The Impact of Generative AI on Collaborative Open-Source Software Development: Evidence from GitHub Copilot

Fangchen Song, Ashish Agarwal, Wen Wen

Abstract

Generative artificial intelligence (AI) has opened the possibility of automated content production, including coding in software development, which can significantly influence the participation and performance of software developers. To explore this impact, we investigate the role of GitHub Copilot, a generative AI pair programmer, on software development in open-source community, where multiple developers voluntarily collaborate on software projects. Using GitHub's dataset for open-source repositories and a generalized synthetic control method, we find that Copilot significantly enhances project-level productivity by 6.5%. Delving deeper, we dissect the key mechanisms driving this improvement. Our findings reveal a 5.5% increase in individual productivity and a 5.4% increase in participation. However, this is accompanied with a 41.6% increase in integration time, potentially due to higher coordination costs. Interestingly, we also observe the differential effects among developers. We discover that core developers achieve greater project-level productivity gains from using Copilot, benefiting more in terms of individual productivity and participation compared to peripheral developers, plausibly due to their deeper familiarity with software projects. We also find that the increase in project-level productivity is accompanied with no change in code quality. We conclude that AI pair programmers bring benefits to developers to automate and augment their code, but human developers' knowledge of software projects can enhance the benefits. In summary, our research underscores the role of AI pair programmers in impacting project-level productivity within the open-source community and suggests potential implications for the structure of open-source software projects.

Keywords: Generative AI, Open-source Software Development, Project-level Productivity, AI Pair Programmer, GitHub Copilot

1. Introduction

The continuous advancements in generative artificial intelligence (AI) are transforming content production across a wide range of domains. Cutting-edge generative AI tools can not only automate mundane tasks but also enhance original content. In the context of software development, generative AI-powered pair programmers like GitHub Copilot, Amazon Q Developer, Google Gemini, and Chat GPT can swiftly generate code based on developers' prompts, parameters, and descriptions. For example, GitHub Copilot, trained on an extensive dataset comprising billions of lines of publicly available code, demonstrates remarkable ability to offer predictive coding suggestions, which are comparable to human's contributions (Dakhel et al. 2023). By reducing common coding errors and repetitive coding needs, these AI pair programmers hold the potential to dramatically influence the software production process. Thus, it is important to assess whether and how AI pair programmers influence software development productivity. A growing body of literature has investigated various impacts of generative AI on well-defined and discrete tasks on individuals, such as writing and customer service tasks (Brynjolfsson et al. 2023, Noy and Zhang 2023). However, there is limited understanding of the role of generative AI in complex tasks that involve iterative development and team collaboration, such as software development. To address this gap, our research focuses on evaluating software productivity at the project level within the context of open-source software development.

As a popular form of software development, open-source software development involves a team of geographically dispersed developers who voluntarily collaborate to develop and refine code for a software project (Levine and Prietula 2014). Any developer can participate in the project by making individual contributions that are then integrated into the final software codebase. Some studies suggest that generative AI can improve the software development activities of individual developers (Dohmke et al. 2023, Peng et al. 2023). However, whether this translates into higher productivity at the project level is not obvious as there is a need for coordination among developers for software development within teams. This is particularly challenging in the context of open-source development due to its open and voluntary nature of developer participation.

Moreover, the open-source development typically involves two types of developers: core developers and peripheral developers (Setia et al. 2012). Core developers set high-level objectives, design the architecture, write code and maintain full control over the codebase, whereas peripheral developers enhance the existing codebase by fixing bugs and adding new features (AlMarzouq et al. 2005, Setia et al. 2012, Gousios et al. 2014, Medappa and Srivastava 2019), which have to be integrated by the core developers. Existing literature has highlighted the heterogenous impacts of generative AI on individuals with varying skill levels (e.g., Brynjolfsson et al. 2023) for well-defined and discrete tasks. However, these findings may not readily enhance our understanding on how generative AI affects core developers versus peripheral developers, as the distinction lies more in the nature of the tasks and roles played by these two types of developers rather than their skills.

Motivated by these observations, in this study we aim to answer the following research questions. *First, how do AI pair programmers affect open software development productivity at the project level? Second, how do AI pair programmers affect the relative contributions of core developers compared to peripheral developers?* These questions are crucial to study, as our findings may provide useful implications on how to leverage generative AI tools to support collaborative and distributed software development in the open-source community.

Building on the literature on generative AI and the literature on open-source software development, our key theoretical argument is that AI pair programmers may yield individual productivity gain and encourage developers to participate more in software development. However, AI pair programmers could also increase coordination and integration costs due to the higher volume of code contributions from increased participation and individual productivity. As a result, the overall impact of AI pair programmers on project-level productivity depends on the net effect of these competing forces. Furthermore, due to the differences in the tasks taken by core developers versus peripheral developers, greater complementarity can be achieved between AI pair programmers and core developers than between AI pair programmers and peripheral developers. Therefore, AI pair programmers could lead to relatively more contributions and participation by core developers compared to peripheral developers. However, this relative effect again depends on how core developers choose to participate in the integration activities.

To empirically investigate our research questions, we examine how the AI pair programmer GitHub Copilot influences the productivity of open-source projects on GitHub. GitHub is one of the largest code-hosting repositories based on the Git version control system (Dabbish et al. 2012, Borges et al. 2016). In this setup, a repository is a fundamental unit that typically contains the source code and resource files for a software project, along with information related to the project's evolution history, high-level features, and developer details (Zhang et al. 2017). Such repositories are often used to investigate collaborative development practices (Dabbish et al. 2012, Kalliamvakou et al. 2015, Celińska 2018). Our unit of analysis is at the repository-month-level, with the sample period from January 2021 to December 2022. To examine the impact of Copilot on code contributions to open-source repositories on GitHub, we use a combination of publicly available data on GitHub repositories and proprietary data on Copilot use provided by the GitHub Team. Our main empirical approach is based on the generalized synthetic control method (GSCM), which allows for staggered treatment turn-on times for repositories with a well-matched control sample. Our treatment group consists of repositories where Copilot was both supported by local coding environments and used by developers to code. Thus, the post-treatment period includes the months during which Copilot was supported and used in a focal repository, and the pre-treatment period includes all other months. The control group includes repositories where Copilot was not used throughout the sample period. We validate our results with alternative matching techniques and samples, and developer-level analysis.

Our empirical results show that the use of Copilot increased the number of successful code contributions in repositories by 6.5%, suggesting an overall positive impact on project-level software development productivity. Detailed analyses on the underlying mechanisms suggest that such an improvement on overall code contributions was accompanied with a significant increase in individual productivity and a significant increase in developer participation, both of which contribute to the increase in the project-level productivity. Interestingly, we also find that the use of Copilot increased the integration time within the development process by 41.6%, potentially due to higher coordination costs. These pieces of evidence highlight the trade-off between improved code contributions and increased coordination efforts resulting from the use of AI pair programmers.

Furthermore, we find that Copilot use led to relatively greater increases in participation, contributions, and productivity by core developers, as compared to peripheral developers. A plausible explanation is higher complementarity between an AI pair programmer and a core developer than between an AI pair programmer and a peripheral developer. Specifically, core developers' understanding of their software projects' overarching objectives and structure may enable them to more effectively leverage an AI pair programmer to complement their development capabilities. In contrast, peripheral developers, whose main roles involve enhancing existing codebase, may lack a comprehensive understanding of the intricacies of a project and thus experience less effective use of an AI pair programmer compared to core developers.

Our study offers several important contributions to the literature. First, it contributes to the literature on generative AI in software development (Imai 2022, Barke et al. 2023, Biswas 2023, Peng et al. 2023). Prior research has shown the positive impact of generative AI on individual productivity for specific coding tasks (Peng et al. 2023). In contrast, our study takes a first step to understand how generative AI tools affect project-level productivity. In the context of open-source software development, we demonstrate the role of generative AI on different aspects of a software project such as participation, individual productivity, and integration. Additionally, we emphasize the importance of knowledge of a software project, as exhibited by core developers, in harnessing the benefits of generative AI for collaborative software development.

Second, our study contributes to the literature on developer participation in open-source software development (Bagozzi and Dholakia 2006, Roberts et al. 2006). Prior research has focused on benefits of developer participation (Hann et al. 2002, Lakhani and Wolf 2003, Shah 2006, David and Shapiro 2008, Spaeth et al. 2015) but paid limited attention on the potential costs of participation. Our study complements this work by demonstrating how generative AI could encourage more participation, potentially due to AI's capabilities to reduce participation costs and barriers and increase developer ability to contribute. This increased participation addresses an important concern[1] that AI, with its automated code writing capabilities, may replace developers in open-source software contributions and consequently discourage developer participation. Additionally, our study shows differential effects of generative AI on core developers and peripheral developer participation.

Third, our study contributes to the literature on pair programming (Astels et al. 2002, Williams and Kessler 2003, Vidgen and Wang 2009). This body of literature primarily focused on human pair programming and showed that it increases code development costs but reduces integration costs (Dawande et al. 2008). In contrast, our study shows that AI pair programming has an opposite effect from human pair programming in the open-source context. Specifically, we show that AI pair programming leads to more integration costs, due to the higher volume of code contributions facilitated by AI in an open-source setup.

[1] For example, see https://www.nytimes.com/2023/06/02/opinion/ai-coding.html.

Our study provides several managerial implications. It underscores the importance of AI pair programmers in enhancing project-level productivity for software development teams. At the same time, it emphasizes the need for managing team coordination and streamlining the integration process to fully benefit from AI pair programmers. It also highlights the importance of familiarity with the project in leveraging the AI capabilities, as demonstrated by our comparison of core and peripheral developers. Additionally, our findings have implications for the structure of open-source software teams as higher participation by core developers may reduce the need for peripheral developers' involvement over time.

2. Literature review

Our study draws from and adds to three streams of literature: the impact of generative AI tools on content production, developer participation in open-source software development, and pair programming. Next, we discuss these streams in greater detail.

2.1 Impact of Generative AI Tools

Many researchers have started exploring the specific productivity gains achieved by generative AI. For instance, Noy and Zhang (2023) find that Chat GPT speeds up writing tasks, improves writing quality, and minimizes disparities in writing grades. Particularly, it helps lower-ability workers produce higher-quality work and enables higher-ability workers to maintain quality while increasing their speed (Noy and Zhang 2023). Similarly, Brynjolfsson et al. (2023) examine the impact of AI assistance on customer support and find that AI boosts productivity without compromising resolution rates or customer satisfaction. The largest productivity gains are observed among less skilled and newer workers, as AI helps them accelerate the learning curve, whereas highly skilled workers may find AI to be distracting.

Few researchers have considered the impact of generative AI on software development. Some studies have focused on individual software development productivity for specific tasks. Imai (2022) finds that GitHub Copilot helps to write more code as compared to human pair programmer for a task involved creation of minesweeper using Python. However, this also resulted in more code deletions, potentially indicating lower code quality. Peng et al. (2023) find that GitHub Copilot enables individual developers to implement an HTTP server 55.8% faster than those not using the tool. Hoffmann et al. (2024) show that

GitHub Copilot causes individual developers to shift focus towards tasks that do not involve collaborative effort with other developers. Despite these insights, research on how generative AI influences project-level outcomes for complex tasks involving multiple developers remains limited. Yeverechyahu et al. (2024) investigate the innovation capabilities of generative AI, particularly its role in extrapolative versus interpolative thinking, and compare its influence on innovation in Python versus R.

In contrast, our research examines the impact of generative AI on open-source software development in a multi-developer setup and its differential effects on two groups: core developers and peripheral developers. Unlike previous studies, we focus on open-source perspectives and contribute to the open-source software literature. More specifically, to our knowledge, we are the first to investigate generative AI's influence on developer participation in open-source projects and its impact on code integration costs, which are critical for assessing the overall project-level productivity of open-source software development.

2.2 Open-Source Software Participation

Open-source software development involves integrating voluntary code contributions from individual developers to form the final software codebase (Atkins et al. 2002, Kornilov and Safonov 2018). Previous studies have considered benefits for developer voluntary participation, such as opportunities to express their creativity and experience for a great sense of satisfaction and achievement (Lakhani and Wolf 2003), satisfy needs for software improvements (Shah 2006), acquire knowledge (David and Shapiro 2008), build reputation (Hann et al. 2002), and enhance social identification (Spaeth et al. 2015). These studies assume that developers can contribute to open-source projects and focus on the motivation and benefits of participation. With the advent of generative AI, which assists in code development, it is important to understand how this technology influences developer participation by potentially affecting the costs and barriers in open-source development. In addition, research has highlighted the distinct roles of developers and suggested both core developers and peripheral developers are important to the long-term success of the open-source software development (Setia et al. 2012). However, there is a gap in understanding how generative AI influences participation by different types of developers in the open-source community. Due to the differences in their tasks and roles in a software project between core developers and peripheral developers, there could be differences in the extent to which they can effectively leverage the features of generative AI to complement their tasks.

2.3 Pair Programming

Human pair programming involves two developers simultaneously working on the same piece of code (Beck 2000). Typically, one member of the pair writes the code, while the other observes the creation of the code, suggests improvements in structure, and points out strategic defects (Williams and Kessler 2003, Dawande et al. 2008). The objective is to keep code clean and simple, and ensure flexibility when confronted with changing requirements (Astels et al. 2002). It has been argued that although human pair programming increases the effort to develop a piece of code as compared to solo programming, this extra effort is often compensated by lower system integration and testing efforts (Dawande et al. 2008). This is because many integration decisions are made collaboratively during the code generation process.

However, the results obtained from human pair programming may not be applicable in the setting of open-source software development with AI pair programmers. On the one hand, by swiftly providing code suggestions, AI pair programmers can potentially lead to a significant reduction in code development costs. On the other hand, AI pair programmers may increase participation, which in turn may raise integration costs as developers need to take more time to determine valuable contributions among the increased inputs facilitated by AI pair programmers. Therefore, it is important to assess whether and how AI pair programming influences software development productivity.

3. Theoretical Background

In this section, we first describe the open-source software development process and the role of AI pair programmers, followed by the theoretical motivation for our analysis.

3.1 Open-Source Software Development

Open-source software development is characterized by a fully decentralized and open environment, where a group of voluntary contributors work together to develop and refine code, subsequently making it accessible to both fellow contributors and the broader community (Levine and Prietula 2014). Besides making code contributions to the software codebase, developers also need to spend a significant amount of time and efforts managing project-level dependencies and ensuring cohesive and aligned actions among team members. We refer to these efforts related to code integration, communication, planning, and resolving conflicts stemming from task interdependencies as integration costs (Howison and Crowston 2014, Lindberg et al. 2016, Medappa and Srivastava 2019, Shaikh and Vaast 2023). In the open-source context, productivity is often measured based on successful code contributions, which refer to the amount of code submitted by developers to propose modifications or new features that is eventually merged to the codebase (Smirnova et al. 2022). Therefore, project-level productivity is affected by participation in open-source development, individual developers' productivity in making code contributions, and integration costs.

The open-source software development process typically involves two major categories of developers: core and peripheral (Setia et al. 2012). These developers exhibit varying levels of understanding and control over the projects. Core developers, who often include the projects' administrators and key maintainers, are responsible for defining the overarching objectives and ensuring the delivery of the final code product. As shown in Figure 1, software development usually begins with core developers designing the primary codebase and hosting it on a platform for open collaboration (Singh and Phelps 2013). Core developers, who have the write access and full control over the projects, can submit code and either directly integrate it into the primary codebase or have it reviewed by other core developers for better integrity (Gousios et al. 2015).

Peripheral developers, on the other hand, typically contribute to a project by enhancing the existing codebase such as fixing bugs and adding new features (AlMarzouq et al. 2005, Setia et al. 2012). They primarily contribute through code submissions. These submissions are reviewed by core developers, who decide whether to merge the contribution into the primary codebase, request additional modifications, or reject them altogether (Gousios et al. 2014, Medappa and Srivastava 2019).

[insert Figure 1 here]

3.2 AI Pair Programmers in the Open-Source Software Development

AI pair programmers, guided by explicit prompts from developers, offer immediate code suggestions and corrections. These tools support software developers by fixing bugs, proposing enhancements, recommending best practices, and facilitating the transfer of coding knowledge across various domains.[2] AI pair programmers use extensive databases and advanced machine learning algorithms to provide suggestions that follow best practices and coding standards, reducing search time. This helps lower the time developers spend writing code and waiting for peer support, thereby accelerating the production process. Moreover, these tools also serve as exploration aids when developers are unsure how to proceed, helping them explore potential options (Barke et al. 2023). Recent research has documented the beneficial effects of AI pair programmers on individual software developers' productivity (Imai 2022). In particular, AI pair programmers allow developers to complete coding tasks faster than those who do not use such tools (Peng et al. 2023). The benefits of AI pair programmers in improving individual productivity in software development can also extend to the open-source community.

AI pair programmers can also increase developer participation by lowering the barriers and associated costs. According to the expectancy value models, contributors evaluate the expectancy of the outcomes and its value to determine if they would participate in certain software projects (Atkinson 1957, Hertel et al. 2003, Setia et al. 2012). Given their limited time, developers have to select projects that are both meaningful and rewarding, balancing the costs of participation against the potential benefits (Wen et al. 2013). Traditionally, developers needed specific knowledge to effectively contribute to open-source software projects, creating significant entry barriers. AI pair programmers can mitigate these barriers by shortening the learning curve, thereby enabling developers to contribute to projects across diverse knowledge domains. Thus, AI pair programmers can lower the participation costs by improving individual productivity and alleviate concerns about the time and effort required from developers, thereby encouraging greater participation in open-source software development.

[2] The GitHub website discusses the use cases of Copilot: https://github.com/features/copilot. https://github.blog/2022-09-14-8-things-you-didnt-know-you-could-do-with-github-copilot/.

However, productivity at the project level is not merely the sum of individual productivity across developers. Although open-source development typically relies on modular design and parallel work to create independent layers of work (Howison and Crowston 2014), a significant amount of development interdependency and developer interdependency still exists (Lindberg et al. 2016). Therefore, it is crucial to take the associated integration and coordination costs into consideration.

We have argued above that generative AI may facilitate more code contributions generated by individual developers; at the same time, such tools could also attract a larger group of contributors with diverse objectives and developmental approaches. Overall, we would expect a higher volume of code submissions after the use of generative AI by developers. However, these contributions can potentially compromise the compatibility of existing codebases, as managing code interdependency becomes challenging with an increased number of contributions (Shaikh and Vaast 2023). Generative AI also has limited ability to coordinate suggestions across parallel work and integrate code contributions at the project level. Therefore, human effort is still needed to integrate independent code contributions into the codebase and to ensure the cohesiveness and compatibility of the final software products. Given the larger volume of submissions enabled by generative AI, developers would need to invest more effort in reviewing and integrating these contributions into the main codebase. This represents a significant cost that must be considered when evaluating productivity at the project level. Therefore, the overall project-level productivity in software development should reflect the net effect of individual productivity and participation gains, balanced against integration and coordination costs. We will empirically evaluate the overall effect of generative AI on open-source software development productivity.

3.3 Developer Roles in the Open-Source Community

Within the open-source community, the collaborative dynamics of projects are greatly enriched by the diverse contributions from core developers and peripheral developers (Setia et al. 2012). Given the roles played by core developers versus peripheral developers, as well as the capabilities of AI pair programmers, we expect a stronger complementarity between AI pair programmers and core developers than that between AI pair programmers and peripheral developers, for reasons as follows.

One important shortcoming of AI pair programmers is their limited understanding on the overarching objectives of a software project compared with human developers. As a result, the code suggestions may not align well with a project's design principles, dependencies, and performance considerations (Vaithilingam et al. 2022, Adamson and Bägerfeldt 2023). Because core developers are the ones who design the overall architecture and determine the development trajectory of a project, they have a deep understanding on the projects' style, patterns, and previously implemented solutions. This would allow them to effectively provide context-rich prompts and adjust AI-generated code. Meanwhile, core developers may be better in anticipating long-term compatibility and maintenance issues from using certain code suggestions. Accordingly, they can choose AI-generated code that not only addresses some immediate problems but also align well with future development plans and scalability considerations. As a result, AI pair programmers could effectively improve core developers' individual productivity by complementing their roles within a software project. Moreover, according to expectancy value models, core developers may have greater incentives to participate because they have clearer expectations regarding software development outcomes compared to peripheral developers. Since AI pair programmers can effectively reduce their participation costs and boost their individual productivity, core developers may participate more to enjoy complementarity with AI pair programmers.

On the other hand, peripheral developers often contribute to projects on an irregular or part-time basis (Bagozzi and Dholakia 2006, Howison and Crowston 2014). Unlike core developers, peripheral developers may lack a comprehensive understanding of the project's intricacies. Therefore, they may not provide effective prompts to generate intended code and at the same time, the code could potentially run into compatibility issues so some of the contributions may not get accepted into the final codebase. In sum, while AI pair programmers could still improve peripheral developers' individual productivity and encourage them to participate in software projects, the increase in their individual productivity and participation could be lower than that of core developers.

4. Data

Our objective is to evaluate the impact of AI pair programmers on software development productivity in the open-source community. We are particularly interested in project-level outcomes which rely on participation from multiple developers. Our empirical analysis focuses on GitHub data at the level of repository, which serves as the basic unit for collaborative software development (Dabbish et al. 2012, Kalliamvakou et al. 2015, Celińska 2018). We investigate how the introduction of AI pair programmer Copilot impacts productivity for GitHub repositories and the plausible underlying mechanisms.

Copilot's introduction occurred in stages, commencing with limited availability in June 2021 and a formal public launch in June 2022[3]. During the one-year period between June 2021 and June 2022, Copilot underwent a technical preview phase. Interested developers needed to sign up for a waitlist, and those selected developers received an invitation email containing a download link for Copilot.[4] Based on this timeline, we collect panel data of GitHub repositories from the publicly available GitHub Archive Dataset, spanning a two-year timeframe, from January 2021 to December 2022[5].

To ensure the generalizability of our analysis, we follow established procedures in the literature (Kalliamvakou et al. 2014, AlMarzouq et al. 2020) to identify active repositories in the open-source community during our panel period. Specifically, we select repositories that meet the following criteria: they have a non-zero size, specify at least one programming language and one license, include repository descriptions, and are not mirrors or personal stores. To further ensure the repositories are not ghost or abandoned, we require at least one code submission from developers every six months from 2021 to 2022, and at least one other activity, such as a release, creation, or deletion, each year. As we are interested in evaluating project-level software development productivity involving multiple developers, we focus on repositories with at least three developers contributing each month. Additionally, as mentioned in greater

[3] GitHub launched the technical preview of Copilot in June 2021: https://github.blog/2021-06-29-introducing-github-copilot-ai-pair-programmer/. It then announced the formal launch and public availability of Copilot in June 2022: https://github.blog/2022-06-21-github-copilot-is-generally-available-to-all-developers/.
[4] GitHub waitlist page: https://github.com/features/preview/copilot-enterprise.
[5] We choose this endpoint to ensure our results are not influenced by the rise in popularity of Chat GPT, which began in early 2023.

detail below, we need IDE[6] information to identify repositories in the treatment group versus those in the control groups, so we further restrict our sample to those that disclose IDE information. These criteria result in a total of 9,244 repositories.

To identify the repositories where Copilot was used by developers (i.e., the treatment group), we collaborate with the GitHub Team, who provides us with proprietary aggregated Copilot usage data for each repository. This data indicates the proportion of developers who submitted code contributions to a focal repository and used Copilot during each month.[7] Moreover, to use Copilot, the focal repository must also use supported IDEs. During our analysis period, only a limited number of IDEs supported Copilot: Visual Studio Code, the JetBrains suite of IDEs, Neovim, and Visual Studio.[8] We check the webpages of the repositories in our sample to gather information on the IDE usage by developers involved in the development of these repositories. We categorize repositories with developers using Copilot and using any of these supported IDEs as our treatment group, and those using unsupported IDEs and not using Copilot as our control group. In total, our sample includes 5,687 repositories in the treatment group and 3,557 repositories in the control group. We designate the first month when Copilot was supported by IDEs and used by a non-zero proportion of developers as the treatment start time for each repository. The months before this time are defined as the pre-treatment period and the months after this time as the post-treatment period for each repository, covering the two-year span from 2021 to 2022.

5. Empirical Analysis

5.1 Measures

We use merged pull requests of each repository as our primary dependent variable to capture project-level software development productivity. Pull requests represent code changes submitted by developers and need further evaluation by core developers. This evaluation results in either approval, leading to merged pull

---

[6] IDE stands for integrated development environment, which is a software application that provides local environments for coding, testing, and debugging.

[7] We do not know the identity of individual Copilot users because of privacy concerns. Although such Copilot usage is measured at the GitHub platform level, developers are likely to integrate Copilot into their workflows extensively and thus would use it for all repositories where it is feasible (Serenko 2008, Marangunić and Granić 2015).

[8] GitHub lists the supported IDEs for Copilot: https://github.com/features/copilot.

requests, or rejection, leading to closed but unmerged pull requests. Thus, merged pull requests reflect successful code contributions that have been accepted and incorporated into the development of repositories (Gousios et al. 2014, Tsay et al. 2014, Kononenko et al. 2018) and are commonly used in the literature to assess productivity (Bertoncello et al. 2020). To account for the time it takes for core developers to evaluate submitted pull requests (Gousios et al. 2015), we calculate the number of pull requests submitted in each month that are eventually merged within six months[9].

As discussed earlier, Copilot may influence individual productivity and developer participation, which impact project-level software development productivity. Additionally, the integration costs might offset the positive effects of AI pair programmers. To provide a comprehensive understanding of Copilot's impact on the number of merged pull requests, we utilize three additional dependent variables. We assess individual productivity through the average number of submitted pull requests per developer (Subramaniam et al. 2009). We gauge participation by counting the number of developers who submitted pull requests (Krishnamurthy 2002, Subramaniam et al. 2009). Finally, we measure the integration costs by calculating the average time difference in minutes between pull requests submission and acceptance (Yu et al. 2016). The unit of analysis in our baseline models is repository-month and we further demonstrate the robustness of our results by conducting developer-level analyses where the unit of analysis is developer-month.

5.2 Model and Estimation

Our objective is to determine how Copilot influences project-level software development productivity, which is measured by the number of merged pull requests for a repository. In our setup, repositories adopt Copilot at different times after it is available. There could be systematic differences between repositories where Copilot was used and repositories where Copilot was not used. Moreover, because the repositories in the treatment group were treated at different points in time and we have more repositories in the treatment group than those in the control group, it is difficult to employ traditional matching techniques to construct

[9] Developers can take several months to review code contributions, request modifications, and integrate them into the primary codebase. We use a six-month window because most pull requests are either merged or rejected within six months.

a matched control sample. Therefore, we use the generalized synthetic control method (GSCM) (Xu 2017, Wang et al. 2021, Mader and Rüttenauer 2022, Wang et al. 2023). This approach combines the idea of a synthetic control (Abadie et al. 2010) with interactive fixed effects (Bai 2009). This allows us to address multiple treated units with staggered treatment times while accounting for time-varying unobservables. The GSCM shares core assumptions with the standard synthetic control method, effectively managing unobserved time-variant confounders by giving more weight to control units that mirror the pre-treatment trends of the treatment group.

We use the following linear factor model (Bai 2009, Xu 2017) to determine the effect of Copilot on the software development productivity:

$$Y_{it} = \delta_{it} D_{it} + X_{it}'\beta + \lambda_i' f_t + \varepsilon_{it} \tag{1}$$

where Y_{it} represents the outcome variable and is measured by the log number of merged pull requests of repository i in month t. We take the log transformation to reduce its skewness. D_{it} is the treatment indicator which equals one if repository i has been developed with Copilot by month t and zero otherwise. The parameter of primary interest is δ_{it}, which signifies the dynamic impact of Copilot on the log number of merged pull requests. δ_{it} is the heterogeneous treatment effect and its subscripts i and t indicate that the estimates vary across repositories and months. In addition, X_{it}' represents the observed time-varying control variable, the size of repository i in month t, since repository size may affect the contributions and participation from developers, and β is the corresponding estimate.

In the above equation, $f_t = [f_{1t}, \dots, f_{rt}]'$ is an $(r \times 1)$ vector of unobserved common factors associated with factor loadings $\lambda_i' = [\lambda_{i1}, \dots, \lambda_{ir}]$ and ε_{it} is the error term with a mean of zero. The factor component $\lambda_i' f_t$ can be expressed as $\lambda_i' f_t = \lambda_{i1} f_{1t} + \lambda_{i2} f_{2t} + \cdots + \lambda_{ir} f_{rt}$. The factor component nests a range of unobserved heterogeneities including additive unit and time fixed effects and unit-specific linear and quadratic time trends. Note that a two-way fixed effects specification is a special case of the factor component, where $r = 2$, $f_{1t} = 1$, and $\lambda_{i2} = 1$, so that $\lambda_i' f_t = \lambda_{i1} + f_{2t}$. Here, λ_{i1} represents the repository fixed effects, while f_{2t} is the month fixed effects. We specify the two-way fixed effects to account for heterogeneity across repositories and time, while considering other unobserved latent factors.

We estimate the optimal number of latent factors using a cross-validation procedure (Xu 2017) Briefly, this involves first estimating the parameters of model using the control group data only and employing a cross-validation procedure to determine the number of latent factors r. Next, the optimal number of factor loadings for each treated unit is estimated by minimizing the mean squared errors of the predicted treated outcomes in the pre-treatment periods. We provide a detailed explanation of the model and the estimation including the determination of the optimal latent factors in Online Appendix A. After accounting for time and repository fixed effects, the optimal number of unobserved factors determined by the cross-validation technique is zero. This suggests that the fixed effects setting has effectively accounted for any unobserved time-varying characteristics (Xu 2017).

The GSCM estimator predicts the counterfactuals for treated units in the post-treatment periods using the parameter estimates obtained in the previous two steps. The causal effect of the treatment is calculated as the average treatment effect on the treated (ATT), based on the differences between the observed outcome of a treated unit $Y_{it}(1)$ and its constructed counterfactual $\hat{Y}_{it}(0)$:

$$ATT_t = \frac{1}{|\mathcal{T}|}\sum_{i\in\mathcal{T}}\left[Y_{it}(1) - \hat{Y}_{it}(0)\right] = \frac{1}{|\mathcal{T}|}\sum_{i\in\mathcal{T}}\delta_{it} \quad (2)$$

where $\mathcal{T}$ denotes the set of treated units and $|\mathcal{T}|$ represents the number of units in $\mathcal{T}$.

To create well-matched synthetic controls, GSCM automatically dropped repositories with very different values in matching variables during the pre-treatment period, resulting in a final sample size of 8,965 repositories used for the GSCM estimation, with 5,435 repositories in the treatment group and 3,530 repositories in the control group. Table 1 provides descriptions and summary statistics for repository-month-level variables based on the sample used for the GSCM estimation.

[insert Table 1 here]

Identification: In our study, there may be dynamic unobservables which could influence the outcome. GSCM synthesizes a weighted control unit that closely mirrors the data pattern of the log number of merged pull requests during the pre-treatment period for the treated unit. The outcome variable of this synthetic control unit during the post-treatment period serves as the counterfactual prediction for the treated unit. By modeling the trend of the outcome variable, the GSCM can naturally accommodate the influence of unobservable confounders that evolve over time. Furthermore, it allows each treatment unit to have a different treatment period and can efficiently construct synthetic control units from a relatively small control sample.

We further apply the equivalence test to examine the presence of a pre-treatment trend in GSCM (Pan and Qiu 2022, Egami et al. 2023, Wang et al. 2023). This is a standard way to test the performance of GSCM (Liu et al. 2024), as the equivalence test better incorporates substantive considerations of what constitutes good balance on covariates and placebo outcomes compared to traditional tests (Hartman and Hidalgo 2018). Specifically, we use the two-one-sided t (TOST) test. The test includes an equivalence range within which differences are deemed inconsequential (Hartman and Hidalgo 2018, Lakens et al. 2018). The test is considered passed if the average prediction error for any pre-treatment period is within the equivalence range (Liu et al. 2024). The result of the equivalence test is shown in Figure B.1 of Online Appendix B. We observe that the average prediction error with 90% confidence intervals (the gray dotted line) is within the equivalence range (the red dotted line). Thus, we can reject the null of inequivalence (equivalence test p-value is 0) and conclude that there exists no pre-treatment trend. This indicates that a sufficient set of confounders has been controlled to address the endogeneity concerns and that GSCM provides a good control group. We also validate our results using alternative samples, matching technique and a developer-level analysis as described in the robustness section.

6. Results

6.1 Main Results

The estimation result on the overall impact of Copilot on repository-level software development productivity is shown in column 1 of Table 2. The result shows that following the integration of Copilot into the development process, there was a significant 6.5 % increase in the number of merged pull requests in a repository. This suggests that Copilot led to increase in productivity for the repositories in our sample.

As discussed earlier, the increase in the number of merged pull requests can be attributed to several key factors. First, Copilot might enhance individual developer productivity by providing useful code suggestions and auto-completions, thereby streamlining the coding processes. This efficiency gain would accelerate pull request creation and submissions, significantly enhancing project-level productivity. Second, Copilot may reduce the participation barriers and costs for developers, enabling them to contribute more to repositories. We explore these potential drivers in detailed mechanism analysis in Section 6.3.

6.2 Robustness Checks

To further investigate the robustness of our findings, we employ a range of empirical methods to examine the relationship between Copilot use and the number of merged pull requests at project-level. A summary of these checks is provided in Table C.1 of Online Appendix C.

Within-Treatment Analysis: In our baseline identification, we use IDEs that supported Copilot during our sample period as part of the criteria to identify repositories in the treatment group and consider repositories developed with IDEs that did not support Copilot for the control group. To address concerns that there could be some systematic and unobservable differences between repositories with IDEs that supported Copilot and repositories with IDEs that did not, we validate our results by focusing on the repositories in the treatment group, i.e., those with IDEs all supporting Copilot.

Specifically, we use repositories that adopted Copilot in the first half of the period from July 2021 to December 2022 when Copilot was available, i.e., July 2021 - March 2022, as the treatment group. Accordingly, we conduct this analysis using only the period from January 2021 to March 2022, so repositories that adopted Copilot after March 2022 can be considered as the control group. We choose March 2022 as the endpoint so that there are enough repositories with Copilot usage during this time window (i.e., a total of nine months from July 2021 to March 2022). Another advantage of this approach is that repositories in the treatment group and the control group would exhibit a similar likelihood of being treated (i.e., being affected by Copilot), with only the difference on the timing of the treatment. This analysis includes 3,749 repositories in the treatment group and 1,686 repositories in the control group, a total of 5,435 repositories. The treatment turn-on time for each repository in the treatment group is decided in the same manner as in our baseline analysis above. The result in column 2 of Table 2 shows that Copilot increased merged pull requests, consistent with our main finding.

Refined Sample: As developers work on multiple repositories, it is possible that some may be working on both treatment and control repositories. Approximately 6% of the selected repositories have developers involved in both groups. Although these developers might not be using Copilot for the control repositories due to IDE restrictions, there is potential for knowledge transfer, which could bias our results. To eliminate this bias, we refine our sample to exclusively assess the impact of Copilot on repositories where no developer is involved in both groups. Corresponding result is shown in column 3 of Table 2, indicating that Copilot increased the number of merged pull requests, which is qualitatively similar to our main result.

PSM and DID: We further validate our result using an alternative matching and estimation approach. Specifically, we use the difference-in-differences (DID) estimation combined with the propensity score matching (PSM). In this analysis, we use June 2021, the first availability date of Copilot, as the single treatment turn-on time for all repositories classified into the treatment group in the baseline analysis. Hence, this approach may alleviate concerns about non-random treatment turn-on time for Copilot usage in the treatment group in the baseline analysis. However, we acknowledge that because developers may sign up gradually after the initial availability date, this approach may consider the period when Copilot was actually not used as the post-treatment period, leading to an underestimation of the positive effect.

We calculate each repository's propensity score, defined as the probability of utilizing Copilot, using a logit regression model. We estimate the model using the following characteristics of repositories in the pre-treatment period (January to June 2021): the number of merged pull requests each month, the average number of submitted pull requests per developer each month, the number of developers who submitted pull requests each month, programming language, and license. Subsequently, we employ the nearest-neighbor matching algorithm with replacement and set a caliper of 0.05 to match each treated repository with a control unit. Following this matching process between repositories where Copilot was used and those that Copilot was not used, we arrive at a dataset comprising 7,322 repositories, consisting of 5,650 treatments and 1,672 controls. We analyze the impact of Copilot on the number of merged pull requests using the following regression specification:

$$Y_{it} = \beta_0 + \beta_1 Copilot_i \times Post_t + x_{it} + \alpha_i + \mu_t + \varepsilon_{it} \quad (3)$$

where the dependent variable Y_{it} represents the log number of merged pull requests of repository i in month t and we again take the log transformation to reduce its skewness; $Copilot_i$ is a binary indicator that is denoted with "1" for repository i where Copilot was used and "0" otherwise; $Post_t$ is a binary indicator that is set to "1" in months after June 2021 and "0" otherwise. x_{it} is the control variable, the size of repository i in month t. To control for time-invariant heterogeneity across repositories and time trends, we include repository-level fixed-effects α_i and month-level fixed-effects μ_t.

The main coefficient of interest is β_1, as it indicates the change in the number of merged pull requests of repositories before and after the availability of Copilot, relative to the changes of repositories where Copilot was not used at all throughout the sample period. Additionally, we confirm that the parallel trend assumption holds for the DID model (details of the parallel pre-trend test conducted using the relative time model are provided in Online Appendix D). We report the result in column 4 of Table 2. The result shows a 7.1% increase in the number of merged pull requests following the availability of Copilot, which is qualitatively consistent with our finding from the main analysis.

[insert Table 2 here]

6.3 Mechanism Analysis

To further elucidate the reasons behind this increase in project-level productivity measured by the number of merged pull requests, we delve into three key aspects: individual productivity, developer participation, and integration costs.

Individual Productivity: The increase in project-level productivity is likely driven by individual productivity improvements, particularly as AI pair programmers primarily support software development activities of a developer. To determine whether Copilot influences individual productivity, we replicated our main analysis using the average number of submitted pull requests per developer as the dependent variable. The corresponding results are presented in column 1 of Table 3. We find that following the use of Copilot, there was a 5.5% increase in the average number of pull requests submitted per developer. Such evidence is consistent with our hypothesis that generative AI such as Copilot may lead to a notable enhancement in individual developer productivity, and this can partly explain the increase in repository-level or project-level productivity.

Developer Participation: If more developers contribute to repositories, it is likely to increase the amount of code generated, thereby potentially improving project-level productivity. To test whether Copilot influences developer participation, we used our main model to analyze its impact on the number of developers who submitted pull requests. The results, shown in column 2 of Table 3, indicate that this effect is significant. Specifically, we find a 5.4% increase in the number of developers submitting pull requests in repositories using Copilot compared to those in repositories not using Copilot. This confirms that Copilot can increase overall developer participation.

Integration Costs: The results so far suggest that Copilot can increase both individual productivity and developer participation. However, the increased productivity and participation place additional demands on core developers to review a larger volume of code contributions, leading to longer integration times. To evaluate the outcome, we analyze the effect of Copilot on the average time difference in minutes between pull request submission and acceptance. Column 3 of Table 3 shows that Copilot had a positive and significant effect on integration time, increasing it by 41.6%. This indicates that the rise in participation and individual productivity is accompanied by higher coordination and integration costs, resulting in a lower overall increase in project-level productivity.

We report the robustness checks for the above analyses across alternative samples and matching approaches in Online Appendix E.

[insert Table 3 here]

6.4 Differential Effects Among Core and Peripheral Developers

We also evaluate the differential effects of Copilot on core developers and peripheral developers within the open-source community. Developers are classified as core or peripheral based on whether they have the write access to a repository. To assess the differential effects of Copilot on these two groups, we evaluate how Copilot influences the relative productivity of core developers compared to all developers.[10] Since all developers fall into either the core or peripheral groups, these metrics can indicate the effect on core developers relative to peripheral developers.

First, we calculate the proportion of merged pull requests by core developers to the total number of merged pull requests for each repository in a month. Using this variable as the outcome variable in Equations (1) and (2), a positive estimate of the average treatment effect of Copilot would indicate that after using Copilot, the proportion of code generated by core developers among all code merged in a repository becomes higher, i.e., core developers achieve greater project-level productivity gains compared to peripheral developers. To shed light on how Copilot leads to greater increase in productivity by core developers than peripheral developers, we create two additional metrics. The first metric is the ratio of average number of submitted pull requests per core developer to the overall average number of submitted pull requests per all types of developer. Using this metric as the outcome variable in Equations (1) and (2) can help determine the relative effect of Copilot on core developers' individual productivity compared to peripheral developers. The second metric is the proportion of core developers to the total number of developers in a repository. A positive estimate of the average treatment effect of Copilot on this variable would indicate that Copilot may encourage more core developers to participate than peripheral developers.

We report the results in Table 4. The results show that the proportion of merged pull requests by core developers increased significantly by 0.033 or 8.3%[11]. Core developers also showed a significant increase in their average number of submitted pull requests relative to the overall average individual submission in a repository, an increase of 0.038 or 5.2%[12]. Additionally, the proportion of core developers

[10] This allows us to incorporate zero values for some peripheral developer activities.
[11] The mean proportion of merged pull requests by core developers to the total number of merged pull requests, as reported in Table 1, is 0.4. This proportion increased by 0.033, representing an 8.3% increase.
[12] The mean ratio of the average number of pull requests submitted per core developer to the average number submitted per all developers, as reported in Table 1, is 0.726. This ratio increases by 0.038, representing an 5.2% increase.

in development teams showed a significant increase of 0.032 or 9.4%[13]. These results indicate that core developers achieve higher project-level productivity gains compared to peripheral developers because they benefit more from Copilot in terms of individual productivity and participation, leading to a potential shift in the team composition of software projects.

[insert Table 4 here]

***Alternative Explanations*:** It is possible that observed differential effects in Table 4 are either due to lower increases or reductions in activities of peripheral developers. To investigate this further, we define six measures to explore the impact of Copilot on the absolute changes in project-level productivity, individual productivity, and participation by core developers and peripheral developers, respectively. Specifically, for a given repository in a month, we calculate the number of merged pull requests by core developers (and peripheral developers), average number of submitted pull requests per core developer (and per peripheral developer), the number of distinct core developers (and peripheral developers) who submitted pull requests.

We report the results in Table 5. The results indicate that both core and peripheral developers benefit from using Copilot. Specifically, core developers improved their project-level productivity gains by 4.83%, individual productivity by 4.26%, and participation by 4.28%, while peripheral developers improved their project-level productivity gains by 1.31%, individual productivity by 1.54% and participation by 2.29%. Note that as GSCM generates different synthetic controls for these different measures, we cannot directly compare the magnitudes of core developers with that of peripheral developers in Table 5. However, these findings corroborate the results in Table 4 and suggest that the relative more proportion of code contributions and participation by core developers after the use of Copilot is not driven by a reduction in code contribution and participation by peripheral developers.

[insert Table 5 here]

[13] The mean proportion of core developers to the total number of developers, as reported in Table 1, is 0.342. This proportion increased by 0.032, representing a 9.4% increase.

It is also possible that the higher number of merged pull requests for core developers, compared to peripheral developers, is due to their prioritizing the review and integration of their own code contributions over those from peripheral developers. To rule out this alternative explanation, we calculate the ratio of the average integration time for core developers' code contributions to the overall average integration time and evaluate how this relative integration time is influenced by Copilot. Additionally, we explore the impact of Copilot on the absolute changes in integration time for core and peripheral developers, respectively. Specifically, we use the average integration time for code contributions from core developers and the average integration time for code contributions from peripheral developers. The results, based on Equations (1) and (2) with these three variables as the dependent variables, are reported in Table F.1 of Online Appendix F. The table shows that both groups experienced an increase in the average integration time, with a 20.6% increase for core developers and a 19.3% increase for peripheral developers. However, there is no effect on the ratio of average integration time for core developers' code contributions relative to the overall average integration time. This finding seems not supporting the possibility that core developers derive enhanced benefits because they have the write access and prioritize their own code over peripheral contributions. Instead, it indicates that their familiarity with the software projects is likely the primary reason for their higher levels of contribution in the presence of Copilot.

6.5 Developer-Level Analysis

In the baseline repository-level analyses, we focus on a set of active repositories based on several criteria, such as having IDE information and a team size of three or more developers. To demonstrate the observed effect is not limited to the set of repositories used in the baseline analyses, we also conduct developer-level analysis to further validate our results. More specifically, we explore the effect on individual developers' activities across all repositories on the GitHub platform.

Due to privacy concerns, we do not have data on the status of Copilot adoption by individual developers. Instead, we infer it from the aggregated repository-level Copilot use data provided by the GitHub Team. To identify a representative sample of developers who are interested in team-based open-source software collaboration, we start with repositories for which the team size is equal or greater than three. Then, among these repositories, based on the Copilot usage information, we identify developers in the treatment group versus developers in the control group. We then collect data on their individual contribution and participation behavior across all open-source repositories on GitHub.

More specifically, while we know the proportion of developers associated with a repository who used Copilot in a particular month, we do not know the identities of these Copilot users due to privacy concerns. Thus, to conservatively estimate the effect, we identify 206 repositories with over 80% of developers using Copilot. Then we consider developers associated with these repositories as the treatment group.[14] This results in 915 developers in the treatment group, as most repositories have Copilot adoption rates lower than 80%. Similarly, we identify repositories with inferred zero percent of developers using Copilot and consider developers associated with those repositories as the control group. This approach leads to 79,257 developers in the control group. We extract data on all GitHub repositories to which these 80,172 developers submitted code contributions, a total of 1,130,901 repositories.

In addition to inferring the set of developers in the treatment group, we also need to infer the Copilot adoption time for each of these developers, as such information is not provided to us either due to privacy concerns. We infer the adoption time based on the timing of Copilot adoption at the level of repositories that each developer contributed to. This allows us to calculate a range that indicates the possible adoption times for each developer in the treatment group. For example, suppose a developer contributed to three repositories, which show the proportion of Copilot users became greater than zero in July 2021, March 2022, and July 2022. Then, we assume this developer could adopt Copilot at any of these time points. We use July 2021 as the earliest point in this developer's adoption time range, assuming the developer belongs to the group of developers who adopted Copilot in July 2021. Conversely, July 2022 is identified as the latest point in this developer's adoption time range, assuming this developer belongs to the group of developers who adopted Copilot in July 2022.

[14] The reason we do not use developers associated with repositories with inferred 100% of developers using Copilot as the treatment group is that this would lead to too few observations. We acknowledge that our approach could include some developers who did not adopt Copilot in our treatment group. However, this would lead to an underestimation of the positive effects of Copilot on developers' individual productivity and participation.

In our baseline analyses, we use the latest point in this developer's adoption time range as the turn-on time for Copilot use. The period from this time point to December 2022 is considered as the post-treatment period. To ensure a clean pre-treatment period when developers were not using Copilot, we designate January to June 2021, the months prior to Copilot's availability, as the pre-treatment period. Consequently, we exclude the months between July 2021 and the latest possible adoption time point from our analysis. For robustness checks, we also use the earliest possible point in this developer's adoption time range as the turn-on time for Copilot use, and we obtain qualitatively similar results.

We measure each developer's overall productivity gains by the number of merged pull requests across all repositories to which a focal developer contributed. To examine the underlying mechanisms through which Copilot may influence a developer's overall productivity, similar to the previous repository-level analyses, we consider three factors: the average number of pull requests per repository submitted by a focal developer, the number of distinct repositories to which a focal developer submitted pull requests, and the average time difference in minutes between the submission and acceptance of pull requests. These metrics measure developers' average productivity per project, participation in different software projects, and the average integration time associated with their contributions, respectively.

Similar to our baseline repository-level analyses, we employ GSCM for developer-level analyses. As noted earlier, to create well-matched synthetic controls, GSCM automatically dropped developers with very different values in matching variables during the pre-treatment period. Thus, we have a final sample size of 63,470 developers used for the GSCM estimation, with 418 developers in the treatment group and 63,052 developers in the control group. Table G.1 in Online Appendix G provides descriptions and summary statistics for developer-month-level variables based on the GSCM estimation sample.

The results based on the GSCM approach are reported in Table 6 and are consistent with our repository-level analysis. It shows that a developer increased the total number of merged pull requests across repositories by 13.2% after using Copilot. Moreover, following the use of Copilot, there was an 11.1% increase in the average number of submitted pull requests per repository and a 9.1% increase in the number of repositories participated by these developers. However, the average integration time also had a 65.6%

increase. Overall, this evidence supports our argument that the use of Copilot enables individual developers to achieve overall productivity gains by improving their productivity within each project and increasing their participation in different software projects, despite the tradeoff of slower feedback from software projects' core teams.

[insert Table 6 here]

Additionally, developers may hold varying roles across different projects, serving as core developers in some and peripheral developers in others. To compare difference in productivity gains between projects where developers act as core developers and those where they act as peripheral developers, we further examine the differential effect of Copilot based on developer's roles. Similar to the repository-level analyses, we create three metrics by calculating the level of contributions to projects where developers are core developers relative to all projects they contribute to, to avoid issues of no activity in projects where they are peripheral developers. Specifically, we use the following metrics: the proportion of merged pull requests in repositories where they are core developers to the total number of their merged pull requests in all repositories; the ratio of the average number of submitted pull requests per repository where they are core developers to their overall average of submitted pull requests per repository; and the proportion of repositories where they are core developers to their total number of repositories.

The results based on the GSCM estimation are presented in Table 7. As shown in column 1 in Table 7, following the use of Copilot, developers increased the proportion of pull requests submitted as core developers that were eventually merged by 0.045 or 20.7%[15]. As shown in column 2, if we focus on the average pull requests submitted, regardless of whether they were eventually merged, still, developers increased the ratio of the average pull requests submitted per repository as core developers relative to overall average submitted pull requests per repository by 0.079 or 28.3%[16]. Furthermore, as shown in column 3,

[15]The mean proportion of merged pull requests in repositories where developers are core developers to the total number of merged pull requests, as reported in Table G.1 in Online Appendix G, is 0.217. This proportion increased by 0.045, representing a 20.7% increase.

[16] The mean ratio of the average number of submitted pull requests per repository where developers are core developers to the average number of submitted pull requests per all repositories, as reported in Table G.1 in Online Appendix G, is 0.279. This ratio increased by 0.079, representing a 28.3% increase.

the proportion of repositories where they were core developers expanded by 0.04, or 18.9%[17]. Overall, these developer-level results align with our repository-level findings. That is, when using Copilot, developers achieved greater overall productivity gains in projects where they were core developers compared to those where they were peripheral developers.

[insert Table 7 here]

As noted earlier, to provide a conservative measure of Copilot's impact, we also use the earliest possible point in the focal developer's adoption time range as the treatment turn-on time. Using such an approach would lead to inclusion of observations without actual Copilot adoption as post-treatment observations, leading to an underestimation of the positive effect of Copilot on developer's productivity and participation. Nevertheless, we obtain qualitatively similar results, which are presented in Online Appendix H.

6.6 Impact on Quality

In addition, we check the impact of Copilot on code quality, as code quality is the bedrock for the effective utilization of software in various activities. While the influence of Copilot on merged pull requests can indirectly reflect its impact on code quality, given that merged pull requests typically denote high-quality code contributions, we opt to directly measure Copilot's influence on code quality by using the metric of the number of new issues standardized by repository size (Krishnan et al. 2000). We report the results in Table 8. We observe that Copilot did not significantly impact the number of new issues when standardized by repository size. Concerns that increased productivity might compromise code quality is not supported, as these results suggest that Copilot improved project-level productivity without concurrently increasing the incidence of new issues.

[insert Table 8 here]

7. Discussion and Conclusions

[17] The mean proportion of repositories where developers are core developers to their total number of repositories, as reported in Table G.1 in Online Appendix G, is 0.211. This proportion increased by 0.04, representing a 18.9% increase.

The rise of AI pair programmers has garnered increasing attention due to their potential to transform software development. In this study, we explore the effects of AI pair programmers within the context of open-source software, with a focus on its impact on project-level software development productivity. Using repository-level data from GitHub, we evaluate the impact of GitHub Copilot, an AI pair programmer, on the development of open-source repositories on GitHub and its underlying drivers. Our results highlight the positive influence of Copilot on the number of merged pull requests, signaling an improvement in project-level software development productivity. Detailed analyses on the underlying mechanisms reveal that the improved project-level productivity is accompanied by increased participation and improved individual productivity among developers. However, we also observe an increase in integration time due to Copilot, indicating a trade-off in the benefits of using Copilot.

Moreover, we examine Copilot's differential effects on core developers and peripheral developers within the open-source community. We show that the proportion of code contributions from core developers became higher after the use of Copilot, indicating core developers derive greater benefits from Copilot than peripheral developers. We further investigate the underlying mechanisms and find that this increase in the proportion of code contributions from core developers could be driven by both a higher increase in individual productivity and greater participation by core developers compared to peripheral developers. Overall, this evidence is consistent with our argument that the complementarity between AI and developer capabilities could be higher for core developers than for peripheral developers, as the former are more familiar with the overall design and code associated with the software projects.

Additionally, we broaden our investigation to assess Copilot's impact on code quality, addressing concerns that increased productivity might lead to more bugs and errors. Our findings, however, do not seem to support the concerns that Copilot may enhance software development productivity but at the same time compromise code quality.

7.1 Implications

Our paper has several implications. First, the findings of our paper suggest that organizations should invest in AI pair programmers, as the tools enhance overall software development productivity at the project level by increasing individual productivity and participation. It is also advisable for companies to develop strategies to manage team coordination effectively and streamline integration processes to mitigate the increased costs associated with AI pair programmers. Effective management of these integration costs is essential to ensure that the benefits of increased individual productivity and participation are not negated by the complexity of merging diverse inputs. Thus, a synergistic balance between individual productivity, developer participation, and manageable integration costs is crucial for optimizing project-level productivity.

Secondly, it is evident that core developers in the open-source community, due to their familiarity with software projects, derive greater benefits from using AI pair programmers. This underscores the importance of integrating human intelligence with the automation and augmentation capabilities of AI pair programmers. However, this also raises concerns about potential long-term challenges for open-source contributions. As AI pair programmers encourage more active participation and contributions from core developers than peripheral developers, these AI tools could alter the team dynamics within open-source software development. Over time, these tools might widen the gap of contributions between core developer and peripheral developers, potentially changing the team composition and skewing the community balance. Given the open-source ethos of broadening participation and incorporating diverse perspectives, the prevalent use of AI pair programmers could fundamentally change the culture and collaborative model within the community.

7.2 Limitations and Future Work

Our study has several limitations that warrant further consideration. Due to developer privacy concerns, we infer developer-level Copilot usage for our analysis. Future work should confirm the results with more precise AI usage data at the developer level. Another concern is the identification challenge given the observational nature of the data used in the study. However, it is impractical to randomly provide generative AI tools to developers in different repositories on a large scale. To address this concern, we implement a wide range of analyses both at the repository level and at the developer level to demonstrate the robustness of our results. Our study is among the first to provide large-scale evidence on how generative AI shapes collaborative open-source software development at the project level.

Our work represents the first step in examining how different types of developers (core versus peripheral) may be affected differently by generative AI. However, more nuances may exist among various developers regarding how AI-generated code suggestions are received and utilized. Future studies could conduct detailed analyses to identify the distinct needs and preferences of different developer groups and tailor AI suggestions accordingly. Moreover, the current study emphasizes the quantitative benefits of increased productivity. Future research could explore the subjective experiences of developers interacting with AI pair programmers to discern the qualitative changes in their work. Such qualitative assessments could enrich the understanding of the human-centric impacts of AI pair programmers in software development settings. Last, the technological design and potential biases of AI pair programmers has not been thoroughly addressed. Future research should critically explore the implications of algorithmic bias and propose frameworks for ethical AI usage in software development.

References

Abadie A, Diamond A, Hainmueller J (2010) Synthetic control methods for comparative case studies: Estimating the effect of California's tobacco control program. *Journal of the American statistical Association* 105(490):493-505.

Adamson V, Bägerfeldt J (2023) Assessing the effectiveness of ChatGPT in generating Python code.

AlMarzouq M, AlZaidan A, AlDallal J (2020) Mining GitHub for research and education: challenges and opportunities. *International Journal of Web Information Systems* 16(4):451-473.

AlMarzouq M, Zheng L, Rong G, Grover V (2005) Open source: Concepts, benefits, and challenges. *Communications of the Association for Information Systems* 16(1):37.

Astels D, Miller G, Novak M (2002) *The practical guide to extreme programming* (Prentice Hall PTR).

Atkins DL, Ball T, Graves TL, Mockus A (2002) Using version control data to evaluate the impact of software tools: A case study of the version editor. *IEEE Transactions on Software Engineering* 28(7):625-637.

Atkinson JW (1957) Motivational determinants of risk-taking behavior. *Psychological review* 64(6p1):359.

Bagozzi RP, Dholakia UM (2006) Open source software user communities: A study of participation in Linux user groups. *Management science* 52(7):1099-1115.

Bai J (2009) Panel data models with interactive fixed effects. *Econometrica* 77(4):1229-1279.

Barke S, James MB, Polikarpova N (2023) Grounded copilot: How programmers interact with code-generating models. *Proceedings of the ACM on Programming Languages* 7(OOPSLA1):85-111.

Beck K (2000) *Extreme programming explained: embrace change* (addison-wesley professional).

Bertoncello MV, Pinto G, Wiese IS, Steinmacher I (2020) Pull requests or commits? which method should we use to study contributors' behavior? *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)* (IEEE), 592-601.

Biswas S (2023) Role of ChatGPT in Computer Programming.: ChatGPT in Computer Programming. *Mesopotamian Journal of Computer Science* 2023:8-16.

Borges H, Hora A, Valente MT (2016) Predicting the popularity of github repositories. *Proceedings of the The 12th international conference on predictive models and data analytics in software engineering*, 1-10.

Brynjolfsson E, Li D, Raymond LR (2023) Generative AI at work. Report.

Celińska D (2018) Coding together in a social network: collaboration among github users. *Proceedings of the 9th international conference on social media and society*, 31-40.

Dabbish L, Stuart C, Tsay J, Herbsleb J (2012) Social coding in GitHub: transparency and collaboration in an open software repository. *Proceedings of the ACM 2012 conference on computer supported cooperative work*, 1277-1286.

Dakhel AM, Majdinasab V, Nikanjam A, Khomh F, Desmarais MC, Jiang ZMJ (2023) Github copilot ai pair programmer: Asset or liability? *Journal of Systems and Software* 203:111734.

David PA, Shapiro JS (2008) Community-based production of open-source software: What do we know about the developers who participate? *Information Economics and Policy* 20(4):364-398.

Dawande M, Johar M, Kumar S, Mookerjee VS (2008) A comparison of pair versus solo programming under different objectives: An analytical approach. *Information Systems Research* 19(1):71-92.

Dohmke T, Iansiti M, Richards G (2023) Sea Change in Software Development: Economic and Productivity Analysis of the AI-Powered Developer Lifecycle. *arXiv preprint arXiv:2306.15033*.

Egami N, Yamauchi S (2023) Using multiple pretreatment periods to improve difference-in-differences and staggered adoption designs. *Political Analysis* 31(2):195-212.

Gousios G, Pinzger M, Deursen Av (2014) An exploratory study of the pull-based software development model. *Proceedings of the 36th international conference on software engineering*, 345-355.

Gousios G, Zaidman A, Storey M-A, Van Deursen A (2015) Work practices and challenges in pull-based development: The integrator's perspective. *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering* (IEEE), 358-368.

Hann I-H, Roberts J, Slaughter S, Fielding R (2002) Delayed returns to open source participation: An empirical analysis of the Apache HTTP Server Project. *Located at: http://www. idei. asso. fr/Commun/Conferences/Internet/OSS2002/Papiers/Hann. PDF* (Citeseer).

Hartman E, Hidalgo FD (2018) An equivalence approach to balance and placebo tests. *American Journal of Political Science* 62(4):1000-1013.

Hertel G, Niedner S, Herrmann S (2003) Motivation of software developers in Open Source projects: an Internet-based survey of contributors to the Linux kernel. *Research policy* 32(7):1159-1177.

Hoffmann M, Boysel S, Nagle F, Peng S, Xu K (2024) Generative AI and Distributed Work: Evidence from Open Source Software. Working Paper.

Howison J, Crowston K (2014) Collaboration through open superposition: A theory of the open source way. *Mis Quarterly* 38(1):29-50.

Imai S (2022) Is github copilot a substitute for human pair-programming? an empirical study. *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, 319-321.

Kalliamvakou E, Damian D, Blincoe K, Singer L, German DM (2015) Open source-style collaborative development practices in commercial projects using GitHub. *2015 IEEE/ACM 37th IEEE international conference on software engineering* (IEEE), 574-585.

Kalliamvakou E, Gousios G, Blincoe K, Singer L, German DM, Damian D (2014) The promises and perils of mining github. *Proceedings of the 11th working conference on mining software repositories*, 92-101.

Kononenko O, Rose T, Baysal O, Godfrey M, Theisen D, De Water B (2018) Studying pull request merges: A case study of shopify's active merchant. *Proceedings of the 40th international conference on software engineering: software engineering in practice*, 124-133.

Kornilov AS, Safonov IV (2018) An overview of watershed algorithm implementations in open source libraries. *Journal of Imaging* 4(10):123.

Krishnamurthy S (2002) Cave or community?: An empirical examination of 100 mature open source projects. *First Monday*.
Krishnan MS, Kriebel CH, Kekre S, Mukhopadhyay T (2000) An empirical analysis of productivity and quality in software products. *Management science* 46(6):745-759.
Lakens D, Scheel AM, Isager PM (2018) Equivalence testing for psychological research: A tutorial. *Advances in methods and practices in psychological science* 1(2):259-269.
Lakhani KR, Wolf RG (2003) Why hackers do what they do: Understanding motivation and effort in free/open source software projects. *Open Source Software Projects (September 2003)*.
Levine SS, Prietula MJ (2014) Open collaboration for innovation: Principles and performance. *Organization Science* 25(5):1414-1433.
Lindberg A, Berente N, Gaskin J, Lyytinen K (2016) Coordinating interdependencies in online communities: A study of an open source software project. *Information Systems Research* 27(4):751-772.
Liu L, Wang Y, Xu Y (2024) A practical guide to counterfactual estimators for causal inference with time-series cross-sectional data. *American Journal of Political Science* 68(1):160-176.
Mader S, Rüttenauer T (2022) The effects of non-pharmaceutical interventions on COVID-19 mortality: a generalized synthetic control approach across 169 countries. *Frontiers in Public Health* 10:820642.
Marangunić N, Granić A (2015) Technology acceptance model: a literature review from 1986 to 2013. *Universal access in the information society* 14:81-95.
Medappa PK, Srivastava SC (2019) Does superposition influence the success of FLOSS projects? An examination of open-source software development by organizations and individuals. *Information Systems Research* 30(3):764-786.
Noy S, Zhang W (2023) Experimental evidence on the productivity effects of generative artificial intelligence. *Available at SSRN 4375283*.
Pan Y, Qiu L (2022) How ride-sharing is shaping public transit system: A counterfactual estimator approach. *Production and Operations Management* 31(3):906-927.
Peng S, Kalliamvakou E, Cihon P, Demirer M (2023) The impact of ai on developer productivity: Evidence from github copilot. *arXiv preprint arXiv:2302.06590*.
Roberts JA, Hann I-H, Slaughter SA (2006) Understanding the motivations, participation, and performance of open source software developers: A longitudinal study of the Apache projects. *Management science* 52(7):984-999.
Serenko A (2008) A model of user adoption of interface agents for email notification. *Interacting with Computers* 20(4-5):461-472.
Setia P, Rajagopalan B, Sambamurthy V, Calantone R (2012) How peripheral developers contribute to open-source software development. *Information Systems Research* 23(1):144-163.
Shah SK (2006) Motivation, governance, and the viability of hybrid forms in open source software development. *Management science* 52(7):1000-1014.
Shaikh M, Vaast E (2023) Algorithmic interactions in open source work. *Information Systems Research* 34(2):744-765.
Singh PV, Phelps C (2013) Networks, social influence, and the choice among competing innovations: Insights from open source software licenses. *Information Systems Research* 24(3):539-560.
Smirnova I, Reitzig M, Alexy O (2022) What makes the right OSS contributor tick? Treatments to motivate high-skilled developers. *Research Policy* 51(1):104368.
Spaeth S, von Krogh G, He F (2015) Research note—perceived firm attributes and intrinsic motivation in sponsored open source software projects. *Information Systems Research* 26(1):224-237.
Subramaniam C, Sen R, Nelson ML (2009) Determinants of open source software project success: A longitudinal study. *Decision Support Systems* 46(2):576-585.
Tsay J, Dabbish L, Herbsleb J (2014) Influence of social and technical factors for evaluating contribution in GitHub. *Proceedings of the 36th international conference on Software engineering*, 356-366.

Vaithilingam P, Zhang T, Glassman EL (2022) Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. *Chi conference on human factors in computing systems extended abstracts*, 1-7.

Vidgen R, Wang X (2009) Coevolving systems and the organization of agile software development. *Information Systems Research* 20(3):355-376.

Wang Q, Qiu L, Xu W (2023) Informal Payments and Doctor Engagement in an Online Health Community: An Empirical Investigation Using Generalized Synthetic Control. *Information Systems Research*.

Wang Y, Ramachandran V, Liu Sheng OR (2021) Do fit opinions matter? The impact of fit context on online product returns. *Information Systems Research* 32(1):268-289.

Wen W, Forman C, Graham S (2013) Research Note—The impact of intellectual property rights enforcement on open source software project success. *Information Systems Research* 24(4): 1131-1146.

Williams L, Kessler RR (2003) *Pair programming illuminated* (Addison-Wesley Professional).

Xu Y (2017) Generalized synthetic control method: Causal inference with interactive fixed effects models. *Political Analysis* 25(1):57-76.

Yeverechyahu D, Mayya R, Oestreicher-Singer G (2024) The Impact of Large Language Models on Open-Source Innovation: Evidence from GitHub Copilot. Working Paper.

Yu Y, Yin G, Wang T, Yang C, Wang H (2016) Determinants of pull-based development in the context of continuous integration. *Science China Information Sciences* 59:1-14.

Zhang Y, Lo D, Kochhar PS, Xia X, Li Q, Sun J (2017) Detecting similar repositories on GitHub. *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)* (IEEE), 13-23.

Figures and Tables:

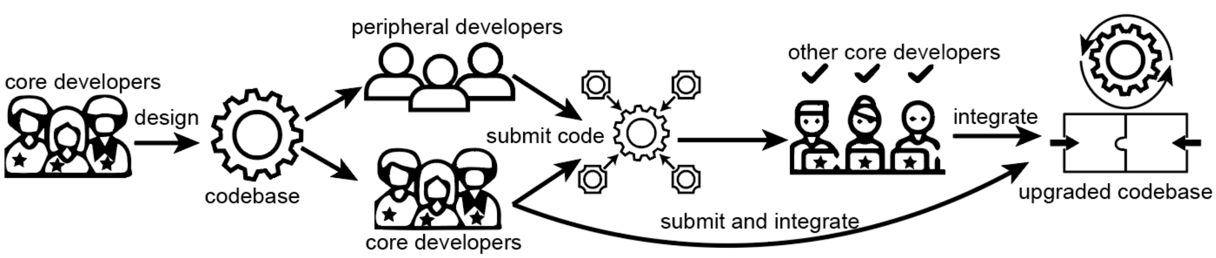

Figure 1: Open-Source Software Development Process

Table 1: Definitions and Summary Statistics, Repository-Month-Level

Variables	**Description**	**Obs**	**Mean**	**Std.**	**Min**	**Max**
Merged_PR	The number of accepted code contributions.	215160	15.83	58.68	0	3930
Avg_PR	The average number of submitted code contributions per developer.	215160	2.577	7.893	0	1020
Dev_count	The number of distinct developers who submitted code contributions.	215160	5.217	14.83	0	991
Avg_merge_time	The average time difference (in minutes) between code submission and acceptance.	215160	9028	22565	0	260111
Core_to_all_mergedPR	The proportion of the accepted code contributions by core developers to the total number of accepted code contributions.	215160	0.400	0.434	0	1
Core_to_avg_PR	The ratio of the average number of code submissions per core developer to the overall average number of code submissions per all types of developer.	215160	0.726	0.966	0	56.21
Core_to_total_dev	The proportion of core developers to the total number of developers.	215160	0.342	0.395	0	1
Core_to_avg_merge_time	The ratio of core developers' average time difference (in minutes) between code submission and acceptance to the overall average time difference (in minutes) between code submission and acceptance.	215160	0.681	0.632	0	31.37
Core_mergedPR	The number of accepted code contributions by core developers.	215160	10.499	38.82	0	2023
Peri_mergedPR	The number of accepted code contributions by peripheral developer.	215160	5.329	31.28	0	2558
Core_avg_PR	The average number of submitted code contributions per core developer.	215160	2.92	9.719	0	1020
Peri_avg_PR	The average number of submitted code contributions per peripheral developer.	215160	1.214	2.507	0	275
Core_dev	The number of core developers who submitted code contributions.	215160	1.945	4.972	0	183
Peri_dev	The number of peripheral developers who submitted code contributions.	215160	3.272	12.47	0	988
Core_merge_time	The average time difference (in minutes) between core developers' code submission and acceptance.	215160	4295	13491	0	253788
Peri_merge_time	The average time difference (in minutes) between peripheral developers' code submission and acceptance.	215160	6489	18667	0	260090
Repo_size	The size of repository in megabytes of code.	215160	111710	532788	5	26103950
New_issues_by_size	The ratio of new issues to repository size (size is normalized by 10,000).	215160	6.943	69.90	0	10747.7

Table 2: The Impact of Copilot on Repository-Level Software Development Productivity

	(1) GSCM Main Analysis Log (Merged_PR)	(2) GSCM Within-Treatment Analysis Log (Merged_PR)	(3) GSCM Refined Sample Log (Merged_PR)	(4) PSM and DID Full Sample Log (Merged_PR)
ATT of Copilot	0.065***	0.019*	0.062***	0.071***
	(0.011)	(0.010)	(0.012)	(0.016)
Repository FE	Yes	Yes	Yes	Yes
Month FE	Yes	Yes	Yes	Yes
Control	Yes	Yes	Yes	Yes
# of repositories	8,965	5,435	8,449	7,322
Observations	215,160	81,525	202,776	175,728

Robust standard errors clustered at repository level in parentheses. *** p<0.01, ** p<0.05, * p<0.1.

Table 3: Mechanism Analysis

	(1) Individual Productivity Log (Avg_PR)	(2) Developer Participation Log (Dev_count)	(3) Integration Time Log (Avg_merge_time)
ATT of Copilot	0.055***	0.054***	0.416***
	(0.008)	(0.006)	(0.042)
Repository FE	Yes	Yes	Yes
Month FE	Yes	Yes	Yes
Control	Yes	Yes	Yes
# of repositories	8,965	8,965	8,965
Observations	215,160	215,160	215,160

All estimations are based on the GSCM method with the baseline repository sample. Robust standard errors clustered at repository level in parentheses. *** p<0.01, ** p<0.05, * p<0.1.

Table 4: Differential Effect of Copilot on Core versus Peripheral Developers

	(1) Project-Level Productivity Core_to_all_mergedPR	(2) Individual Productivity Core_to_avg_PR	(3) Developer Participation Core_to_total_dev
ATT of Copilot	0.033***	0.038***	0.032***
	(0.004)	(0.009)	(0.004)
Repository FE	Yes	Yes	Yes
Month FE	Yes	Yes	Yes
Control	Yes	Yes	Yes
# of repositories	8,965	8,965	8,965
Observations	215,160	215,160	215,160

All estimations are based on the GSCM method with the baseline repository sample. Robust standard errors clustered at repository level in parentheses. *** p<0.01, ** p<0.05, * p<0.1.

Table 5: The Effect of Copilot on Core versus Peripheral Developers (Absolute Changes)

	Core Developers			Peripheral Developers		
	(1) Project-level Productivity Log (Core_mergedPR)	(2) Individual Productivity Log (Core_avg_PR)	(3) Developer Participation Log (Core_dev)	(4) Project-level Productivity Log (Peri_mergedPR)	(5) Individual Productivity Log (Peri_avg_PR)	(6) Developer Participation Log (Peri_dev)
ATT of Copilot	0.0483***	0.0426***	0.0428***	0.0131***	0.0154***	0.0229***
	(0.006)	(0.006)	(0.004)	(0.004)	(0.005)	(0.004)
Repository FE	Yes	Yes	Yes	Yes	Yes	Yes
Month FE	Yes	Yes	Yes	Yes	Yes	Yes
Control	Yes	Yes	Yes	Yes	Yes	Yes
# of repositories	8,965	8,965	8,965	8,965	8,965	8,965
Observations	215,160	215,160	215,160	215,160	215,160	215,160

All estimations are based on the GSCM method with the baseline repository sample. Robust standard errors clustered at repository level in parentheses. *** p<0.01, ** p<0.05, * p<0.1.

Table 6: The Impact of Copilot on Software Development Productivity, Developer-Level Analysis

	(1) Developer Overall Productivity Gains Log (Merged_PR)	(2) Individual Project Productivity Log (Avg_PR)	(3) Project Participation Log (Repo_count)	(4) Integration Time Log (Avg_merge_time)
ATT of Copilot	0.132***	0.111***	0.091***	0.656***
	(0.024)	(0.019)	(0.017)	(0.104)
Developer FE	Yes	Yes	Yes	Yes
Month FE	Yes	Yes	Yes	Yes
# of developers	63,470	63,470	63,470	63,470
Observations	1,518,467	1,518,467	1,518,467	1,518,467

All estimations are based on the GSCM method with the developer sample. Robust standard errors clustered at developer level in parentheses. *** p<0.01, ** p<0.05, * p<0.1.

Table 7: Differential Effect of Copilot on Core versus Peripheral Activities, Developer-Level Analysis

	(1) Developer Overall Productivity Gains Core_to_all_mergedPR	(2) Individual Project Productivity Core_to_avg_PR	(3) Project Participation Core_to_total_repo
ATT of Copilot	0.045***	0.079***	0.040***
	(0.012)	(0.017)	(0.012)
Developer FE	Yes	Yes	Yes
Month FE	Yes	Yes	Yes
# of developers	63,470	63,470	63,470
Observations	1,518,467	1,518,467	1,518,467

All estimations are based on the GSCM method with the developer sample. Robust standard errors clustered at developer level in parentheses. *** p<0.01, ** p<0.05, * p<0.1.

Table 8: The Impact of Copilot on Code Quality

	(1) GSCM Main Analysis New_issues_by_size	(2) GSCM Within-Treatment Analysis New_issues_by_size	(3) GSCM Refined Sample New_issues_by_size	(4) PSM and DID Full Sample New_issues_by_size
ATT of Copilot	0.494	-0.011	0.509	2.508
	(0.718)	(0.363)	(0.528)	(2.093)
Repository FE	Yes	Yes	Yes	Yes
Month FE	Yes	Yes	Yes	Yes
# of repositories	8,965	5,435	8,449	7,322
Observations	215,160	81,525	202,776	175,728

Robust standard errors clustered at repository level in parentheses. *** p<0.01, ** p<0.05, * p<0.1.

ONLINE APPENDIX

Appendix A: Details of the Main Estimation

Our main causal inference uses the generalized synthetic control method (Xu 2017). Below, we provide a brief overview of its estimation process and relate it to our study.

Model: The GSCM combines the Interactive Fixed Effects (IFE) model (Bai 2009), which considers latent factors with effects that can vary across time and units, with the Synthetic Control (SC) method (Abadie et al. 2015), which creates synthetic control units to act as counterfactuals for the treated units. This combination enables GSCM to (1) relax the assumption of pre-treatment parallel trends, and (2) consider potential unobserved factors that vary over time at the unit level. Furthermore, GSCM offers several advantages over other estimation methods. Unlike the SC method, which is limited to a single treated unit, GSCM is designed for multiple treated units, eliminating the need to construct synthetic control units for each treatment unit one by one. Additionally, unlike the typical difference-in-differences model combined with a specific matching technique, which is most effective for a single treatment turn-on time and a large control group, GSCM allows each treatment unit to have a different treatment period and can efficiently construct synthetic control units from a relatively small control sample. These advantages make GSCM particularly well-suited to our data, as projects adopted Copilot at different points in time and we have relatively fewer projects in the control sample than in the treatment sample.

First, the GSCM adopts a linear IFE framework to model the latent factors. In our study, the outcome Y_{it}, the log number of merged pull requests of repository i in month t, can be expressed as:

$$Y_{it} = \delta_{it} D_{it} + X_{it}'\beta + \lambda_i' f_t + \varepsilon_{it} \qquad (1)$$

where D_{it} is the treatment indicator which equals one if repository i has been developed with Copilot by month t and zero otherwise; the parameter of primary interest is δ_{it}, which signifies the dynamic impact of Copilot on the log number of merged pull requests. δ_{it} is the heterogeneous treatment effect and its subscripts i and t indicate that the estimates vary across units and time. In addition, X_{it}' represents the observed control variable, size of repository i in month t, since repository size may affect the contributions and participation from developers, and β is the corresponding estimate. Let r be the number of latent factors, then f_t represents the $(r \times 1)$ vector of unobserved common factors, and λ_i' is the $(1 \times r)$ vector of unknown factor loadings. The factor component $\lambda_i' f_t$ can be expressed as $\lambda_i' f_t = \lambda_{i1} f_{1t} + \lambda_{i2} f_{2t} + \cdots + \lambda_{ir} f_{rt}$. A two-way fixed effects specification is a special case of the factor component, where $r = 2$, $f_{1t} = 1$, and $\lambda_{i2} = 1$, so that $\lambda_i' f_t = \lambda_{i1} + f_{2t}$. Here, λ_{i1} represents the repository fixed effects, while f_{2t} is the month fixed effects. We specify the two-way fixed effects to control for heterogeneity across repositories and time, while considering other unobserved latent factors. In general, $\lambda_i' f_t$ is estimated by an iterative factor analysis of the residuals from the model. If optimal number of unobserved latent factors determined by the cross-validation technique is zero, it means that the fixed effects setting has effectively accounted for any unobserved time-varying characteristics (Xu 2017). Lastly, ε_{it} is the error term with a mean of zero.

Estimation of Latent Factors: An essential task when utilizing the IFE framework is to identify the number of latent factors, denoted as r. Xu (2017) suggests a predictive analytics method for this purpose. For clarity, we divide the total number of repositories into two groups: Tr for treated repositories and Co for control repositories. The latent-factor selection algorithm first applies Equation (1) to data from control units only, covering both pre-treatment and post-treatment time periods, using the following equation:

$$Y_{it} = X_{it}'\beta + \lambda_i' f_t + \varepsilon_{it}, \forall i \in Co \qquad (2)$$

Since the control repositories never received treatment during the data-collection period, $\delta_{it} D_{it}$ is excluded from Equation (2). The value of r can vary within a range of candidate values specified by the researchers. For each specified r, the algorithm runs Equation (2) and derives the estimates $\hat{\beta}$ and $\hat{f_t}$. Subsequently, a cross-validation procedure is carried out for all treated repositories. Specifically, let s be the index of a pre-treatment period, ranging from one (the first pre-treatment period) to t^0 (the last pre-treatment period). Equation (3) is then applied to each pre-treatment period, starting with $s = 1$, for all repositories in the treatment group by leaving one period out and using the remaining periods to estimate the factor loadings.

$$\hat{\lambda}_{i,-s} = (\hat{f}_{-s}^{0\prime} \hat{f}_{-s}^{0})^{-1} \hat{f}_{-s}^{0\prime}(Y_{i,-s}^{0} - X_{i,-s}^{0\prime}\hat{\beta}),\ i \in Tr,\ s = 1, \dots, t^0 \qquad (3)$$

In Equation (3), $\hat{f}_{-s}^{0}$ and $\hat{\beta}$ are the estimates obtained from Equation (2). The superscript 0 indicates periods before the introduction of the treatment, and the subscript $-s$ denotes all periods except for s. Based on these estimates, the algorithm predicts the outcome for treated unit i in period s using $\hat{Y}_{is}(0) = X_{is}'\hat{\beta} + \lambda_{i,-s}'\hat{f}_s$ and records the out-of-sample error $e_{is} = Y_{is}(0) - \hat{Y}_{is}(0)$ for all $i \in Tr$. In the final step, the algorithm calculates the Mean Squared Error (MSE) of the prediction, summed over all pre-treatment periods s, for each candidate value of r. The value of r that minimizes this prediction error is selected, and the corresponding set of latent factors will be used in the causal inference process.

Estimation of Average Treatment Effects: The GSCM algorithm uses the estimated function to predict the counterfactuals for treated units, denoted as $\hat{Y}_{it}(0)$, representing the outcome of treated repositories had they not received treatment in the post-treatment period. The causal effect of the treatment is quantified as the Average Treatment effect on the Treated unit (ATT), calculated mathematically as:

$$ATT_t = \frac{1}{|\mathcal{T}|}\sum_{i\in\mathcal{T}}\left[Y_{it}(1) - \hat{Y}_{it}(0)\right] \text{ for } t > t^0$$

where $\mathcal{T}$ denotes the set of treated units and $|\mathcal{T}|$ represents the number of units in $\mathcal{T}$. $Y_{it}(1)$ is the observed outcome for treated repository i at time t. The essence of GSCM lies in the fact that if the predicted outcomes for the treated repositories during the pre-treatment periods are accurate, the algorithm can produce a valid counterfactual for each treated unit during the post-treatment periods. Finally, the GSCM uses bootstrapping to estimate confidence intervals and standard errors (Xu 2017).

Unlike typical econometric modeling, the latent-factor selection algorithm in GSCM prioritizes models with the lowest out-of-sample prediction error to accurately predict counterfactuals, rather than those with the best fit based on information criterion. As a result, models with more latent factors may not be selected, even if they account for more confounding factors. Moreover, the GSCM without any latent factor still yields a more valid estimate than a difference-in-differences (DID) model, except under two conditions: (1) the treatment is randomly assigned, satisfying the parallel trend assumption; and (2) no treatment heterogeneity exists. If either of these conditions is not met, the DID estimate is likely to be invalid.

Appendix B: Equivalence Test of GSCM

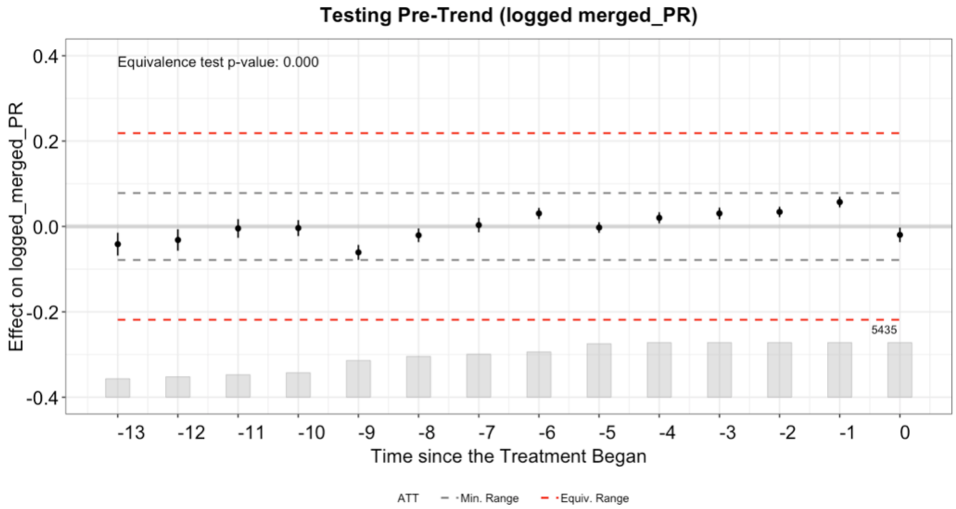

Figure B.1: Pre-trend test (TOST) of logged merged_PR in GSCM

Appendix C: Summary of Empirical Methods

Table C.1: Overview of Empirical Methods

Empirical methods	Description	Key takeaways
Generalized Synthetic Control Method in the Main Analysis	We use the GSCM to construct a weighted control unit that matches the outcome variable of the treated unit during the pre-treatment period.	Because the GSCM models the trend of the outcome variable, our results are robust to the effects of unobservable confounders that change over time.
Generalized Synthetic Control Method in a Within-Treatment Analysis	We use the repositories that adopted Copilot later as the control for repositories that adopted Copilot before March 2022. The treatment time for each repository is defined by its adoption time.	Our results remain robust in this within-treatment analysis, which addresses concerns about the unobserved differences between repositories that used supported IDEs and repositories that used unsupported IDEs.
Generalized Synthetic Control Method in a Refined Sample	We filter the main sample to exclude repositories associated with developers who participate in both treatment and control repositories.	Our results are robust with this refined sample and address the concern that the knowledge transfer of developers might bias the results.
Propensity Score Matching and Difference-in-Differences	We use the PSM to construct a comparable control group with a single treatment turn-on time and analyze the effect using the DID estimation.	Our results are consistent after addressing the concerns about non-random treatment turn-on time for Copilot usage in the baseline analysis.
Equivalence Test	We conduct equivalence test to evaluate the performance of GSCM and eliminate the concern of pre-trend issue.	Our results pass the test, indicating that a sufficient set of confounders has been controlled to address the endogeneity concerns and show that GSCM provides a good control group.
Relative Time Model	We use the RTM to verify whether the parallel trend assumption holds in the analysis of PSM and DID.	Our results pass the parallel pre-trend test, indicating that we have constructed a comparable control group in the analysis of PSM and DID.

Appendix D: Relative Time Model (RTM)

We employ the RTM to examine whether there is any pre-treatment trend in the analysis of PSM and DID, i.e., whether the outcome variable of repositories in the treatment group is different from that in the control group even before the treatment (i.e., the availability of Copilot) happens. The RTM has been used in the economics and information systems literature to validate the pre-treatment parallel trend assumption (Lu et al. 2019, Alyakoob and Rahman 2022). More specifically, we use the following RTM (Autor 2003).

$$Y_{it} = \beta_0 + \beta_1 Copilot_i \times Post_t + \sum_{\eta=1}^{q} \beta_{2,\eta} \left(Copilot_i \times Post_{t-\eta}\right) + x_{it} + \alpha_i + \mu_t + \varepsilon_{it}$$

In this model, the sum on the right-hand side represents q lead indicators (anticipatory effects), where Y_{it} denotes the log number of merged pull requests for repository i in month t. For our analysis of PSM and DID, we set $q = 2$ to compare all post-treatment periods with the two months before treatment. When the pre-treatment trend exists, the lead indicators would be able to predict changes in the dependent variable, the log number of merged pull requests. Conversely, if the pre-treatment trend is unlikely to be a concern, the lead indicators would not predict the dependent variable (Autor 2003, Angrist and Pischke 2009). The estimation results of the RTM are shown in Table D.1. We observe that the coefficients for the two lead indicators are statistically insignificant, indicating that there may not exist a pre-treatment trend.

Table D.1: Pre-trend Test for PSM and DID Analysis

	(1) PSM and DID Log (Merged_PR)
$Copilot_i \times Pre2m$	0.003
	(0.020)
$Copilot_i \times Pre1m$	0.022
	(0.022)
$Copilot_i \times Post_t$	0.075***
	(0.018)
Repository FE	Yes
Month FE	Yes
Control	Yes
# of repositories	7,322
Observations	175,728

Robust standard errors clustered at repository level in parentheses. *** p<0.01, ** p<0.05, * p<0.1.

Appendix E: Robustness Checks for Mechanism Analysis

In order to understand how Copilot leads to an increase in project-level productivity measured by the number of merged pull requests, in our mechanism analysis we examine how Copilot influences the following three metrics, all of which may influence project-level productivity: individual productivity, developer participation, and integration costs. Besides using the GSCM method with the main sample to conduct the mechanism analysis, we also conduct the same set of robustness checks as those in section 6.2 to investigate whether the results on the underlying mechanisms still hold. The results are shown in Tables E.1, E.2, and E.3 below using individual productivity, developer participation, and integration time as the dependent variable, respectively. They are all qualitatively similar to our main results.

Table E.1: Robustness Checks for Individual Productivity

	(1) GSCM Within-Treatment Analysis Log (Avg_PR)	(2) GSCM Refined Sample Log (Avg_PR)	(3) PSM and DID Full Sample Log (Avg_PR)
ATT of Copilot	0.018**	0.054***	0.041***
	(0.008)	(0.007)	(0.011)
Repository FE	Yes	Yes	Yes
Month FE	Yes	Yes	Yes
Control	Yes	Yes	Yes
# of repositories	5,435	8,449	7,322
Observations	81,525	202,776	175,728

Robust standard errors clustered at repository level in parentheses. *** p<0.01, ** p<0.05, * p<0.1.

Table E.2: Robustness Checks for Developer Participation

	(1) GSCM Within-Treatment Analysis Log (Dev_count)	(2) GSCM Refined Sample Log (Dev_count)	(3) PSM and DID Full Sample Log (Dev_count)
ATT of Copilot	0.023***	0.056***	0.052***
	(0.007)	(0.008)	(0.009)
Repository FE	Yes	Yes	Yes
Month FE	Yes	Yes	Yes
Control	Yes	Yes	Yes
# of repositories	5,435	8,449	7,322
Observations	81,525	202,776	175,728

Robust standard errors clustered at repository level in parentheses. *** p<0.01, ** p<0.05, * p<0.1.

Table E.3: Robustness Checks for Integration Time

	(1) GSCM Within-Treatment Analysis Log (Avg_merge_time)	(2) GSCM Refined Sample Log (Avg_merge_time)	(3) PSM and DID Full Sample Log (Avg_merge_time)
ATT of Copilot	0.156***	0.345***	0.382***
	(0.043)	(0.048)	(0.063)
Repository FE	Yes	Yes	Yes
Month FE	Yes	Yes	Yes
Control	Yes	Yes	Yes
# of repositories	5,435	8,449	7,322
Observations	81,525	202,776	175,728

Robust standard errors clustered at repository level in parentheses. *** p<0.01, ** p<0.05, * p<0.1.

Appendix F: Differential Effect on Integration Time

Table F.1: Differential Effect on Integration Time Among Core and Peripheral Developers

	(1) Relative Integration Time Core_to_avg_merge_time	(2) Core Developers' Integration Time Log (Core_merge_time)	(3) Peripheral Developers' Integration Time Log (Peri_merge_time)
ATT of Copilot	0.008	0.206***	0.193***
	(0.006)	(0.028)	(0.029)
Repository FE	Yes	Yes	Yes
Month FE	Yes	Yes	Yes
Control	Yes	Yes	Yes
# of repositories	7,947	7,947	7,947
Observations	190,728	190,728	190,728

All estimations are based on the GSCM method with the baseline repository sample. Robust standard errors clustered at repository level in parentheses. *** p<0.01, ** p<0.05, * p<0.1. Note: We exclude repositories with unmerged pull requests only as they lack merge times. Our analysis focuses on repositories with merged pull requests to examine the effect on their integration time.

Appendix G: Definitions and Summary Statistics of Developer-Month-Level Variables

Table G.1: Definitions and Summary Statistics, Developer-Month-Level

Variables	Description	Obs	Mean	Std.	Min	Max
Merged_PR	The number of accepted code contributions.	1518467	2.475	39.02	0	40806
Avg_PR	The average number of submitted code contributions per repository.	1518467	1.092	7.686	0	8173.6
Repo_count	The number of repositories to which developers submitted code contributions.	1518467	1.047	27.27	0	25603
Avg_merge_time	The average time difference (in minutes) between code submission and acceptance.	1518467	3686	14496	0	262199
Core_to_all_mergedPR	The proportion of accepted code contributions in repositories where developers are core developers to the total number of accepted code contributions across all repositories.	1518467	0.217	0.396	0	1
Core_to_avg_PR	The ratio of the average number of code submissions per repository by developers when acting as core developers to their overall average number of code submissions per repository.	1518467	0.279	0.506	0	21.64
Core_to_total_repo	The proportion of repositories where developers are core developers to the total number of repositories to which developers submitted code contributions.	1518467	0.211	0.382	0	1

Appendix H: Developer-Level Analysis (Earliest Possible Time of Copilot Adoption)

In this Appendix, we use the earliest possible point in the focal developer's adoption time range as the treatment turn-on time. Therefore, the period from this earliest possible time point to December 2022 is considered as the post-treatment period. Similar to our baseline analyses, we use January to June 2021, the months prior to Copilot's availability, as the pre-treatment period. That is, we exclude months between July 2021 and the earliest possible time point from our analysis.

We employ the same GSCM approach as well as the same outcome variables as in our baseline developer-level analyses. Note that since GSCM generates different synthetic controls for the earliest time point and the latest time point, we cannot directly compare the magnitudes from this set of results with those in Table 6 and Table 7.

The results are shown in Tables H.1 and H.2. Overall, it shows that developers increased their total number of merged pull requests across repositories by 15.5%. This increase is comprised of an 11.5% increase in the average number of submitted pull requests per repository and a 10.8% increase in the number of repositories to which developers submitted pull requests, while also being associated with a 74.4% increase in integration time.

Additionally, we note that developers increased the proportion of merged pull requests in repositories where they are core developers by 0.052 or 24%[1]. Each developer increased the ratio of average number of submitted pull requests in repositories where they were core developers relative to the overall average submitted pull requests per repository by 0.085 or 30.5%[2]. Moreover, the proportion of repositories where they were core developers grew by 0.043 or 20.4%[3]. The results are qualitatively similar to the developer-level results in our main paper and further validate our repository-level findings.

[1] The mean proportion of merged pull requests in repositories where developers are core developers to the total number of merged pull requests, as reported in Table G.1 in Online Appendix G, is 0.217. This proportion increased by 0.052, representing a 24% increase.

[2] The mean ratio of the average number of submitted pull requests per repository where developers are core developers to the average number of submitted pull requests per all repositories, as reported in Table G.1 in Online Appendix G, is 0.279. This ratio increased by 0.085, representing a 30.5% increase.

[3] The mean proportion of repositories where developers are core developers to their total number of repositories, as reported in Table G.1 in Online Appendix G, is 0.211. This proportion increased by 0.043, representing a 20.4% increase.

Table H.1: The Impact of Copilot on Software Development Productivity, Developer-Level Analyses (Earliest Possible Time of Copilot Adoption)

	(1) Developer Overall Productivity Gains Log (Merged_PR)	(2) Individual Project Productivity Log (Avg_PR)	(3) Project Participation Log (Repo_count)	(4) Integration Time Log (Avg_merge_time)
ATT of Copilot	0.155***	0.115***	0.108***	0.744***
	(0.020)	(0.015)	(0.014)	(0.093)
Developer FE	Yes	Yes	Yes	Yes
Month FE	Yes	Yes	Yes	Yes
# of developers	63,470	63,470	63,470	63,470
Observations	1,520,064	1,520,064	1,520,064	1,520,064

All estimations are based on the GSCM method with the developer sample. Robust standard errors clustered at developer level in parentheses. *** p<0.01, ** p<0.05, * p<0.1

Table H.2: Differential Effect of Copilot on Core versus Peripheral Activities, Developer-Level Analyses (Earliest Possible Time of Copilot Adoption)

	(1) Developer Overall Productivity Gains Core_to_all_mergedPR	(2) Individual Project Productivity Core_to_avg_PR	(3) Project Participation Core_to_total_repo
ATT of Copilot	0.052***	0.085***	0.043***
	(0.009)	(0.018)	(0.012)
Developer FE	Yes	Yes	Yes
Month FE	Yes	Yes	Yes
# of developers	63,470	63,470	63,470
Observations	1,520,064	1,520,064	1,520,064

All estimations are based on the GSCM method with the developer sample. Robust standard errors clustered at developer level in parentheses. *** p<0.01, ** p<0.05, * p<0.1

Notes: The earliest possible time analysis has more observations than the latest possible time analysis in the main paper because the identified adoption time for each developer could be earlier. We exclude the months between July 2021 and the developers' adoption time to minimize noise. To explore the effect, we use the months before July 2021, the initial availability of Copilot, as the pre-treatment period and months after the identified adoption time of each developer as the post-treatment period. Thus, the earliest possible time analysis has longer post-treatment periods, resulting in more observations.

References

Abadie A, Diamond A, Hainmueller J (2015) Comparative politics and the synthetic control method. *American Journal of Political Science* 59(2):495-510.

Alyakoob M, Rahman MS (2022) Shared prosperity (or lack thereof) in the sharing economy. *Information Systems Research* 33(2):638-658.

Angrist JD, Pischke J-S (2009) *Mostly harmless econometrics: An empiricist's companion* (Princeton university press).

Autor DH (2003) Outsourcing at will: The contribution of unjust dismissal doctrine to the growth of employment outsourcing. *Journal of labor economics* 21(1):1-42.

Bai J (2009) Panel data models with interactive fixed effects. *Econometrica* 77(4):1229-1279.

Lu Y, Gupta A, Ketter W, Van Heck E (2019) Information transparency in business-to-business auction markets: The role of winner identity disclosure. *Management Science* 65(9):4261-4279.

Xu Y (2017) Generalized synthetic control method: Causal inference with interactive fixed effects models. *Political Analysis* 25(1):57-76.