Tampere University

Aide Barniskyte

GITHUB COPILOT DEVELOPER EXPERIENCE: A STUDY COMPARING NOVICE AND EXPERIENCED DEVELOPERS' PERSPECTIVES

Faculty of Information Technology and Communication Sciences
M. Sc. Thesis
November 2025

ABSTRACT

Barniskyte Aide: Github Copilot Developer Experience: A Study Comparing Novice And Experienced Developers' Perspectives
M.Sc. Thesis
Tampere University
Master's Degree Programme in Computer Sciences
November 2025

With the ever-growing role that AI coding tools play in software development, evaluating the developer experience of using AI-based coding assistants becomes more important than ever before. So far, little attention has been paid to how the developer experience with AI coding assistants may differ among less experienced and more experienced developers. This thesis focuses on GitHub Copilot, a popular AI coding assistant, and investigates the differences in the developer experience among novice and experienced developers and asks whether or not the AI assistant should take into account the developer's programming experience level. It also explores how the usage of acceleration and exploration modes differ among novice and experienced developers. In the Related Work chapter, existing literature on AI coding assistants and developer experience is reviewed, covering usability, interaction modes, educational impact, and cognitive load. Acceleration and exploration modes are introduced, and the definitions of developer experience are discussed. The Methodology chapter describes the research methods used in the thesis – the online developer survey and developer user study – and justifies their choice. This chapter details online survey distribution, the questions asked, use study participant selection, and the programming tasks given to the participants. The Results chapter presents findings from both the online survey and the user study. Novices used Copilot primarily for debugging and learning, while experienced developers applied it more broadly, however, they were also more critical of context-awareness and security. Exploration mode was more prevalent than acceleration mode among both novices and experienced developers, and experienced developers were more likely to enter acceleration mode. The Discussion chapter interprets the results in relation to the research questions. It concludes that both novices and experienced developers wished to remain in control and were unlikely to delegate all the work to GitHub Copilot. Moreover, the importance of improved onboarding for AI coding assistants is emphasized. A "learning mode" is proposed for AI coding assistants to adapt to developers' familiarity with tools and frameworks, enhancing onboarding and personalization. The Conclusions chapter presents final reflections, threats to validity (such as small sample sizes) and future research directions (such as different programming languages and frameworks).

Key words and terms: Coding assistant, developer experience, user experience, GitHub Copilot, artificial intelligence.

USE OF AI IN THESIS

I have utilised AI tools in my thesis:

☐ No
☒ Yes

The AI tools utilised in my thesis and their purposes are described below:

Names and versions of AI tools: GitHub Copilot, Perplexity, Microsoft Copilot

Purpose of using AI tools: GitHub Copilot was used as the study object during the user study to evaluate their user experience. Perplexity was used as an aid in summarizing the overall results of the survey and drawing conclusions from them, as well as an aid in paraphrasing and summarizing excerpts from scientific papers (for example, summarizing a bullet-point list into 2-3 sentences). Microsoft Copilot was used to help make some parts of the text more concise.

Sections where AI tools were used: Related Work, Methodology, Results

I acknowledge that I am fully responsible for the entire content of my thesis, including the parts generated by AI, and accept accountability for any violations of ethical standards in publications.

Contents

1. Introduction

1.1. Background

In recent years, technological advancements in large language models (LLMs) have prompted a considerable spike of interest in utilizing artificial intelligence (AI) tools for product development and task completion in various industries, including software development. For example, many software developers are turning to AI-powered coding assistants to increase their productivity. According to the survey on AI in software development conducted by GitHub, over 97% of software developers and other professionals working in the software development field have used AI-based coding tools at work [1]. Aside from increased productivity, which allows software development teams to "design systems, collaborate more and meet customer requirements better" [1], the survey also indicates other benefits of AI-based coding tools, such as improved software security, increased code quality, faster test case creation and improved test coverage and faster adoption of programming languages [1]. Moreover, over a third of developers claim that their company is supportive of and "actively promoting" the use of AI-based coding tools [1]. These findings strongly suggest that AI-based coding tools will be continue to be increasingly more present. Therefore, research evaluating the usability and broader user experience (UX) of AI-based coding tools, including AI-powered coding assistants, is crucial to understanding their impact on developers' work experience and satisfaction. Existing research (e.g. Barke et al [2]) has explored how users interact with coding assistants but not always specifically among software developers or software developers with different experience levels, with one paper identifying two modes of interaction with a coding assistant: acceleration mode (when the developer already knows what to do and uses the assistant to write the code faster) and exploration mode (when the developer does not know what to do and uses the assistant for guidance) [2].

1.2. Research Goals and Questions

This paper aims to address this question by exploring how the developer experience of GitHub Copilot, one of the most well-known AI-powered coding assistants, differs among novice and experienced developers and provide insights into future improvements to AI-based coding tools. To achieve this, the paper focuses on the following research questions:

1. How does the developer experience with GitHub Copilot differ between novice and experienced developers?
2. Should coding assistants take into account the user's programming experience level? If yes, how?

3. Are there any noteworthy differences in how the acceleration and exploration modes are used based on the developers' experience level?

1.3. Research Approach and Methodology

The work will be conducted following the empirical research approach, combining both qualitative and quantitative methods. The data collection for qualitative data analysis will be conducted in the form of a user study where participants will be given software development tasks to complete using GitHub Copilot. The participants will be professional software developers, and the number of novice and experienced developers will be approximately the same. In addition to qualitative data obtained by the user study, to provide additional context and perspectives, quantitative data will be collected by an online survey on the use of GitHub Copilot for professional software developers. Developers with 5 years of experience or less will be classified as novices because this period in a developer's career is often seen by the practitioners as formative, when foundational skills are developed and the transition to a more higher-level and independent problem-solving work takes place.

1.4. Structure and Contents of the Thesis

This paper consists of four key chapters which provide a comprehensive view of current research into the UX of coding assistants and developer experience, as well as seek to answer the research questions outlined above. The Related Work chapter provides an overview of relevant research into the user experience of coding assistants and developer experience, identifying key findings, research directions and knowledge gaps. This chapter provides the scientific basis for the thesis work and positions the thesis within the broader scientific context. The Methodology chapter describes the user study process and the online survey in detail, outlining the research design. The Results chapter analyzes the obtained qualitative and quantitative data, highlighting the key findings emerging from the conducted research and presenting a detailed interpretation of the collected data. Finally, the Conclusions chapter addresses the research questions in relation to the findings and discusses possible future research and development work.

2. Related Work

This chapter presents an overview of relevant research into the topics of user experience of coding assistants (also referred to as AI coding assistant tools, or ACATs) and developer experience. It covers research on topics such as ACAT usability and interaction modes, tool evaluation and improvement, educational impact and learnability, developer experience frameworks, flow state and motivation, and cognitive load and onboarding.

2.1. User Experience of Coding Assistants

2.1.1. Usability and Interaction Modes

A paper by Tan et al [3] explored the "effectiveness and usability" of three different "AI coding assistant tools (ACATs)" by conducting user study that mimicked real software development tasks with computer science students as participants. The study found that AI coding assistants improve task completion times and code quality, especially for novices. However, experienced developers may not benefit as much in terms of speed. Aside from code completion, 44% of participants have also been found to use other features of ACATs, such as using natural language prompts to generate code, debugging, searching the ACAT's knowledge base for information and generating test units. Most participants reported perceiving programming while using an ACAT as less difficult, requiring less external help and having a better developer experience. 88-97% of participants perceived the syntax correctness, similarity to correct code and readability of the generated code as its most important features. The functional challenges faced by the participants when using ACATs include poor performance on complex logic tasks, limited natural language processing capabilities and lack of multi-type completion (f.ex. suggesting relevant packages and APIs aside from providing basic code completions). Non-functional challenges, such as slow response time and user interface (UI) issues such as inconvenient switching between suggestions, hard-to-notice auxiliary functions and disappearing suggestions have also been reported. The expectations for ACATs expressed by developers in the study demonstrate that developers view ACATs as not just merely code completion tools but rather as multifunctional programming assistants. Overall, the study points to ACATs being helpful to developers beyond enhancing productivity by making programming less exhausting. The study also suggests the importance of tailoring assistant features to user experience level. [3]

Barke et al. identified two interaction modes with Copilot: acceleration and exploration [2]. Acceleration mode is entered when developers are working on familiar tasks, allowing fast, uninterrupted coding that boosts productivity and allows faster task completion. Exploration mode is more common when developers face unfamiliar tasks and trust the assistant to help them achieve their goals and involves slower, more deliberate

interactions and more extensive suggested code validation. Exploration mode also features more natural language prompts and even validation with external sources. Developers are more likely to accept a suggestion that is mostly correct and requires little modification while in exploration mode rather than acceleration mode. During acceleration mode, developers prefer short suggestions focused on a single logical unit, such an end-of-line suggestion or a single "for" loop and tend to reject long suggestions for entire sequences. Conversely, long suggestions from Copilot while in acceleration mode are distracting for developers, sometimes to the point where the developers disable Copilot entirely. Aside from the interaction modes, Barke et al also provide suggestions on how to improve GitHub Copilot's developer experience, including things like allowing the developer to have more explicit control over the context that is taken into account by GitHub Copilot, fine-tuning Copilot to perform the cross-language translation task, Copilot being aware of the current interaction mode and acting accordingly, improved UI for the multi-suggestion pane, generating only high-level control structures and thus leaving holes for the developer to implement themselves, and better validation for the generated code. [2]

Mozannar et al. introduced the CodeRec User Programming States (CUPS) framework to analyze developer interactions with Copilot [4]. To study developer interactions with GitHub Copilot, Mozannar et al conducted a user study where participants first completed the programming tasks and then retroactively labelled their activities in the CUPS taxonomy while watching the video playback. The study found that the CUPS stages that are unique to ACATs, such as "Waiting for Suggestion" or "Verifying Suggestion" take up 51,5% of the programming session's time on average. Moreover, the suggestion acceptance rate among more experienced developers (those with 6 years of experience or more) was around 7% lower than among more novice developers. Meanwhile, the acceptance rate for those who have used GitHub Copilot before was around 8% higher than among those who have not used it before. However, due to the small number of participants, it was not determined whether said differences were caused by the years of experience or familiarity with the ACAT. The study showed that developers are likely to verify the ACAT's output, either right after accepting or later during the task. The post-study survey revealed that while developers are concerned about the quality of generated code, they also feel like Copilot helps them stay in the flow and spend less time searching for information. Concerns about wrong or outdated suggestions have also been raised. To better assist developers, Mozannar et al suggest identifying the current CUPS state and adjust the displaying of code suggestions accordingly. [4]

Vaithilingam et al. found that poor UI design limits discoverability of ACAT features, such as code changes [5]. According to the authors, the code change suggestions went unnoticed by developers due to the user interface. To improve the user experience and

discoverability of the code change feature for Visual Studio Intellicode the authors performed a design exploration study for various forms of code changes. Based on the results of the study, the authors have launched a new version of Intellicode with two of the improved interfaces, which lead to "a significant increase in usage of the corresponding tools" [5]. Moreover, the study outlines five design principles for inline code suggestions: ensuring visibility (glanceable suggestions), juxtaposing existing and proposed code for clarity, leveraging familiar interface elements to reduce cognitive load, providing sufficient visibility for validation before commitment, and allowing users to snooze suggestions to avoid interruptions. The last principle comes from the observation that some participants may find the inline suggestions intrusive and obstructive if they are coding in a more explorative manner at the given moment without the intention of refactoring or editing the code. [5]

Conclusions. The current literature is unanimous in its support for the notion that ACATs enhance usability and developer experience. Studies reviewed above agree that ACATs reduce cognitive effort, improve task completion speed, and offer valuable support through features like natural language prompting and code suggestions. However, while Tan et al. emphasize the productivity benefits for novices [3], Barke et al. [2] and Mozannar et al. [4] highlight the nuanced challenges experienced developers may face, such as distraction from long suggestions and the need for better context-awareness, especially if the behaviour of the ACAT does not match the needs of the developer's current interaction mode. Vaithilingam et al. further stress the importance of UI design in feature discoverability, which aligns with the usability concerns raised in other studies [5]. Overall, the literature is largely in agreement about the benefits of ACATs, though it diverges on how these benefits manifest across experience levels and how usability challenges should be addressed.

2.1.2. Tool Evaluation and Improvement

Miah and Zhu proposed a "user-centric approach to the testing and evaluation of LLMs as code generation tools" [6]. The authors propose a methodology where a developer tries to achieve their desired goal by completing their given task using the smallest number of prompts (attempts) possible. The testing process continues until a satisfactory solution is achieved or the maximum number of allowed attempts is reached, leading to two performance metrics: number of attempts and completion time. This allows for a user-centered evaluation as opposed to a single-dimensional assessment of the output performed by other LLM testing methodologies. However, the methodology overlooks broader UX concepts beyond usability. The user study conducted to validate the methodology also has limitations, such as having only a single participant, testing only one programming language, and using tasks from a textbook rather than real-world scenarios. Additionally, it focuses on general-purpose LLMs rather than ACATs, highlighting the

need for further research to improve its applicability for ACATs that run inside the integrated development environment (IDE). [6]

Liang et al. surveyed developers on the usability of ACATs to understand which usability aspects the ACATs struggle and succeed at [7]. Upon analyzing the results, Liang et al concluded that the most important reasons for developers to use ACATs are reducing keystrokes, finishing the tasks quicker and better recalling of syntax. The most important reasons for developers to not use ACATs are the generated code not meeting the functional or non-functional requirements and the difficulty in controlling the ACAT to generate the desired output. Additionally, the developers believe their experience with ACATs could be improved by the ability to provide feedback to the tool, better context awareness and better understanding of programming languages and application programming interfaces (APIs). The most popular improvements requested by the participants include better personalization, including ability to provide feedback to the ACAT and ability to change the ACAT's configuration settings, better context awareness, better output, explanation of the output and accounting for non-functional requirements. The study by Liang et al outlines the most important usability issues facing GitHub Copilot, as well as ACATs in general. [7]

Zhou et al. analyzed common issues developers encounter with GitHub Copilot based on the messages in StackOverflow, an online platform for developers to seek help with programming-related issues [8]. Most problems were operational issues (issues related to the usage of core Copilot functionality itself, such as code suggestions not being shown or authentication failures) and compatibility issues (the issues of compatibility between Copilot and its operating environment). Feature requests for Copilot are aimed at improving the user experience and consist of suggestions to add or improve functionalities, UI modifications, or to create a professional version that could be deployed on-premises. These requests reflect users' needs for greater flexibility, broader platform support, better interface design, and more robust, enterprise-level features. Some of the feature requests mentioned in the paper, such as Copilot taking into account the context of multiple files or the entire project, have since been implemented. Suggested content issues, referring to issues related to the code generated by Copilot, are significantly less prevalent, likely due to users being less likely to report them compared to usage-related problems. The findings show that Copilot internal errors and network connection errors are the most common causes for the issues encountered by the users. While most issues were solved by the Copilot team, the most common solutions employed by the users themselves were modifying a configuration/setting, using suitable version of Copilot or the IDE, and reinstalling/restarting/reauthorizing Copilot. Overall, the study provides insights into the user experience of GitHub Copilot both in coding-related and plugin-related ways. [8]

Contextualized AI assistants are optimized for specific tasks such as optimizing database queries or analyzing a codebase. The contextualized AI assistants also have access to "proprietary, domain-specific" knowledge, which the general-purpose AI assistants, such as GitHub Copilot, often lack [9]. Pinto et al analyze the findings of an online user study of StackSpot AI (a pseudonym), a contextualized AI coding assistant that takes into account the individual needs of the developer and the specifics of the codebase [9]. The key benefits of StackSpot identified during the study were accurate generated code that allowed for swift integration with the codebase, time efficiency achieved by centralizing information access, and easier access to documentation and guidelines within the IDE. The key challenges were multiple knowledge sources being required to achieve accurate responses, inconsistent response to identical prompts, and difficulties related to generating complex coding structures. Moreover, the study confirmed the benefits of shortcuts for experienced users, as the participants' views of the "quick commands" shifted from a "minor convenience" to "time-savers, greatly aiding in automating mundane aspects of coding and allowing them to focus on more complex tasks" [9]. An important difference between GitHub Copilot and StackSpot is that StackSpot requires a more complicated setup before it can be used. More precisely, StackSpot requires the user to manually input documents and information related to the codebase into a web browser interface before the AI assistant can be used inside the IDE. This has been noted as a negative aspect of the StackSpot developer experience by some participants. Moreover, adding different sources may in some cases negatively affect the quality of the generated code. Similarly to other studies, the participants in the study by Pinto et al emphasize the need to use the assistant "correctly", as only then will the productivity gains be unlocked. For example, one participant is quoted as saying that if StackSpot is "used carelessly or by less experienced people, it may result in more work for more experienced developers" [9]. This points to potential usability and developer experience issues, as the operation of ACATs is not seen intuitive, especially for novice developers, and is seen as something to master and requiring of effort. Overall, some of the developer experience issues pertaining to contextualized AI coding assistants are similar to those identified in general-purpose AI coding assistants in other studies, outlining some of the common expectations and requirements developers may have for both contextualized and general-purpose ACATs. [9]

Sergeyuk et al conducted a survey of over 400 developers (most of them experienced developers with over 10 years of experience) on their practices of using ACATs [10]. The results showed that most respondents used some AI tools in their programming work at least occasionally, with the most popular being ChatGPT, GitHub Copilot and JetBrains. Most respondents described the code generated by ACATs as "usable" and "accurate" but not "secure". The activities that the respondents enjoyed the least and are most likely

to delegate to ACATs were writing tests and natural language artifacts (such as code comments), while implementing new features was the task that was enjoyed the most and was least likely to be delegated. The findings indicate that developers are more likely to use ACATs for generation and summarizing rather than directly applying changes, possibly indicating a wish to remain in control and/or a certain level of distrust of AI-generated code. This is further supported by "remaining in control" and "lack of trust" appearing in the list of reasons for not using ACATs provided by the respondents that indicated they do not use AI tools for programming. Other UX issues identified by the authors include the "time ineffectiveness" of needing to write the correct prompt to receive the required output, flow state disruptions caused by using ACATs, issues with output not adhering to non-functional requirements and the lack of conversational/pair-programming capabilities for ACATs. [10]

Conclusions. Among different studies, there is a shared recognition of the need for user-centric evaluation methods and usability improvements for ACATs. Miah and Zhu proposed an evaluation framework focused on the number of prompts a developer needs to achieve their goal [6], while Liang et al. [7], Zhou et al. [8] and Sergeyuk et al [10] provided empirical insights into usability challenges and user expectations. Pinto et al. echoed similar findings in their study of a contextualized AI assistant, revealing that certain usability issues are present in the contextualized assistants too, underscoring the importance of ACAT usability research and improvements. A recurring theme among several papers is the importance of personalization, context-awareness, and feedback mechanisms. The current literature reveals the multifaceted nature of ACAT usability and the importance of evaluating ACATs from the usability/user experience perspective and tailoring tools to diverse developer needs.

2.1.3. Educational Impact and Learnability

Jayagopal et al. explored learnability of program synthesizers (including ACATs) by novices and found that novices struggled more with synthesizers that required a separate process to produce a specification compared to those that derived specifications from the developer's usual programming activity [11]. Synthesizers that exposed more control over triggering synthesis-related activities were more confusing, and participants often did not know what constituted meaningful progress or a good specification. Additionally, in familiar programming environments, participants borrowed behaviours from pre-synthesizer programming. Even in environments where the synthesizer was to be invoked with a separate process, the participants' first intentions were often to write their own code, expecting the synthesizer to derive the specification, pointing to the common mental model for program synthesizers being the derived specification. This may also mean that developers may prefer environments that allow them to both use program synthesizers and write their own code manually, exercising their pre-existing knowledge and practices.

Asking the developer to take actions that do not visibly move them toward task completion might also be learnability issue. Additionally, since novice developers tend to bring their pre-existing behaviours from other environments, the design of program synthesizers could benefit from identifying and working with common user behaviours. Novice programmers engage with synthesizer output by reading, tracing, and refining specifications before accepting it, making program synthesizers a potentially valuable tool for enhancing code comprehension and specification skills. [11]

Pan et al explored the factors affecting the willingness of college students to use ACATs [12]. The study developed a research model based on the Technology Acceptance Model (TAM) to examine the willingness to adopt ACATs, emphasizing perceived usefulness, ease of use, and behavioural intention as core factors, as well as four external variables (perceived trust, perceived risk, self-efficacy, and dependence worry) added based on prior research. The results suggest that the willingness to use ACATs is positively influenced by trust in the ACAT, while risk assessment (for example, the perceived quality of the generated code) has a negative influence, with the concerns over skills acquisition and retainment playing a significant role as well. Additionally, a user's self-efficacy negatively impacts dependency worries and positively affects perceived ease of use. These results highlight opportunities for improving ACATs by focusing on meeting users' educational and long-term growth requirements in addition to their immediate need for efficient programming, such as by taking into account the developer's programming proficiency level. [12]

Sun et al explored how the usage of ChatGPT would affect college students' "programming behaviors, performances, and perceptions" [13]. The study compared students' programming behaviours in students who engaged in CFP (ChatGPT-Facilitated Programming) and those who engaged in SDP (Self-Directed Programming). Both modes showed similarities, such as frequent manual coding in Python, but differed in specific behaviours: SDP students read console feedback more often and debugged less, while CFP students engaged more in code debugging, in which they relied heavily on ChatGPT. Students in the CFP mode achieved slightly higher average scores on their final programming projects compared to those in the SDP mode, but the difference was not statistically significant. The CFP mode students largely saw ChatGPT-aided programming as advantageous, and ChatGPT as beneficial for learning programming. Aside from code-related tasks, such as code generation and debugging, many students also used ChatGPT for general tasks such as information gathering. Students used ChatGPT throughout various programming stages, most frequently for coding, debugging, and learning new concepts, while also valuing its support in problem decomposition and project design. Regarding the user experience of ChatGPT, students highlighted several advantages of using the

chatbot, including its extensive programming knowledge, contextualized and varied responses, accurate and efficient feedback, and human-like interaction, which they found superior to traditional search engines and programming websites. However, they also reported issues such as inaccurate and outdated code, limited input/output capabilities (such as allowing only text as input or output), challenges in formulating effective queries, and occasional technical problems. Overall, while ChatGPT was valued for its support and efficiency, students recognized its limitations in handling complex projects and nuanced language understanding. While students' perceptions of ChatGPT's usefulness, ease of use, and intention to use improved significantly with experience, these gains did not translate into more positive attitudes toward its use, revealing a gap between practical acceptance and emotional response. As a result of their findings, the authors make several recommendations on how ChatGPT could be effectively used in programming education, some of which are related to the improvements to ChatGPT itself, such as an implementation of a "heuristic guidance" mode that "guide[s] students toward solving problems by asking sequential and probing questions" instead of providing a code answer immediately after prompt submission. [13]

Conclusions. The literature on educational impact and learnability of ACATs gives support to the idea that these tools can be useful to novice and student programmers, but also introduce risks related to over-reliance and skill retention. Jayagopal et al. [11] and Pan et al. [12] emphasized the importance of tool simplicity, intuitive design and educational needs in fostering ACAT adoption among novices, while Sun et al. [13] showed that ChatGPT can enhance learning outcomes and engagement. The studies show that novice programmers are keen on using ACATs as learning aid and do not fully automate their tasks. Several studies also directly suggested or alluded to the idea of ACATs taking the educational needs of students and novice programmers into account. The analyzed studies highlighted different aspects of the educational experience, and, collectively, their findings suggest that ACATs have strong potential in educational contexts if designed to support learning and maintain developer autonomy.

2.2. Developer Experience

2.2.1. Developer Experience Frameworks

Fagerholm and Münch set out to define developer experience as a "means for capturing how developers think and feel about their activities within their working environments" [14]. The concept of developer experience is understood similar to user experience. A "developer" is defined as anyone who is involved in writing code and developing software, and "experience" is defined as the involvement. Several factors are thought to influence developer experience, which in turn affect the outcome of the project. Developer experience includes "not only affective aspects, but also cognitive, conative, and

social aspects of experience" [14]. The cognitive aspects include factors related to how a developer experiences the development infrastructure intellectually, such as specific interactions with development tools. The affective dimension involves factors that shape how developers feel about their work, such as respect, belonging, and attachment. The conative dimension relates to how developers perceive the value of their contributions. When personal goals align with team goals through intentional activity, it is likely to boost purpose, motivation, and commitment, further enhancing developer experience. When designing development tools, such as ACATs, developer experience may offer insights into increasing the attractiveness of said tools to developers. Figure 1 below shows the conceptual framework presented in the paper. [14]

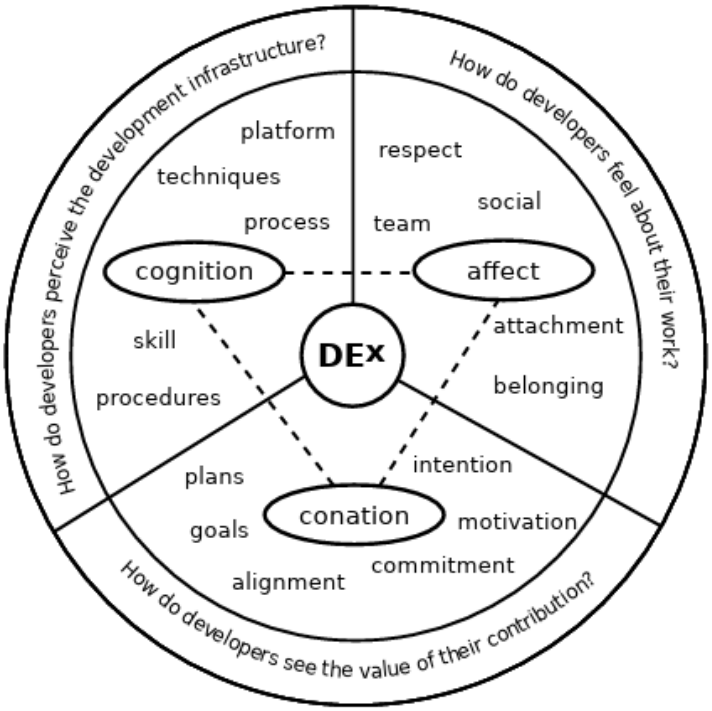

Figure 1. Developer experience: conceptual framework proposed by Fagerholm and Münch [14].

Greiler et al examined the factors affecting developer experience using semi-structured interviews with software developers [15]. Their paper presents the "DX Framework", which consists of factors affecting developer experience (DX factors), contextual factors that determine the importance of DX factors, barriers to improving the developer experience, improvement strategies, and coping mechanisms for when the developer experience cannot be improved. The DX factors concern development and release (factors related to codebases and tools, such as codebase health, development environment, automated testing coverage and others), product management (clearly-defined goals and scope, iterative work, unreasonable deadlines), collaboration and culture (factors related

to the relationships between team members, like support, knowledge sharing, feeling connected) and developer flow and fulfilment (factors that influence how easily and enjoyably the developers can complete their tasks, such as autonomy, stimulating work, progressing without obstacles). Support as a DX factor is especially important to junior developers and new team members. Contextual characteristics are defined as presence of problems, expectations, seniority, personal interests, company goals, company maturity and frequency of problems. The seniority factor describes how the experience level of a developer affects the importance they place on certain factors, as senior developers were seen as having more experience and mental capacity to care about factors such as team culture or release process, while junior developers were more focused on flow and codebase health. The barriers to developer experience improvement were found to mainly be organizational (such as low prioritization) rather than technical. Improvement strategies and coping mechanisms were also non-technical. The study examined the multifaceted factors that influence developer experience and related issues and demonstrated the areas in which the developer's seniority level may influence developer experience. [15]

Saboor et al set out to define developer personas and the developer experience factors influencing developers' choices in which frameworks and technologies to use [16]. The paper identifies four main characteristics of the application developer persona, representing the most important factors relating to choosing a tool: Ease Of Use, Eco-System, Framework, and Quality & Maturity. Developers are likely to choose tools that are easy to use and offer API hooks that allow outside access when necessary, with a strong preference for open-source solutions and robust support for debugging capabilities. A well-developed ecosystem with active community support, comprehensive documentation, and easy integration with other systems that are in use is also important for developers. Frameworks that are quick to learn, flexible, and not tied to specific brands are preferred. Moreover, developers prioritize performance, portability, reliability, and security in their tools and seek solutions that are auditable and interoperable across platforms. The authors recommend prioritizing ease of use in software tools and advocate for developer-first approaches in design decisions to enhance the developer experience. Building collaborative ecosystems that foster co-innovation, knowledge sharing, and the removal of barriers to ecosystem growth are also emphasized. The authors suggest creating stable, mature software interfaces and visual programming tools to enhance development. Finally, the importance of tools and frameworks providing interoperability and auditability of code, thus increasing a developer's performance and productivity is highlighted. Overall, the study presents insights into the key aspects of the work tools that developers value and that lead to developers choosing to work with certain tools. [16]

Conclusions. Fagerholm and Münch's individual-oriented conceptual model [14] emphasized similar themes as Greiler et al.'s DX Framework [15], with the latter taking

a more organization-oriented approach and expanding the former by adding contextual factors such as developer's seniority, organizational barriers to improving developer experience, suggestions for improvement and coping mechanisms. Saboor et al. contributed by identifying developer personas and tool selection criteria, reinforcing the importance of ease of use, ecosystem support, and tool maturity [16]. Collectively, the studies agree that developer experience is multifaceted and influenced by both individual and organizational contexts and build a layered understanding of developer experience that supports the exploration of proficiency-level differences and tool design implications.

2.2.2. Flow State and Motivation

Kuusinen et al measured developer experience as the relationship between developers' flow, intrinsic motivation and user experience [17]. The study discovered that both autotelic experience and intrinsic motivation related to both software development and the specific tools used were important factors in predicting developers' overall user experience. Need fulfilment could be anticipated based on autotelic experience, sense of control, and intrinsic motivation. Practical, hedonic, and general quality collectively were also significant predictors of need fulfilment when analysed collectively, but not individually. Developers who felt they had more freedom in choosing their tools rated both overall user experience and need fulfilment higher than those who felt required to use specific tools. Furthermore, those with less perceived choice reported enjoying the tool less. This suggests that developers with less choice were less motivated to use the tools. Additionally, developers with lower perceived choice experienced frustration more frequently. The study suggests that intrinsic motivation and flow state have significant influence over the developer experience, and the perceived choice affects intrinsic motivation towards using a specific tool. [17]

Calais and Franzini explored how test-driven development (TDD) impacts the flow state and developer experience [18]. The paper examines the relationship between TDD steps (writing a failing test, writing code to pass the test, refactoring code) and the preconditions necessary to achieve the flow state (clear goals, unambiguous feedback loop, challenge-skill balance, sense of control), which has been strongly linked to factors impacting developer experience. Clear goals are achieved by writing tests first before writing any code (thus making the goal to be passing the test), unambiguous feedback loop is created with the tests failing and passing, the challenge-skill balance is ensured by the developer focusing on smaller problems defined by the tests rather than the entirety of the system being developed, and the sense of control is provided by the presence of tests that the developer can run at any time to see whether the system is working as intended. Therefore, TDD as a practice aligns with the pre-requisites of achieving flow state. Additionally, the results of the survey presented in the paper show that TDD practitioner developers experience flow state, with those who have more experience in TDD experiencing

flow more often. Lastly, “self-report data collected from popular online tech forums and blog posts” provides testimonies of developers experiencing flow state. Overall, the paper shows how engineering practices such as TDD can work in fostering flow state, thus (positively) affecting developer experience. [18]

Forsgren et al set out to investigate the impact that flow state, feedback loops and reduced cognitive load – all components of developer experience – have on “developer, team and organization outcomes” [19]. Their study found that improvements in all three components correlated with better outcomes for individual developers, teams, and organizations. More prevalent flow state improved productivity, creativity, code quality, and organizational innovation. Fast feedback loops, such as quick code reviews and answers to questions, notably reduced technical debt and improved team performance, although their direct effect on individual and organizational outcomes was limited. Lowered cognitive load, achieved through intuitive tools, clear processes, and understandable code, improved developers’ productivity and innovativeness, lead to better team outcomes, and improved organizational success. The research showed that flow state and reduced cognitive load were especially powerful levers for improving developer experience. Investments in developer experience not only benefited developers but also translated into higher retention, profitability, and goal achievement for organizations. The findings of the study supported the benefits of improving flow state and cognitive load reduction at all levels, while improved feedback loops were shown to be particularly beneficial for teams. The study recommends improving developer experience by enabling flow state, improving processes and tools to reduce cognitive load, and accelerating feedback loops. Organizations could benefit from using data-driven approaches to identify pain points, set clear goals, and foster a culture of continuous learning and engagement to maximize productivity, innovation, and employee retention. These strategies are shown by the study to benefit individual developers, teams, and organizations. Overall, the study by Forsgren et al underscores and quantifies the positive effects of investing in developer experience and its components. [19]

Conclusions. The reviewed literature strongly supports the role of flow state and intrinsic motivation in shaping positive developer experience, with all studies demonstrating that flow state correlates with increased productivity, satisfaction, and innovation. The studies collectively show that practices like test-driven development and intuitive tooling foster flow by providing clear goals, feedback loops, and a sense of control. Forsgren et al. extend this by linking flow and cognitive load reduction to organizational outcomes, suggesting that developer experience improvements benefit not just individuals but entire teams and companies [19]. There is consensus across the papers that enabling flow and reducing cognitive load are key for enhancing developer experience.

2.2.3. Tool Usability and Evaluation

Kuusinen et al. have investigated the developer experience of a graphical user interface designer [20]. Their findings showed that developers who rated their experience as good and developers who rated their experience as bad differed in whether they were doing things spontaneously, felt frustrated, enjoyed using the tool and were skilled in using the tool. This implies that developer experience may be affected by the spontaneity of the developer's actions (which could be related to the exploration and acceleration modes as defined in [2]), how frustrated they are, and how enjoyable they find their tools to be, as well as their skill level. Other research has also found that hedonic qualities are a better predictor of developer experience than pragmatic qualities. The key flow state and intrinsic motivation factors contributing to developer experience include autotelic experience (how rewarding the experience is), interest/enjoyment (enjoyable tools, not boring tasks), perceived competence (being skilled with the tool), challenge-skill balance (feeling competent), sense of control (voluntary use), and action awareness (spontaneous and automatic). The findings of the study point to the importance of hedonic qualities of tool use to the developer experience and the potential for further studies in this regard. [20]

Kuusinen et al. have investigated the developer experience of a specific IDE [21]. While efficiency, flexibility, informativeness and intuitiveness were all found to be the best qualities of the IDE that was analysed, flexibility and informativeness were simultaneously regarded as being the areas that need improvement the most. The authors speculate that this might point to flexibility and informativeness are being "central for an IDE" [21]. The best qualities of the IDE were perceived as being fast and easy-to-use (especially emphasized as important for novice users). The IDE was also seen as simple, unbloated, well-designed and supportive of developers' work. On the other hand, developers wished the IDE was more flexible (easier to integrate with other tools, more plugin support) and more stable (especially in terms of the debugger). Both the lack of stability and the lack of flexibility were seen as breaking the flow state, as it necessitated moving away from the IDE and switching to other tools, and often waiting. Other areas of improvement, such as intelligent code completion, were also identified. While the developers' open-ended responses concentrated on the pragmatic qualities rather than the hedonic, the IDE was also described "with affection", as developers "expressed that the IDE made them to feel at home and created a friendly atmosphere" [21]. The authors propose that the results of the survey may underscore the importance of flow state for the developers' workflow, and the role an IDE may play in supporting flow state. The study offers insights into the qualities developers value in their development tools. [21]

The "State of developer experience 2024" report by software company Atlassian examined the situation of developer experience in the industry and provided advice on how

developer experience can be improved by companies [22]. The report revealed that developers are losing significant time primarily due to technical debt, insufficient documentation, and inefficient processes, negatively affecting their developer experience. There is a clear disconnect between leadership and developers: while most leaders believe improving developer experience is essential for attracting and retaining talent, less than half of developers feel their organizations truly prioritize it, and even fewer are satisfied with their company's current investments in developer experience. Although leaders are optimistic about the impact of AI tools and believe them to be key in improving developer experience, most developers in the report said that they are yet to see meaningful productivity gains from them. The report underscores the need for organizations to move beyond traditional productivity metrics when evaluating developer experience and invest in targeted improvements that address real developer pain points. To improve developer experience, the report recommends focusing on three core elements: establishing strong feedback loops for continuous improvement, reducing cognitive load, and enabling developers to achieve the flow state. Additionally, the report emphasizes the importance of directly engaging developers to identify friction points, investing in platform engineering teams, and shifting from traditional productivity metrics for measuring developer experience to more meaningful indicators like perceived ease of software delivery and overall job satisfaction. Organizations that invest in these targeted improvements see gains in both productivity and developer morale. Overall, the report displays the discrepancies that exist between the outlook on developer experience among developers and leadership and underscores the need for developer involvement and direct feedback in developer experience initiatives. [22]

Conclusions. The studies in this subsection agreed on the importance of usability in shaping developer experience. Kuusinen et al. show that hedonic qualities such as enjoyment, spontaneity, and perceived competence are stronger predictors of developer experience than efficiency alone [20], [21]. Developers value tools that support flow and feel intuitive and pleasant, even if they are not the most feature-rich. The Atlassian industry report highlighted the disconnect between leadership and developers regarding developer experience priorities and the limited impact of AI tools on productivity so far [22]. While the academic studies focus on individual perceptions, the industry report emphasizes organizational strategies, such as feedback loops and platform engineering. The studies collectively address developer experience from both individual and team perspectives, underscoring the need for tools that offer both good technical performance and pleasant hedonic experiences.

2.2.4. Cognitive Load and Onboarding

Chandrasekaran has examined the strategies aimed at improving the developer experience by reducing cognitive load [23]. The emergence of new tools and practices such as

DevOps has increased the developers' cognitive load, negatively affecting their productivity and developer experience. As more tools become available, developers need to master a higher number of technologies, thus increasing the cognitive demands associated with managing and understanding the complex relationships among them. Developers, especially those new to a codebase or a tool, face significant cognitive load as they often need to navigate documentation, set breakpoints, and understand complex code structures before commencing their daily tasks. Higher cognitive demands can lead to more errors, slower development, and reduced productivity, while prolonged high cognitive load is associated with increased stress, burnout, and job dissatisfaction. Individual experiences of cognitive load may also vary based on factors such as expertise, experience, and cognitive abilities. Chandrasekaran proposes four cognitive load reduction strategies: platform engineering, proper documentation, internal developer portal and standardized development practices. These strategies are mostly aimed at streamlining the development process by standardizing and automating tasks related to the tool selection, setup and maintenance, as well as having standardized processes to follow for faster and more efficient development, which underscores the role familiarity with tools and processes plays in developer experience. [23]

Conclusions. Chandrasekaran's work emphasized reducing the cognitive load in onboarding, lending support to the idea that those with less experience may face higher levels of cognitive load, and thus may need extra measures or features in ACATs to achieve positive developer experience. Chandrasekaran focuses on organizational strategies, while other papers (such as Forsgren et al. [19]) explore the link between cognitive load and individual productivity and satisfaction. Collectively, the findings suggest that reducing cognitive load through better onboarding and tool design is essential for improving developer experience, especially for novice developers.

2.3. Summary of Related Work

The existing research on the **user experience of ACATs** highlights that AI coding assistants like GitHub Copilot generally improve productivity and code quality, particularly for less experienced developers. However, highly skilled users may not always see time savings and sometimes even experience slower task completion. Developers interact with coding assistants in two main ways: acceleration mode (for familiar tasks, maintaining flow and speed) and exploration mode (for unfamiliar tasks, requiring more validation and prompting). Key user experience challenges include worse performance on complex logic, limited natural language processing capabilities, and discoverability issues with certain features of ACATs. Developers prioritize reduced keystrokes, syntax recall, and faster task completion, but reject suggestions failing functional or non-functional requirements and tend to only delegate tasks they enjoy less to the ACATs rather than fully ceding control. Using ACATs in classroom settings generally makes programming feel

less difficult for students, reduces the need for external help, and improves code quality and task completion speed. However, it may also introduce challenges such as over-reliance and skill retention issues. The effectiveness of and student willingness to use ACATs are strongly influenced by perceived usefulness, trust, and efficiency. **Developer experience** is shaped by cognitive, affective, and motivational factors, with flow state and reduced cognitive load being linked to higher productivity and job satisfaction. Technical debt, poor documentation, and inefficient processes remain major barriers to positive developer experience, and, according to some industry surveys, AI tools have yet to deliver widespread productivity gains across the industry. Seniority influences developer experience, as novice developers focus on codebase health and flow, while experienced developers prioritize team culture and release processes. Hedonic qualities such as enjoyment and spontaneity outweigh pragmatic features in predicting positive developer experience. Organizational investment in developer experience through flow support, cognitive load reduction, and enhanced feedback loops correlates with higher productivity, innovation, and retention.

3. Methodology

This chapter describes the methodology for the online quantitative survey of novice and experienced developers on GitHub Copilot developer experience, as well as the methodology of the user study looking into the developer experience of GitHub Copilot in novice and experienced developers. The quantitative survey was conducted to gather broad general insights into GitHub Copilot developer experience among novices and experienced developers, while the qualitative user study was chosen to perform a deeper exploration of individual perceptions during real coding tasks. These methods combined allowed for and in-depth understanding and comprehensive evaluation of the developer experience of GitHub Copilot and its differences among developers of different career stages.

Figure 2 below presents the timeline of the research process.

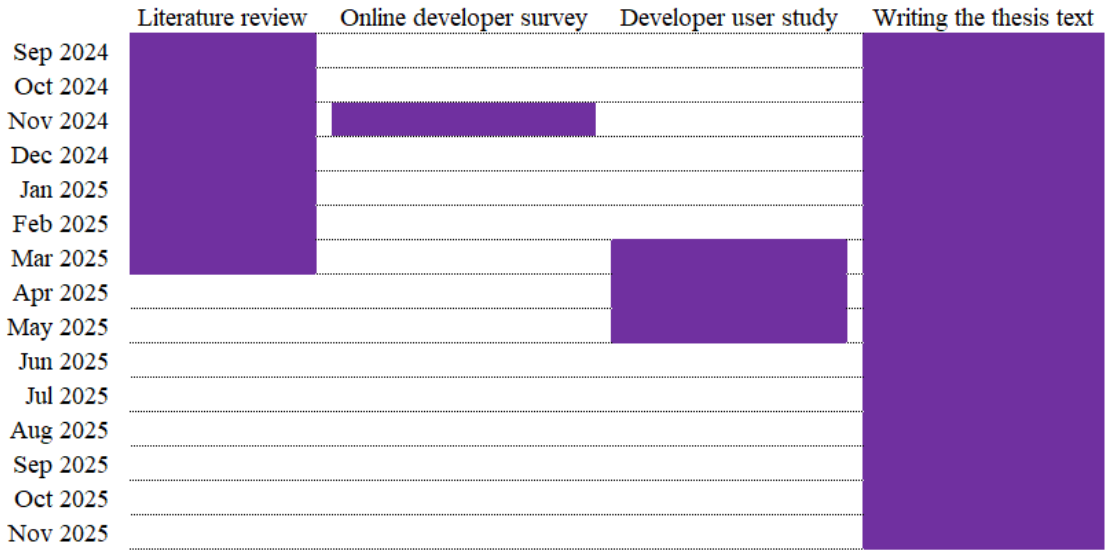

Figure 2. Timeline of the research process

3.1. Online Developer Survey

To conduct a general overview of common perceptions of GitHub Copilot as a programming assistant among professional developers, an online survey was conducted on November 2024. The survey was distributed in online spaces and communities frequented by software developers, engineers and other professionals, such as programming-related Discord servers and online forums. The survey was also shared on the survey-sharing platform SurveyCircle. The first part of the survey contained the questions related to the respondent's professional background. The background questions were as follows:

- **Years of Experience:** How many years of professional software development experience do you have?
- **Job Title:** What is your job title?

- **Use of Copilot:** Do you use GitHub Copilot in your job?

More information regarding the background questions, such as question type and answer options, is presented in the Table 1 of Appendix 1.

Aside from collecting background information, the Years of Experience question was utilised as a separation criterion for separating the answers from novice and experienced developers. Respondents who indicated their years of professional experience as "0-5 years" were considered novice developers, and those who selected "5+ years" as their answer were considered experienced developers. Moreover, the Use of Copilot question was also used as a filtering question, selecting only the respondents that answered "Yes" to proceed to the second part of the survey, which contained questions related to the GitHub Copilot developer experience. The GitHub Copilot developer experience questions were as follows:

- **IDEs used:** Which of the following integrated development environments (IDEs) do you use in your job?
- **AI Tool Usage:** How often do you use the following AI-powered tools in your job? [list of different tools]
- **Other AI Tools (optional):** Do you use any other AI-powered tools in your job? If yes, how often?
- **Software Development Lifecycle Stages:** Which stages of software development lifecycle do you use AI-powered tools for in your job?
- **Programming Activities:** Which programming activities do you use AI-powered tools for in your job?
- **Descriptors for Copilot:** When using GitHub Copilot in my job, I find it to be... [list of different qualities]
- **Satisfying Copilot Qualities:** I am satisfied with GitHub Copilot's... [list of different features]
- **Most Important Improvements (up to 5 options allowed):** Choose the most important areas of improvement for GitHub Copilot: [list of options]

More information on developer experience questions is presented in the Table 2 of Appendix 1.

In total, 39 responses were submitted, of which 28 passed the filtering question and were included in the analysis dataset. Out of the 28 included responses, 11 were from novice developers and 17 were from experienced developers.

No data that could identify the respondent (such as the respondent's name, email, location, employer or other information) was collected or recorded during the survey and the survey was fully anonymous.

The results of the online developer survey are analysed in chapter 4.1.

Analysis methods. After the survey responses were collected, the valid responses that passed the filtering question were extracted from the dataset. Responses from novice and experienced respondents were separated into different datasets to allow for analysis based on experience level. After that, the data was analyzed by question and experience level, by visualizing it to allow for seeing trends and identifying differences between novice and experienced participants. From the visualization, the online developer survey findings were drawn.

3.2. Developer User Study

User study investigating the developer experience of GitHub Copilot as experienced by novices and experienced developers was conducted in April 2025 and May 2025. The user study participants were recruited through the author's personal, professional and academic networks, as well as online spaces and communities visited by software developers. In total, 11 user study sessions were conducted, each with one participant, 5 of them conducted with novices and 6 of them conducted with experienced developers. To enable as many participants to take part in the user study as possible, 3 of the study sessions were conducted in-person, and 9 were conducted online. The participants were able to choose whether to take part in the user study in-person or online. The in-person user study sessions were conducted at Tampere University City Centre campus in Tampere, Finland. The online user study sessions were conducted using Microsoft Teams. In both in-person and online study sessions, the participants completed their tasks using a laptop provided by the researcher, with online participants using the remote control functionality provided by Teams. Each study session lasted for 1 hour.

The participants had the option to choose their preferred IDE from the list of IDEs that had a GitHub Copilot plugin available as of March 25th, 2025, which were Microsoft Visual Studio (version 17.13.4, GitHub Copilot plugin version 17.13.441.19478), Microsoft Visual Studio Code (version 1.99.2, plugin 1.303.0 and earlier), Neovim (version 0.10.4, plugin 1.44.0), and JetBrains IntelliJ IDEA Community Edition (version 2024.3.5, plugin 1.5.40-243).

Each user study began by participant being explained the aim of the research and testing procedure and signing the consent form. The participants were asked to use think-aloud method while completing the programming task, which means voicing out any and all thoughts that come to their head.

Background questions. The study sessions began by the participant being asked these background questions:

1. What is your current job position?

2. How often do you use GitHub Copilot in your job? How long have you been using it?
3. Which integrated development environments (IDEs) do you use in your job?
4. Do you use any other AI coding assistants or AI tools in your job besides GitHub Copilot? If yes, which ones and how often?

Test tasks. After the background questions, the participant was given their programming task to complete using GitHub Copilot. Each participant was given one task to complete during the user study, and they had the possibility to abandon the task and ask for a different task once. Before proceeding with the task, the participants were asked to read the task instructions out loud. The tasks involved performing programming tasks that a developer would normally perform as part of their job, such as refactoring or debugging, and were implemented using React or .NET frameworks. The task set included two tasks focused on the front-end (React-based) and two tasks focused on the back-end (.NET-based). The task set was as follows:

Task 1. Open the folder *user-study-task1* in the IDE. Complete the React app by displaying the one-day weather forecast for Tampere, starting from today, using the AccuWeather API. Display the forecast in the homepage. Write integration tests (optional). Refer to the README.md on instructions on how to use the AccuWeather API.

Task 2. Open the folder *user-study-task2* in the IDE. The repository contains a React app that includes 3 bugs. Debug the app to fix the issues. Write unit tests (optional). Refer to the README.md on instructions on how to setup and run the app.

Task 3. Open the folder *user-study-task3* in the IDE. Complete the .NET app by implementing endpoints for a REST API to create, read, update and delete objects of class *Note* in an in-memory database (a List). Write unit tests (optional). Refer to the README.md on instructions on how to setup and run the app.

Task 4. Open the folder *user-study-task4* in the IDE. Refactor the .NET app to follow SOLID principles, implement input validation and error handling. Write unit tests (optional). Refer to the README.md on instructions on how to setup and run the app.

The tasks were designed to mimic real software development tasks a developer may face in their day-to-day activities, such writing code or refactoring, while still being possible to complete in less than one hour. An attempt was made to match tasks to participants based on their familiarity with the frameworks used by the tasks – for example, a participant that was familiar with React would have been given a React-based task. Participants who were familiar with both or neither of the frameworks could choose whether they preferred React or .NET task. Information on the test setup and the tasks attempted and/or finished by each participant is presented in the Table 3 in Appendix 3.

Post-test questions. Immediately after finishing the task, the participant was asked the post-test questions, which were as follows:

1. What is your overall impression of GitHub Copilot?
2. If you had to pick 3 words to describe your experience with GitHub Copilot, what would they be? *(hints, if participant required them: intuitive, unreliable)*
3. What would be your suggestions for improvement for GitHub Copilot?
4. What did you like about GitHub Copilot?
5. What did you dislike about GitHub Copilot?

The user study concluded with a debrief, where a participant was asked to share any final thoughts and/or questions should they have any.

The analysis of the user study results is presented in the chapter 4.2 in the "Results" section.

Analysis methods. After all interview sessions were conducted, the screen recordings were reviewed by the author, and thematic coding was performed using the inductive thematic analysis framework. During the coding process, attention was paid to both to participants' actions and any comments voiced out, as well as the questions they might have asked. After the coding process, the emergent codes were grouped by their themes (for example: usability issues, positive aspects of Copilot) and from the themes, the insights were formed.

4. Results

This chapter presents the analysis of the results for both the quantitative online developer survey and qualitative user study.

4.1. Online Developer Survey

As mentioned in chapter 3.1, 28 valid responses were collected and analyzed, 11 of them coming from novice developers and 17 of them submitted by experienced developers.

4.1.1. Respondent Profiles

Job Title question revealed that the most common job title among novices was front-end, back-end or full-stack developer (36,4%), while among experienced developers it was software developer or engineer (41,2%), which was also the most common job title overall.

IDEs used question results showed that Visual Studio Code was the most popular IDE overall (82,1%), used by 90,9% of novices and 76,5% of experienced developers, followed by Visual Studio and IntelliJ IDEA.

AI Tool Usage question results found ChatGPT to be the most frequently used AI tool besides GitHub Copilot, with 46,4% using it daily, with Visual Studio Intellicode and Bing AI being the second and third most popular choices respectively. Novices used ChatGPT more frequently (72,7% daily) than experienced developers (29,4%). GitHub Copilot is used frequently by the respondents, with 12 of them (42,9%) using it daily, 7 respondents using it a few times per week, 5 respondents (17,9%) using it a few times per month, and the remaining 4 respondents using it less than a few times per month. 19 respondents (67,9%) use GitHub Copilot a few times per week or more often. GitHub Copilot usage trends among novice respondents largely mimic the general trends, while experienced respondents tend to use GitHub Copilot more frequently than novices.

Most of the answers that came for the **Other AI Tools** question were given by experienced participants. Tools included in the answer span both general-purpose assistants and specialized platforms. One experienced respondent used Cursor, an IDE that is optimized specifically for AI-powered coding [24], during a free trial. Another reported using "own RAG-tuned LLMs" weekly, indicating custom-built solutions combining Retrieval-Augmented Generation and fine-tuning [25]. Other tools mentioned by experienced developers include HuggingFace (a platform for hosting and training LLMs) [26], Microsoft Copilot (an AI assistant), NotebookLM (an AI-powered tool for research and note-taking by Google) [27], Microsoft Azure AI tools (a set of AI tools for cloud services) [28], and the OpenAI API (an API for integrating OpenAI's services to third-party apps). These tools support tasks ranging from coding to research and integration, indicating a wide

range of applicability for AI in the respondents' work. The only novice response mentioned using a Visual Studio Code extension for BLACKBOX.AI, an AI coding assistant that works as an extension for Visual Studio Code [29], occasionally. Overall, the responses show that experienced developers tend to use more professional-grade or customizable AI tools, while novices stick to mainstream or beginner-friendly options.

4.1.2. AI Tool Usage Habits

Software Development Lifecycle Stages question results revealed that 85,7% of respondents used AI tools during the development stage, followed by 64,3% during testing. Novices are 13,9% more likely to use AI-powered tools in the testing stage, 12,8% more likely to use AI-powered tools in the deployment stage, and 16% more likely to use AI-powered tools in the maintenance stage, while experienced respondents used AI tools 20,3% more in the requirements gathering stage. Overall, respondents tended to use AI-powered tools way more often for stages that are more focused on technical implementation and coding, such as development and testing rather than stages that are focused on more on domain knowledge and/or interaction with the client(s) and/or other non-IT stakeholders, such as requirements gathering.

Programming Activities question results showed that writing code (89,3%) and debugging (75%) were the most common activities for AI tool usage. Novices favoured using AI tools for debugging (90,9%), while experienced respondents used them more for writing code (94,1%) and writing tests (64,7%).

4.1.3. Views on GitHub Copilot

Descriptors for Copilot question results indicated that most respondents found GitHub Copilot easy to use and intuitive (53,6%, agree, 35,7% strongly agree) and easy to learn and beginner-friendly (57,1% agree, 25% strongly agree), but had concerns about its security and reliability, with the aforementioned aspects being rated the most negatively. Accessibility, reliability and customizability received neutral and somewhat-negative reviews. Speed was rated rather negatively, as 32,1% disagreed that GitHub Copilot is fast and robust, and 3,6% strongly disagreed. GitHub Copilot being secure and trustworthy received the most disagreement of all statements, with 9 participants (32,1%) disagreeing and 1 participant strongly disagreeing, while 12 participants (42,9%) neither agreed nor disagreed, 5 participants agreed and 1 participant strongly agreed. These results show that the respondents are most satisfied with GitHub Copilot being easy to use, learn to use and integrate into their workflow, while having reservations on the security and reliability aspects. Copilot's customizability, accessibility and inclusiveness are also regarded largely positively. However, it is also worthwhile to observe that the "Neither agree nor disagree" answer option, indicating neutral or close-to-neutral opinion, was a popular choice in many statements, in some cases accruing the largest number of answers.

This indicates that while developers may not be outright dissatisfied with certain aspects of GitHub Copilot's developer experience, they may not be fully satisfied either, meaning there could still be improvements that could be made. The biggest areas of disagreement between experienced and novice respondents are security and trustworthiness, pleasantness and likeability, as well as context-awareness and adaptiveness.

Satisfying Copilot Qualities question results highlighted satisfaction with the easiness of completing tasks (67,9% agreed that they are satisfied with it and 14,3% strongly agreed) and clarity and informativeness of system messages (50% agree, 3,6% strongly agree), but dissatisfaction with mistake prevention (21,4% disagree that they are satisfied with it). The results show that the mistake prevention, documentation and user-defined shortcuts are areas with the most dissatisfaction and thus the highest potential for improvement. While the satisfaction trends among novices are similar to the trends among all respondents, novices may also have a more strongly expressed (dis)satisfaction in certain areas, such as support for undoing or redoing actions or mistake prevention. Likewise, experienced respondents are less likely to express satisfaction across various aspects compared to novices, indicating that they may have higher expectations or encounter different challenges.

Most Important Improvements question results showed that better output (64,3%) and context awareness (57,1%) were the most requested improvements. Novices prioritized debugging and prompt comprehension, while experienced developers emphasized context awareness and testing capabilities. Improvements directly targeting the user experience, such as better UI (14,3%), and better documentation (10,7%), are less popular, indicating that the generated code quality and related concerns are seen as more urgent and more impactful for the developer experience. However, a relatively higher number of respondents choosing better support for personalization highlights to respondents potentially wanting to tailor their GitHub Copilot developer experience to their needs. Better context awareness as an improvement is 20,8% less popular among novices than all respondents, better debugging capabilities are 7,8% more popular, and better support for personalization is almost as popular among novices than among all respondents, with only 0,6% difference. This may be related to the way novice developers and professionals approach debugging, as novices are less likely to build complex mental models of software systems and use them while debugging [30], leading to less appreciation for the context awareness capabilities of coding assistants, or the tendency among novice professionals to work on tasks that concern smaller codebases (such as a single feature) rather than large systems. Overall, the improvements chosen by experienced respondents and novice respondents correlate with the programming activities in which the experienced and novice respondents respectively tend to use the AI-powered tools, and experienced respondents tend to choose more improvements than novices.

A more detailed overview of GitHub Copilot developer experience among all, novice and experienced participants is available in Figures 3, 4 and 5 respectively, which are presented below.

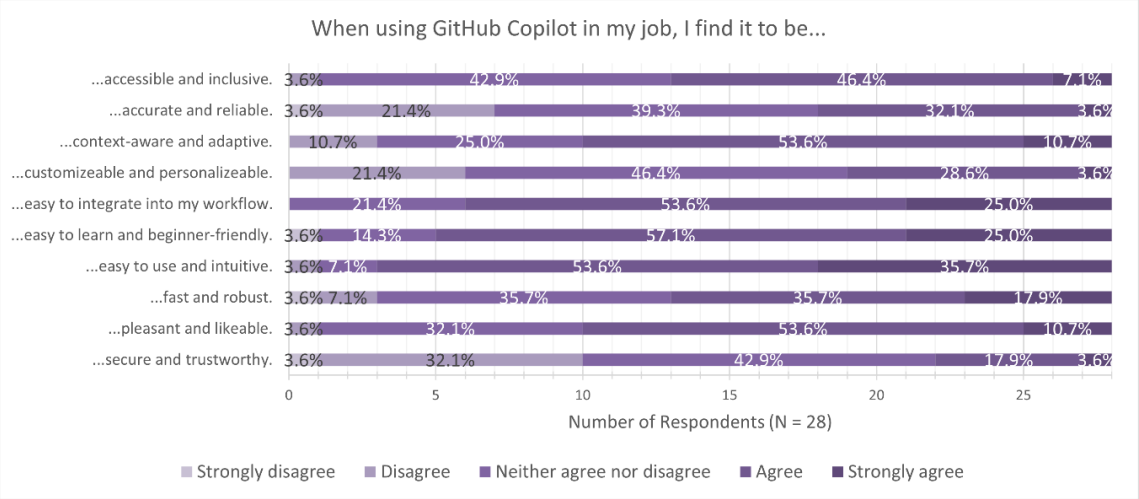

Figure 3. GitHub Copilot developer experience (all respondents)

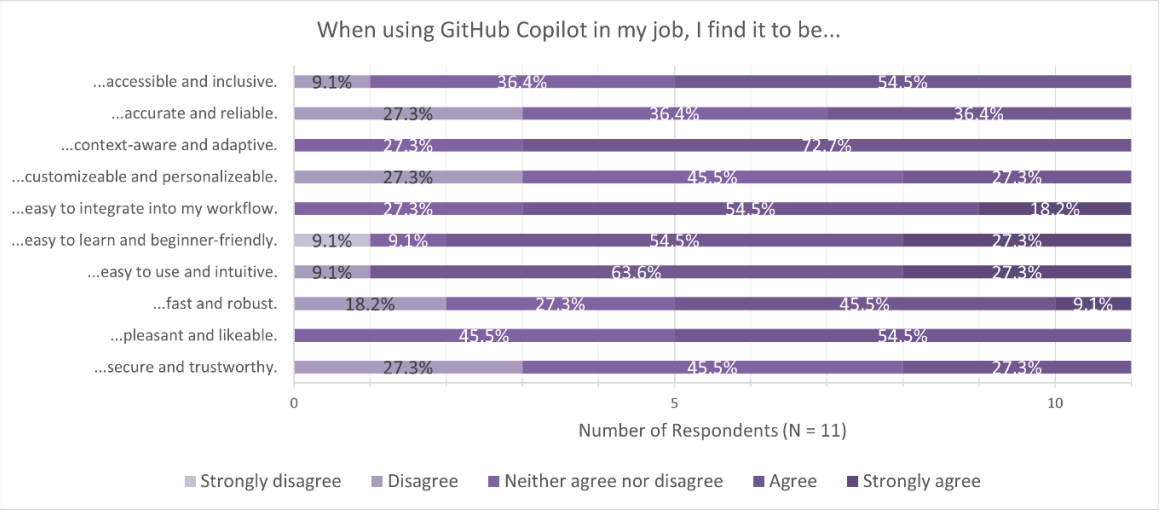

Figure 4. GitHub Copilot developer experience (novice respondents)

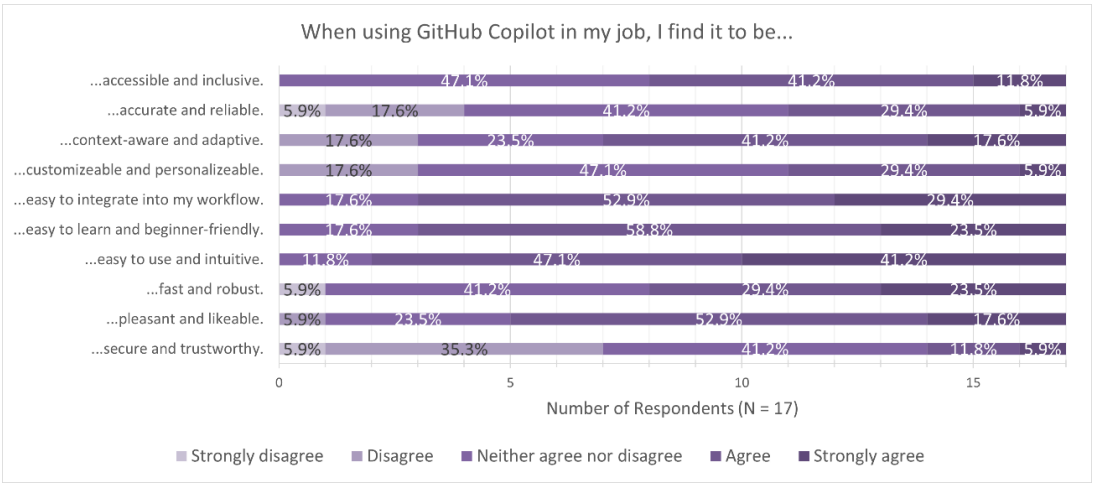

Figure 5. GitHub Copilot developer experience (experienced respondents)

More detailed information on satisfaction with different qualities among all respondents, novices and experienced respondents is presented in Figures 6, 7 and 8 below.

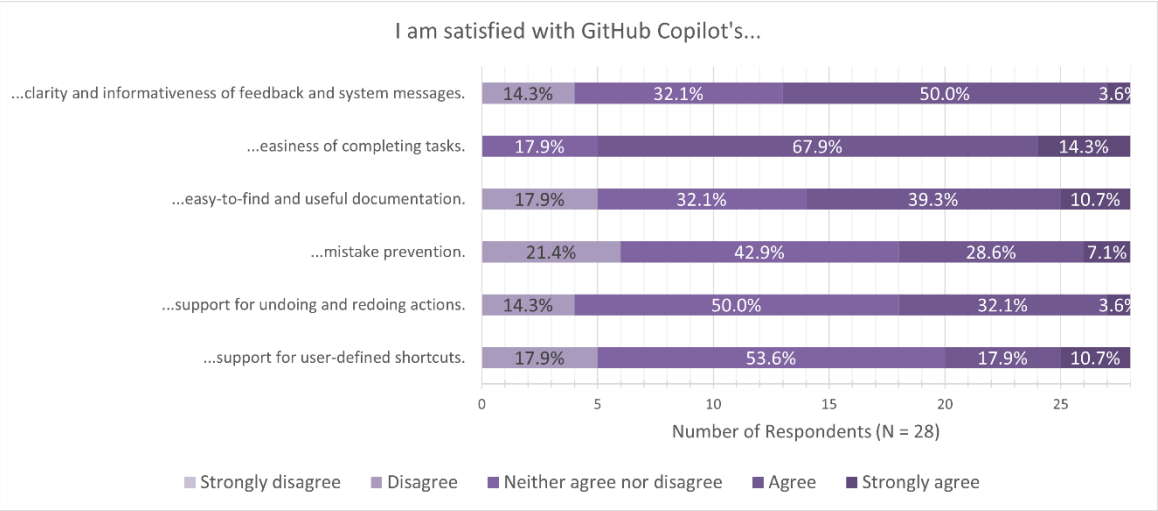

Figure 6. GitHub Copilot satisfaction (all respondents).

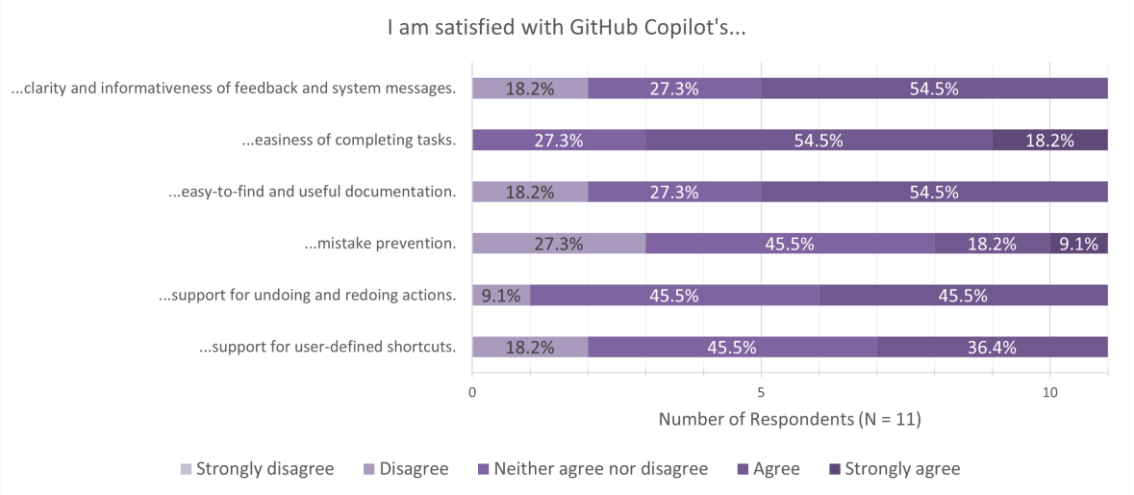

Figure 7. GitHub Copilot satisfaction (novice respondents)

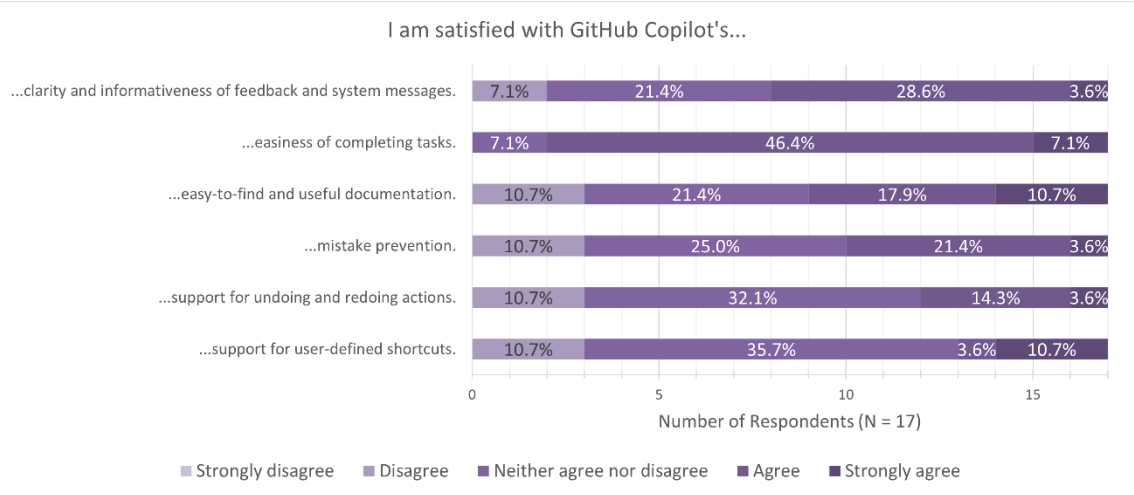

Figure 8. GitHub Copilot satisfaction (experienced respondents)

Figures 9, 10 and 11 below present a more detailed overview of the most important improvements among all respondents, novices and experienced respondents respectively.

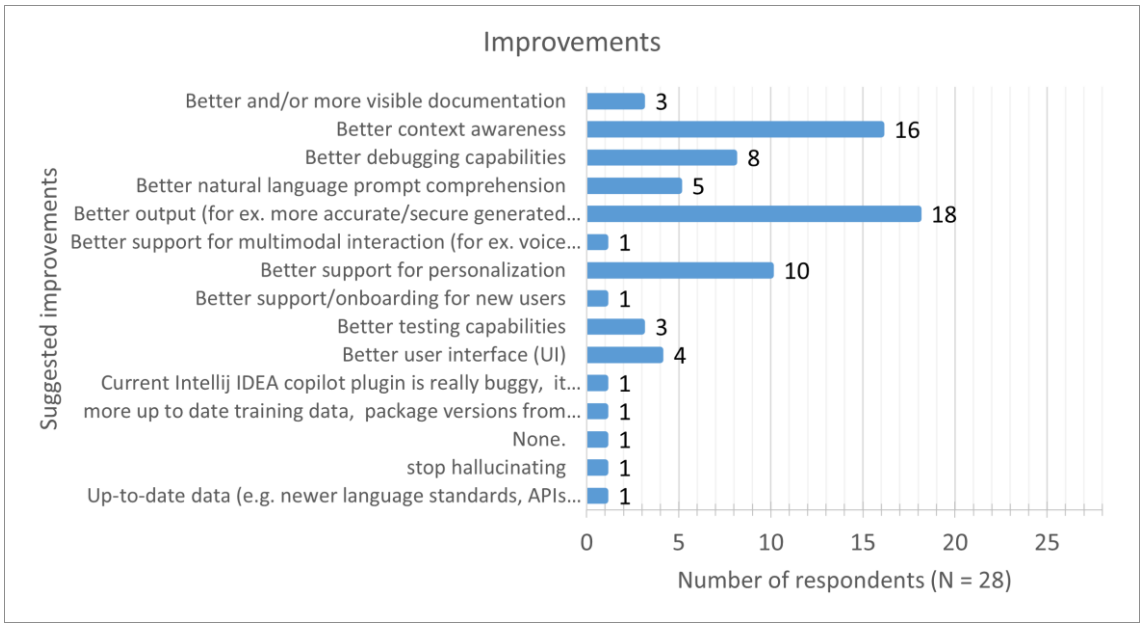

Figure 9. Most important improvements for GitHub Copilot (all respondents)

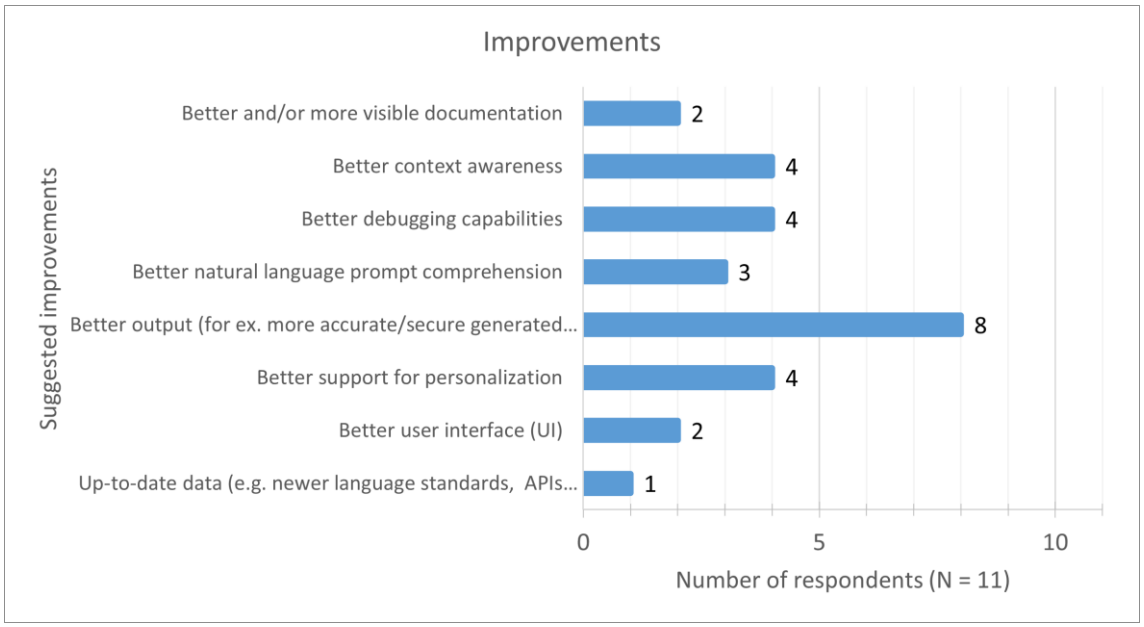

Figure 10. Most important improvements for GitHub Copilot (novice respondents)

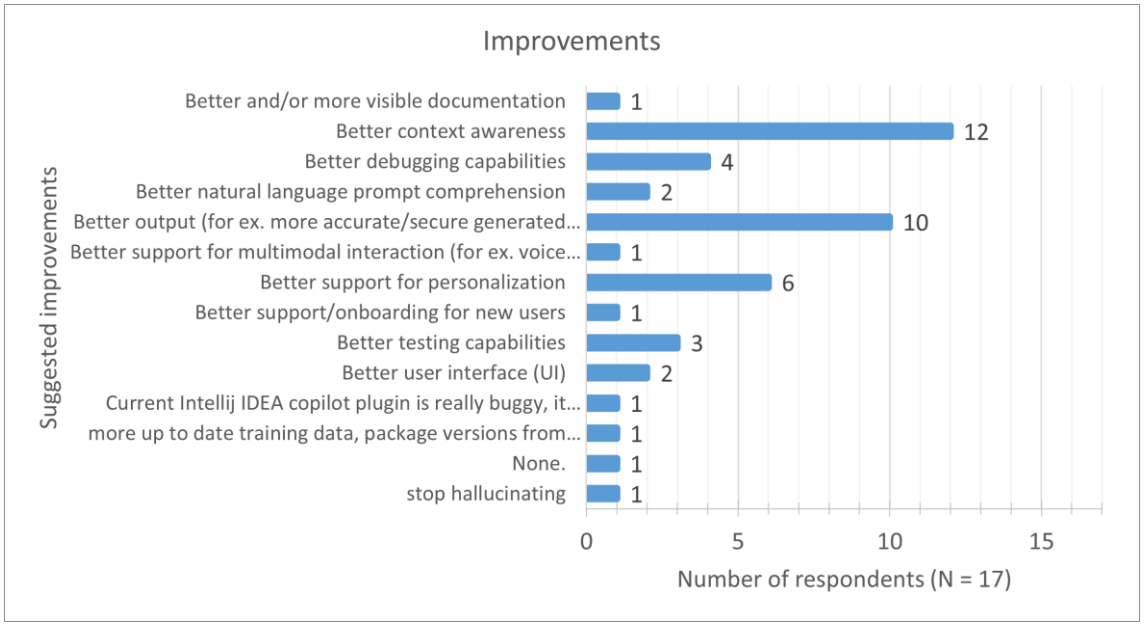

Figure 11. Most popular improvements for GitHub Copilot (experienced respondents)

4.1.4. Summary of the Survey Results

Implications for future research and development. The implications from the GitHub Copilot developer experience online survey suggest several potential future research and development directions. One of them is enabling GitHub Copilot and other AI-powered coding tools to tailor behaviour based on developer experience level (for example, providing more robust debugging assistance for novices and enhanced context awareness for experienced users). Another research and development suggestion is developing GitHub Copilot and other AI coding assistants to generate code with built-in security features, explain suggestions and provide confidence metrics to address security and trustworthiness concerns. Moreover, improving GitHub Copilot capabilities in regards to requirements analysis, architecture design and other programming activities beyond writing code; improving GitHub Copilot's understanding of complex codebases and project structures; and examining how developers' relationships with AI tools evolve as they gain more experience are also potential areas of improvements and future research worthy of exploration.

Conclusions. The survey demonstrates that novices tend to use GitHub Copilot as a learning aid and problem-solving assistant (especially for debugging), while experienced developers integrate it more broadly across different programming activities such as code analysis and writing tests. Most respondents held neutral-to-positive view on many aspects of GitHub Copilot's developer experience, finding it easy to use and intuitive, easy to learn, and easy to integrate into their workflow. However, satisfaction levels also varied by career stages, with novices being generally more positive, while experienced developers had higher expectations and more nuanced critiques. Aside from offering experience-tailored modes that emphasize different capabilities and needs of novices and experienced

developers, addressing shared concerns around output quality, security, and context awareness would improve developer experience for all users regardless of experience level. Said concerns echo the findings in other studies, such as [7]. Moreover, attention should be paid to GitHub Copilot's mistake prevention, documentation, and developing user-defined shortcut functionality.

4.2. Developer User Study

As stated in chapter 3.2, 11 participants took part in the qualitative user study. Five of them were novices and six of them experienced developers.

4.2.1. Background Questions

Participants' profiles according to the background questions are described below.

Each participant held a unique job title. Novice participants included representatives from academia, entrepreneurship and industry, such as a Researcher, Senior Researcher, Early-Stage Startup Founder, Freelance Full-Stack Developer, and an unemployed job seeker in the tech field. Most experienced participants held technical positions, including titles such as a Technical Product Manager, Full-Stack Engineer, Software Developer, Senior Software Developer, Doctoral Researcher in Computer Engineering, and a Scrum Master. This shows that the study reflects the needs and usage patterns of a wide variety of role and professions.

Novices had varied GitHub Copilot usage patterns: some had never used Copilot, while others used it rarely, occasionally, or frequently, with one using it "all the time" for five months. Experienced participants ranged from never having used Copilot to daily use, with one participant having used it for three years. Regular and long-term use was more common among experienced participants.

Visual Studio Code was the most popular IDE choice among both novices and experienced participants. Novices also used PyCharm, Matlab, Visual Studio, and IntelliJ IDEA. Experienced participants used a broader range of IDEs including IntelliJ IDEA, Netbeans, Spyder, Jupyter, RStudio, Sublime Text, Google Project IDX, and Android Studio. Visual Studio Code was the most popular IDE across both groups, but experienced participants tended to use more IDEs tailored to specific languages or workflows.

All participants used ChatGPT, making it the most popular AI tool beside GitHub Copilot. Claude was the second most popular among novices, used by three participants, while Claude, Cursor, and Google Gemini were used by one experienced participant each. Both groups tended to use up to three AI tools, with novices being more likely to use specialized ones like v0 and ManusAI. Some participants also revealed why they use the tools, such as for comparing the output of differrent tools, "checking code", understanding code and writing boilerplate.

Appendix 4 contains Tables 4 and 5, which present the job titles of both novices and experienced developers respectively in a table format. Tables 6 and 7 in Appendix 5 present the data on GitHub Copilot usage patterns among novices and experienced developers in a table format. Appendix 6 contains Tables 8 and 9, which display the data on the IDEs utilized by novice and experienced participants respectively, and Figures 28 and 29 which display said data in a bar chart form. Tables 10 and 11 in Appendix 7 display the data on AI tool usage among novices and experienced participants respectively in a table format, while Figures 30 and 31 display it in a bar chart.

4.2.2. Insights About Github Copilot Developer Experience

The following chapter describes insights regarding the developer experience of GitHub Copilot and its differences between novice and experienced developers observed as the user study participants were completing their programming tasks. The insights are grouped into four themes based on the topic they cover:

- Interaction Modes and Usage Patterns,
- Trust, Control and Developer Autonomy,
- Feature Usability and Discoverability.

All themes are numbered, and the insights belonging to each theme are presented below the title of the theme, with Figure 12 below presenting a chart with the themes and their corresponding insights.

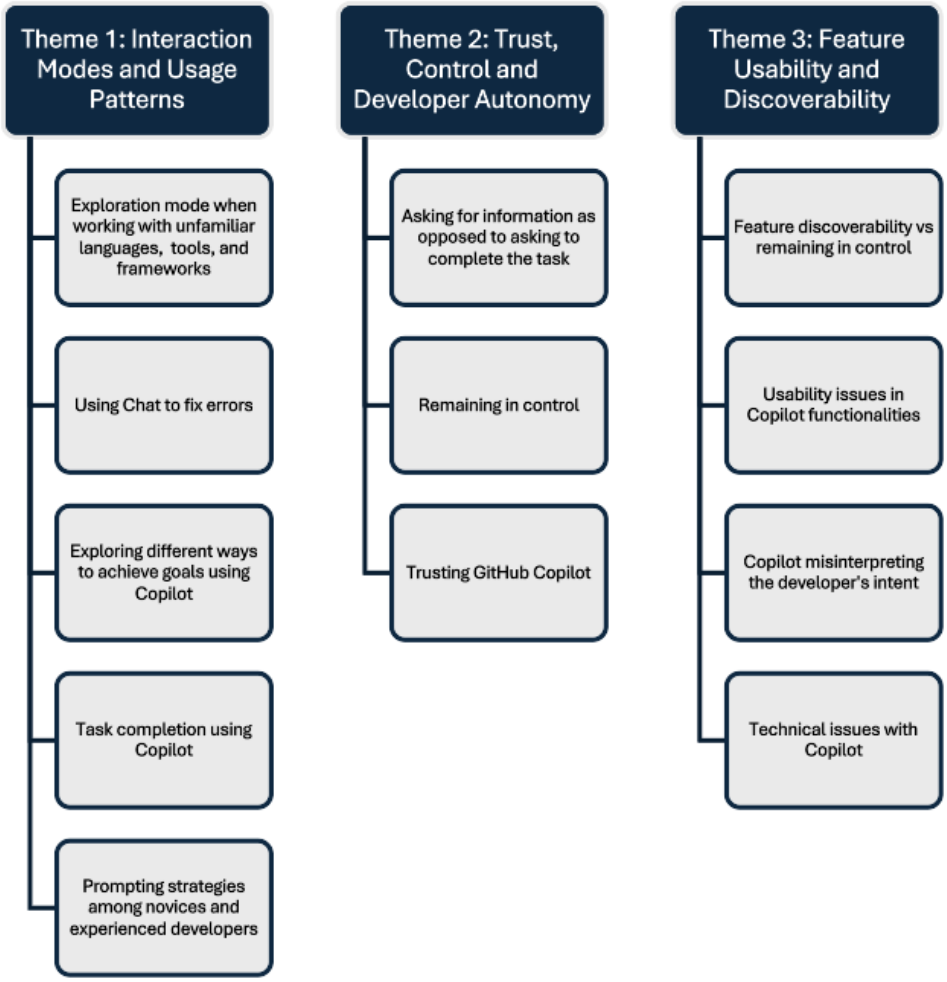

Figure 12. Themes and their corresponding insights

Theme 1: Interaction Modes and Usage Patterns. These insights reflect how developers engage with Copilot in two distinct interaction modes – acceleration mode, where developers use the AI assistant to speed up familiar tasks, and exploration mode, where they rely on it for guidance when facing unfamiliarity – and present analysis of Copilot usage patterns among novice and experienced developers.

Exploration mode when working with unfamiliar languages, tools and frameworks. Participants often entered exploration mode when facing unfamiliar tasks or frameworks, with five experienced and four novice users observed doing so. A novice participant (P5) relied heavily on GitHub Copilot for refactoring and unit testing due to limited knowledge, and thus spending most of the task in exploration mode. Experienced participant P9 similarly used Copilot to understand and implement SOLID principles, input validation, and error handling, spending the session in exploration mode. Several participants (three experienced and one novice) expressed hope that Copilot could help them navigate unfamiliar languages or frameworks. An experienced participant P1 hoped that GitHub Copilot would be able to help them work in C# when they are more familiar with Java, at one point saying "*In Java, I would have used autowire but here, I want to see*

how [Copilot] did it first" while inspecting Copilot's output. Likewise, another experienced participant (P4) said when implementing Copilot's suggestions for refactoring, "*there's something I don't understand in terms of Swagger but I know what Swagger is, so I'll accept [the suggested code]*". Later, the same participant remarked "*I don't have experience in unit testing in C# but I'll trust Copilot*". Trust in Copilot was evident even with participants carefully examining Copilot's output, with participants accepting suggestions even when unsure, like P3 questioning the need for parentheses but still following Copilot's advice. Despite this trust, participants often wanted explanations, treating Copilot as a learning tool, with P1 adding the words "explain your process step by step" to their prompt, saying "*I would love for it to tell me what it's doing*". Exploration mode would sometimes facilitate participants (usually novices) adding features because of Copilot's suggestions, sometimes unnecessarily or erroneously. Novices were more likely to implement Copilot's suggestions even when unsure of their necessity, preferring Copilot's suggestions to their own, sometimes switching strategies based on its output, with P5 accepting Copilot's changes, saying "*maybe not everything is necessary but to be on the safe side, I'll go and implement this*". Copilot itself was a subject of exploration, especially for novices unfamiliar with its capabilities. Some were unsure if Copilot could identify bugs or create files, while experienced participants experimented with its output to test its behaviour. P6 used Copilot's suggestions to generate tests by simply pressing "Enter" and seeing what it offered, saying "*I sometimes press 'Enter' and see what tests Copilot suggests – I have gotten great tests this way*". Overall, exploration mode was common across experience levels, but novices tended to rely more heavily on Copilot as a guide and were more likely to defer to Copilot's solutions rather than trusting themselves, while experienced participants were more confident in their abilities and treated Copilot as a supportive tool, and Copilot itself was explored as well.

Using Chat to fix errors. Six participants (four experienced and two novices) used GitHub Copilot's Chat functionality for debugging and error fixing. This behaviour mostly occurred in exploration mode, where participants were unfamiliar with the errors, languages, or frameworks involved. Only one experienced participant (P1) used Chat in acceleration mode, first giving Copilot specific instructions on what to do to fix the error before ultimately prompting it with "fix this error." Participants often copy-pasted error messages into Chat and added instructions to guide the debugging process. Experienced participants like P11 and P4 carefully examined Copilot's output, using it as a suggestion rather than a definitive fix. Novice P7 cross-checked Copilot's suggestions with the README file, explaining *"[Copilot] tries to cut corners and hard-code values when it would be better not to – you need to proof-read the changes*". P8 used inline chat to debug API errors, trying to manually identify the issue before prompting Copilot to find it. Some participants trusted Copilot to find the root cause of errors (sometimes with prompts like

"Can you tell me why I am getting this error?" and "why do I get the following error?", indicating wanting to learn, or "the files are in the same namespace but it still throws an error", indicating the expectation that Copilot would infer what they want to do), while others used it to automate fixes after manual analysis. P1 and P9 asked Copilot to explain errors step-by-step, indicating a desire to learn through the process. Debugging with Chat was more common among experienced participants, who often combined manual inspection with Copilot's assistance, and three experienced participants used a combination of copy-pasting a code snippet or an error message to the Chat window and adding additional instructions to the prompt for debugging. One time a participant has also entered acceleration mode and copy-pasted the file into Chat and prompted it to "inject service to notesapi", trying to automate the actions they believed they needed to take to fix the error. Analyzing an error manually and then using Chat to fix it has also been observed in P1. Using the inline chat functionality for debugging and fixing errors has been observed in an experienced participant P4 and a novice P8. Novices were less likely to use Chat for debugging, possibly due to unfamiliarity with the tool or uncertainty about its capabilities. Overall, using Chat for debugging was a popular strategy by experienced participants in exploration mode both as a way to find the root causes of the issues and as a way to fix the errors, with prompts frequently including error messages and instructions.

Exploring different ways to achieve goals using GitHub Copilot. Some participants were observed using Copilot to explore different ways to achieve their goals, looking for the best solution. This behaviour usually happened in exploration mode, with five experienced participants and two novices performing actions related to it. The most notable example is the experienced participant P6, who sought to compare different solutions offered by Copilot. The participant would often do so by copy-pasting parts of the solution offered by Chat and using inline suggestions to "*see if [Copilot] would give a different or the same answer*". Similarly, an experienced P1 was observed hoping that Copilot would suggest a better way of doing things than their own attempt when implementing a GetByRating method in the interface, saying "*maybe it will help me do it better*". When implementing the method, the participant also gave some leeway for Copilot in hopes it would help with generating a better answer, remarking "*maybe I shouldn't tell it what the method should do*" when formulating the Chat prompt. In contrast, a novice P3 preferred to think by themselves when deciding whether to use the Axios library or not rather than asking Copilot for advice. When looking for ways to achieve their goals, the participants were also inclined to accept GitHub Copilot's suggestions even when they were not sure about them, with three experienced participants observed doing so, along with a novice P5, who remarked "*I see how fast I am going so I trust [that Copilot] gives good code*". Similarly, P4 said "*I'll still use [the Copilot's suggestion] even though I'm not sure if it's correct*". Overall, using GitHub Copilot to explore different ways to achieve one's goals

or to search for the best solution is quite popular and is done both by comparing Copilot outputs from different ways of prompting and by adjusting prompting strategies.

Task completion using Copilot. Most participants (four novices and four experienced) used Copilot to complete the task, either in full or parts of it. Using various Copilot functionalities to assist with task completion was also popular. Both novice and experienced participants were inclined to prompt Copilot to complete parts of the task for them, dividing the task into parts. For some participants, this may mimic their usual way of tackling their tasks. For others, it may be due to wanting to minimize the likelihood of Copilot making mistakes and wanting to remain in control. As a novice P8 explained, "*I usually try to do things progressively because [Copilot] sometimes makes mistakes*". Some participants would add the README to their prompts, and one participant copy-pasted the task description from the README and used it as a prompt for Copilot. Aside from task completion, participants would also often use Copilot to automate repetitive or easy actions. These include writing subsequent tests once one test has already been written ("*If I get one test working, Copilot can infer how to do the other tests*" – an experienced participant P6), writing docstrings for classes, and creating classes ("*I could've created this whole class myself but I wanted to use Copilot*" – an experienced participant P1). Participants would also use Copilot to complete lower-level tasks, such as figuring out how to run the application (two experienced participants). "Fix using Copilot", a functionality for fixing bugs and errors, was not very popular, with only two experienced participants using it. One experienced participant also said that they would that they would use it for syntax mistakes and build errors rather than logic errors and other higher-level issues, giving more support for the hypothesis that Copilot may perform poorly in error fixing and debugging. The participant also said that they would prefer using Chat for debugging. Another experienced participant tried "Fix using Copilot" but in the end fixed the issue manually, possibly because Copilot gave a large popup window with a lot of code or because they were using Copilot's output as a suggestion for their own solution. "Explain using Copilot" was very unpopular as well, with only one experienced participant using it in an attempt to debug Copilot's code. Participants tended to be pleased with Copilot's output, with experienced participant P11 saying "*Copilot also explains the code and adds comments to the code so we can easily understand what it does*", novice P5 calling Copilot's suggestions "*quite good changes*" and experienced participant P4 remarking "*I think this [error handling] is great concerning the common errors*". Overall, participants were likely to prompt Copilot to complete their task step-by-step, writing one step into one prompt, while not being likely to use specialized functionalities like "Fix using Copilot" and "Explain using Copilot".

Prompting strategies among novices and experienced participants. Some differences have been observed among the prompting strategies in novices and experienced

participants. As opposed to novices who would often hope for Copilot to infer what they want to do, experienced participants tended to make their prompts specific. Half of experienced participants felt like Copilot requires a very specific prompt to generate acceptable output. P6 explained "*I noticed that if you give Copilot specific packages and tell the specific tests to write, it gives you better results <…> If your prompt is not clear enough for Copilot, it is more likely to give you trash*". Grammatical correctness of the prompt was important to some participants too, with experienced P1 and novice P8 both ensuring that their prompts are grammatically correct and do not contain spelling mistakes, with P1 commenting "*The English may not be fully correct but I hope it understands*". This may point to participants' mental model of AI assistants as a tool requiring effort to generate quality output. Notably, many participants would talk to Copilot in the same way as they would talk to a human when prompting Chat or Edits. Their prompts would feature phrases such as "Help me run this program", "Can you show me how to use…", "Please check the <…> files and ensure that…", "Why do I get the following exception?", "Can you replace types with nullable types where it makes sense?", "Can you edit the file for me to do that?". Some participants would choose this strategy due to not being sure if Copilot would be able to do the task and trying to explore Copilot's capabilities. This strategy was about as popular among novices as it was among experienced participants, with four novices and three experienced participants engaging in it at least once. Using comments as prompts was significantly less popular strategy, with only two novices and two experienced participants engaging in this behaviour. However, one experienced participant heavily preferred using comments as prompts over Chat and Edits, while one novice only turned to Chat after using comments as prompts gave only one suggestion. This suggests that while comments as prompts may not be utilized by many, it is used heavily by those who do. Prompt suggestions displayed at the home screen of Chat or above the input box for the prompt in Chat thread were used by one novice and one experienced participant, with the novice modifying the prompt before running. Both participants were using the suggestions for test writing. Overall, while experienced participants preferred specific prompts, speech patterns resembling human-human interaction in prompts was popular among both participant groups, while comments as prompts and prompt suggestions were used rarely.

Theme 2: Trust, Control and Developer Autonomy. Insights in this theme reveal how novice and experienced developers balance automation and autonomy, showing how developers choose between automating tasks and completing them on their own.

Asking for information as opposed to asking to complete the task. Some participants preferred asking GitHub Copilot for guidance or example code rather than requesting it to complete tasks directly. This behaviour was observed in both novice and experienced users with similar frequency, with five experienced and four novice participants doing so

at least once. Experienced participants such as P10 and P9 asked for explanations of programming concepts such as SOLID principles (a set of software design principles aimed at making code more readable and maintainable) and instructions on how to refactor the app in the Chat, using prompts such as "how can I refactor…". As they were reading through the response, P9 remarked, "*I'm trying to figure out how much do I have to do manually or if there's a way to do it easier*" and "*I'm going through the Chat explanation of the refactoring and I'm trying to see if there's a way for me to ask it to do a part of the process for me*", indicating that for some developers, the hesitation to ask Copilot to complete the tasks for them is due to unfamiliarity with its functionalities and capabilities. Other experienced participants used Copilot for instructions on how to perform certain actions such as removing tests or information on terminal command syntax such as a command for installing a package. P9 also used Chat to assist with debugging, later explaining, "*I didn't know if [Copilot] could debug, so I asked it 'can you debug this?'*", indicating unfamiliarity and potential functionality discoverability issues. However, turning to Copilot for help with issues was more prevalent among novices, with P2 using Chat to try to figure out how open the React app in the browser, P7 prompting Chat with "I get an error that I cannot fetch the data in my home.jsx" when a mistake in the API call parameters leads to an error response from the API, and P8 using inline chat to modify the code to log the API response when trying to fix an issue with the API call. Novices were also more likely to ask Copilot to identify problems, likely due to less experience with debugging or experienced participants being more wary of Copilot's debugging capabilities. Sometimes, novices would hope for Copilot to infer that they are trying to identify the issues – for example, P3 prompted Chat with "I am trying to get the weather data but I am getting the location key as undefined". Novice P8 asked for instructions on writing integration tests but chose not to delegate the task due to concerns about Copilot placing the files in the wrong directories, later saying they would prefer Copilot to tell them to create the files rather than creating the files by itself. This reflects a broader trend of developers valuing manual control just as much as convenience. Overall, both groups used Copilot for information-seeking, but novices did so more for debugging, influenced by wanting to remain in control, unfamiliarity with Copilot's features and capabilities and concerns over the quality of task completion.

Remaining in control. All participants displayed behaviours that would allow them to remain in control of the code they were working on. These behaviours included modifying Copilot's inline suggestions after accepting them, implementing part of the generated code from Chat, and examining Chat output before implementing it. When the inline suggestion only partially matched the code an experienced participant P10 wanted, they typed out the part they wanted instead of accepting the suggestion. On another occasion,

Copilot provided them with inline suggestions for multiple constructors, and the participant accepted some while rejecting others, making new lines to get more inline suggestions. While novice P3 relied heavily on inline suggestions and modifying them to complete their task, experienced participant P9 relied on Chat instead and would accept and modify the suggestions generated by Chat, at one point saying *"[the suggestion] wasn't completely correct but it had the correct of idea of what I wanted to do*". Similarly, P3 would copy-paste the code from Chat line-by-line and fix errors and modify coding conventions. Conversely, an experienced participant P6 explained "*I sometimes copy-paste part of the Chat solution to see if Copilot would give a different answer*", pointing to a wish to explore different solutions and pick the one they like most. A novice P7 thoroughly read Copilot's answer in the Chat and cross-checked with the README to ensure code quality and correctness. Two experienced participants would examine the code provided by Chat before manually implementing or copy-pasting the relevant parts of the suggestion, with one of them explaining that they do this because they need to understand the code. Likewise, one novice participant used a combination of reading the README themselves and asking questions from Chat when they had a question regarding the codebase, pointing to the need to understand the code. Another novice participant prompted Chat to check if "the application has correct error handling", remarking "*[Copilot] already did some error handling but I need to make sure it is working correctly*". Overall, all participants would interact with Copilot in a way that allowed them to remain in control as opposed to fully automating the development process, and the reasons for that are ensuring code quality and correctness, need to develop and maintain an understanding of the codebase, and exploring different options to achieve their goals.

Trusting GitHub Copilot. On the other hand, many participants (five novices and four experienced) have also demonstrated behaviours where they would defer to Copilot instead of remaining in control. These behaviours included accepting Copilot's suggestions without checking and seeing what changes Copilot made after accepting them. When faced with unfamiliar situations or when working in environments with which they have little experience, some participants would trust Copilot to provide the correct solution. For example, one novice at one time remarked "*I'll just copy-paste it and see if it works*". Similarly, an experienced participant at one point said "*I'll do what Copilot says and see if it works*". One novice participant checked the code by running it after only quickly skimming it before accepting. They ran it way after they had accepted the Copilot's suggestion, made manual modifications and accepted other suggestions, trusting that Copilot will not introduce errors or bad practices. After the initial skimming, the aforementioned participant explained "*I just want to see if it opened a stream that would need to be closed but I guess we'll see later*". Similarly, an experienced participant trusted Copilot enough to accept a suggestion they were not sure about, saying "*I don't know if*

this breaks SOLID but I'll check the [SOLID principles] at the end". This shows that even when developers trust Copilot, they still try to check the generated code, even if they defer the checking for later. As an experienced participant P11 said "*I'll verify, because we always have to verify the code generated by AI*". However, the trust in Copilot is seemingly high among some participants, as one novice and one experienced participant were observed making significant changes to the codebase to accommodate Copilot's suggestions, such as downgrading a .NET package version to make the project compatible with the code suggested by Copilot. The participant who did this later explained that they would not have been as willing to downgrade packages had they been more familiar with .NET, supporting the hypothesis that developers are more likely to defer to Copilot when faced with unfamiliarity. One experienced participant trusted Copilot by adding the API key to the file for Copilot to use, explaining that this is an artifact specifically for Copilot that does not get included in the production code, and that they only do such things when working with secured LLMs hosted on-site in their workplace. Overall, trusting Copilot and deferring judgement to it was common among both novices and experienced participants and was usually done when facing unfamiliarity and code check was usually still performed at a later stage.

Theme 3: Feature Usability and Discoverability. These insights highlight the usability challenges faced by novice and experienced developers and the strategies developers use to overcome issues.

Feature discoverability vs remaining in control. Participants would sometimes opt to not use Copilot due to wanting to remain in control, with three novices and three experienced participants showing such preference. A novice P5 said, "*maybe I am a bit old-fashioned but I still go and check the code myself – if you leave it all to Copilot, you've lost control*", with experienced participant P1 echoing a similar sentiment – *"I have the old-school in me, I want to figure out things myself"*. Experienced participants P9 and P10 echoed similar sentiments, with the latter citing clashes between their personal code formatting preferences and the code formatting in the code generated by Copilot as the reason for checking and modify Copilot's code. The participant also expressed a wish to remain in control – "*I don't want to be in the backseat*". Most notably, all participants chose to familiarize themselves with their task manually by reading the README. The experienced participant P6 explained, "*I spend a lot of time reading the instructions so that I could later tell if Copilot's response was correct or not*". Other instances of participants not using Copilot were analyzing the code manually (two experienced, one novice), creating files and folders manually (three experienced, two novices), manually comparing the current code and the Copilot suggestions instead of relying on the diffs provided by Copilot, such as copy-pasting the suggestion into the file to compare with the current version (one experienced, two novices), trying to debug or solve issues manually (three

experienced, four novices), manually moving the code to a different file (one novice), and manually correcting Copilot's mistakes instead of asking Copilot to do it (two novices). Some hesitated to use Copilot for error fixing due to perceived poor performance, with at least one (experienced) participant expressing such views. Another participant preferred manual debugging to better understand the code and find opportunities to improve it. Importantly, none of the participants were bothered by doing some things manually, with the novice P5 stating so explicitly – "*I don't think there will ever come a time when coding will be fully AI but as an assistant, [Copilot] is good*". However, sometimes, participants would not use Copilot also due to poor feature discoverability. The participants would sometimes struggle to discover Edits or Chat (two experienced, three novices), with Edits being notably undiscoverable, with many participants using Chat when they could have used Edits. Some participants would not discover the Chat or Edits functionalities until being suggested to use them, and sometimes they would not realize they can use them to achieve their goals – for example, P1 wanted to search for information on SOLID principles online before being suggested to use Chat. The "Add files" functionality in Edits and Chat was also hard to discover for participants, with participants sometimes typing the file names or copy-pasting the file contents in the prompt instead of using it. Overall, while wanting to remain in control was the main motivator for participants to not use Copilot, feature discoverability also played a significant role, with both reasons being similarly prevalent among both novices and experienced participants.

Usability issues in Copilot functionalities. All but one participant encountered usability issues with GitHub Copilot, most commonly due to poor feature discoverability. Participants often struggled to find or understand functionalities like "Add files," "Add folder," Chat, Edits, and "Explain using Copilot." The "Add files" functionality was hard to discover for four novices and two experienced participants, with participants often choosing to copy-paste the file contents into the prompt instead. One experienced participant also added "the affected files are" to their prompt, followed by file names, and one novice formulated their prompt as "take the NotesApi.cs file and refactor it to follow the SOLID principles". A novice P7 tried to look for the "Add files" option in the top bar and the top-right corner of the Chat panel rather than in the prompt text box area. Some misunderstood its purpose, thinking it was for creating new files, or didn't realize actions like "Accept changes" were required to apply suggestions. One experienced participant said that they were not aware that the current file is always included in the context. Another significant usability issue was confusion over whether Copilot had saved or implemented changes, leading to overlooked errors or uncertainty regarding Copilot-generated new files, with four experienced and two novices experiencing it. For one novice participant, Copilot drew attention to and fixed a syntax issue, however, the changes were not saved, leading to the user continuing on with the mistake in their subsequent edits. Meanwhile,

an experienced participant was confused over where Copilot created the new file and whether it was created. As mentioned earlier, several participants did not discover Chat and/or Edits without being suggested to use them. One novice participant also had trouble trying to locate Copilot itself, and another novice participant struggled to locate previous threads in Chat. One experienced participant was unaware of the "Explain using Copilot" feature and speculated that it may be due to their company's policy on AI tools. A couple of experienced participants also struggled to understand how to perform their desired actions, with one facing issues with accepting a Copilot-generated file or accepting an inline suggestion, while another struggled to understand whether they should copy-paste the terminal error output in the Chat prompt for debugging. One novice and one experienced participant also had issues understanding Copilot's feedback and output, and one experienced participant made sure to manually check that Copilot's changes were applied. The top-bar options, specifically "Prompt" and "Changes", were confusing to one experienced participant, who said that they can be "*tricky to first-time users*". Overall, both novices and experienced participants faced usability issues when working with Copilot, with experienced participants facing more issues related to saving changes, and novices facing more discoverability issues.

Copilot misinterpreting the developer's intent. Some participants encountered issues when Copilot misinterpreted the intent behind their prompt. This issue, as well as related issues, was faced by four novices and three experienced developers. When the novice P8 asked Copilot to add styling to the data displayed in their React app, they assumed that Copilot would do so by using inline CSS. Instead, Copilot created new CSS files for the styling. This has disappointed the participant, with them commenting "*It created a new file but I don't really want it <…> I don't trust it, it sometimes messes things up*". Later, the participant explained "*I didn't think it was going to create a new file, I thought it would add inline CSS because I didn't tell it to create a new file*". This shows that correctly guessing the developer's intent is one of the components of good developer experience for AI coding assistants. Two experienced participants, when they were using comments as prompts, had Copilot continue generating comments instead of acting on the prompt presented in the comment written by the participant. In these situations, both participants would accept the generated comments that Copilot would provide as suggestions, hoping that Copilot would eventually act on the prompt. However, Copilot would stop the suggestions instead and never act on the prompt. This would leave the participants confused and disappointed that they are unable to use their preferred method of prompting Copilot to perform their tasks and would begrudgingly switch to other methods such as Chat or Edits. Similarly, Copilot would sometimes lack context-awareness and would thus provide incorrect suggestions – this issue was faced by three novices and one

experienced participant and usually manifested in Copilot using the wrong package manager in its response (such as yarn instead of npm) or suggesting .NET packages that are incompatible with the project's .NET version. Notably, while most participants who encountered this issue would get misled by the incorrect suggestion, novice P3 noticed Copilot's mistake – "*It suggested npm, I guess I should use yarn instead*". The participant did not seem bothered by the faulty suggestion. One experienced participant faced issues when they expected Copilot to give inline suggestions when they accepted the inline suggestion for a method GetByTitle and then manually changed it to GetByRating but it did not do that. The participant said that in this situation, they expected Copilot to give them suggestions on how to complete the GetByRating method. Output that does not match developer's expectations but is otherwise correct can also lead to disappointment, as evidenced by the experienced P1 being confused when Copilot did not refactor a method that could not be further refactored. Faulty or context-unaware Copilot output may also mislead developers and get them side-tracked from their goals by, for example, making them implement unnecessary methods. Overall, issues regarding context-awareness or incorrect prompt interpretation were faced by around half of both novices and experienced developers, and had the potential to negatively affect the developer experienced by confusing the developer or misleading them into unnecessary work.

Technical issues with Copilot. Technical issues with Copilot were rare, encountered by only two novices and one experienced participant. The technical issues were mostly related to empty output (where the only changes to a file made by Copilot were adding new empty lines) and Edits not being able to find the files. For the latter error, one experienced participant performed successfully error recovery by prompting Edits with "Add a new method GetByRating to the INotesRepository interface…" instead of copy-pasting the Chat output. Mismatches between Copilot-generated code in different files, such as differences in method names or a method being present in the interface but not in the class that implements it, were also encountered by one novice and one experienced participant. Long response times by Chat and Edits were also noted by some participants. Technical issues would often frustrate the participants and require more time and mental load to fix or work around them, or to debug code discrepancies between different files. Overall, while technical issues were rare, they still had the potential to negatively affect the developer experience and introduce errors to the code.

4.2.3. Post-Test Interview Questions

The following chapter describes the results obtained from the post-interview questions.

Novices generally expressed positive views on GitHub Copilot, praising features like Edits and context awareness, helpfulness with error-fixing and anticipation of the developer's intentions but also expressed differing opinions about code quality and strongly

emphasized the need to manually verify suggestions and make modifications if necessary. Experienced participants were more mixed, appreciating Copilot's code-focused nature and ability to help when working with unfamiliar languages, general usefulness and integration into IDEs, but also expressing discontent about inline suggestions being distractive sometimes and inconsistent output quality.

Novices used terms like "empowering," "useful," and "quick," while also choosing certain nuanced expressions like "has potential", while experienced participants chose descriptors like "precise," "reliable," and "helpful," with some noting issues like "context-unaware" or "silly." Both groups chose words related to Copilot's speed, such as "fast" and "quick". One of the descriptors chosen by the novice P7 was also "intimidating/scary", with the participant explaining that *"[GitHub Copilot] changed the whole game for coding <...> you now need to know less [technical details in order to create software], you can now create beautiful and user-friendly interfaces even though you don't know much about it*", indicating a largely positive view of Copilot that is influenced by its broader effects on software development rather than only momentary user experience or technical capabilities. An experienced participant P11 described Copilot as "reliable", "secure", and "consistent", explaining that Copilot's generated code is secure if generated using an on-premises LLM rather than a public LLM accessed through the Internet connection, showing a strongly positive view of the security of GitHub Copilot. Overall, while both groups expressed positive sentiments in their choice of descriptors, experienced participants appeared more focused on specific qualities and areas such as security, easy-of-use and speed, while novices were more likely to describe broader emotional responses and future potential or Copilot's impact on software development as a whole.

Both groups emphasized UI/UX enhancements (such as improvements to adding context and the top bar options), better discoverability, and improved output quality. Novices also requested more control features like progressive prompting (including progressive voice chat) that would allow the developer to complete the task using Copilot step-by-step while thinking about what they want to do, and a "codebase off" switch ("*with ChatGPT, you can have a fresh conversation where it doesn't know your codebase – here, it's always on but I don't always need it, sometimes I just want to understand a concept or something but it generates code for me*"). Experienced participants suggested real-time code verification to prevent the developer from writing buggy code and tighter IDE integration, such as a possibility to prompt Copilot from the top bar.

When asked what participants liked about GitHub Copilot, novices highlighted inline suggestions, Chat functionality, help with error fixing and debugging and context awareness, while experienced participants appreciated productivity gains, code comprehension support, and seamless IDE integration. An experienced participant P6 also mentioned that

they liked using Copilot for converting between languages (primarily for code comprehension purposes). As P6 explained, "*If I had to do something with Java – and I don't like Java – I have to understand the code, so instead of breaking my head, I would open Chat and say 'explain this code for me' <…> and then I'd ask it 'convert this code to TypeScript for me' and it does so very nicely, and then I am able to understand*".

When asked about what they disliked, novices pointed to hallucinations, long responses, Copilot misunderstanding the prompt or intent and distracting suggestions. P3 said that Copilot may "*start to suggest a lot of different ways and approaches [when debugging] and you get lost*", and P7 talked about inline suggestions sometimes being distracting – "*sometimes I just want to write things manually but then there are all these suggestions popping up constantly*", potentially pointing to the different effects of GitHub Copilot in acceleration and exploration modes as well ("*Many times, it's really useful and I do use it quite a lot and I [accept the suggestion] if it looks correct. But then sometimes it's also a little bit distracting.*"). The participant also noted that inline suggestions are most unhelpful when they are starting to write in a new empty file and when writing API endpoints. Meanwhile, experienced participants criticized lack of file context awareness (which files the project contains, are there any unsaved changes and similar), poor performance in new files, and Copilot's insufficient technical knowledge, as evidenced by Copilot suggesting package install commands that do not correspond to the project's .NET version. Both groups expressed a desire for more reliable and context-sensitive behavior.

Tables 12 and 13 in Appendix 8 contain all the descriptor words used to describe GitHub Copilot by novices and experienced participants respectively.

5. Discussion

5.1. Differences of Github Copilot Developer Experience Among Novices and Experienced Developers (Research Question 1).

Both novice and experienced online developer survey respondents generally reported a positive developer experience with Copilot. However, differences in views on security, trustworthiness, pleasantness, and context-awareness emerged. Novices found Copilot more secure, trustworthy, context-aware and adaptive, while experienced respondents were more likely to rate it as pleasant and likeable. Novices were more satisfied with the undo/redo functionality, whereas experienced respondents were more satisfied with mistake prevention. Novices wanted better debugging and prompt comprehension, while experienced respondents sought enhanced context-awareness and testing features, echoing the findings by Liang et al, who found that developers wished for better context-awareness and better technical knowledge from AI assistants [7]. This shows that experienced developers were more likely to run into issues with Copilot's context-awareness, possibly due to regularly working on more complex tasks than novices. Moreover, novices were less concerned with the security aspect, which likely positively affected their developer experience. Experienced user study participants also pointed out the issues with context-awareness as well as code quality during post-test interviews, however, both novice and experienced participants remained largely positive in their opinions on Copilot. While experienced participants were more likely take notice of Copilot's context-awareness, the prevalence of issues related to context-awareness in user study was similar among both groups. This, combined with novices being more likely to emphasize the need to verify the code provided by Copilot, may show that experienced developers are more used to automation of processes (for example, with the help of CI/CD pipelines), and thus expect their experience with Copilot to require less manual effort. Experienced participants also suggested real-time mistake prevention as a potential new feature for Copilot, underscoring the importance of context-awareness, code quality and the depth of Copilot's technical knowledge. This suggestion resonates with Barke et al suggesting better validation of generated code [2]. One novice user study participant suggested a progressive prompting or voice chat feature, echoing a the lack of step-by-step task completion feature noted in previous research [10]. The user study also revealed motivations behind (not) using various Copilot features, with the main motivation among both participant groups being the wish to remain in control. Remaining in control and using the AI assistant as a helper tool rather than a tool that completely removes the need for human intervention was also noted in previous research, such as Sergeyuk et al [10] and Tan et al [3]. However, issues with feature discoverability also played a role, especially among novice participants. Feature discoverability issues have also been noticed by Tan et al [3]. Lack of discoverability may also explain the unwillingness of participants to use certain (right-click) features

such as “Fix using Copilot”. Likewise, usability issues such as issues with saving and applying changes were more prevalent among experienced participants. Regarding the prompting strategies, both participant groups were inclined to prompt Copilot to complete a part of the task for them instead of completing the entire task, likely due to wanting to remain in control or due to the ease of verifying the generated code. Using comments as prompts and using prompt suggestions was done rarely. However, those who utilized comments as prompts as a prompting strategy relied on it heavily and it was often their preferred prompting method. The aforementioned prompting method was more prevalent among experienced participants. While experienced participants preferred specific, more detailed prompts, both participant groups tended to talk to Copilot as if it were another human. While sometimes this was done out of uncertainty, it may also point to underlying mental models regarding chat windows and chatbots. Some participants were also observed putting significant effort into the wording of their prompts, echoing the findings by Pinto et al, who found that some developers may feel that it is important to use the coding assistant “correctly” [9]. Technical issues were rare but could still negatively affect developer experience. Unlike in Zhou et al, who found that most-reported technical issues with Copilot were related to network connectivity issues [8], the issues encountered by participants were related to Copilot’s output generation. Overall, **while both novices and experienced developers had largely positive experience and shared many key developer experience elements such as wanting to remain in control, similar prompting strategies and issues with feature discoverability, certain differences emerged, as experienced developers were more critical of Copilot’s context-awareness, were more focused on the security aspects of Copilot, and were more likely to wish for Copilot to assist in error prevention**.

5.2. Taking Into Account the User’s Programming Experience (Proficiency) Level (Research Question 2).

A big factor in determining whether the AI coding assistants should take into account the programming experience level of the developer using them is the tendency to remain in control vs trusting the assistant. During the developer user study, both novices and experienced developers would utilize Copilot while remaining in control, with no significant differences between based on programming experience level. The reasons for remaining in control, such as ensuring code quality, having an understanding of the codebase, and exploring different ways to achieve the desired goals, were also largely consistent among both experience levels. On the other hand, the developer user study has also shown that a developer trusting Copilot and deferring to the assistant’s suggestion rather than relying on their own judgement has also been frequently observed among both novices and experienced developers. Both groups of developers were likely to trust Copilot when faced with unfamiliarity, such as programming languages or frameworks they

have little to no experience with. This suggests that while taking into account the developer's overall programming experience level may not bring a lot of benefits, AI coding assistants could tailor their behaviour and suggestions based on how much experience the developer has with the particular languages, frameworks, packages, platforms and other tools that are being used in the current project. For example, if an experienced developer is working on a .NET project while not being well-versed in .NET, the AI assistant could provide more detailed instructions and explanations. Moreover, in such situations, the assistant could also provide its suggestions in smaller parts, allowing the developer to inspect the code and potentially modify it, thus remaining in control and developing a better understanding of the unfamiliar technologies they are working with. These types of learning-facilitating interactions could also be beneficial to novice developers, alleviating fears of AI coding assistants being detrimental to beginners observed by Pan et al [12]. It could also address the wishes for personalization and requests for "progressive prompting" where the developer uses AI to achieve their goals step-by-step. Moreover, it could foster AI coding assistant adoption, as it would create the feeling of the assistant "working together" and being helpful for the developers. To maintain personalization and take different ways of working into account, this kind of "learning mode" should be suggested for developers based either on the analysis of their interactions with the coding assistant (similar suggestions were previously made by Mozannar et al [4]) or developer's answers to the questions the assistants would ask at the start of their work with a codebase, or switched on and off manually by the developer themselves. Moreover, as some user study participants who had little to no experience with GitHub Copilot displayed confusion and were unsure of its capabilities, similar "learning mode" or onboarding process could be helpful for those who are new to working with the AI assistant. Similar "learning mode" for ChatGPT has been proposed by Sun et al [13]. Overall, **while there is little evidence to suggest that tailoring an AI coding assistant's behaviour based on the developer's overall programming experience level would be beneficial, it might be useful to do so based on the developer's familiarity with the technologies they are utilizing at a given moment**.

5.3. Acceleration and Exploration Modes Among Novices and Experienced Developers (Research Question 3).

While both novices and experienced developers entered **exploration** mode often, the reasons for entering it differed. During the developer user study, all participants spent at least some time in exploration mode. However, novices were more likely to enter exploration mode due to unfamiliar tasks such as refactoring or when facing issues and debugging, while in experienced developers, exploration mode was often triggered by facing unfamiliar technologies and frameworks. Exploration mode would often mean asking for information on how to achieve a specific goal rather than asking Copilot to complete the

task itself, with novices being more likely to ask for information on basic tasks such as running the application or making a HTTP request, while experienced developers would be more inclined to explore how to achieve a familiar task in an unfamiliar framework or when encountering knowledge gaps such as not being strongly familiar with SOLID principles. Novices were also more inclined to enter exploration mode when debugging, relying on Copilot to help them solve the issues, and also were more likely to add features based on Copilot's suggestions. While previous research such as Barke et al correlate exploration mode with unfamiliarity and uncertainty [2], the difference between the reasons to enter exploration mode based on the developer's experience level has not been explored. Experienced participants were more likely to use Chat and inline chat functionalities when fixing errors, and were more likely to use Copilot for exploring different ways of achieving their goals. The online developer survey indicated that novices are more likely to use Copilot for debugging, which aligns with the findings of the user study and echoes findings by Sun et al about student programmers being keen to use ChatGPT for debugging [13]. **Acceleration** mode was noticeably less popular among both novices and experienced developers during the developer user study, with two novices and four experienced developers spending some time in acceleration mode. Most sessions where acceleration mode was present in did not feature it prominently, with exploration mode still taking the majority of the session time. The popularity of each mode was not explicitly explored by Barke et al [2]. In the online survey, experienced developers were more likely to use AI tools for writing code and tests. Both of these activities may align with acceleration mode, however, as evidenced by the developer user study, they are often present in exploration mode as well. Similarly, development and testing were the two most popular SDLC stages to use AI tools in during online developer survey, which, as evidenced by the user study, are likely to be performed in exploration mode. Experienced user study participants were more likely to enter acceleration mode, and during it, they would often use Copilot to complete the task or parts of the task (such as refactoring, creating tests or writing comments), create files and classes faster, and sometimes to automate the process of error fixing or check if their code did not violate SOLID principles. Experienced participants were likely to check the Copilot's output thoroughly during acceleration mode, and some would use the output as a basis for writing the rest of the code themselves. For novice participants, acceleration mode usually meant asking Copilot to implement minor changes (such as adding styling to the React app, adding nullable types or adding code for displaying the error messages in the console), or asking Copilot to check if the error handling and input validation are implemented correctly (after checking manually). Overall, **exploration mode is significantly more popular among both novices and experienced developers than acceleration mode, with experienced developers searching for more high-level assistance related to programming concepts or**

technologies, and novices relying on Copilot to help them with error-fixing and syntax issues. Acceleration mode is more frequent with experienced developers, who often seek to automate tasks or reach their goals faster, while novices tend to make small modifications.

6. Conclusions

This chapter summarizes the thesis by presenting reflections, potential threats to validity and possible areas of future research.

6.1. Reflections

The thesis has explored the differences in GitHub Copilot developer experience and the usage of acceleration and exploration modes among novice and experienced developers, and investigated whether the developer's programming experience level should be taken into account by AI coding assistants. The research methods consisted of an online developer survey and developer user study. Both similarities and differences in the developer experience were identified. Novice and experienced developers shared wanting to remain in control, used similar prompting strategies and faced issues with feature discoverability. However, novice developers were less critical of context-awareness, security and error prevention capabilities. Exploration mode was more common than acceleration mode among both novices and experienced developers, with experienced developers utilizing it more for high-level assistance related to programming concepts and technologies, while novices explored error-fixing and syntax issues. Experienced developers were more likely to enter acceleration mode as they sought speed and automation. While it was found that taking into account the developer's overall programming experience level may not be beneficial, it might be useful to take into account the developer's familiarity with the technologies being used, or the AI assistant itself. Therefore, a "learning mode" that would allow the developer to familiarize themselves with programming languages or frameworks, or even the AI coding assistant, was proposed. This paper provides insights into and continues the newly-emerged conversation on the developer experience of AI coding assistants while also identifying potential future AI coding assistant development directions.

6.2. Threats to Validity

Certain potential limitations apply to the findings of this paper. Both the online developer survey and the developer user study had small sample sizes (28 respondents and 11 participants respectively), which raises concerns over possible wide-scale applicability. Moreover, while the user study and the online survey data was being analyzed, several newer versions of the GitHub Copilot plugin for various IDEs used by the user study participants have been released, containing new functionalities and changes in the user interface. Therefore, some findings that apply to the version of GitHub Copilot that was used by the user study participants and online survey respondents may not apply to the newer, updated versions. Regarding the user study, while efforts were made to ensure that the tools used in the task are at least somewhat familiar to each participant, the level of experience with the programming languages and frameworks in which the tasks were

completed (C#/.NET and JavaScript/React) differed among both novice and experienced participants, with some participants having very little familiarity with the languages and frameworks while others had been working with them frequently, which may also affect the results. Novelty bias regarding AI coding assistants may have also affected the user study participants and survey respondents, thus affecting the results. For online developer survey, selection bias may have affected the results, as respondents self-selected to answer the survey. As the survey was distributed over online channels, those who visit online communities more often, as well as those with a positive outlook on AI tools, may be overrepresented. Moreover, the survey only included respondents who answered "Yes" to using GitHub Copilot for their job, which may exclude developers who may have used Copilot in the past, tried it once but never used again, or use Copilot for their personal, academic or other non-work projects, thus potentially altering the survey results, especially toward more favourable or informed opinions. In the user study, sampling bias may be present due to reliance on the author's networks and volunteer bias could also affect the results, as participants who opt into a user study involving GitHub Copilot may already have a positive disposition toward AI tools. Lastly, think-aloud method was used during the user study, which may introduce limitations related to vocalizations of internal thoughts.

6.3. Possibilities for Future Research

The research presented in this paper may be extended and continued in several directions. User study and developer surveys regarding the developer experience of various AI coding assistants and other AI-based tools used for coding and software development may be conducted on a larger scale, with more participants/respondents to achieve better applicability. The user study may also be conducted using different programming languages and frameworks, such as Python or Node.js, and with participants that are moderately or highly familiar with the technologies being used, ensuring better control of participant background. Developer experience of developers who use AI coding assistants frequently could be compared to the developer experience of those that use them rarely or are using it for the first time, thus exploring and highlighting the challenges that may be faced by new users of AI coding assistant tools. Moreover, the use of conversational prompts such as "Can you help me…?" and inclination for human-like interaction among some developers when interacting when interacting with AI assistants could be explored further.

References

[1] K. Daigle, ‘Survey: The AI wave continues to grow on software development teams’. Aug. 20, 2024. Accessed: Sept. 17, 2024. [Online]. Available: https://github.blog/news-insights/research/survey-ai-wave-grows/

[2] S. Barke, M. B. James, and N. Polikarpova, ‘Grounded Copilot: How Programmers Interact with Code-Generating Models’, *Proc. ACM Program. Lang.*, vol. 7, no. OOPSLA1, pp. 85–111, Apr. 2023, doi: 10.1145/3586030.

[3] X. Tan, X. Long, X. Ni, Y. Zhu, J. Jiang, and L. Zhang, ‘How far are AI-powered programming assistants from meeting developers’ needs?’, 2024, *arXiv*. doi: 10.48550/ARXIV.2404.12000.

[4] H. Mozannar, G. Bansal, A. Fourney, and E. Horvitz, ‘Reading Between the Lines: Modeling User Behavior and Costs in AI-Assisted Programming’, in *Proceedings of the CHI Conference on Human Factors in Computing Systems*, Honolulu HI USA: ACM, May 2024, pp. 1–16. doi: 10.1145/3613904.3641936.

[5] P. Vaithilingam *et al.*, ‘Towards More Effective AI-Assisted Programming: A Systematic Design Exploration to Improve Visual Studio IntelliCode’s User Experience’, in *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, Melbourne, Australia: IEEE, May 2023, pp. 185–195. doi: 10.1109/ICSE-SEIP58684.2023.00022.

[6] T. Miah and H. Zhu, ‘User Centric Evaluation of Code Generation Tools’, in *2024 IEEE International Conference on Artificial Intelligence Testing (AITest)*, Shanghai, China: IEEE, 2024, pp. 109–119. doi: 10.1109/AITest62860.2024.00022.

[7] J. T. Liang, C. Yang, and B. A. Myers, ‘A Large-Scale Survey on the Usability of AI Programming Assistants: Successes and Challenges’, in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, Lisbon Portugal: ACM, Feb. 2024, pp. 1–13. doi: 10.1145/3597503.3608128.

[8] X. Zhou *et al.*, ‘Exploring the problems, their causes and solutions of AI pair programming: A study on GitHub and Stack Overflow’, *J. Syst. Softw.*, vol. 219, p. 112204, Jan. 2025, doi: 10.1016/j.jss.2024.112204.

[9] G. Pinto, C. De Souza, T. Rocha, I. Steinmacher, A. Souza, and E. Monteiro, ‘Developer Experiences with a Contextualized AI Coding Assistant: Usability, Expectations, and Outcomes’, in *Proceedings of the IEEE/ACM 3rd International Conference on AI Engineering - Software Engineering for AI*, Lisbon Portugal: ACM, Apr. 2024, pp. 81–91. doi: 10.1145/3644815.3644949.

[10] A. Sergeyuk, Y. Golubev, T. Bryksin, and I. Ahmed, ‘Using AI-Based Coding Assistants in Practice: State of Affairs, Perceptions, and Ways Forward’, *Inf. Softw. Technol.*, vol. 178, p. 107610, 2024, doi: 10.1016/j.infsof.2024.107610.

[11] D. Jayagopal, J. Lubin, and S. E. Chasins, ‘Exploring the Learnability of Program Synthesizers by Novice Programmers’, in *Proceedings of the 35th Annual*

ACM Symposium on User Interface Software and Technology, Bend OR USA: ACM, Oct. 2022, pp. 1–15. doi: 10.1145/3526113.3545659.

[12] Z. Pan, Z. Xie, T. Liu, and T. Xia, 'Exploring the Key Factors Influencing College Students' Willingness to Use AI Coding Assistant Tools: An Expanded Technology Acceptance Model', *Syst. Basel*, vol. 12, no. 5, p. 176, May 2024, doi: 10.3390/systems12050176.

[13] D. Sun, A. Boudouaia, C. Zhu, and Y. Li, 'Would ChatGPT-facilitated programming mode impact college students' programming behaviors, performances, and perceptions? An empirical study', *Int. J. Educ. Technol. High. Educ.*, vol. 21, no. 1, p. 14, Feb. 2024, doi: 10.1186/s41239-024-00446-5.

[14] F. Fagerholm and J. Munch, 'Developer Experience: Concept and Definition', in *Proceedings of the International Conference on Software and System Process*, ACM, 2012, pp. 73–77. doi: 10.5555/2664360.2664372.

[15] M. Greiler, M.-A. Storey, and A. Noda, 'An Actionable Framework for Understanding and Improving Developer Experience', *IEEE Trans. Softw. Eng.*, vol. 49, no. 4, pp. 1411–1425, 2023, doi: 10.1109/TSE.2022.3175660.

[16] E. Al. Abdul Saboor, 'Towards the Definition of Application Developer Persona and Developer Experience Conceptualization', *Int. J. Recent Innov. Trends Comput. Commun.*, vol. 11, no. 9, pp. 3663–3671, Nov. 2023, doi: 10.17762/ijritcc.v11i9.9588.

[17] K. Kuusinen, H. Petrie, F. Fagerholm, and T. Mikkonen, 'Flow, Intrinsic Motivation, and Developer Experience in Software Engineering', in *Agile Processes, in Software Engineering, and Extreme Programming*, vol. 251, H. Sharp and T. Hall, Eds, in Lecture Notes in Business Information Processing, vol. 251. , Cham: Springer International Publishing, 2016, pp. 104–117. doi: 10.1007/978-3-319-33515-5_9.

[18] P. Calais and L. Franzini, 'Test-Driven Development Benefits Beyond Design Quality: Flow State and Developer Experience', in *2023 IEEE/ACM 45th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, Melbourne, Australia: IEEE, May 2023, pp. 106–111. doi: 10.1109/ICSE-NIER58687.2023.00025.

[19] N. Forsgren, E. Kalliamvakou, A. Noda, M. Greiler, B. Houck, and M.-A. Storey, 'DevEx in Action: A study of its tangible impacts', *ACM Queue*, vol. 21, no. 6, pp. 47–77, Dec. 2023, doi: 10.1145/3639443.

[20] K. Kuusinen, 'Are Software Developers Just Users of Development Tools? Assessing Developer Experience of a Graphical User Interface Designer', in *Human-Centered and Error-Resilient Systems Development*, vol. 9856, C. Bogdan, J. Gulliksen, S. Sauer, P. Forbrig, M. Winckler, C. Johnson, P. Palanque, R. Bernhaupt, and F. Kis, Eds, in Lecture Notes in Computer Science, vol. 9856. , Cham: Springer International Publishing, 2016, pp. 215–233. doi: 10.1007/978-3-319-44902-9_14.

[21] K. Kuusinen, 'Software Developers as Users: Developer Experience of a Cross-Platform Integrated Development Environment', in *Product-Focused Software*

Process Improvement, vol. 9459, P. Abrahamsson, L. Corral, M. Oivo, and B. Russo, Eds, in Lecture Notes in Computer Science, vol. 9459. , Cham: Springer International Publishing, 2015, pp. 546–552. doi: 10.1007/978-3-319-26844-6_40.

[22] Atlassian, 'State of developer experience 2024'. 2024. Accessed: June 19, 2025. [Online]. Available: https://www.atlassian.com/software/compass/resources/state-of-developer-2024

[23] S. Chandrasekaran, 'Enhancing Developer Experience by Reducing Cognitive Load: A Focus on Minimization Strategies', *Int. J. Comput. Trends Technol.*, vol. 71, no. 1, pp. 99–103, Jan. 2024, doi: 10.14445/22312803/IJCTT-V72I1P117.

[24] Anysphere, 'Cursor - The AI Code Editor'. 2025. Accessed: Jan. 27, 2025. [Online]. Available: https://www.cursor.com/

[25] Oracle Corporation, 'RAG vs. Fine-Tuning: How to Choose'. 2025. Accessed: Feb. 07, 2025. [Online]. Available: https://www.oracle.com/artificial-intelligence/generative-ai/retrieval-augmented-generation-rag/rag-fine-tuning/

[26] Hugging Face, 'Hugging Face - the AI community building the future'. 2025. Accessed: Feb. 07, 2025. [Online]. Available: https://huggingface.co/

[27] Google, 'Google NotebookLM | Note Taking and Research Assistant Powered by AI'. 2025. Accessed: Feb. 07, 2025. [Online]. Available: https://notebooklm.google/

[28] Microsoft, 'Azure AI Platform - Cloud AI Platform | Microsoft Azure'. 2025. Accessed: Feb. 07, 2025. [Online]. Available: https://azure.microsoft.com/en-us/solutions/ai

[29] Microsoft, 'BLACKBOX AI #1 Coding Agent +10 Million Developers | Visual Studio Market'. 2025. Accessed: Feb. 07, 2025. [Online]. Available: https://marketplace.visualstudio.com/items?itemName=Blackboxapp.blackbox

[30] K. Gupta, 'Debugging Tactics in Novice Programmers: The Familiarity Factor', *J. Softw. Eng. Simul.*, vol. 10, no. 9, pp. 09–15, 2024, doi: 10.35629/3795-10090915.

Appendix 1

Online developer survey questions

Question	Question type	Answer options
Years of Experience: How many years of professional software development experience do you have?	Single-choice	• 0-5 years • 5+ years
Job Title: What is your job title?	Single-choice	• Software Developer / Engineer • Full-Stack / Front-End / Backend Developer / Engineer • Web Developer • Mobile Developer / Engineer • System Architect, System Analyst, System Administrator • Other *(write-in)*
Job Title: Do you use GitHub Copilot in your job?	Single-choice	• Yes • No

Table 1. Online developer survey background questions.

Question	Question type	Answer options
IDEs used: Which of the following integrated development environments (IDEs) do you use in your job?	Multiple-choice	• Visual Studio Code • Visual Studio • IntelliJ IDEA • Notepad++ • Vim • Android Studio • PyCharm • Other *(write-in)*

AI Tool Usage: How often do you use the following AI-powered tools in your job? *1 - I do not use this tool at my job at all* *2 - I use it less than a few times per month* *3 - I use it a few times per month* *4 - I use it a few times per week* *5 - I use it daily* • GitHub Copilot • ChatGPT • Google Gemini • Bing AI • Visual Studio Intellicode • Claude • Codeium	Likert scale	• 1 – do not use at all • 2 • 3 • 4 • 5 – use daily
Other AI Tools (optional): Do you use any other AI-powered tools in your job? If yes, how often?	Text input	N/A
Software Development Lifecycle Stages: Which stages of software development lifecycle do you use AI-powered tools for in your job?	Multiple-choice	• Planning and analysis • Requirements gathering • Design • Development • Testing • Deployment • Maintenance • Other *(write-in)*
Programming Activities: Which programming activities do you use AI-powered tools for in your job?	Multiple-choice	• Writing code • Debugging • Code analysis • Writing tests • Refactoring • Writing natural language artifacts (for ex. comments and README files) • Other *(write-in)*
Descriptors for Copilot: When using GitHub Copilot in my job, I find it to be... • ...easy to use and intuitive.	Likert scale	• Strongly disagree • Disagree

• ...accurate and reliable. • ...fast and robust. • ...customizeable and personalizeable. • ...easy to learn and beginner friendly. • ...secure and trustworthy. • ...context-aware and adaptive. • ...accessible and inclusive. • ...pleasant and likeable. • ...easy to integrate into my workflow.		• Neither agree nor disagree • Agree • Strongly agree
Satisfying Copilot Qualities: I am satisfied with GitHub Copilot's... • ...easiness of completing tasks. • ...clarity and informativeness of feedback and system messages. • ...support for undoing and redoing actions. • ...mistake prevention. • ...support for user-defined shortcuts. • ...easy-to-find and useful documentation.	Likert scale	• Strongly disagree • Disagree • Neither agree nor disagree • Agree • Strongly agree
Most Important Improvements (up to 5 options allowed): Choose the most important areas of improvement for GitHub Copilot:	Multiple-choice	• Better natural language prompt comprehension • Better output (for ex. more accurate/secure generated code) • Better debugging capabilities • Better user interface (UI) • Better support for personalization

		• Better support/onboarding for new users • Better context awareness • Better support for multimodal interaction (for ex. voice chat) • Better and/or more visible documentation • None. • Better testing capabilities. • Other *(write-in)*

Table 2. Online developer survey developer experience questions.

Appendix 2

Online developer survey other questions' results

Figures 13, 14 and 15 below present the percentage distribution of job titles among all respondents, novices and experienced respondents respectively.

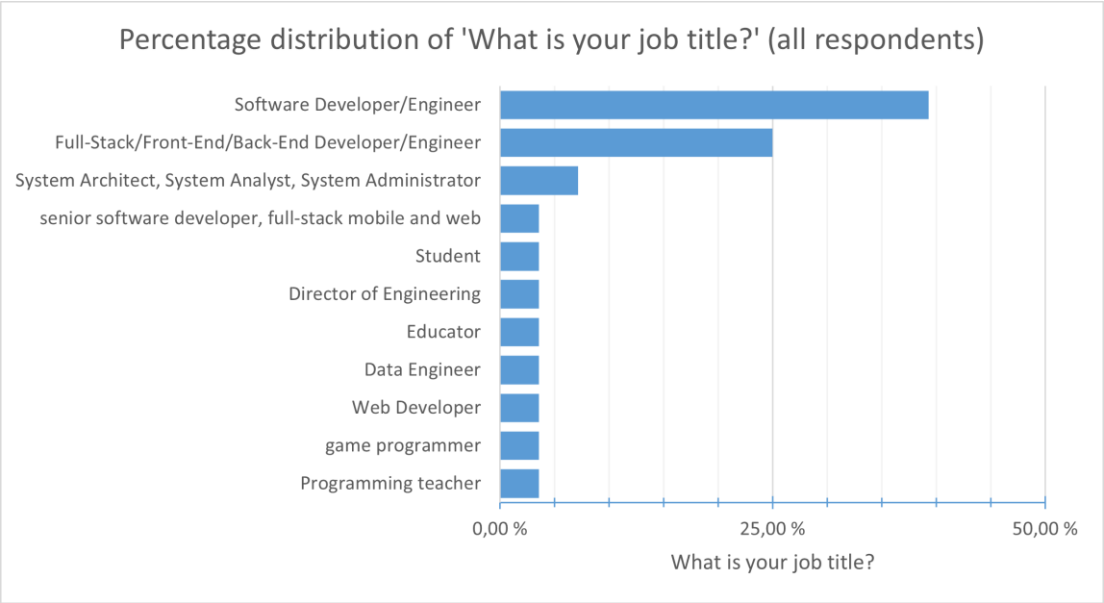

Figure 13. Percentage distribution of "What is your job title?" (all respondents).

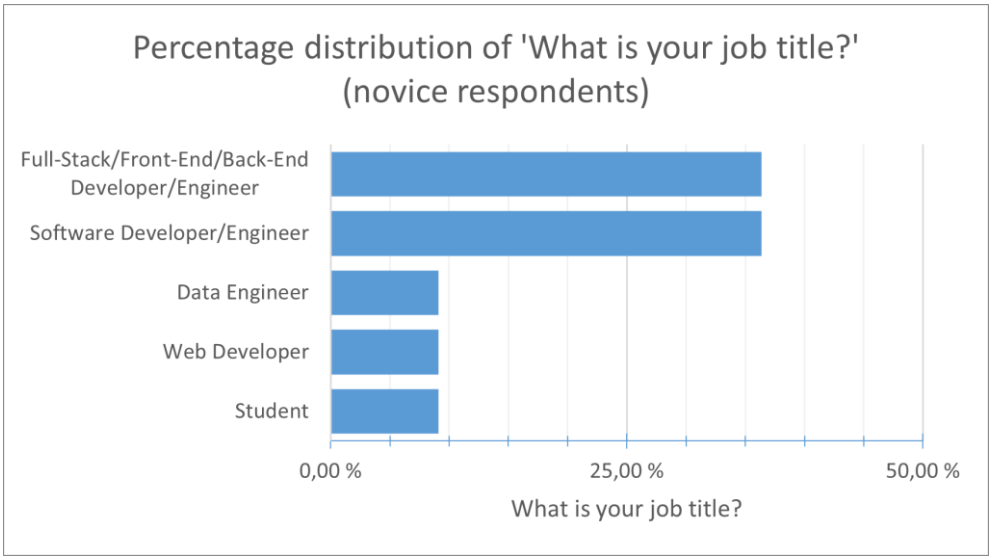

Figure 14. Percentage distribution of "What is your job title?" (novice respondents).

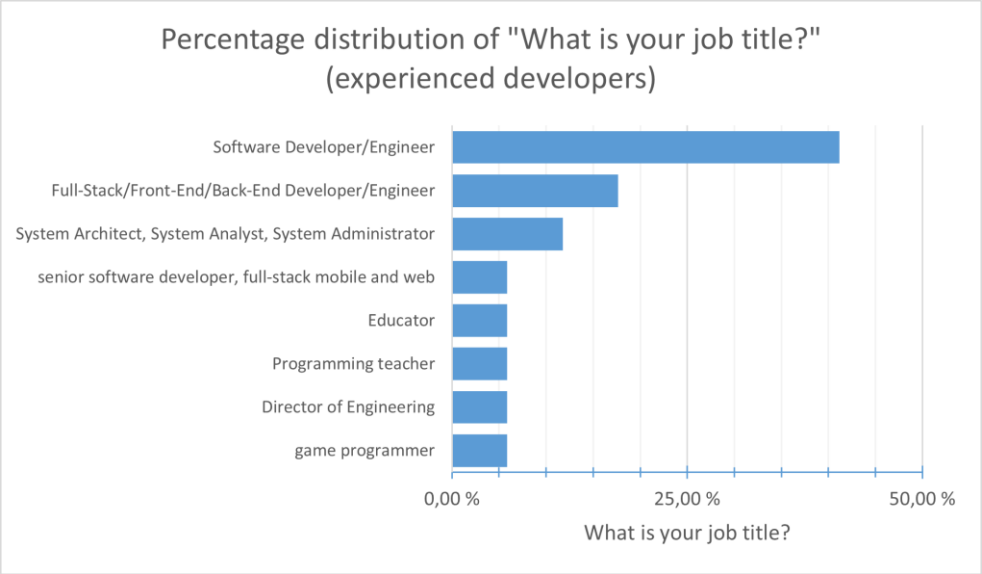

Figure 15. Percentage distribution of "What is your job title?" (experienced respondents).

A more detailed analysis of IDEs used by the respondents, as well the differences between novice and experienced respondents in IDE usage, is presented in Figures 16, 17 and 18 below.

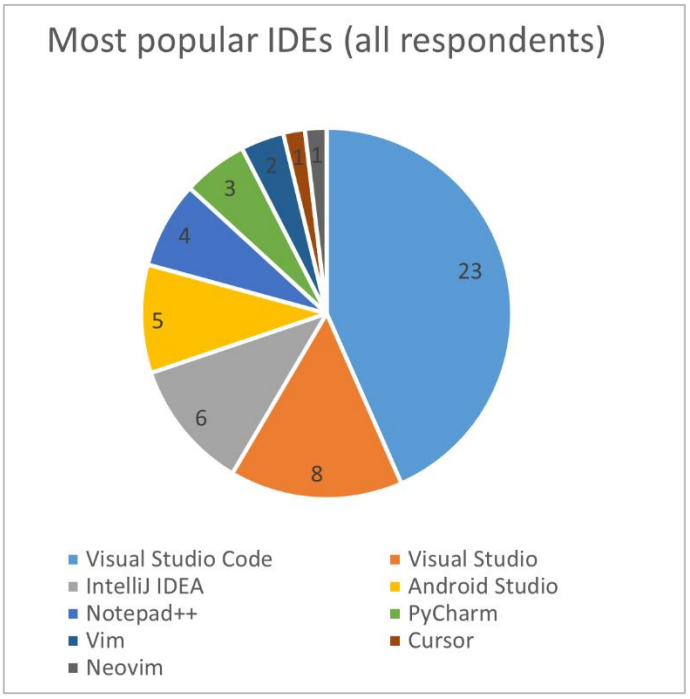

Figure 16. Most popular IDEs (all respondents)

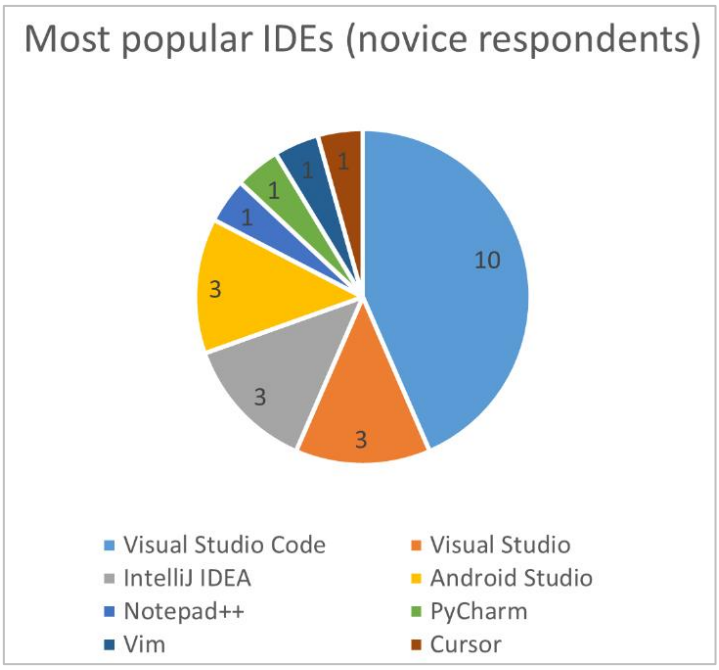

Figure 17. Most popular IDEs (novice respondents)

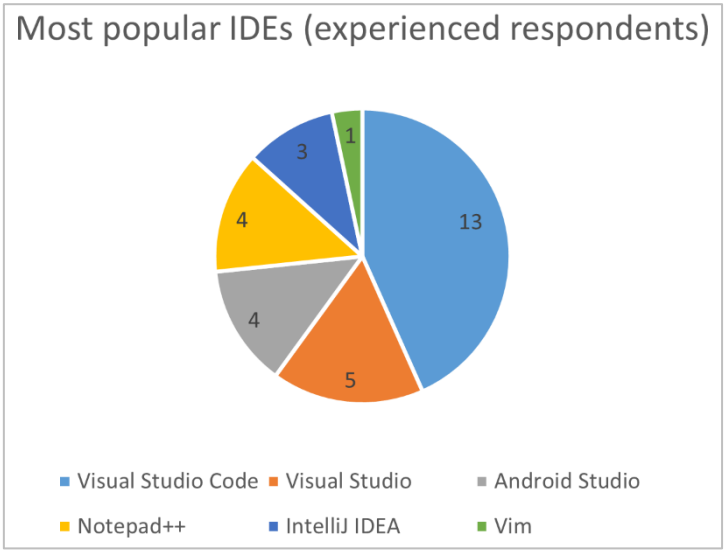

Figure 18. Most popular IDEs (experienced developers)

Figures 19, 20 and 21 below present a detailed overview of AI-powered tools used by all respondents, novices and experienced respondents respectively.

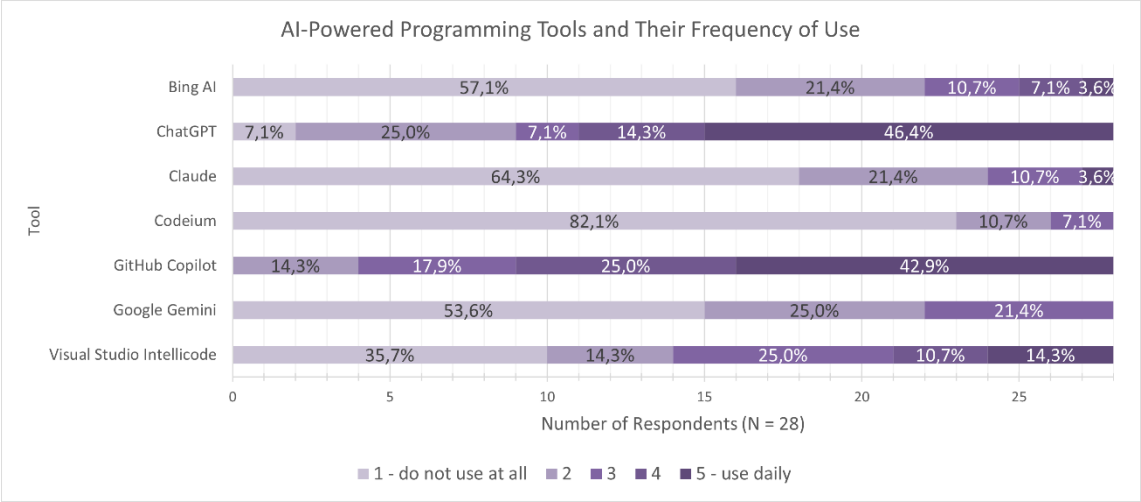

Figure 19. AI-powered programming tools and their frequency of use (all participants)

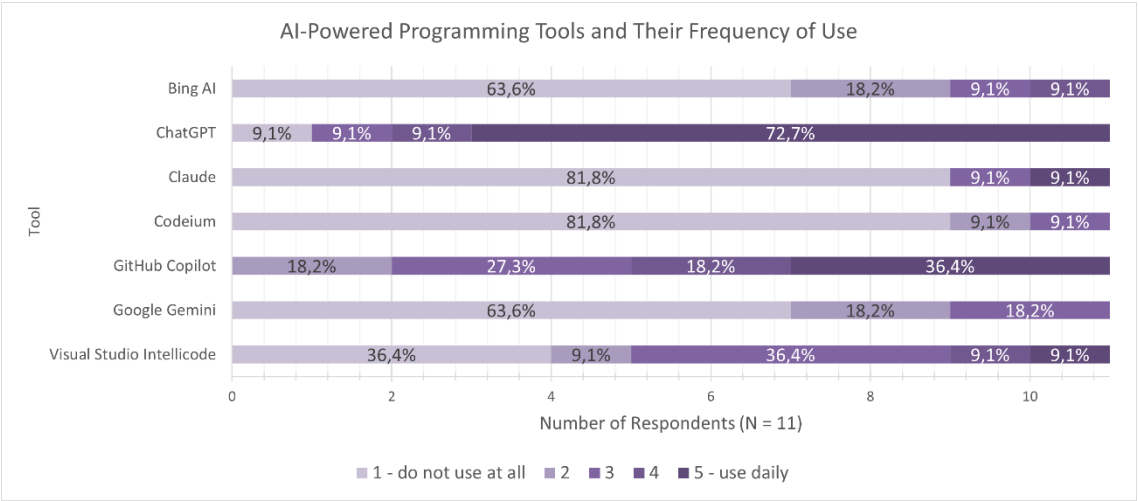

Figure 20. AI-powered programming tools and their frequency of use (novice participants)

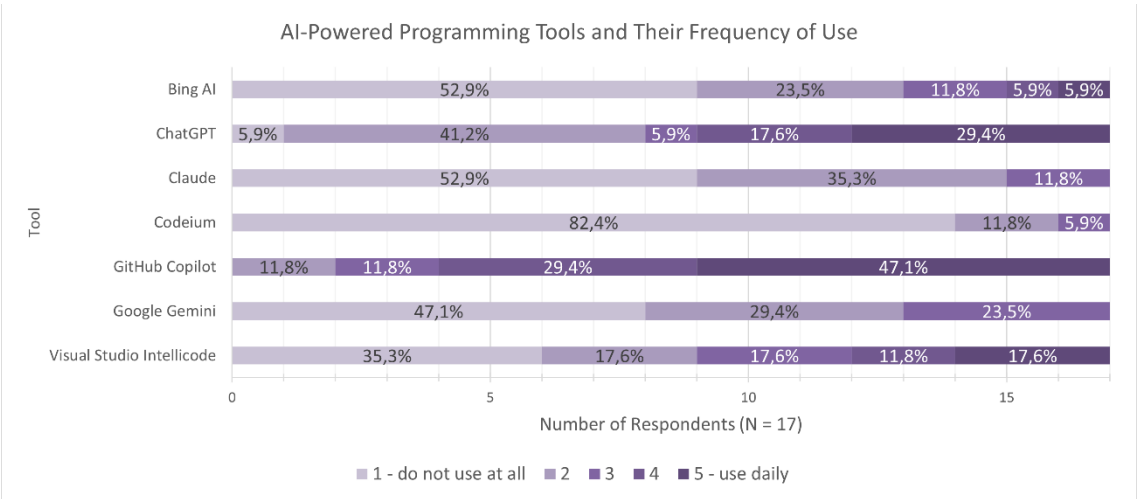

Figure 21. AI-powered programming tools and their frequency of use (experienced participants)

Further analysis of the use of AI-powered tools in different SDLC stages can be found in Figures 22, 23 and 24 below.

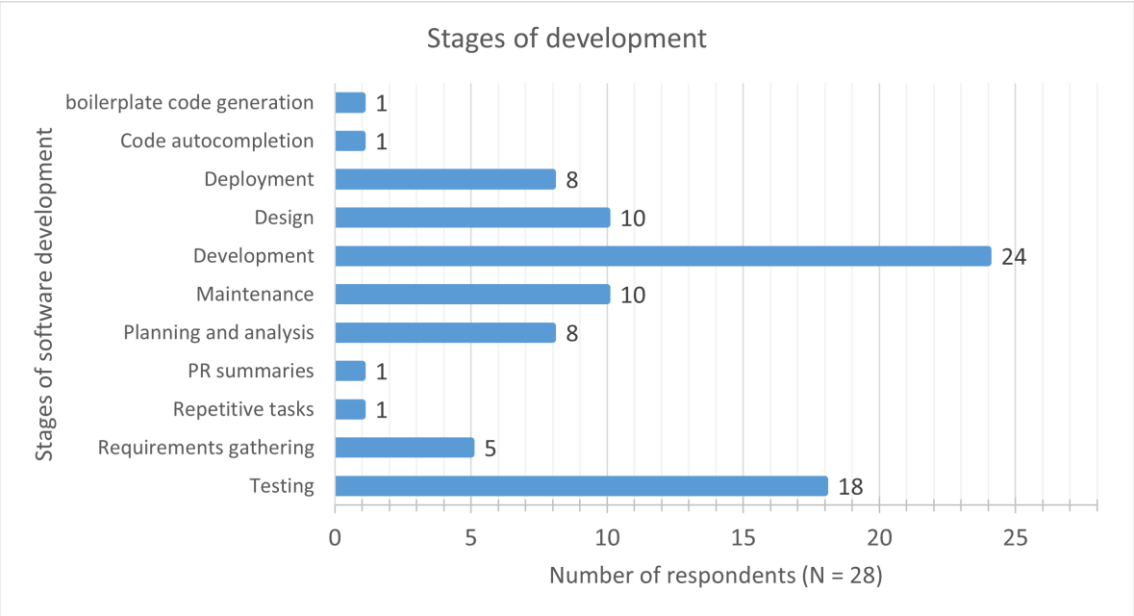

Figure 22. AI-powered tools in SDLC stages (all respondents)

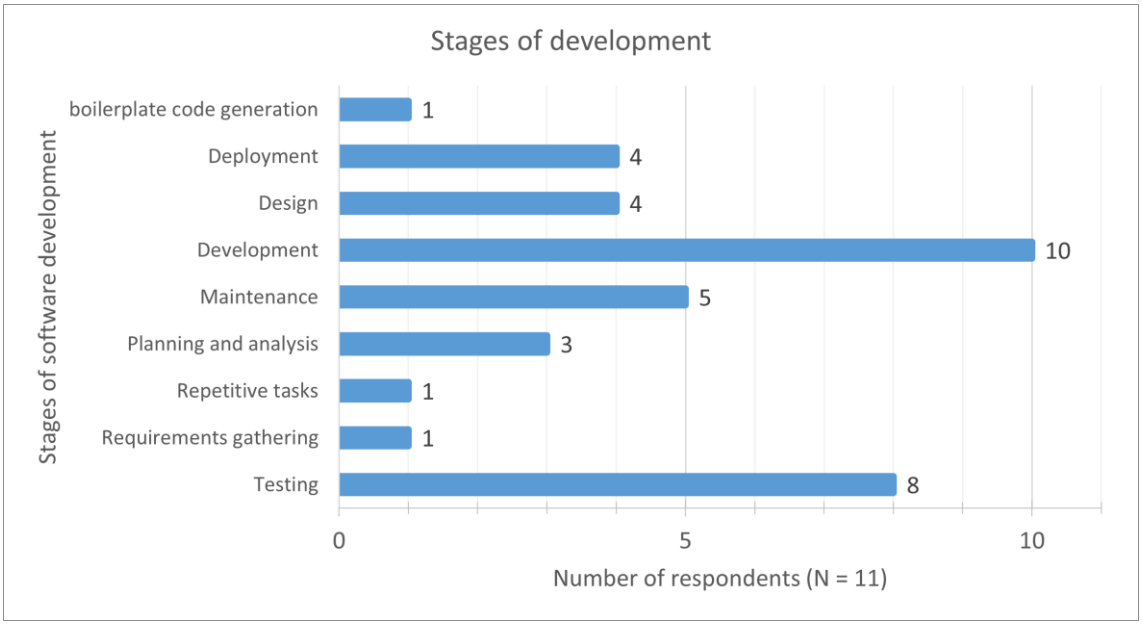

Figure 23. AI-powered tools in SDLC stages (novice respondents)

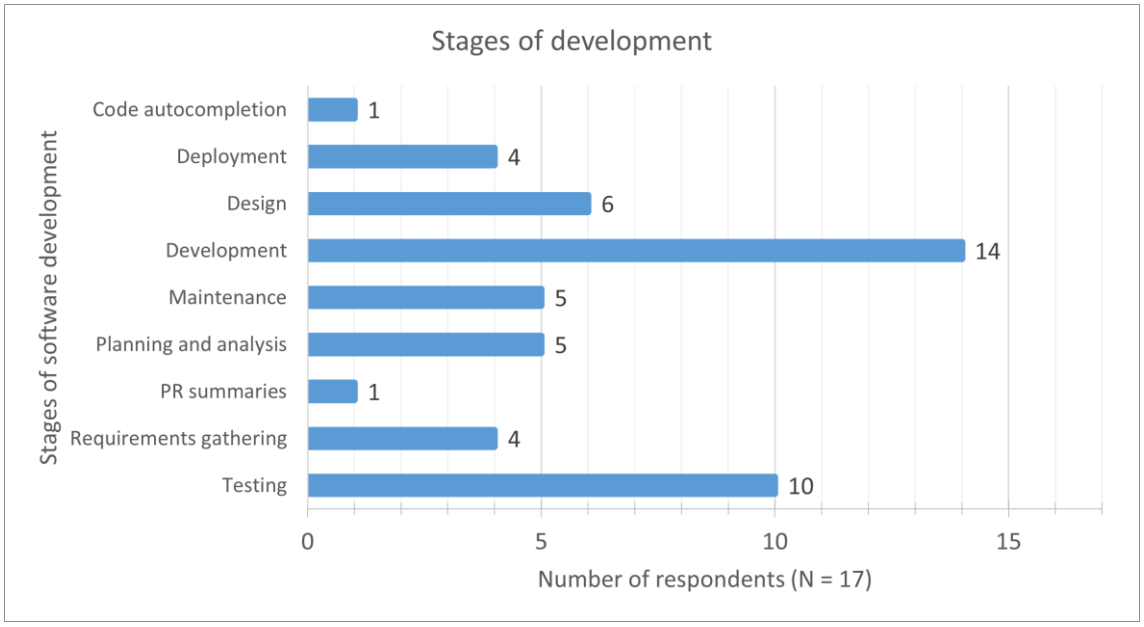

Figure 24. AI-powered tools in SDLC stages (experienced respondents)

Figures 25, 26 and 27 represent a more detailed analysis of the usage of AI-powered tools for different programming activities.

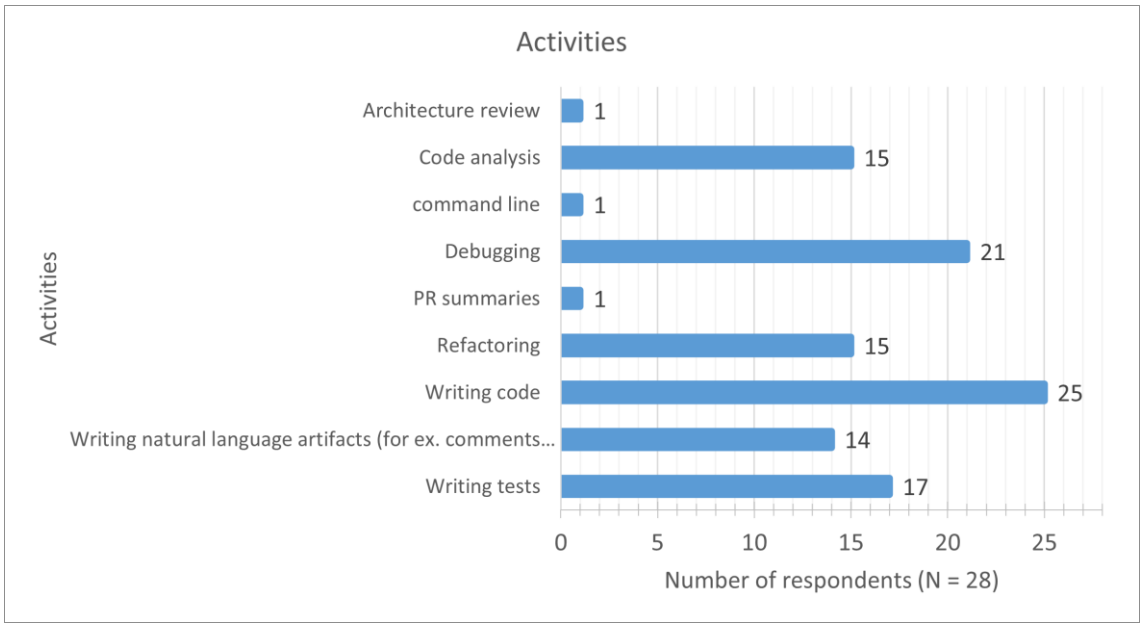

Figure 25. AI-powered tools in programming activities (all respondents).

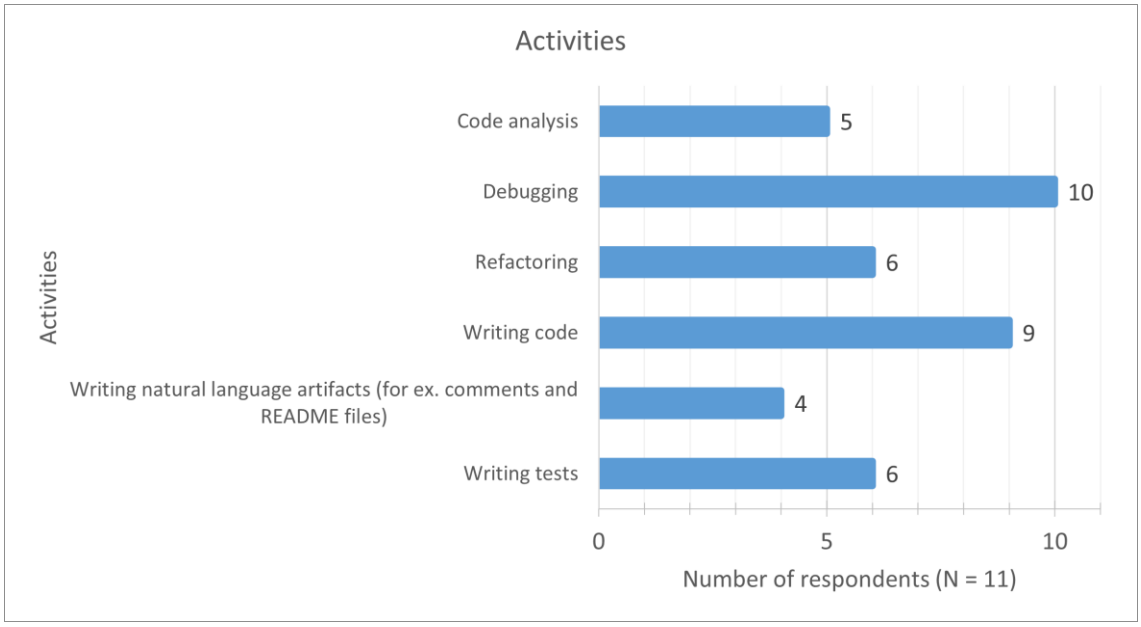

Figure 26. AI-powered tools in programming activities (novice respondents).

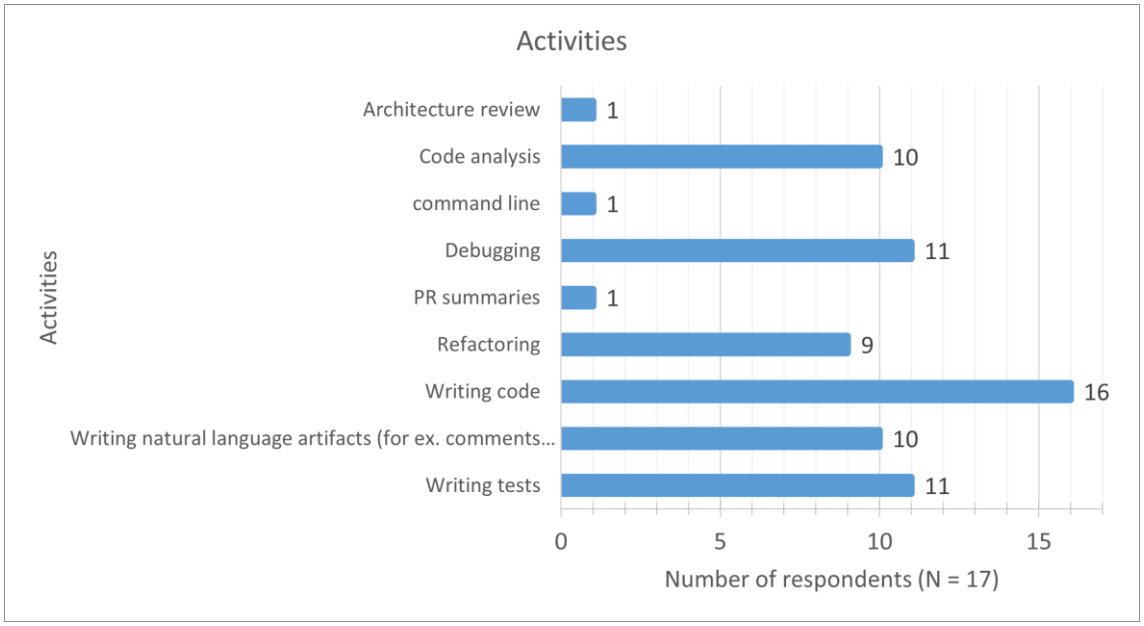

Figure 27. AI-powered tools in programming activities (experienced respondents).

Appendix 3

Developer user study test setup for each participant

Participant	Novice or experienced	Online or in-person	IDE	Task(s) attempted	Task(s) completed
P1	Experienced	Online	IntelliJ IDEA	Task 4	-
P2	Novice	In-person	Visual Studio Code	Task 1, Task 2	Task 2
P3	Novice	Online	Visual Studio	Task 1	-
P4	Experienced	In-person	Visual Studio Code	Task 4	Task 4
P5	Novice	In-person	Visual Studio	Task 4	Task 4
P6	Experienced	Online	Visual Studio Code	Task 1	Task 1
P7	Novice	Online	Visual Studio Code	Task 1	Task 1
P8	Novice	Online	Visual Studio Code	Task 1	Task 1
P9	Experienced	Online	Visual Studio	Task 4	-
P10	Experienced	Online	IntelliJ IDEA	Task 4	-
P11	Experienced	Online	Visual Studio	Task 4	Task 4

Table 3. Developer user study test setup for each participant.

Appendix 4

Job titles of user study participants

Participant	Job title
P2	Researcher
P3	Unemployed job seeker in the technical field
P5	Senior Researcher
P7	Early-Stage Startup Founder
P8	Freelance Full-Stack Developer

Table 4. Job titles for novice participants.

Participant	Job title
P1	Scrum Master
P4	Technical Product Manager
P6	Full-Stack Engineer
P9	Doctoral Researcher in Computer Engineering
P10	Software Developer
P11	Senior Software Developer

Table 5. Job titles for experienced participants.

Appendix 5

GitHub Copilot usage patterns among user study participants

Participant	How often and how long have you been using GitHub Copilot?
P2	Rarely, for about 6 months
P3	Often, for about a year
P5	Never used it
P7	All the time, for 5 months
P8	50% of the time, for about a year

Table 6. GitHub Copilot usage patterns among novice participants.

Participant	How often and how long have you been using GitHub Copilot?
P1	Not actively using, has some experience
P4	Tried a few times, for about a year
P6	Around 3 days a week, for about two years
P9	Never used it
P10	Every day, about 3 years
P11	Daily, 6 months

Table 7. GitHub Copilot usage patterns among experienced participants.

Appendix 6

IDEs used by user study participants

Participant	IDEs
P2	VS Code, IntelliJ IDEA
P3	VS Code, Visual Studio
P5	VS Code, Visual Studio, Matlab (used last year)
P7	VS Code
P8	VS Code, Pycharm

Table 8. IDEs used by novice participants.

Participant	IDEs
P1	IntelliJ IDEA, Netbeans
P4	VS Code, Google Project IDX, Android Studio (previously)
P6	VS Code, Sublime Text, IntelliJ IDEA
P9	Visual Studio, Spyder (Python IDE), Jupyter, RStudio
P10	IntelliJ IDEA
P11	VS Code, Visual Studio

Table 9. IDEs used by experienced participants.

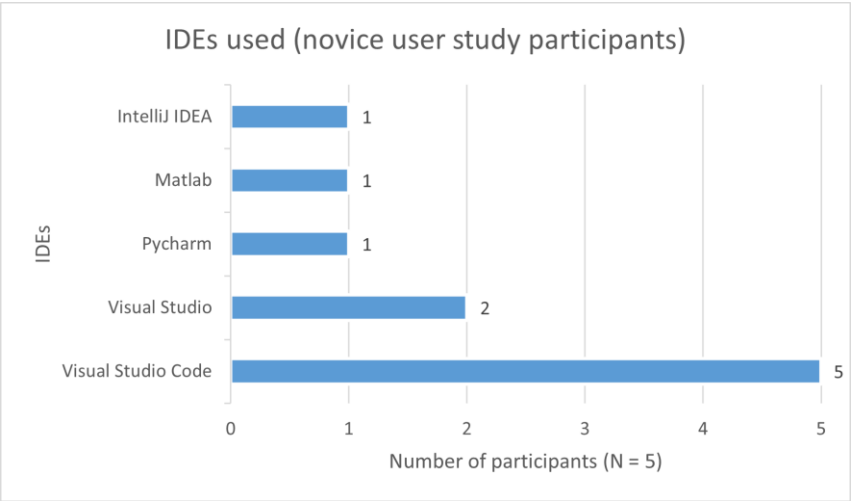

Figure 28. IDEs used by novice user study participants.

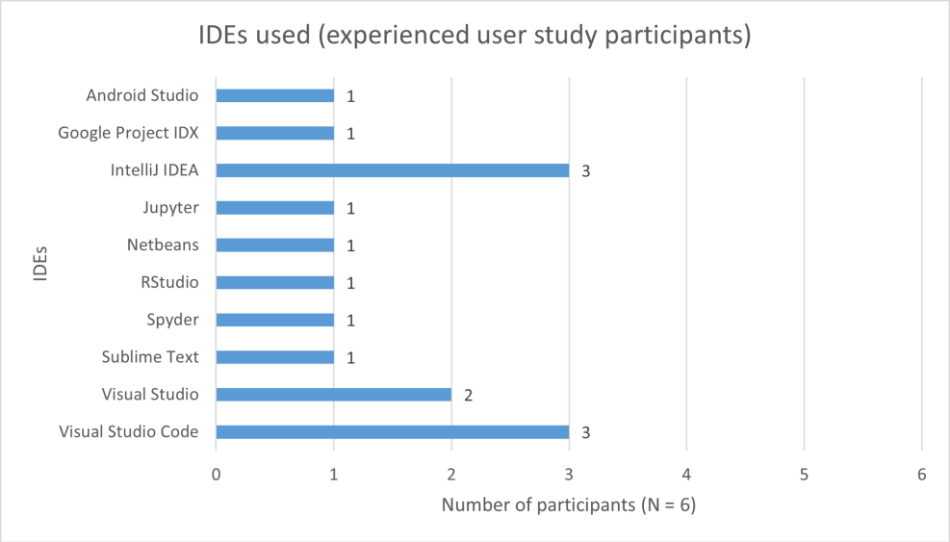

Figure 29. IDEs used by experienced user study participants.

Appendix 7

Other AI tools used by user study participants

Participant	Other AI tools
P2	ChatGPT, Claude
P3	ChatGPT, Claude, DeepSeek (compares answers from different tools)
P5	ChatGPT
P7	ChatGPT, ManusAI
P8	ChatGPT, Claude, v0

Table 10. Other AI tools used by novice participants.

Participant	Other AI tools
P1	ChatGPT, Claude
P4	ChatGPT, Cursor
P6	ChatGPT
P9	ChatGPT, Google Gemini (in Google Collab)
P10	ChatGPT
P11	ChatGPT

Table 11. Other AI tools used by experienced participants.

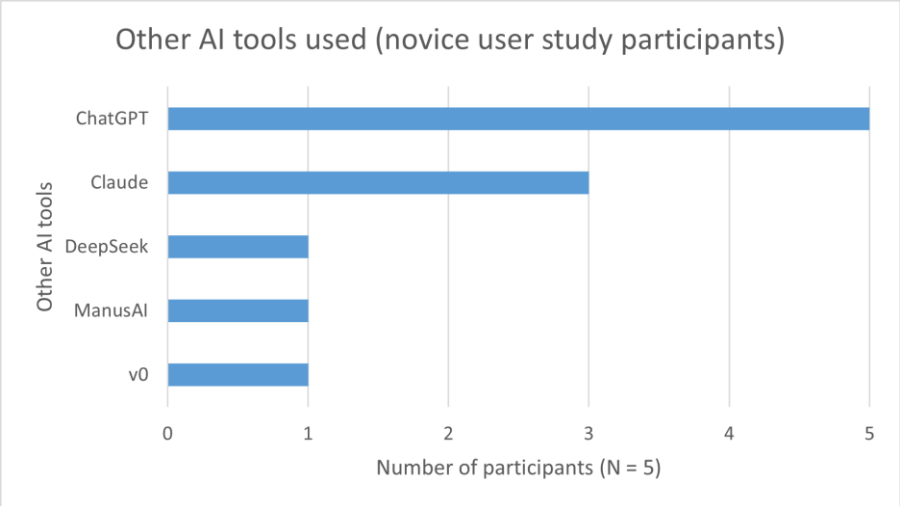

Figure 30. Other AI tools used by novice user study participants.

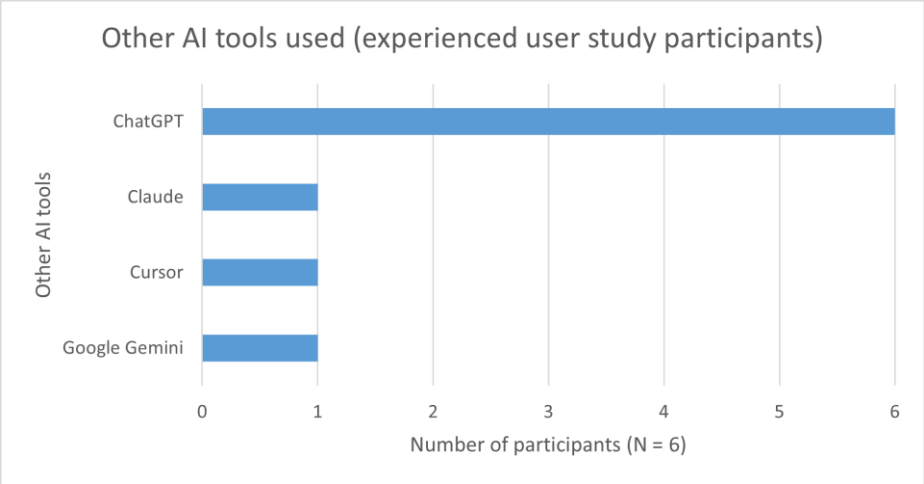

Figure 31. Other AI tools used by experienced user study participants.

Appendix 8

Words to describe GitHub Copilot chosen by user study participants

Participant	Descriptors
P2	Empowering, easy-debugging, good-for-learning
P3	Quick, has potential, room for improvement
P5	Useful, time-saving, pleasant
P7	Interesting, helpful, intimidating/scary
P8	Understanding, relieving, stubborn

Table 12. Words used to describe GitHub Copilot used by novice participants.

Participant	Descriptors
P1	Flirty, simple, “figureoutable”
P4	Precise, quick, context-unaware
P6	Helpful with code completion, nice, silly/naughty
P9	Convenient, interactive, simpler
P10	Easy, fast, helpful
P11	Reliable, secure, consistent

Table 13. Words used to describe GitHub Copilot used by experienced participants.

Note: some participants provided additional context and explanations regarding their choice of descriptors – see chapter 4.2.3 for more information.