When the Pair Becomes Three: An Empirical Study of GitHub Copilot in Pair Programming

Diego Assis da Silva Lisboa
Universidade Federal do Pará
Belém, PA, Brazil
diegolisboa@ufpa.br

Tiago Massoni
Universidade Federal de Campina Grande
Campina Grande, PB, Brazil
massoni@computacao.ufcg.edu.br

Mayara Costa Figueiredo
Universidade Federal do Pará
Belém, PA, Brazil
mcfigueiredo@ufpa.br

Cleidson de Souza
Universidade Federal do Pará
Belém, PA, Brazil
cleidson.desouza@acm.org

Abstract

Pair programming (PP) is a widely adopted Agile practice in which two developers collaborate closely on the same task. Although PP is often promoted for its potential to improve code quality, foster shared understanding, and support learning, empirical evidence suggests that its outcomes are highly context-dependent and that the practice entails significant coordination effort. With the growing adoption of AI-powered coding assistants, it becomes essential to examine how these tools reshape PP and other collaborative programming practices traditionally grounded in human–human interaction. This paper presents findings from a 40-week qualitative study conducted in a software team adopting GitHub Copilot during PP sessions, combining developer diaries, observations and notes from agile meetings.Our analysis shows that Copilot substantially influenced PP practices by mediating how developers coordinated, reasoned about code, and negotiated design decisions. Participants reported higher perceived productivity and code quality, as well as longer, more cognitively demanding PP sessions due to frequent discussions of AI-generated suggestions. Rather than reinforcing fixed roles, these interactions led participants to reconceptualize PP using an AI-assistant as a form of *triadic* collaboration, in which coordination and responsibility are distributed across two human developers and the assistant. Building on these findings, we articulate empirically grounded implications for practice highlighting how AI code assistants reshape coordination and learning in PP.

Keywords

agility, agile methods, ceremonies, pair programming, software modernization, LLM4code, LLMs

1 Introduction

The automation of code generation to support programmers has long been a topic of research [37]. However, recent advances in large language models (LLMs) and the emergence of AI-powered code assistants – such as Tabnine, GitHub Copilot, and Cursor – have made automatic code generation increasingly accessible. Rather than a solo developer working on a single machine, this paradigm involves a programmer and an LLM-based tool working on the same task. This shift raises important questions regarding the impact of using such tools on PP practices, collaborative dynamics, and decision-making processes.

Initial empirical studies on AI assistants highlighted both positive [11, 22, 37, 48] and negative aspects, regarding correctness [24, 45], memorization [44], non-determinism [25], security [26], and biases [18, 36]. Until very recently, these studies rarely addressed the everyday organizational contexts in which software engineers use these tools, being mostly based on laboratory experiments and surveys. Newer studies focused on real-world scenarios using qualitative methods [7, 22], quantitative [48] or quasi-experiments [11].

Meanwhile, Pair Programming (PP) has long been promoted within agile methods as a cornerstone practice, based on its reported potential to improve code quality, foster shared understanding, and support learning within teams [28, 35, 38, 42, 43]. Although prior research highlights these benefits, it also shows that the outcomes of PP are highly context-dependent, varying across educational, experimental, and industrial settings. Additionally, PP requires substantial coordination effort, which may influence how it scales and evolves in modern development environments [4, 42, 43].

With the advent of AI-based coding assistants, PP increasingly involves triadic interactions in which two human developers collaborate in close coordination with an AI assistant. These interactions emerge through ongoing negotiation and AI-mediated sensemaking, introducing new layers of complexity and calling for reexamination of PP dynamics, benefits, and challenges. Understanding these changes is essential for practitioners integrating AI tools into development workflows and for researchers studying the reconfiguration of agile practices in AI-assisted software development.

This paper investigates the impact of AI assistants in a real-world PP setting with an empirical study conducted with professional and intern software developers in a university department that adopted GitHub Copilot during their PP sessions. Our results are based on diaries written by pairs of interns over 40 weeks while they worked on modernizing a legacy system using agile practices; these diaries contained comprehensive records of the project's daily activities and reflections on the use of an AI assistant for code generation. In addition, the first author documented information from team retrospectives and other relevant meetings to complement the analysis. A qualitative analysis of the data was conducted based on grounded theory techniques [32], including open and axial coding. In this context, the present study focuses specifically on aspects of PP involving human developers collaborating with GitHub Copilot, aiming to investigate how the integration of this AI assistant influenced the collaborative dynamics between team members.

Our results indicate that the main benefits of using Copilot during PP sessions include increased perceived productivity and code quality. However, participants also reported negative aspects related to the increased coordination effort introduced by Copilot; in particular, frequent joint discussions around AI-generated code led to longer and more cognitively demanding PP sessions. While these interactions were often perceived as valuable for learning and reflection, they also resulted in fatigue and time overhead, sometimes increasing the risk of accumulating technical debt without clear improvements in code quality. These discussions and work influenced PP dynamics in such a way that a member of the team aptly described the experience as a *"pairing with three people."*

In addition to characterizing how AI code assistants are integrated into PP, this study derives empirically grounded implications for practice, research, and education, based on the strategies and consequences observed in participants' everyday work.

2 Background

Pair Programming (PP) is a collaborative software development practice in which two developers work together on the same task, typically sharing a workstation. Since its introduction within Extreme Programming (XP), PP has been widely discussed in both research and practice. However, empirical evidence accumulated over the past two decades suggests a more nuanced and context-dependent picture than early prescriptive accounts imply. For example, Hannay et al. [15] reported mixed effects of PP on productivity and code quality across controlled experiments, highlighting substantial variability across contexts and study designs. Focusing on educational settings, Salleh et al. [30] identified consistent learning benefits alongside methodological limitations and strong contextual dependencies. Complementing these findings, Vanhanen and Mäntylä [39] showed that evidence from industrial settings remains comparatively scarce, fragmented, and often based on self-reported perceptions rather than direct observation.

Early XP-oriented descriptions emphasized complementary activities within the pair [3], but these accounts were primarily normative and pedagogical rather than empirical. Subsequent qualitative and observational studies have challenged simplified role-based interpretations of PP. Research by Bryant et al. [5, 6], Chong and Hurlbutt [10], Plonka et al. [27], Salinger et al. [29], and Jones and Fleming [17] consistently depicts PP as a socially rich and dynamically negotiated practice, in which developers frequently switch activities, overlap contributions, and jointly engage in problem solving, design reasoning, and code evaluation.

The widespread "driver" and "navigator" terminology illustrates the gap between pedagogical models and empirical observations. While these labels were introduced as an instructional scaffold for novices [41], empirical studies suggest that experienced developers rarely conform to a strict division of labor in practice.

Empirical findings on PP outcomes must also be interpreted in light of the study design. For instance, Begel and Nagappan [4] reported perceived benefits such as improved knowledge sharing, higher code quality, and increased developer satisfaction based on a large-scale retrospective survey at Microsoft. Williams et al. [42], based on controlled experiments with students, reported improvements in code quality and defect reduction, albeit with increased development time. Although valuable, such studies offer limited insight into how PP unfolds in long-term industrial practice.

Taken together, prior research characterizes PP as a complex, socially situated practice whose outcomes depend strongly on context, experience, and task characteristics, and which resists rigid role definitions and simplistic causal claims. This perspective is particularly relevant in AI-assisted software development: if traditional PP already involves fluid roles and negotiated collaboration, the introduction of an AI code assistant further mediates coordination, responsibility, and sensemaking. Based on this understanding, our study examines how human–human collaboration is reconfigured when an AI assistant becomes an active participant in PP sessions.

3 Related Work

With the emergence of AI code assistants, new configurations of pair programming have emerged in which human developers may collaborate not only with each other but also with a virtual partner. This extension of PP introduces additional complexity and potential by mediating collaboration through AI systems. Recent research has primarily examined dyadic human–AI collaborations in which one developer "pair" with an AI code assistant. In a controlled experiment, Imai [16] reported increased productivity when using GitHub Copilot, as measured by lines of code added, alongside indications of lower code quality reflected in subsequent code removal. In an industrial setting, Chen [9] observed that developers pairing with AI assistants in joint coding sessions reported improved code quality, reduced repetitive work, and increased satisfaction, particularly for tasks such as code completion and refactoring. Additionally, Zhou et al. [47] analyzed large-scale GitHub online discussions and Stack Overflow related to human–AI pairing in programming. Their qualitative analysis identified recurring challenges, including context misunderstandings, over-reliance on AI-generated suggestions, and inconsistencies in generated code, as well as mitigation strategies such as prompt refinement, contextualization, and selective acceptance of suggestions. Together, these studies highlight both the benefits and the cognitive and coordination costs associated with treating AI code assistants as programming "partners".

Importantly, existing work focuses almost exclusively on dyadic human–AI interactions. For example, Stray et al. [33] investigated how generative AI tools influence solo and pair programming workflows, reporting productivity gains in solo settings but more nuanced and sometimes limited use in pair programming contexts. While such configurations suggest gains in productivity and satisfaction, they may also compromise critical evaluation [9, 47] and collaborative reflection—central elements of traditional human–human PP. Building on this previous research, our study investigates a configuration in which two human developers pair with an AI code assistant throughout long-term PP sessions.

4 Methodology

4.1 Study Definition and Research Questions

Our study aims to investigate the following research question: "*How does the use of AI code assistants impact pair programming dynamics?*". To answer this question, we employed a qualitative approach, broadly presented in Figure 1.

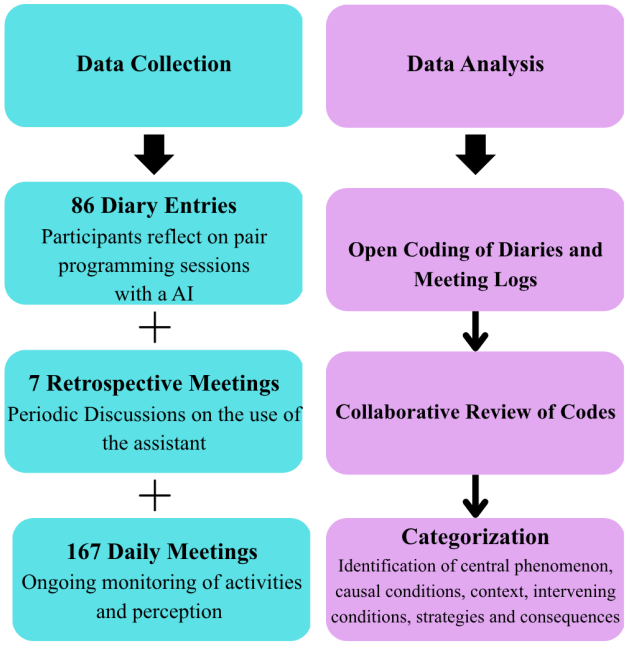

Figure 1: Visual representation of our methodology.

We describe below the details of the site and the software development project in which participants worked during the study, as well as our data collection and analysis methods.

4.2 Context

The study was conducted in a university software development department under the direction of the Dean of Undergraduate Studies. This department is responsible for developing and evolving five different applications that are used by several thousand students, faculty, and staff. This study focused on only one of these projects; however, during the research period, participants were simultaneously involved in tasks related to the other projects.

During the research period, the team consisted of seven members: two full-time public servants working 40 hours per week (including the first author), and five interns, who were students from computer science, information systems, or computer engineering programs, working 20 hours per week. All team members were male. The average age was over 30 years for the public servants and around 20 years for the interns. In terms of experience, the public servants had been on the team for over five years, while all the interns had been working in this department for at least one year.

The department has adopted agile methods in its workflow for over five years, which means that all interns in this study had at least one year of experience in agile methods. The adopted agile practices include stand-up meetings, code refactoring, team retrospectives, pair programming, and automated testing. But PP stands out for helping the team reduce the learning curve among team members, enhancing interaction and knowledge sharing among them.

4.3 Participants

The entire team was invited and agreed to participate in the study. Most interns had little or no experience with AI code assistants, and some had prior industry experience through other internships. The adoption of the AI code assistant was not introduced as part of the research design. Instead, the development team independently decided to incorporate the tool into their workflow, and the researchers chose to study how its use unfolded in practice.

The five interns had spent between one and two and a half years in the department, meaning they were already well-integrated into the team's agile and pair programming routines. Within this group, background experience varied slightly: I1 and I2 each had two and a half years of tenure and had held prior internships elsewhere. In contrast, the other three participants (I3, I4, and I5), with tenures ranging from one to one and a half years, were stepping into their very first professional roles in software development.

Our study focused on a modernization project for an internal application, the "internship center." This project aimed to improve user experience and code maintainability by migrating the application from PHP to Python with the Django framework.

PP sessions were carried out exclusively by interns and constituted a regular part of the team's development workflow. Interns typically worked in pairs during implementation tasks related to the modernization project, while pairing configurations varied over time as team members rotated across tasks. PP was not continuously applied in all activities, but it was primarily used for core development tasks. Professional developers did not participate directly in PP coding sessions; instead, they contributed through requirements elicitation, testing, code reviews, and participation in daily and retrospective meetings. Over the 40-week period, PP sessions occurred regularly as part of ongoing development work, resulting in repeated and sustained exposure of interns to PP practices.

4.4 Data Collection

We adopted a qualitative empirical research approach grounded in participatory observation of the software development team, as the first author is one of the public servants who is a member of the team. Note that the first author did *not* participate in the study as a developer and did not contribute to writing the PP session diaries. As reported in 4.3, his involvement was limited to other project-related activities, such as testing, code reviews, and participation in daily and retrospective meetings.

Pairs of interns wrote diaries [7] for approximately 40 weeks. These diaries specifically recorded information related to the use of the AI assistant (in this case, GitHub Copilot) during PP sessions, including experiences, perceptions, and examples of interactions with the assistant. During this period, interns were instructed to complete diaries at the end of each work shift, detailing the activities performed and their experiences with Copilot. As shown in Table 1, a total of 86 journal entries were collected and recorded.

In addition to the 86 diary entries, we also collected observational data from agile meetings throughout the 40-week study period. Specifically, the first author conducted participant observation during 7 retrospective meetings and 167 daily meetings, producing informal field notes focused on discussions related to GitHub Copilot use during PP sessions. These notes captured reflections

raised during retrospectives, issues discussed in daily meetings, and team-level perceptions regarding the integration of the AI assistant into PP practices. The observational notes were not treated as a primary data source nor analyzed independently. Instead, they were used as complementary material to contextualize, triangulate, and support the interpretation of diary data during the analysis process. Information from team retrospectives and other relevant meetings related to Copilot use was also included. Each record contained information like: Date; Author; Activity Description; Detailed Notes; Input Prompt; AI Suggestion; Whether or not the Suggestion Was Accepted; and Additional Notes. Figure 2 illustrates a diary entry after a PP activity on the student profile login feature, with the initial search prompts that did not return what was expected, so the participant continued to refine the prompt to obtain better results.

Instead of analyzing entire diary entries, tasks, or isolated Copilot interactions, we defined our unit of analysis as any coded excerpt with a meaningful episode with GitHub Copilot during a session. This means a single diary entry or set of meeting notes could yield several coded excerpts, each capturing different behaviors or outcomes—such as writing prompts together, accepting or rejecting AI code, or reflecting on learning and fatigue. These distinct excerpts served as the foundation for our coding and category development.

4.5 Data Analysis

As shown in Figure 1, the data analysis adopted grounded theory (GT) [32] techniques, i.e., the goal of this study was *not* to develop a theory, but to use GT procedures to organize, interpret, and categorize the collected data. The process began with *open coding* of the diaries and meeting records. Specifically, the first author performed a detailed inspection of the data to select entries related to Copilot use during PP sessions. This was done through a manual coding process, which resulted in an initial list of codes identifying keywords or relevant aspects in the data.

In the second phase, based on this initial set of codes, all the authors conducted repeated collaborative analyses to identify similarities among codes. We collaboratively performed a thorough review of the codes and made changes to their descriptions when necessary, to ensure they more clearly represented the information found in our data. Divergent interpretations were resolved through an analytic collaborative and consensus-based process grounded in the data, rather than through statistical agreement measures.

Then, we moved to *axial coding* [32] to create the following categories: central phenomena, causal conditions, context, intervening conditions, strategies, and consequences. The *Central Phenomenon* refers to your main focus — the core event, issue, or situation being studied. Meanwhile, *Causal conditions* describe why the phenomenon occurs — the underlying triggers or reasons behind it. The *Context* outlines the specific environment or circumstances in which the phenomenon takes place. Then, the *Intervening conditions* encompass broader factors that shape or influence what happens, such as personal background or available resources. After that, the *Strategies* capture how individuals or groups respond to or manage the situation. Finally, *Consequences* highlight the outcomes of those actions — both the expected and the unforeseen results. In other words, our final model contains this set of categories. In general, Strauss and Corbin [32] call this set of categories the paradigm (pg. 128) and this is "*nothing more than a perspective taken toward data, another analytic stance that helps to systematically gather and order data*". The objective of this analysis was to understand how the use of GitHub Copilot is impacting the traditional dynamics of PP.

Figure 3 illustrates an example of our coding process for analyzing qualitative data from both diaries and observations of meetings. While privacy and confidentiality constraints prevent us from sharing the complete raw dataset, we are fully committed to research transparency. We have therefore provided a replication package (see Section Artifact Availability) that carefully documents our coding process.

5 Results

The main output of this research is the conceptual model shown in Figure 4. Each of the following subsections describes a category from the paradigm [32].

5.1 Context and Conditions

At the outset of the project, GitHub Copilot was adopted during PP sessions as part of the team's development environment, motivated by an interest in understanding how AI-based coding assistants integrate into existing collaborative practices. The tool was not introduced as part of a controlled intervention, but rather to observe how PP practices evolve within their existing development workflow. This section describes the context and overarching conditions observed in the use of Copilot during PP sessions, which were categorized in the following categories: Context, Causal Conditions, and Intervening Conditions [32].

Our data showed the emergence of a triadic collaboration, which existed within a specific organizational **context**. The department already adopted agile practices in its workflow – including daily meetings, refactoring, retrospectives, and automated testing – and heavily relied on PP to promote knowledge sharing among team members. The imbalance in numbers and expertise among team members (see 4.3) created natural opportunities for mentorship and knowledge transfer in PP sessions, as reported by I1: *"The knowledge exchange between pairs during the analysis, review, and adjustment of Copilot-suggested code has helped reduce the learning curve among team members working with the new programming language."*

During the modernization of the legacy PHP system to Python-Django, the team began using Copilot during PP sessions to support coding tasks. The combination of an established PP culture, the uneven distribution of experience among team members, the modernization task, and the integration of Copilot — constituted the **context** that shaped the phenomenon under analysis.

The **causal conditions** included (i) the team's need to accelerate modernization tasks without compromising code quality and (ii) the limited team expertise in the new programming language, which led participants to rely more heavily on Copilot for guidance (as we discuss later in section 6.2). Meanwhile, we identified several **intervening conditions**: limited prior experience with Copilot, and limited knowledge of the new programming language: *"Since we are still learning the new programming language, our pairing sessions took longer because we were trying to understand the logic behind Copilot's suggestions before accepting them."* (I2)

Activity Diary

Date	Initial Time	End Time	Author
10/26/2023	9:22	12:34	I1
10/26/2023	9:16	12:13	I4

4. Activity Description

Implementation of the Student Login Page.

2. Did you use Intelligent Assistants to perform the activity? (If yes, which one?)

Yes, Copilot was used for this task.

3. Details

This task will begin with the implementation of the graphical interface for the student login page, but it will only be completed once the login page is fully functional. Therefore, we will start working on the back-end of the application and the database.

4. What is the input prompt?

4.1 - *"Create a login template in HTML using Django. The form should contain fields for email and password, as well as a submit button and a password recovery option below the password input."*

4.2 *- "Use Django forms to dynamically generate form fields."*

4.3 - *""I have a form in Django and I want to make it functional. Considering that this framework uses the MTV software architecture model, and that the login form contains the email and password fields, how do I make this login form functional; that is, perform a database query to verify the credentials and allow or deny access?"*

5. What suggestion was provided by the assistant?

Suggestion 1:

```
<!DOCTYPE html>
<html>
<head>
  <title>Login</title>
  <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/boo-
tstrap/4.5.0/css/bootstrap.min.css">
</head>
<body>
  <div class="container mt-5">
```

Figure 2: Example of activity diary entry.

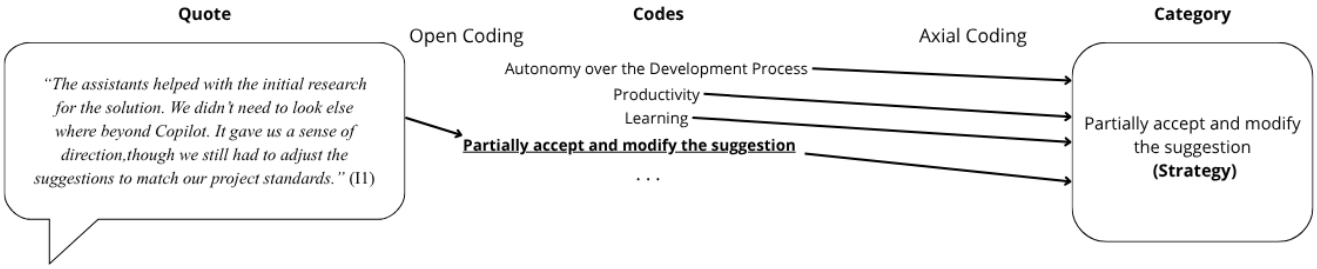

Figure 3: Coding process applied to the analysis of activity data.

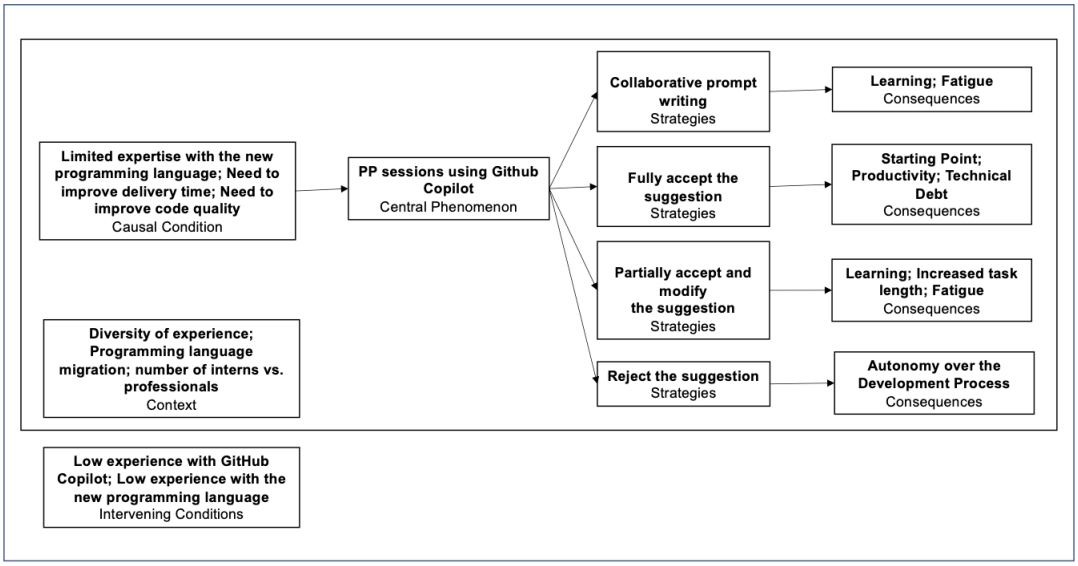

Figure 4: Visual Representation of our results: perceived positive and negative aspects.

5.2 PP Sessions using GitHub Copilot

Together, all the factors described in the previous section shaped the emergence of the **central phenomenon** we studied: "PP sessions using Copilot." The studied team managed this phenomenon by adopting a few *Strategies*, which include Collaboratively Writing Prompts, Fully Accepting Copilot's suggestions, Partially accepting the suggestions, or Rejecting these suggestions (see Figure 4. Following Strauss's and Corbin's concept of paradigm [32], each one of these *Strategies* or Actions has **Consequences** that are the outcomes of those actions, which, for our participants, lead to positive or negative results, suggesting an interesting trade-off.

5.3 Collaborative Prompt Writing

One of the clearest shifts in PP dynamics involved the emergence of **collaborative prompt writing**, in which both participants jointly crafted, refined, and evaluated prompts used to query the AI. Rather than being driven by stable role assignments, this activity unfolded through ongoing negotiation and shared sensemaking, transforming the PP sessions into an exploratory process of prompt engineering. This process extended beyond traditional practices such as task decomposition, implementation planning, or consulting official documentation and external tutorials. Interns reported that engaging in this shared prompt construction fostered greater reflection and mutual learning. For example, I1 mentioned: *"First, we write the input prompts and consult Copilot, then we discuss whether the suggestions are suitable for the context, and if needed (which is almost always the case...), we keep refining the solution."*

Participants described collaborative prompt writing as a practice that supported reflection and a shared sense of **learning**. By working together, pairs reported developing a shared understanding of the new programming language. As highlighted by I5:*"Discussing the prompts and the generated code helped me understand why certain approaches work better in Python. It felt like learning together while solving the task."* Despite its benefits, this strategy also introduced cognitive demands. The continuous joint evaluation of Copilot-generated code prolonged sessions and led to reports of **fatigue**: *"It often felt like we were pairing with three people – way more detailed and tiring"* (I1).

It is important to remember that I1 had already been conducting PP sessions for more than two years. He and I2 (another very experienced intern) often guided the process by proposing initial prompts and reviewing suggestions collaboratively. While this leadership facilitated learning among less experienced interns, it also increased the cognitive effort required, leading to fatigue for the more experienced members. This indicates that although prior experience enabled effective prompt construction, it came at a higher mental cost for those guiding the sessions.

5.4 Fully Accepting the Suggestion

The strategy of **fully accepting** Copilot's **suggestions** without modifications occurred in two distinct situations. First, when suggestions were deemed appropriate by the pairs, it led to positive outcomes, serving as a useful **starting point** and increasing interns' perceived **productivity** because they did not need to consult other sources of information, as explained by I1: *"The assistant has been helping me, allowing me to be more productive. It's good to have more search options for day-to-day development problems."* And, second, when the pairs were uncertain whether GitHub Copilot's suggestions were appropriate for the given context, and, despite that, they chose to accept these suggestions fully. Although this approach temporarily accelerated progress, it often resulted in **technical debt** [12] and subsequent rework, which developers only recognized later in the project (week 30 of 40). For example, participant I1, again one of the more experienced interns, acknowledged the risks involved: *"We accepted some recommendations related to application security, but we still have technical debt in that area."*

In PP sessions where Copilot's suggestions were fully accepted, less experienced interns, such as I3, often relied on Copilot's outputs as a starting point, perceiving it as a way to increase productivity without the cognitive load of writing code from scratch. However, more experienced interns occasionally also fully accepted suggestions. These occurrences were particularly noticeable in the context of tasks associated with application security, which they perceived as requiring specialized knowledge beyond their own expertise.

In summary, although full acceptance of Copilot's suggestions could boost short-term productivity, the lack of knowledge about the implemented solutions sometimes led to increased long-term maintenance costs.

5.5 Partially Accepting the Suggestion

To partially accept and modify the suggestion represented the most frequent strategy (see section 5.7). In most cases, the pair initially integrated Copilot's suggestion and then collaboratively modified it to ensure alignment with project standards and other project constraints. Typical examples can be seen in the reports of participants I2 and I1, respectively: *"After accepting it* [Github Copilot's suggestion], *we had to make some modifications so that the final result would meet the established criteria."*; *"The assistant's suggestions were too robust for what we needed, so we took some time to understand and adapt them to our context."*

Participants **reported experiencing collective learning and increased knowledge sharing** when collaboratively modifying Copilot's suggestions, especially those less experienced with the new programming language. As mentioned by P1: *"Sharing ideas while analyzing and improving Copilot's code helped the developers learn the new language faster."*

However, intense peer interactions to review code suggestions generated by Copilot were cited as a factor contributing to **increased task length**. As reported by I3: *"We spent much longer than usual finishing the task because every suggestion required checking and rewriting before it actually worked."*

These longer discussions, compared to the ones without Copilot, and the rework cycles also contributed to **fatigue**, especially during consecutive days of intensive pairing, as participant I4 explained: *"By the end of the week, we were exhausted. Reviewing Copilot's suggestions took more energy than writing code from scratch."*

Partially accepting and modifying Copilot's suggestions was a strategy observed among all participants. However, it was more predominant among the more experienced interns: I1 and I2. They frequently adjusted Copilot's suggestions to ensure compliance with project standards, improve readability, and maintain overall code quality. Less experienced interns also employed this strategy,

but less consistently, often relying on guidance from their more experienced peers to determine how to modify the suggestions.

While this approach allowed greater control over code quality and facilitated learning for less experienced interns, it also increased task duration and cognitive load. Several participants, especiallhy those with more experienced, reported fatigue as a consequence of the additional effort required to carefully review and adapt Copilot's output. These findings indicate that partial acceptance balances the benefits of quality and learning with increased effort, and that prior experience influences how this strategy is applied within the team.

5.6 Rejecting the Suggestion

Finally, a last strategy we observed was that sometimes pairs **rejected the suggestion**. These rejections were particularly common in front-end development or environment configuration tasks, where Copilot's suggestions were perceived as irrelevant or overly generic, as participant I2 explained: *"The assistant's suggestion for working with the API just confused us."*

Front-end developers highlighted their preference to adhere to the organization's established style guides, while also expressing a strong desire to preserve **autonomy over the development process**, as I2 explained: *"We're sticking more to our pre-established style guide than to Copilot's suggestions for design-related activities."*

All participants occasionally rejected Copilot's code suggestions when they conflicted with their personal understanding or coding preferences. Among them, intern I2 stood out as the most experienced in front-end tasks and he frequently preferred to maintain greater control over the development process. On several occasions, he rejected Copilot's suggestions not because they were incorrect, but to ensure that the code adhered to established standards and his own implementation approach.

Less experienced interns also rejected suggestions, primarily when they were uncertain about the appropriateness of the AI's suggestions or when guided by their partners. This pattern highlights that prior experience not only shapes the decision to accept or reject AI-generated code but also influences the level of autonomy interns seek during PP sessions.

5.7 Overall Results

As mentioned before, the team wrote 86 journal entries, or simply reports, which were organized into twelve main dimensions grouped under two categories: *Strategies and Consequences* (see section 4.5). Specifically, the collaborative prompt writing strategy (29 reports) was associated with the consequences of learning (26) and fatigue (3). Fully accepting the suggestion strategy (26), led to consequences such as productivity (11), starting point (3), and technical debt (12). The partial acceptance and modification of the suggestion strategy (16) was related to learning (4), increased task length (11), and fatigue (1). Finally, the reject the suggestion strategy was exclusively associated with autonomy over the development process (15). These relationships are illustrated in Table 1.

Table 1: Summary of strategies, associated consequences, and number of quotes.

Strategy	Consequences (quotes)
Collaborative prompt writing (29)	Learning (26); Fatigue (3)
Fully accept suggestion (26)	Productivity (11); Starting point (3); Technical debt (12)
Partial acceptance (16)	Learning (4); Increased task length (11); Fatigue (1)
Reject suggestion (15)	Autonomy over development process (15)

6 Discussion

The research question explored in this study is: "*How does the use of AI code assistants impact pair programming dynamics?*" It can be answered through the results presented in Figure 4: first, we identified the overarching scenario - including the context, the causal, and intervening conditions [32] - that influence how the PP sessions using Copilot take place (see section 5.1). This scenario directly influences our central phenomenon: the usage of Copilot in PP sessions. Then, we highlighted how the studied development team adopted four different *Strategies* while dealing with this phenomenon: (i) they collaboratively wrote prompts; (ii) they fully accepted Copilot suggestions when they found these suggestions appropriate or when they did not have enough experience in a particular domain (e.g., in security) to assess them; (iii) they accepted the suggestions, but changed them because these code suggestions were good starting points, but required adaptation; or (iv) they rejected the suggestions, especially if these suggestions were about front-end aspects. These strategies are not isolated or mutually exclusive; participants used many of them during the same PP section.

Another point to consider is the timeline of the technology itself. While this study captures GitHub Copilot at a specific moment in time, AI assistants have moved forward rapidly since our data was collected—especially in their context awareness and conversational features. Even so, the core value of this research lies not in the software's exact version, but in the social and collaborative dynamics that happen when an AI tool enters a pair programming setup.

To be clear, we are not suggesting that outcomes like learning, productivity, fatigue, or development challenges are exclusive to AI-assisted setups—traditional pair programming literature has explored these themes for decades. Our focus is instead on how these familiar phenomena are fundamentally reconfigured when an AI assistant enters the loop. Specifically, we highlight how the collaboration becomes triadic through shared practices like collaborative prompting, collective evaluation of AI outputs, and the constant negotiation over whether to trust, tweak, or reject AI code.

The **collaborative prompt writing** *Strategy* illustrates that PP interactions were not organized around fixed role assignments. Rather, both participants actively engaged in the joint construction, refinement, and evaluation of prompts, reflecting a shared and negotiated approach to coordinate interaction with the AI assistant. In general, existing studies emphasize the importance of prompt writing, but they either describe individual scenarios or implicitly assume that this is carried out individually [23, 46]. Our observations, however, are in line with recent studies that suggest that writing prompts is a collaborative task [14]. In fact, we present

evidence that experienced agile teams, such as the one we studied, share experiences and prompts while working together. These observations also suggest that, as AI code assistants evolve, new responsibilities might be added to PP roles to support this new step in PP sessions, with a potential shift of "responsibilities" to the AI code assistants — for instance, the assistant may take over the task of initiating code solutions, while the human pair focuses on interacting with the AI code assistant to refine this process. An interesting research aspect might be to observe these changes as they evolve with technology use, i.e., how the reported "pairs of three" may change the logic of pair programming. This also means that current and new software development tools (e.g., [34]), should be augmented to support this collaboration and minimize tool switching for a single task.

As for the other three *Strategies* regarding the acceptance or rejection of Copilot suggestions, each one of them has either positive or negative *Consequences*, i.e., the use of an AI code assistant during PP sessions can have *positive* or *negative* impacts. In the following sections, we discuss aspects that participants themselves described as beneficial or challenging when using GitHub Copilot during pair programming. These 'positive' and 'negative' aspects reflect participants' perceptions as reported in diaries and meetings, and are interpreted by the authors through qualitative analysis. Building on this discussion, we subsequently articulate empirically grounded implications for practice, research, and education in Section 5.4.

6.1 Aspects Perceived as Beneficial

Before diving into the benefits and challenges, we need to clarify the distinction between the practices we observed and the consequences perceived by the participants. On one hand, actions like collaborative prompt writing or the choice to accept, modify, and reject AI suggestions emerged directly from our qualitative analysis of diaries and notes. On the other hand, outcomes such as fatigue, learning opportunities, perceived productivity, or worries about technical debt are drawn from the participants' own subjective accounts of their experience.

Regarding aspects perceived as *positive* by the interns, participants frequently described Copilot suggestions as useful **starting points** [2, 22, 37] for certain development tasks whenever they decided to partially accept the code suggestions. There was also evidence that the use of the AI code assistant improved the **perceived productivity** of the pair, which also supports previous studies [11]. We extend prior results by describing how PP with Copilot was associated with participants' perceptions of **increased interaction between pair members**. According to the interns, AI-generated suggestions often triggered joint reflection, questioning, and discussion, which required pairs to collectively interpret and evaluate the assistant's output. Such increased interaction contributed to enhanced team learning outcomes and increased knowledge sharing. The reason for increased interaction, compared to traditional PP sessions, is that pairs now have an AI code assistant easily offering code suggestions for them, and they have to analyze this code and decide whether they should use it or not.

As we discussed in section 2, increased team interaction has been reported in the literature [28, 35, 38] as a benefit of PP. However, our results are different: the entire development team already had at least one year of experience with PP, and, despite that, they reported greater interaction among them during the PP sessions using Copilot. In summary, our results suggest that PP using an AI assistant leads to even more interaction than pairing without one.

Our results are based on participants' experiences during a modernization project using GitHub Copilot; therefore, we do not have prior data for a direct comparison with "traditional" PP sessions. However, the participants themselves reported differences. In particular, we observed that pairs relied on Copilot to learn the new programming language adopted, in a behavior analogous to the typical use of Stack Overflow [8]. It is important to highlight that we did not ask participants to focus on specific aspects, such as types of discussions or information-seeking patterns; instead, we asked them to record general aspects of their interaction with Copilot. As a result, information-seeking behavior practices were organically discussed by participants and strongly present in the data. Still, when comparing PP sessions with and without Copilot based on participants' reports, we identified notable differences in the nature of discussions and sources of information used. In traditional PP, conversations usually focused on task decomposition, implementation strategies, and consultation of official documentation or external tutorials. With Copilot, however, the focus of discussions shifted to the analysis, adaptation, or rejection of the assistant's suggestions. This shift intensified the interaction within pairs, who began to collaborate not only in code review but also in writing prompts and critically evaluating AI responses.

6.2 Aspects Perceived as Challenging

There are also *negative* aspects associated with the adoption of an AI code assistant in PP sessions. Several participants described longer and more detailed sessions as **challenging or tiring**, which they perceived as a negative aspect of using Copilot during a PP session. Importantly, aspects such as longer and more detailed sessions were not labeled as negative solely by the researchers; rather, participants themselves repeatedly described these experiences as tiring and cognitively demanding in their diary entries. In general, increased development time and effort for task completion have been reported in previous PP literature [4, 42, 43]. As mentioned in section 2, to the best of our knowledge, there are no other studies specifically examining PP in a triadic setting, that is, sessions in which two human developers collaborate simultaneously with an AI code assistant. The most closely related studies include Imai [16], Chen [9], and Zhou et al. [47]. Imai reported that having a copilot and a human as pair members resulted in slower task completion compared to two human programmers without an AI code assistant. Similarly, Chen found that while AI pair programmers improved developer satisfaction and reduced repetitive work, they also introduced challenges related to overreliance on AI-generated suggestions and decreased critical evaluation of generated code. Likewise, Zhou et al. identified coordination difficulties and inconsistencies produced by AI code assistants, and cognitive overload when developers interacted with them during collaborative programming tasks. It is important to mention that these studies were conducted under controlled or limited-scope conditions, whereas our study took place in a real-world setting with interns developing a full software application over 40 weeks. While Imai conducted a controlled

experiment, Chen examined the adoption of AI pair programmers in an industrial environment, and Zhou et al. analyzed large-scale developer discussions from GitHub and Stack Overflow. Despite the contextual differences, all three studies converge in showing that coordination with AI code assistants can slow down development due to increased cognitive effort and discussion overhead.

Another interesting result of our study is the problematic **over-reliance** on an AI code assistant, which in this case is instantiated by blindly accepting code suggestions because the pair lacked sufficient knowledge about the problem. While team members *did* recognize this situation as problematic because it could potentially lead to the emergence of **technical debt** [12], they only recognized it very late in the project (week 30 of 40). Therefore, only after that, they actively sought strategies to mitigate this problem. Although LLMs can introduce technical debt when used indiscriminately, they may also be part of the solution. For instance, Sheikhaei et al. [31] reported inspiring results on automatic identification and classification of existing technical debt found in code comments, and AlOmar et al. [1] discuss refactoring legacy of low-quality code into cleaner and more modern versions using ChatGPT.

It is important to mention that this **over-reliance** happened in a very specific context: the team was learning a new programming language, and the task was one about security aspects, a topic in which most software developers had limited knowledge [40]. Despite this specificity, other teams adopting AI assistants might need to be careful about this over-reliance and be prepared to adopt work practices to avoid it from the very beginning of the project.

6.3 Comparison with a Human+AI Pair

Previous studies [16], examined scenarios in which a single programmer works together with Copilot, thus forming a pair composed of a human and an AI. They suggest that AI code assistants can enhance developer productivity and satisfaction, although the effects on code quality remain mixed — with some studies reporting improvements in readability or maintainability ([37], [9], [2], [11]), while others [47] highlight coordination challenges, the absence of systematic peer review, and the risk of unnoticed errors. More recent studies [9, 19] reported similar trade-offs: although developers often experienced faster task completion, they also noted challenges related to excessive reliance on the AI code assistant, reduced critical thinking, and potential risks to code quality.

In contrast, in our study, GitHub Copilot did *not* replace a human partner but was embedded in traditional PP sessions, creating a dynamic of "three participants." This difference is relevant: while in the human+AI pair, there is a higher risk of uncritically accepting suggestions, in the human+human+AI trio, we observed more frequent review and discussion of the AI code assistant suggestions. This promoted learning and knowledge sharing but also led to longer sessions, increased fatigue, and more complex interactions. In other words, the human+AI pair tends to privilege speed, whereas the human+human+AI trio privileges reflection, debate, and potentially deeper learning—though at the cost of higher effort.

6.4 Implications for Practice

Our observations suggest that AI-assisted PP is neither a simple productivity boost nor a neutral extension of traditional PP. It reshapes coordination among the pair, redistributes cognitive effort, and introduces new risks alongside new opportunities. Based on our longitudinal observations, we highlight five practical implications.

Treat Prompt Writing as a First-Class Activity: Prompt writing is not a peripheral interaction with a tool; it is a collaborative design practice. Teams should explicitly recognize prompting as part of their development workflow. This means prompts should be treated as lightweight development artifacts, i.e., making prompting explicit helps transform it from ad hoc experimentation into deliberate "collaboration" between the pairs and the AI assistant. Agile teams should also reflect on prompt formulation during retrospective meetings to share the lessons learned among the team.

Explicitly Discuss Decisions: Rejecting or accepting AI-generated code – whether fully or partially – should *not* be implicit. Our results show that unexamined full acceptance can introduce subtle technical debt. Thus, pairs should always discuss and articulate the reasons when making a decision about a prompt (accept, modify or reject). Our study suggests that discussing decisions reduces hidden complexity and helps prevent long-term maintainability issues.

Anticipate Cognitive Load: While AI assistance often increases *perceived* productivity, it also shifts effort toward evaluation and validation, i.e., our results indicate that continuous assessment of generated code can be cognitively demanding. In summary, productivity gains should not come at the expense of sustained mental overload. To handle this, it is important to schedule short breaks during long AI-assisted sessions; alternate who leads evaluation discussions; and be mindful of decision fatigue.

Develop Shared Norms Around Trust: Trust in AI-generated suggestions should not be treated as a fixed property of the tool, but as something that evolves through use. Our observations show that, over time, pairs do not simply become more or less trusting of the assistant; instead, they developed shared and situational norms about when its suggestions could be relied upon and when they required careful scrutiny. In practice, this means that trust becomes calibrated based on factors such as task complexity, familiarity with the codebase, and perceived risk. For example, pairs tended to rely more on the assistant for simple or repetitive tasks, while treating its suggestions as provisional in more critical parts of the system. In design and UX-related tasks, developers often chose to disregard the assistant altogether due to its limited usefulness in these contexts.

These norms are not established individually, but ongoingly negotiated between the pair. Discussions around accepting, adapting, or rejecting suggestions play a central role in shaping a shared understanding of the assistant's strengths and limitations, echoing prior findings on how developers actively interpret and ground AI-generated code in practice [2]. As a result, trust becomes a collective and continuously evolving aspect of collaboration, rather than an implicit or static assumption, reinforcing broader observations that effective use of generative AI in software development requires continuous human evaluation and critical judgment [13]. This means agile teams should explicitly discuss when AI should be used more and when human reasoning should prevail; revisit trust assumptions periodically; and encourage healthy skepticism without dismissing AI contributions outright.

7 Limitations

One of the authors is a member of the studied team. While this enabled close access to everyday practices and facilitated trust and openness among participants, it may also have introduced biases. The researcher's presence may have influenced which interactions with Copilot were considered noteworthy and thus documented in the diaries, potentially emphasizing reflective, problematic, or unusual episodes over routine or unremarkable uses. Although the study spanned approximately 40 weeks—allowing practices to stabilize over time—this influence cannot be fully ruled out.

Also, the study's primary data source consists of participants' self-reported diary entries, which do not represent exhaustive logs of all interactions with GitHub Copilot. Instead, each diary entry typically reflects participants' retrospective and selective accounts of interactions they considered salient during a PP session. Such sessions often lasted several hours and could involve multiple Copilot suggestions; consequently, a single diary entry may summarize or amalgamate several interactions rather than documenting them individually. As a result, the 86 diary entries should not be interpreted as capturing the full frequency or distribution of Copilot usage, but rather as highlighting interactions that participants themselves deemed meaningful, challenging, or worth reflecting upon.

Additionally, diary entries were typically written collaboratively by the pairs, resulting in negotiated accounts rather than independent individual perspectives. While this collaborative reporting aligns with the PP setting and foregrounds shared interpretations, it may also mask individual differences in perception, disagreement between pair members, or asymmetric experiences with Copilot. Individual diary entries could provide more fine-grained views of how different participants experienced the same interaction, but such data were not systematically collected in this study.

In addition, our study was conducted with a team composed mostly of interns with at least one year of experience at the organization. These interns can be regarded as junior professionals. Also, all team members were male, which might affect how the LLM answered queries [21]. We also did not control for race, which might have implications during the PP interactions [20]. Future studies could diversify the studied population to better explore diversity aspets like gender and race, and AI proficiency.

Another limitation concerns the composition of the studied team. Pair programming sessions were conducted exclusively by junior developers (interns), while senior professionals contributed through mentoring, requirements elicitation, testing, code reviews, and Agile meetings rather than participating directly in coding sessions. Therefore, our findings reflect AI-assisted pair programming among junior developers and may differ in settings where senior developers actively participate in pair programming activities.

Finally, the findings presented in this paper are grounded in participants' perceived experiences and in the researchers' interpretive analysis of self-reported data, rather than in independently measured performance metrics or controlled comparisons with pair programming sessions conducted without GitHub Copilot. As such, claims regarding learning, collaboration, or task characteristics should be understood as reflecting how participants made sense of their experiences within a specific organizational context, rather than as objective or universally applicable effects. Consequently, the results of this study are not intended to support statistical generalization. Instead, they contribute analytically and conceptually, offering insights that may support **theoretical generalization** to similar contexts where PP and AI-based code assistants are adopted.

8 Conclusions and Future Work

This paper described an empirical study about the actual use of AI code assistants in pair programming (PP) sessions during a 40-week long modernization task. Data collected from diaries and observation of meetings allowed us to identify the context and conditions that affect this usage, as well as the strategies developers use to deal with this situation and their associated positive and negative consequences. The positive consequences, or benefits, included learning, increased perceived productivity, among other aspects. Meanwhile, negative aspects included more time-consuming and tiring work, and even increased technical debt. This is a trade-off that requires additional exploration in future studies. Based on these findings, we articulated empirically grounded implications for practice highlighting how AI-assisted pair programming reshapes coordination, learning, and workload in collaborative software development.

Furthermore, while previous studies emphasized the importance of prompt writing by either describing individual scenarios or implicitly assuming the software development task was carried out by a unique developer, we present evidence that prompt writing also occurs *collaboratively*, i.e., experienced agile teams share experiences about prompt writing and the prompts themselves when working together. This suggests interesting venues for future research including tool support for prompt writing.

As future work, several variations of our study context can be explored. For instance, interviews with a more diverse group of developers could help assess how the coordination demands, learning dynamics, and workload implications observed in this study manifest in other real-world settings. In addition, conducting large-scale surveys with a broader spectrum of professionals may help examine the extent to which the perceived benefits, challenges, and trade-offs identified here hold across different levels of experience and organizational contexts.

Future research could also compare PP practices across different AI code assistants beyond GitHub Copilot, in order to better understand how specific tool features shape collaborative dynamics. In particular, recently introduced agentic software development tools, such as Claude Code, OpenAI Codex, and similar systems, warrant investigation, as their increased autonomy may further amplify or transform the coordination, responsibility, and cognitive load implications discussed in this paper.

ARTIFACT AVAILABILITY

The replication package including the study materials required to understand and replicate the analysis is publicly available at https://doi.org/10.5281/zenodo.21272097.

ACKNOWLEDGEMENTS

We would like to thank the members of the studied software development team for their help. This work was partially supported by CNPq grants 420406/2023-9, 444802/2024-0, and 308101/2025-1.

References

[1] Eman Abdullah AlOmar, Anushkrishna Venkatakrishnan, Mohamed Wiem Mkaouer, Christian Newman, and Ali Ouni. 2024. How to refactor this code? An exploratory study on developer-ChatGPT refactoring conversations. In *Proceedings of the 21st International Conference on Mining Software Repositories* (Lisbon, Portugal) *(MSR '24)*. Association for Computing Machinery, New York, NY, USA, 202–206. doi:10.1145/3643991.3645081

[2] Shraddha Barke, Michael B. James, and Nadia Polikarpova. 2023. Grounded Copilot: How Programmers Interact with Code-Generating Models. *Proc. ACM Program. Lang.* 7, OOPSLA1, Article 78 (April 2023), 27 pages. doi:10.1145/3586030

[3] K Beck. 2000. *Extreme Programming Explained.* Massachusetts: Addison-Wesley.

[4] Andrew Begel and Nachiappan Nagappan. 2008. Pair programming: what's in it for me?. In *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement* (Kaiserslautern, Germany) *(ESEM '08)*. Association for Computing Machinery, New York, NY, USA, 120–128. doi:10.1145/1414004.1414026

[5] Susan Bryant, Pedro Romero, and Benedict du Boulay. 2006. The Collaborative Nature of Pair Programming. In *Extreme Programming and Agile Processes in Software Engineering (Lecture Notes in Computer Science, Vol. 4044)*. Springer, 53–65. doi:10.1007/11774129_6

[6] Susan Bryant, Pedro Romero, and Benedict du Boulay. 2008. Pair Programming and the Mysterious Role of the Navigator. *International Journal of Human-Computer Studies* 66, 7 (2008), 519–529. doi:10.1016/j.ijhcs.2007.03.005

[7] Jenna Butler, Jina Suh, Sankeerti Haniyur, and Constance Hadley. 2024. Dear Diary: A randomized controlled trial of Generative AI coding tools in the workplace. arXiv:2410.18334 [cs.SE] https://arxiv.org/abs/2410.18334

[8] Jing Chen and Zhenchang Xing. 2016. Mining Technology Landscape from Stack Overflow. In *Proceedings of the 13th International Conference on Mining Software Repositories (MSR)*. 220–231. doi:10.1145/2901739.2901757

[9] Tianyi Chen. 2024. The Impact of AI-Pair Programmers on Code Quality and Developer Satisfaction: Evidence from TiMi studio. In *Proceedings of the 2024 International Conference on Generative Artificial Intelligence and Information Security* (Kuala Lumpur, Malaysia) *(GAIIS '24)*. Association for Computing Machinery, New York, NY, USA, 201–205. doi:10.1145/3665348.3665383

[10] Jan Chong and Tom Hurlbutt. 2007. The Social Dynamics of Pair Programming. In *Proceedings of the 29th International Conference on Software Engineering (ICSE)*. IEEE Computer Society. doi:10.1109/ICSE.2007.87

[11] Zheyuan Cui, Mert Demirer, Sonia Jaffe, Leon Musolff, Sida Peng, and Tobias Salz. 2025. The Effects of Generative AI on High-Skilled Work: Evidence from Three Field Experiments with Software Developers. doi:10.2139/ssrn.4945566 Available at SSRN: https://ssrn.com/abstract=4945566.

[12] Ward Cunningham. 1992. The WyCash portfolio management system. In *Addendum to the Proceedings on Object-Oriented Programming Systems, Languages, and Applications (Addendum)* (Vancouver, British Columbia, Canada) *(OOPSLA '92)*. Association for Computing Machinery, New York, NY, USA, 29–30. doi:10.1145/157709.157715

[13] Christof Ebert and Panos Louridas. 2023. Generative AI for Software Practitioners. *IEEE Software* 40, 4 (2023), 30–38.

[14] Li Feng, Ryan Yen, Yuzhe You, Mingming Fan, Jian Zhao, and Zhicong Lu. 2024. CoPrompt: Supporting Prompt Sharing and Referring in Collaborative Natural Language Programming. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems* (Honolulu, HI, USA) *(CHI '24)*. Association for Computing Machinery, New York, NY, USA, Article 934, 21 pages. doi:10.1145/3613904.3642212

[15] Jo E. Hannay, Tore Dybå, Erik Arisholm, and Dag I. K. Sjoberg. 2009. The Effectiveness of Pair Programming: A Meta-Analysis. *Information and Software Technology* 51, 7 (2009), 1110–1122. doi:10.1016/j.infsof.2009.02.001

[16] Saki Imai. 2022. Is GitHub copilot a substitute for human pair-programming? an empirical study. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 319–321. doi:10.1145/3510454.3522684

[17] Simon Jones and Scott Fleming. 2013. What Use Is a Backseat Driver? A Qualitative Investigation of Pair Programming. In *Proceedings of the 2013 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE. doi:10.1109/VLHCC.2013.6645252

[18] Kei Koyanagi, Dong Wang, Kotaro Noguchi, Masanari Kondo, Alexander Serebrenik, Yasutaka Kamei, and Naoyasu Ubayashi. 2024. Exploring the Effect of Multiple Natural Languages on Code Suggestion Using GitHub Copilot. In *Proceedings of the 21st International Conference on Mining Software Repositories* (Lisbon, Portugal) *(MSR '24)*. Association for Computing Machinery, New York, NY, USA, 481–486. doi:10.1145/3643991.3644917

[19] Qianou Christina Ma, Tongshuang (Sherry) Wu, and Kenneth R. Koedinger. 2023. Is AI the better programming partner? Human–Human Pair Programming vs. Human–AI pAIr Programming. In *Empowering Education with LLMs—The Next-Gen Interface and Content Generation (AIED 2023 Workshop) (CEUR Workshop Proceedings, Vol. 3487)*. CEUR-WS.org, Tokyo, Japan. https://ceur-ws.org/Vol-3487/paper3.pdf

[20] Shandler A. Mason and Sandeep Kaur Kuttal. 2023. Investigating Interracial Pair Coordination During Remote Pair Programming. In *2023 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 260–262. doi:10.1109/VL-HCC57772.2023.00047

[21] Alexander McAuliffe, Jacob Hart, and Sandeep Kaur Kuttal. 2022. Evaluating Gender Bias in Pair Programming Conversations with an Agent. In *2022 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 1–4. doi:10.1109/VL/HCC53370.2022.9833146

[22] Wendy Mendes, Samara Souza, and Cleidson De Souza. 2024. "You're on a bicycle with a little motor": Benefits and Challenges of Using AI Code Assistants. In *Proceedings of the 2024 IEEE/ACM 17th International Conference on Cooperative and Human Aspects of Software Engineering* (Lisbon, Portugal) *(CHASE '24)*. Association for Computing Machinery, New York, NY, USA, 144–152. doi:10.1145/3641822.3641882

[23] Aditi Mishra, Bretho Danzy, Utkarsh Soni, Anjana Arunkumar, Jinbin Huang, Bum Chul Kwon, and Chris Bryan. 2025. PromptAid: Visual Prompt Exploration, Perturbation, Testing and Iteration for Large Language Models. *IEEE Transactions on Visualization and Computer Graphics* 31, 10 (2025), 6946–6962. doi:10.1109/TVCG.2025.3535332

[24] Nhan Nguyen and Sarah Nadi. 2022. An empirical evaluation of GitHub copilot's code suggestions. In *Proceedings of the 19th International Conference on Mining Software Repositories* (Pittsburgh, Pennsylvania) *(MSR '22)*. Association for Computing Machinery, New York, NY, USA, 1–5. doi:10.1145/3524842.3528470

[25] Shuyin Ouyang, Jie M. Zhang, Mark Harman, and Meng Wang. 2025. An Empirical Study of the Non-Determinism of ChatGPT in Code Generation. *ACM Transactions on Software Engineering and Methodology* 34, 2 (Jan. 2025), 1–28. doi:10.1145/3697010

[26] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2022. Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions. In *2022 IEEE Symposium on Security and Privacy (SP)*. 754–768. doi:10.1109/SP46214.2022.9833571

[27] Laura Plonka, Helen Sharp, and Janet van der Linden. 2011. Collaboration in Pair Programming: Driving and Switching. In *Agile Processes in Software Engineering and Extreme Programming (Lecture Notes in Computer Science, Vol. 212)*. Springer, 32–47. doi:10.1007/978-3-642-20677-1_4

[28] Bernhard Rumpe and Astrid Schröder. 2002. Quantitative Survey on Extreme Programming Projects. arXiv:1409.6599 [cs.SE] https://arxiv.org/abs/1409.6599

[29] Sharon Salinger, Laura Plonka, and Helen Sharp. 2013. Liberating Pair Programming Research from the Oppressive Driver/Observer Regime. In *Proceedings of the 35th International Conference on Software Engineering (ICSE)*. IEEE. doi:10.1109/ICSE.2013.6606678

[30] Norsaremah Salleh, Emilia Mendes, and John Grundy. 2011. Empirical Studies of Pair Programming for CS/SE Teaching in Higher Education: A Systematic Literature Review. *IEEE Transactions on Software Engineering* 37, 4 (2011), 509–525. doi:10.1109/TSE.2010.59

[31] Mohammad Sadegh Sheikhaei, Yuan Tian, Shaowei Wang, and Bowen Xu. 2024. An empirical study on the effectiveness of large language models for SATD identification and classification. *Empirical Softw. Engg.* 29, 6 (Oct. 2024), 34 pages. doi:10.1007/s10664-024-10548-3

[32] Anselm Strauss and Juliet Corbin. 1990. *Basics of Qualitative Research: Grounded Theory Procedures and Techniques.* Sage Publications, Newbury Park, CA.

[33] Viktoria Stray, Nils Brede Moe, Nivethika Ganeshan, and Simon Kobbenes. 2025. Generative AI and Developer Workflows: How GitHub Copilot and ChatGPT Influence Solo and Pair Programming. In *Proceedings of the 58th Hawaii International Conference on System Sciences (HICSS)*. Hawaii International Conference on System Sciences, 7381–7390. doi:10.24251/HICSS.2025.XXX

[34] Juan Pablo Sáenz and Luigi De Russis. 2022. Dear Diary: On Documenting Novices' Development Process. In *2022 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 1–3. doi:10.1109/VL/HCC53370.2022.9833003

[35] Bjørnar Tessem. 2003. Experiences in learning XP practices: a qualitative study. In *Proceedings of the 4th International Conference on Extreme Programming and Agile Processes in Software Engineering* (Genova, Italy) *(XP'03)*. Springer-Verlag, Berlin, Heidelberg, 131–137.

[36] Christoph Treude and Hideaki Hata. 2023. She Elicits Requirements and He Tests: Software Engineering Gender Bias in Large Language Models. arXiv:2303.10131 [cs.SE] https://arxiv.org/abs/2303.10131

[37] Priyan Vaithilingam, Tianyi Zhang, and Elena L. Glassman. 2022. Expectation vs.nbsp;Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models. In *Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems* (New Orleans, LA, USA) *(CHI EA '22)*. Association for Computing Machinery, New York, NY, USA, Article 332, 7 pages. doi:10.1145/3491101.3519665

[38] J. Vanhanen and C. Lassenius. 2005. Effects of pair programming at the development team level: an experiment. In *2005 International Symposium on Empirical Software Engineering, 2005.* 10 pp.–. doi:10.1109/ISESE.2005.1541842

[39] Jari Vanhanen and Mika V. Mäntylä. 2013. A Systematic Mapping Study of Empirical Studies on the Use of Pair Programming in Industry. *International Journal of Software Engineering and Knowledge Engineering* 23, 8 (2013), 1101–1132. doi:10.1142/S0218194013500381

[40] Charles Weir, Ingolf Becker, and Lynne Blair. 2021. A Passion for Security: Intervening to Help Software Developers. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 21–30. doi:10.1109/ICSE-SEIP52600.2021.00011

[41] Laurie Williams and Robert Kessler. 2002. *Pair Programming Illuminated*. Addison-Wesley Longman Publishing Co., Inc., USA.

[42] Laurie Williams, Robert R. Kessler, Ward Cunningham, and Ron Jeffries. 2000. Strengthening the Case for Pair Programming. *IEEE Softw.* 17, 4 (July 2000), 19–25. doi:10.1109/52.854064

[43] Laurie Williams, Robert R. Kessler, Ward Cunningham, and Ron Jeffries. 2001. The Costs and Benefits of Pair Programming. In *Extreme Programming Examined*, Laurie Williams and Robert R. Kessler (Eds.). Addison-Wesley Longman Publishing Co., Inc., USA, 223–243.

[44] Zhou Yang, Zhipeng Zhao, Chenyu Wang, Jieke Shi, Dongsun Kim, Donggyun Han, and David Lo. 2024. Unveiling Memorization in Code Models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 72, 13 pages. doi:10.1145/3597503.3639074

[45] Burak Yetistiren, Isik Ozsoy, and Eray Tuzun. 2022. Assessing the quality of GitHub copilot's code generation. In *Proceedings of the 18th International Conference on Predictive Models and Data Analytics in Software Engineering* (Singapore, Singapore) *(PROMISE 2022)*. Association for Computing Machinery, New York, NY, USA, 62–71. doi:10.1145/3558489.3559072

[46] J.D. Zamfirescu-Pereira, Richmond Y. Wong, Bjoern Hartmann, and Qian Yang. 2023. Why Johnny Can't Prompt: How Non-AI Experts Try (and Fail) to Design LLM Prompts. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems* (Hamburg, Germany) *(CHI '23)*. Association for Computing Machinery, New York, NY, USA, Article 437, 21 pages. doi:10.1145/3544548.3581388

[47] Xiyu Zhou, Peng Liang, Beiqi Zhang, Zengyang Li, Aakash Ahmad, Mojtaba Shahin, and Muhammad Waseem. 2025. Exploring the problems, their causes and solutions of AI pair programming: A study on GitHub and Stack Overflow. *Journal of Systems and Software* 219 (2025), 112204. doi:10.1016/j.jss.2024.112204

[48] Albert Ziegler, Eirini Kalliamvakou, X. Alice Li, Andrew Rice, Devon Rifkin, Shawn Simister, Ganesh Sittampalam, and Edward Aftandilian. 2022. Productivity assessment of neural code completion. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming* (San Diego, CA, USA) *(MAPS 2022)*. Association for Computing Machinery, New York, NY, USA, 21–29. doi:10.1145/3520312.3534864