answer on the Stack Overflow forums or on the corresponding documentation pages.

The rest of the time, P2 and P3 tended just to code themselves and accept real-time suggestions of Copilot or keep typing and therefore rejecting the AI completions.

An interesting trend emerged in how P2 and P3 interacted with Copilot's multi-line code suggestions. While maintaining a focus on writing the core logic themselves, they demonstrated a willingness to utilize Copilot's capacity to generate substantial code chunks. The programmers exhibited a behavior I would call "acceptance and refinement." They would accept these suggestions, even if not entirely perfect, as a starting point. Subsequently, they would meticulously edit the generated code to ensure it aligned with their specific requirements (sometimes even deleting the entire suggestion after a careful review).

```
Example 7. P3 made a joke about described above strategy:
```

"Deleting the code works faster than typing it in"

This "acceptance and refinement" approach likely offered a time-saving benefit compared to writing the entire multi-line block from scratch. However, it is important to acknowledge that this strategy necessitates additional effort in the form of close scrutiny and code revision. Further research is needed to quantify the exact trade-off between time saved and time spent refining Copilot's suggestions.

Also interesting was their reaction to the long suggestions: often it was very vivid, both in positive and negative ways.

Example 8. P3 was caught off guard by a suggestion. The code looked suspiciously similar to something he might have encountered earlier.

"Oh God, how does it know? I bet it's pulling data from somewhere!"

Example 9. P2 had accepted four sequential, single-line code suggestions from Copilot. However, the next suggestion offered by Copilot didn't seem to make sense within the current coding context.

"And now it lies because it's stupid. No, now we need a 'back team member' go afterwards"

In many cases, the programmers frequently vocalized their thoughts while interacting with Copilot, resembling communication with a human partner. It is unclear whether this behavior stemmed from my request to "think aloud" during the sessions or if it reflected their admitted habit of talking to their development environment (including Copilot) throughout the coding process (as P2 and P3 mentioned in their post-session interviews; this was also a case for P1; more about this in the next subsection).

Example 10. P3 manually typed in a beginning of the function definition.

"So, what would you suggest?"

Example 11. P2 entered the first line of the class definition.

"Any ideas?"

Both programmers P2 and P3 also spent a lot of time verifying the shorter code snippets generated by Copilot. This was noticeable, thanks to the multiple factors: time spent on reading the suggestions, their “think aloud” comments, deleting or editing of the accepted code snippets. In the interviews after the session, P2 and P3, independently of each other as well, emphasized the need to be constantly aware and check literally every accepted

suggestion as a drawback of Copilot, since it still can be a good-looking piece of code, the running of which can cause errors later on.

Exactly this behavior makes the most significant difference between the sessions of seasoned programmers P2 and P3 and the novice programmers P1 and P4 sessions.

5.3. Novice programmers' sessions: differences and similarities

As shown in Table 3, the most frequent action during the programming session with P1 and P4 was reading Copilot's suggestions (in the Copilot Chat), and – strongly connected to the reading – *thinking* (mostly about general design, its implementation, and the particular code lines/functions). Interestingly, for P1, the number of prompts sent to the Copilot Chat during the session was almost twice less than the *reading* cases, since the programmer could not always understand Copilot's responses thoroughly, and he had to read them again and again, often after a small action in between (thinking, editing the code, testing, etc.).

For P4, the reading time for Copilot's suggestions was remarkably brief, raising concerns about thorough code review (4 seconds for 8 lines of code and two paragraphs of Copilot's natural language explanations; 4 seconds for 15 lines of code and two paragraphs of natural language text; 3 seconds for 17 lines of code and two short natural language paragraphs; and remarkably 5 seconds of reading for 60 lines of the refactored code). The brief reading times of Copilot's suggestions indicate that *at least some code* might have been accepted without proper assessment by P4. She tried to compensate for not reading the accepted code by testing it, reading the output in the terminal, and then checking the generated code snippets. This strategy worked in some cases, but it also led to a dead end as soon as the project's complexity grew to a certain level.

In the case of P1, he often accepted Copilot's suggestions without thoroughly understanding what exactly the generated code did. This led, as with P4, to debugging challenges later.

Another similarity between P1 and P4's sessions was their occasional confusion regarding Copilot's suggestions. This was particularly evident with P1, whose background lies outside computer science. This lack of formal training sometimes led to misunderstandings of even fundamental concepts, like data types.

```
Example 12. While trying to understand why the client received data in
an unexpected format, P1 verbalized his thought:

"No, I'm not sending a dictionary from the server, it's a list"
```

The piece of code he was talking about actually was:

```
{'data': [name, email, phone, location]}
```

In other words, P1 correctly interpreted that the part `[name, email, phone, location]` was a list, but ignored the rest of the code, completely missing that the list was within a dictionary.

Due to the striking nature of P1's explicit verbal confusions which demonstrated misunderstandings, relevant excerpts have been provided in Appendix 4. While some quotes highlight P1's knowledge gaps, others amusingly illustrate the relatable human experience of an aspiring programmer learning to code with an AI pair programmer, which is why those particular examples were included.

In contrast to P1's occasional confusion, this was not the case with P2 and P3, the experienced programmers, at all.

```
Example 13. P3 received an inaccurate suggestion from Copilot.

"Why? I don't need this!"
```

```
Example 14. P2 deemed a multi-line suggestion from Copilot to be
entirely inappropriate for the situation.

"Yeah, now it's fantasizing again. Totally bizarre stuff! Uh!"
```

With other words, seasoned programmers did not show any confusion. This suggests a strong grasp of fundamental programming concepts, allowing them to quickly identify and reject unsuitable suggestions.

Both P1 and P4 exhibited a pattern of not actively driving the development process during the initial stages of their sessions. P1 spent a significant amount of time in the planning phase, relying on Copilot for both design discussions and code generation. Similarly, P4 initially adopted a "navigator" role, letting Copilot take the lead through natural language prompts and code generation for over an hour. This behavior suggests a tendency for both participants to view Copilot as a primary source of direction and code creation, potentially hindering their own initiative and decision-making during the development.

The most significant difference between the sessions with P1 and P4 was their approach during the sessions. Even if both of the participants relied on natural language prompts in Copilot Chat, P1 in fact opted for a so-called “big bang” approach: P1 built the entire functionality at once before initiating any testing.

The writing of the code took one hour and 8 minutes and 27 prompts P1 sent to Copilot Chat during this time. Subsequently, he spent the rest of the session trying to make the code run. He was interrupted by me without reaching the initial goal, since the programmer had been debugging a rabbit hole for more than an hour without any visible chances for improvement and without understanding where the actual problem was.

Unlike P1, P4 adopted a classic iterative development approach. She created small, testable functionalities, ran them in terminal, debugged them as needed, and then moved on to the next small piece of code.

It is important to elaborate on *how* exactly she created the code. In the beginning, P4 primarily acted as a navigator, letting Copilot take the lead (driver role). Remarkably, she did not write any new Python code herself for the first hour and six minutes! Even for minor tasks, unquestionably faster to complete manually, P4 chose natural language communication.

```
Example 15. Instead of a simple and quick manual cut-and-paste
operation, P4 decided to instruct Copilot to move constant strings to
another place in the file through a natural language prompt.

"Move the constants from the function to the top of the file"
```

As shown, P4 heavily relied on natural language communication in Copilot Chat during the initial phase, sending 27 out of 34 prompts for this session.

```
Example 16. P4 started with a general task.

"Write a code which visits a website, goes to all subpages, and prints
its content"
```

```
Examples 17-19. Following the initial generation of the web scraper
code, P4 continued to employ natural language prompts to iteratively
refine the scraper's functionalities.

"Add function to not visit the same subpage twice"
```

```
"Instead of printing the content of the subpages, print the list of
subpages"
```

```
"For each subpage found here, find subpages which link text contains
'organization'. Ignore the case"
```

After each response from Copilot in the chat, P4 would test the generated code and continue building upon it with new prompts.

Despite the differences in the coding and testing approach, novice programmers P4 and P1 availed themselves of the support of Copilot in a very similar way.

If I were to use the “classical” definition of pair programming based on driver-navigator relationship, I would certainly say that both P1 and P4 were “Non-Drivers” for at least a

significant part of their programming sessions, even in the moments when they had to write code, make decisions or choose a way to go. I honestly did not expect this kind of behavior during a human-AI programming session.

I would rather characterize the roles' distribution during the session as a *novice learner* who fully relied on the help of an experienced, fluent-in-programming *mentor*. Notably, P1 did not properly consider the fact that this mentor was not a human, but a large language model susceptible to occasional inaccuracies in its code prediction capabilities.

The participant's interaction with GitHub Copilot closely mirrored that of engaging with a human collaborator.

Examples 20-21. P1's used natural language prompts that resembled human conversation to seek guidance from Copilot:

"Do I still need this piece of code?"

"Well, what if I just disregard the 400 errors?"

Example 22. After completing the code, P1 paused before initiating the tests. He then queried Copilot to validate whether the task was finished. This behavior could indicate that P1, to some extent, viewed Copilot as capable of making decisions about task completion, not just providing suggestions.

"Do I have anything left to do?"

Examples 23-24. P1 occasionally engaged in dialogues that resembled conversation with another human (rather than LLM) or probed the AI's perception.

"Great! Uh, apparently you are capable of helping human beings! Wow! Now keep on working!"

"Do you actually see all the code, uh, inside my folder?"

These instances reflect an anthropomorphization of Copilot, suggesting a perceived fluidity between AI and human roles in the coding process.

In contrast to P1's human-like interaction with Copilot, P4's communication style was markedly different. She refrained from anthropomorphization, focusing on concrete requests and instructions.

Examples 25-26. A lot of P4’s prompts were short and direct, in the same vein as Google searches or voice assistant requests.

"Refactor this function, break it into smaller ones"

"Add a function which checks if the tag is present in the page content"

Nevertheless, in the post-session interview, P4 noticed that at least Copilot's answers in Chat were imitating human communication.

Example 27. P4 expressed her amusement with Copilot's behavior.

"It's always a bit funny, like it says uh 'You're correct!' in the beginning, just trying to say something nice to me"

Similarly to P1, P4's approach resembled working with a highly-skilled AI programming mentor who could potentially complete given tasks independently. This also highlights the flexibility of user interaction with Copilot, catering to both those seeking a human-like collaborative partner and those preferring a more directive approach.

In conclusion, despite their contrasting communication styles, both P1 and P4 exhibited a similar tendency to over-rely on Copilot during their programming sessions. This was evident in their initial reluctance to take control of the development process and their limited code review practices. P1's anthropomorphization of Copilot and P4's focus on concrete tasks both highlight the flexibility of user interaction with the tool, catering to different preferences in collaborative styles. However, their approach also reveals a potential vulnerability to errors and a hindrance to the development of crucial programming skills.

The analysis of these novice programmers' sessions underscores the importance of striking a balance between leveraging the benefits of AI-assisted tools like GitHub Copilot and fostering the growth of essential programming skills. While Copilot can provide valuable support and guidance, an over-reliance on the tool may impede the development of code comprehension, independent problem-solving, and critical code review practices. As AI-assisted programming tools become increasingly prevalent, it is crucial for novice programmers to cultivate these fundamental skills alongside the efficient use of AI assistance.

6. Discussion

The analysis of programmer interactions with GitHub Copilot in this study aligns with the "acceleration" and "exploration" modes identified by Barke et al. [47]. Seasoned programmers (P2 and P3) primarily used Copilot for acceleration. This can be attributed to the nature of their tasks, which involved familiar programming languages and well-defined goals. They focused on leveraging Copilot's ability to generate small, correct code snippets to expedite development.

On the other hand, novice programmers (P1 and P4) preliminarily exhibited exploration behavior. Faced with complex (for them) tasks, they used Copilot to design discussions, explore functionalities, and seek guidance on algorithms and implementation. This highlights a potential limitation of the novice/seasoned programmer categorization I used in this thesis. It worked well for data I had for my analysis. However, it also indicates that task complexity, rather than pure programmer experience, might be a stronger factor influencing the choice between acceleration and exploration modes.

Conversely, this bachelor’s thesis serves as an independent validation of these modes described by Barke et al., and further emphasizes the potential influence of task complexity on user behavior. Future research could explore this aspect in more detail, investigating how programmers adapt their interaction styles based on both experience and task characteristics.

The findings of my study provide valuable insights into the dynamics of pair programming with AI-powered coding assistants like GitHub Copilot. The analysis throughout this project reveals a complex interplay between human expertise and AI-generated suggestions, highlighting both the benefits and challenges that arise from this collaboration.

The results align with existing literature on the impact of AI on software development practices. Previous research has emphasized the potential benefits of AI-assisted programming, such as increased productivity, improved code quality, and automation of repetitive tasks (e.g., Sarkar et al. [40]). My study corroborates these findings, as participants acknowledged the efficiency gains and helpful suggestions provided by Copilot. However, the tendency of novice programmers to over-rely on Copilot's suggestions, as observed in the cases of P1 and P4, resonates with concerns raised about the potential negative impact of AI on programming skill development (Dakhel et al. [54, p. 25]). The limited understanding of generated code and the potential for skill stagnation underscore the need for a balanced approach to AI integration in software development.

Interestingly, the observed behaviors in this thesis study also align with the concepts of “shepherding” and “drifting” introduced by Prater et al. [48, pp. 10-11]. Similar to “shepherding”, novice programmers in our study exhibited a tendency to spend a significant amount of time manipulating Copilot's suggestions to achieve their desired outcomes, rather than independently driving the development process. Furthermore, this thesis indings echo the idea of 'drifting,' as participants moved from one suggestion to another, often without a clear direction. This reinforces the notion that novice programmers might struggle to

effectively integrate Copilot into their workflow, potentially hindering the development of crucial programming skills.

The findings of this study have important implications for the use of LLMs in pair programming and software development in general. While AI-powered coding assistants like GitHub Copilot offer significant benefits, it is crucial to recognize their limitations and the potential drawbacks of over-reliance on AI suggestions. The experiences of the participants in this study emphasize the importance of human oversight, critical thinking, and continuous learning in the context of AI-assisted programming.

To ensure the effective integration of AI in pair programming, it is essential to develop strategies that promote a balanced collaboration between human programmers and AI-powered coding assistants. This may involve training programmers to critically evaluate AI suggestions, encouraging them to actively engage in code comprehension and problem-solving, and fostering a culture of continuous learning and skill development. Additionally, the design of AI-powered coding assistants should prioritize transparency and explainability, enabling programmers to understand the reasoning behind the generated code and make informed decisions.

The limitations of my study provide avenues for future research. The small sample size limits the generalizability of the findings. Future studies could explore the experiences of programmers with different AI-powered tools and across a wider range of programming languages and development environments. Additionally, longitudinal studies could investigate the long-term impact of AI-assisted pair programming on the skills and knowledge acquisition of programmers, providing insights into the potential implications for the programming profession.

Additionally, future research could delve into the development of pedagogical approaches and training programs that effectively integrate AI-powered coding assistants into the learning process. By designing educational interventions that promote a critical and reflective engagement with AI suggestions, researchers can contribute to the development of best practices for AI-assisted programming education.

In conclusion, this study sheds light on the intricate dynamics of integrating AI coding assistants into pair programming practices. While AI tools like Copilot can provide tangible benefits, the findings accentuate the need for judicious integration strategies that balance AI assistance with human agency. As the software development landscape continues to evolve with AI, it is crucial to prioritize research that fosters synergistic human-AI collaboration models. These models should emphasize developing critical thinking skills, promoting code comprehension, and cultivating a culture of continuous learning among programmers. By harmonizing AI capabilities with human expertise, the programming profession can harness the potential of AI while ensuring its long-term sustainability and growth.

7. Conclusion

This bachelor’s thesis has investigated the dynamics of pair programming with AI-powered coding assistants, specifically focusing on GitHub Copilot. By analyzing the interactions of both professional and novice programmers, the study sheds light on the intricate relationship between human expertise and AI-generated suggestions.

The findings highlight the potential of AI assistants to enhance programmer productivity and expedite development processes. However, the research also underscores the importance of critical human oversight to mitigate potential drawbacks like over-reliance and skill stagnation.

Moving forward, the programming community should strive for a balanced integration of AI tools. This can be achieved through educational initiatives that equip programmers to critically evaluate AI suggestions and prioritize active code comprehension. Additionally, the design of AI-powered assistants might emphasize explainability, allowing programmers to understand the reasoning behind generated code.

Beyond that, AI tools could incorporate user-customizable settings that tailor the level of assistance to the programmer's experience and understanding of the problem at hand. By considering these factors, AI assistants can provide more targeted suggestions that are appropriate for the programmer's skill level.

By fostering a culture of critical engagement with AI and promoting continuous learning, the programming profession can harness the power of these tools while safeguarding the essential skills and expertise that define human programmers. This ensures sustainable growth within the field, as AI continues to reshape the landscape of software development.

References

[1] F. Zieris and L. Prechelt, “Explaining pair programming session dynamics from knowledge gaps,” in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, Jun. 2020. Accessed: Apr. 28, 2024. [Online]. Available: http://dx.doi.org/10.1145/3377811.3380925

[2] A. Cockburn and L. Williams, “The Costs and Benefits of Pair Programming,” presented at the 1st International Conference on eXtreme Programming and Flexible Processes in Software Engineering, Sardinia, Italy, Feb. 2000. Accessed: Apr. 28, 2024. [Online]. Available: https://www.researchgate.net/publication/2333697_The_Costs_and_Benefits_of_Pair_Programming/references

[3] A. Begel and N. Nagappan, “Pair programming: What’s in it for Me?,” in *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, Oct. 2008. Accessed: Apr. 28, 2024. [Online]. Available: http://dx.doi.org/10.1145/1414004.1414026

[4] A. Bolboacă, *Practical Remote Pair Programming: Best practices, tips, and techniques for collaborating productively with distributed development teams*. Packt Publishing Ltd, 2021.

[5] S. Salinger, F. Zieris, and L. Prechelt, “Liberating pair programming research from the oppressive driver/observer regime,” in *2013 35th International Conference on Software Engineering (ICSE)*, May 2013. Accessed: Apr. 28, 2024. [Online]. Available: http://dx.doi.org/10.1109/icse.2013.6606678

[6] S. A. Fincher and A. V. Robins, *The Cambridge Handbook of Computing Education Research*. Cambridge University Press, 2019. Accessed: Apr. 28, 2024. [Online]. Available: https://www.cambridge.org/core/books/cambridge-handbook-of-computing-education-research/F8CFAF7B81A8F6BF5C663412BA0A943D

[7] K. Beck and C. Andres, *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, 2004.

[8] C. McDowell, L. Werner, H. E. Bullock, and J. Fernald, “Pair programming improves student retention, confidence, and program quality,” *Communications of the ACM*, vol. 49, no. 8, pp. 90–95, Aug. 2006, doi: 10.1145/1145287.1145293.

[9] C. Rich and R. C. Waters, *The Programmer’s Apprentice*. Association for Computing Machinery (ACM), 1990. Accessed: Apr. 28, 2024. [Online]. Available: https://doi.org/10.7551/mitpress/1425.001.0001

[10] A. Vaswani *et al.*, “Attention Is All You Need,” *arXiv.org*, Jun. 12, 2017. https://arxiv.org/abs/1706.03762

[11] T. Eloundou, S. Manning, P. Mishkin, and D. Rock, “GPTs are GPTs: An Early Look at the Labor Market Impact Potential of Large Language Models,” *arXiv.org*, Mar. 17, 2023. https://arxiv.org/abs/2303.10130

[12] Y. Mehdi, “Reinventing search with a new AI-powered Microsoft Bing and Edge, your copilot for the web,” *The Official Microsoft Blog*, Feb. 07, 2023. https://blogs.microsoft.com/blog/2023/02/07/reinventing-search-with-a-new-ai-powered-microsoft-bing-and-edge-your-copilot-for-the-web/ (accessed Apr. 28, 2024).

[13] Microsoft, “How Copilot works, technically speaking.” Nov. 21, 2023. Accessed: Apr. 28, 2024. [Online]. Available: https://www.microsoft.com/en-us/bing/do-more-with-ai/how-bing-chat-works?form=MA13KP

[14] J. Ribas, “Building the New Bing,” Feb. 21, 2023. https://www.linkedin.com/pulse/building-new-bing-jordi-ribas/ (accessed Apr. 28, 2024).

[15] S. Pichai, “An important next step on our AI journey,” *Google*, Feb. 06, 2023. https://blog.google/technology/ai/bard-google-ai-search-updates/ (accessed Apr. 28, 2024).

[16] E. Collins, “LaMDA: our breakthrough conversation technology,” *Google*, May 18, 2021. https://blog.google/technology/ai/lamda/ (accessed Apr. 28, 2024).

[17] S. Pichai, “Our next-generation model: Gemini 1.5,” *Google*, Feb. 15, 2024. https://blog.google/technology/ai/google-gemini-next-generation-model-february-2024/#architecture (accessed Apr. 28, 2024).

[18] Meta AI, “Code Llama: Open Foundation Models for Code,” *Meta*, Jan. 31, 2024. https://arxiv.org/html/2308.12950v3 (accessed Apr. 28, 2024).

[19] Y. Wang, H. Le, A. D. Gotmare, N. D. Q. Bui, J. Li, and S. C. H. Hoi, “CodeT5+: Open Code Large Language Models for Code Understanding and Generation,” *arXiv.org*, May 13, 2023. https://arxiv.org/abs/2305.07922

[20] Replit, “Replit — Ghostwriter AI & Complete Code Beta,” *Replit Blog*, Sep. 22, 2022. https://blog.replit.com/ai (accessed Apr. 28, 2024).

[21] Assistiv AI LTDA, “Codi Model Management,” *Your AI coding assistant*, 2024. https://askcodi.com/llms/ (accessed Apr. 28, 2024).

[22] Tabnine, “How Tabnine works,” *Tabnine: Support Page*, 2023. https://support.tabnine.com/hc/en-us/articles/5407860396817-How-Tabnine-works (accessed Apr. 28, 2024).

[23] Replit, “Replit — Replit’s new AI Model now available on Hugging Face,” *Replit Blog*, Oct. 10, 2023. https://blog.replit.com/replit-code-v1_5 (accessed Apr. 28, 2024).

[24] Accubits Technologies Inc, “Polycoder,” *Accubits Technologies Inc*, Mar. 27, 2023. https://accubits.com/open-source-program-synthesis-models-leaderboard/polycoder/ (accessed Apr. 28, 2024).

[25] B. Berabi, A. Gronskiy, V. Raychev, G. Sivanrupan, V. Chibotaru, and M. Vechev, “DeepCode AI Fix: Fixing Security Vulnerabilities with Large Language Models,” *arXiv.org*, Feb. 19, 2024. https://arxiv.org/abs/2402.13291

[26] AWS, “Amazon CodeWhisperer FAQs – How does CodeWhisperer works?,” *Amazon Web Services, Inc.*, 2024. https://aws.amazon.com/codewhisperer/faqs/ (accessed Apr. 28, 2024).

[27] Anthropic, “The Claude 3 Model Family: Opus, Sonnet, Haiku,” *Anthropic Research*, Mar. 04, 2024. Accessed: Apr. 29, 2024. [Online]. Available: https://www.anthropic.com/claude-3-model-card

[28] WPAI, Inc. , “CodeWP Modes,” *CodeWP*, Oct. 20, 2023. https://codewp.ai/modes/ (accessed Apr. 29, 2024).

[29] B. McKinzie *et al.*, “MM1: Methods, Analysis & Insights from Multimodal LLM Pre-training,” *arXiv.org*, Mar. 14, 2024. https://arxiv.org/abs/2403.09611

[30] S. Huang *et al.*, “Language Is Not All You Need: Aligning Perception with Language Models,” *arXiv.org*, Feb. 27, 2023. https://arxiv.org/abs/2302.14045

[31] H. Mozannar, G. Bansal, A. Fourney, and E. Horvitz, “Reading Between the Lines: Modeling User Behavior and Costs in AI-Assisted Programming,” *arXiv.org*, Oct. 25, 2022. https://arxiv.org/abs/2210.14306

[32] S. Altman, “ChatGPT crossed 1 million users,” *Official Twitter Account*, Dec. 05, 2022. https://twitter.com/sama/status/1599668808285028353 (accessed Apr. 29, 2024).

[33] K. Hu, “ChatGPT sets record for fastest-growing user base - analyst note,” *Reuters*, Feb. 02, 2023. Accessed: Apr. 29, 2024. [Online]. Available: https://www.reuters.com/technology/chatgpt-sets-record-fastest-growing-user-base-analyst-note-2023-02-01/

[34] OpenAI, “OpenAI Codex,” Aug. 10, 2021. https://openai.com/blog/openai-codex (accessed Apr. 29, 2024).

[35] S. Zhao, “GitHub Copilot now has a better AI model and new capabilities,” *The GitHub Blog*, Feb. 14, 2023. https://github.blog/2023-02-14-github-copilot-now-has-a-better-ai-model-and-new-capabilities/ (accessed Apr. 29, 2024).

[36] T. Dohmke, “GitHub Copilot X: The AI-powered developer experience,” *The GitHub Blog*, Mar. 22, 2023. https://github.blog/2023-03-22-github-copilot-x-the-ai-powered-developer-experience/ (accessed Apr. 29, 2024).

[37] S. Zhao, “Smarter, more efficient coding: GitHub Copilot goes beyond Codex with improved AI model,” *The GitHub Blog*, Jul. 28, 2023. https://github.blog/2023-07-28-smarter-more-efficient-coding-github-copilot-goes-beyond-codex-with-improved-ai-model/ (accessed Apr. 29, 2024).

[38] “GitHub Copilot – November 30th Update,” *The GitHub Blog*, Nov. 30, 2023. https://github.blog/changelog/2023-11-30-github-copilot-november-30th-update/ (accessed Apr. 29, 2024).

[39] Tabnine Inc., “We’re Tabnine,” *Official website*, Apr. 04, 2024. https://www.tabnine.com/about (accessed Apr. 29, 2024).

[40] A. Sarkar, A. D. Gordon, C. Negreanu, C. Poelitz, S. S. Ragavan, and B. Zorn, “What is it like to program with artificial intelligence?,” *arXiv.org*, Aug. 12, 2022. https://arxiv.org/abs/2208.06213

[41] Q. Ma, T. Wu, and K. Koedinger, “Is AI the better programming partner? Human-Human Pair Programming vs. Human-AI pAIr Programming,” *arXiv.org*, Jun. 08, 2023. https://arxiv.org/abs/2306.05153

[42] Z. C. Ani, Z. A. Hamid, and N. N. Zhamri, “The Recent Trends of Research on GitHub Copilot: A Systematic Review,” *Springer Nature Singapore*, Jan. 01, 2024. https://link.springer.com/chapter/10.1007/978-981-99-9589-9_27

[43] C. Bird *et al.*, “Taking Flight with Copilot,” *Queue*, vol. 20, no. 6, pp. 35–57, Dec. 2022, doi: 10.1145/3582083.

[44] OpenAI, “OpenAI and Microsoft extend partnership,” Jan. 23, 2023.

https://openai.com/blog/openai-and-microsoft-extend-partnership (accessed Apr. 29, 2024).
[45] S. Imai, “Is GitHub copilot a substitute for human pair-programming?,” in *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, May 2022. Accessed: Apr. 29, 2024. [Online]. Available: http://dx.doi.org/10.1145/3510454.3522684
[46] P. Vaithilingam, T. Zhang, and E. L. Glassman, “Expectation vs. Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models,” in *CHI Conference on Human Factors in Computing Systems Extended Abstracts*, Apr. 2022. Accessed: Apr. 29, 2024. [Online]. Available: http://dx.doi.org/10.1145/3491101.3519665
[47] S. Barke, M. B. James, and N. Polikarpova, “Grounded Copilot: How Programmers Interact with Code-Generating Models,” *arXiv.org*, Jun. 30, 2022. https://arxiv.org/abs/2206.15000
[48] J. Prather *et al.*, “‘It’s Weird That it Knows What I Want’: Usability and Interactions with Copilot for Novice Programmers,” *arXiv.org*, Apr. 05, 2023. https://arxiv.org/abs/2304.02491
[49] G. Foley and V. Timonen, “Using Grounded Theory Method to Capture and Analyze Health Care Experiences,” *Health Services Research*, vol. 50, no. 4, pp. 1195–1210, Dec. 2014, doi: 10.1111/1475-6773.12275.
[50] K.-J. Stol, P. Ralph, and B. Fitzgerald, “Grounded theory in software engineering research,” in *Proceedings of the 38th International Conference on Software Engineering*, May 2016. Accessed: Apr. 29, 2024. [Online]. Available: http://dx.doi.org/10.1145/2884781.2884833
[51] B. G. Glaser and A. L. Strauss, *Discovery of Grounded Theory: Strategies for Qualitative Research*. Routledge, 2017.
[52] A. L. Strauss and J. M. Corbin, *Grounded Theory in Practice*. SAGE, 1997.
[53] Microsoft, “Visual Studio Code,” *Microsoft*, Nov. 03, 2021. https://code.visualstudio.com/ (accessed Apr. 29, 2024).
[54] A. M. Dakhel *et al.*, “GitHub Copilot AI pair programmer: Asset or Liability?,” *arXiv.org*, Jun. 30, 2022. https://arxiv.org/abs/2206.15331

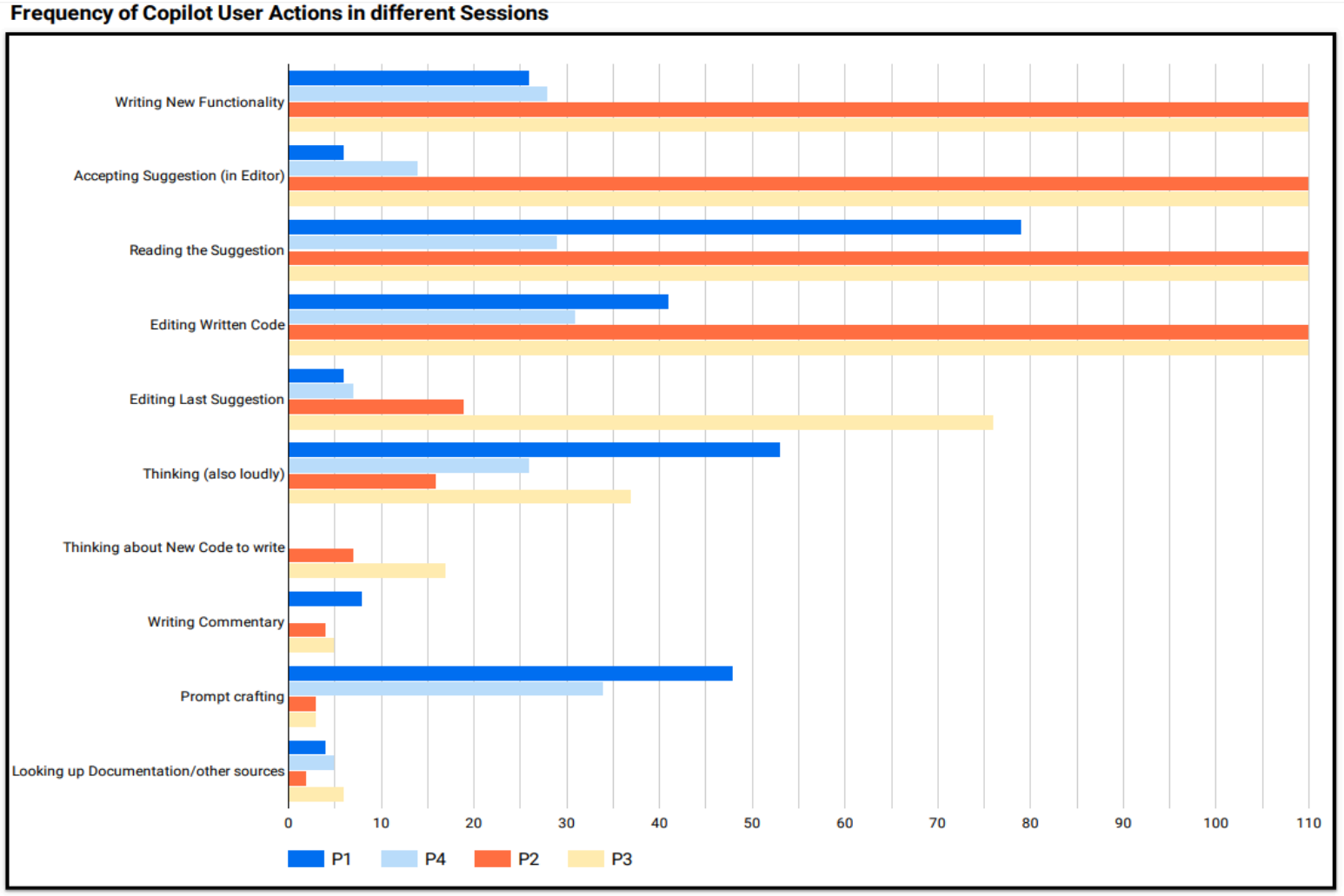
Appendix 1. Pre-Session Questionnaire

- Information about the developer:
    - Experience as a software developer in years?
    - Experience with AI tools? How often have you worked with them before?
- What project are you working on today?
    - Hobby or Commercial?
    - Size of the project (rough estimate in lines of code, number of classes or similar)
    - How many people are working on it?
    - How long have you been working on the project?
- What is the task you want to work on?
    - Description of the objective for this session?
    - What difficulties do you expect or what is still unclear?
    - How do you think an AI can help you with that?
    - What can the AI not help with?

Appendix 2. Post-Session Questionnaire

- Summary of the session:
    - How would you divide the session into phases (e.g., 1st Phase: High-Level Design, 2nd Phase: Concrete implementation of method X, 3rd Phase: Reading documentation, 4th Phase: Debugging)
    - What were the essential decisions to be made or the essential findings to move forward?
    - Looking back, would you have done anything differently?
    - Where did the AI help you?
    - Where did the AI surprise you (positively / negatively)?
    - Where did the AI not help you or even hinder you (e.g., by a very poor suggestion)?
    - Overall, was working with the AI an enrichment for the process or an obstacle?

Appendix 3. Frequency of Copilot User Actions in different Sessions (Chart)



Visual analog of the Table 3. *Also available as [Google Looker Studio Report](#).*

Appendix 4. Some of verbal confusion expressed by P1

"I think, the step three is not really clear"

"I don't know"

"At this point feel kind of lost"

"I don't know if it will change anything"

"I don't understand what it means when a client connects to the server"

"I don't understand yet"

"I think uh GitHub Copilot could confuse me"

"I feel very stupid now"

"I have no idea what they are used for"

"I have not a single clue why I need them"

"I have some sort of a confusion here"

"I still have no idea why it doesn't work"

"I'm lost"

"Now I cann't get it"

"I don't understand what this all is about"

```
"Here I really feel very lost because I don't understand what it all does"
```


```
"Here I really feel very lost because I don't understand what it all does"
```


```
"I don't know, I don't understand now"
```

```
"It's clear that it's not clear"
```

```
"Maybe I am having a confusion because I never hosted a website"
```

```
"No, I don't know"
```

```
"No, I still don't really understand"
```

```
"Okay, clear, that is not clear"
```

```
"Okay, great news. Why is that?"
```

```
"Okay, what does Cloudflare have to do with this? I have not a single clue"
```

```
"What do I need to check? Where do I need to look? I don't know"
```

```
"What the heck SocketIO application is? I don't know"
```