Speed at the Cost of Quality: How Cursor AI Increases Short-Term Velocity and Long-Term Complexity in Open-Source Projects

Hao He
Carnegie Mellon University
Pittsburgh, PA, USA
haohe@andrew.cmu.edu

Courtney Miller
Carnegie Mellon University
Pittsburgh, PA, USA
courtneymiller@cmu.edu

Shyam Agarwal
Carnegie Mellon University
Pittsburgh, PA, USA
shyamaga@andrew.cmu.edu

Christian Kästner
Carnegie Mellon University
Pittsburgh, PA, USA
kaestner@cs.cmu.edu

Bogdan Vasilescu
Carnegie Mellon University
Pittsburgh, PA, USA
vasilescu@cmu.edu

Abstract

Large language models (LLMs) have demonstrated the promise to revolutionize the field of software engineering. Among other things, LLM agents are rapidly gaining momentum in software development, with practitioners reporting a multifold increase in productivity after adoption. Yet, empirical evidence is lacking around these claims. In this paper, we estimate the *causal* effect of adopting a widely popular LLM agent assistant, namely Cursor, on *development velocity* and *software quality*. The estimation is enabled by a state-of-the-art difference-in-differences design comparing Cursor-adopting GitHub projects with a matched control group of similar GitHub projects that do not use Cursor. We find that the adoption of Cursor leads to a statistically significant, large, but transient increase in project-level development velocity, along with a substantial and persistent increase in static analysis warnings and code complexity. Further panel generalized-method-of-moments estimation reveals that increases in static analysis warnings and code complexity are major factors driving long-term velocity slowdown. Our study identifies quality assurance as a major bottleneck for early Cursor adopters and calls for it to be a first-class citizen in the design of agentic AI coding tools and AI-driven workflows.

CCS Concepts

• **Software and its engineering** → *Development frameworks and environments*; • **Computing methodologies** → *Intelligent agents*.

ACM Reference Format:
Hao He, Courtney Miller, Shyam Agarwal, Christian Kästner, and Bogdan Vasilescu. 2026. Speed at the Cost of Quality: How Cursor AI Increases Short-Term Velocity and Long-Term Complexity in Open-Source Projects. In *23rd International Conference on Mining Software Repositories (MSR '26), April 13–14, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 13 pages.
https://doi.org/10.1145/3793302.3793349

1 Introduction

Large language models (LLMs) have demonstrated remarkable capabilities in code generation, achieving near-human performance across various software engineering tasks [47, 63]. Among emerging applications, LLM agent assistants—tools that combine LLMs with autonomous capabilities to inspect project files, execute commands, and iteratively develop code—represent a particularly promising direction for integrating LLMs into software development. For example, Cursor [8], a popular LLM agent assistant, has generated considerable enthusiasm among practitioners, with developers self-reporting multi-fold (as large as 10x) productivity increases and claiming transformative workflow impacts [10, 15].

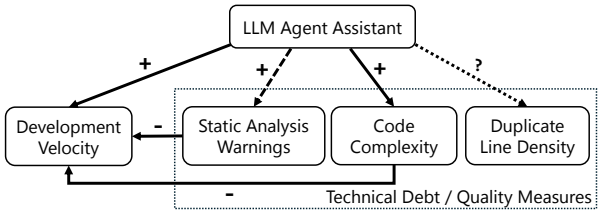


Figure 1: Our theory around how LLM agent assistants impact software development. Solid lines indicate causal relationships supported by our analysis; dashed lines indicate partial evidence; and dotted lines indicate inconclusive evidence.

However, substantial concerns have been raised about the quality of LLM-generated code and the long-term consequences of AI-driven development workflows. Studies documented that AI coding assistants can produce code with security vulnerabilities [92, 94], performance issues [74], code smells [100], and increased complexity [77]. Yet, these findings, derived from evaluations of early Codex models in controlled experiments [94], completion-based tools like early GitHub Copilot [74, 92, 100], or chat-based interfaces like ChatGPT [77], may not generalize to modern LLM agent assistants, which represent a qualitative shift in architecture and integration, not simply incremental improvement. Completion tools suggest individual lines based on immediate context; chat-based assistants require context-switching to formulate queries and integrate responses. Both operate at the periphery of the development workflow, with developers remaining the primary agents and maintaining oversight of AI-generated code at a granular level.

LLM agent assistants like Cursor, by contrast, are tightly integrated into the IDE with persistent codebase awareness, autonomously navigating files, proposing multi-file refactorings, and implementing features spanning dozens of files—all within the development environment. This architectural difference has profound implications that cannot be extrapolated from prior studies: Automation scope shifts from accelerating typing to automating entire workflows; seamless integration may enable both productivity gains and over-reliance or reduced code review rigor; quality implications of reviewing large, AI-generated multi-file changes differ fundamentally from reviewing line-by-line completions; and temporal dynamics may involve longer-term effects as technical debt accumulation. Consequently, prior findings about security vulnerabilities

This work is licensed under a Creative Commons Attribution 4.0 International License.

MSR '26, Rio de Janeiro, Brazil

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2474-9/26/04
https://doi.org/10.1145/3793302.3793349

in Copilot completions or complexity issues in ChatGPT-generated functions provide limited insight into whether—and how—these issues manifest in agentic tools that generate code at a substantially larger scale with different developer oversight patterns.

This gap between generations of AI coding tools also shows up in recent research. For example, Becker et al. [26] show through controlled experiments that early-2025 AI tools, including Cursor, do not help experienced open-source developers solve real day-to-day tasks faster, pointing to potential slowdown mechanisms such as developer over-optimism, low AI reliability, and high task complexity. Their findings contradict substantial prior literature on earlier-generation AI coding assistants [39, 45, 62, 65, 90, 93, 101, 104, 106, 109, 111], which generally found modest velocity improvements from code completion tools like early GitHub Copilot.

Most recently, Watanabe et al. [108] take an important step toward understanding modern agentic tools by examining 567 pull requests generated by Claude Code, finding that 83.8% are accepted and merged. However, their analysis focuses on PR acceptance rates and task types rather than on longitudinal, project-level effects of tool adoption on development velocity and code quality. Whether high acceptance rates of individual agent-generated PRs translate into sustained productivity gains and maintained quality at the project level—or whether quality degradation accumulates as teams integrate these tools into everyday workflows over months—remains an open empirical question. To address this gap, we ask:

RQ: *How does the adoption of LLM agent assistants impact project-level development velocity and code quality?*

We focus on Cursor, one of the earliest and most widely adopted LLM agent assistants [3], as our empirical case. Our key insight is that the presence of `.cursorrules` configuration files in GitHub repositories signals the adoption of Cursor's multi-file editing and agentic capabilities, and thus the date of the first commit touching these configuration files serves as a proxy for the adoption date of the modern Cursor client with agentic coding features.[1] By scanning Cursor configuration files across GitHub, we identify 806 repositories that adopted the modern Cursor client, with most of the adoptions happening between August 2024 and March 2025.

We then use a *difference-in-differences* (DiD) design with staggered adoption [28, 30], comparing repositories that adopt Cursor at different times to a matched control group that never adopts during our observation period. This *quasi-experimental* approach uses naturally occurring variation in adoption timing to identify causal effects while controlling for repository-specific characteristics and common temporal trends. To construct a comparable control group, we use propensity score matching [25] to select 1,380 similar repositories from those never adopting Cursor during our observation period. Our matching model incorporates the dynamic history of repository characteristics—activity levels, contributor counts, and development patterns over the six months preceding potential adoption—ensuring treated and control repositories exhibit similar observable trajectories before adoption.

To estimate treatment effects, we use *the Borusyak et al. [28] imputation estimator*, a modern DiD approach designed for staggered adoption that avoids biases from traditional two-way fixed effects models. Using this approach, we estimate the impact of Cursor adoption on two velocity outcomes (commits and lines added) and three code quality outcomes (static analysis warnings, code complexity, and duplicate line density). Finally, to test temporal interactions between outcomes, we also estimate the impact of changes in code quality on future development velocity (and vice versa) using *panel generalized method of moments* (GMM) models [22].

Our findings reveal a concerning picture among GitHub open-source projects adopting Cursor. First, the adoption of Cursor leads to significant, large, but transient velocity increases: Projects experience 3-5x increases in lines added in the first adoption month, but gains dissipate after two months. Concurrently, we observe persistent technical debt accumulation: Static analysis warnings increase by 30% and code complexity increases by 41% post-adoption according to the Borusyak et al. [28] DiD estimator. Panel GMM models reveal that accumulated technical debt subsequently reduces future velocity, creating a self-reinforcing cycle. Notably, Cursor adoption still leads to significant increases in code complexity, even when models control for project velocity dynamics.

These findings carry important implications for research and practice. Our longitudinal evidence of how Cursor affects real-world software projects reveals complex temporal dynamics between AI-augmented velocity gains and quality outcomes (Figure 1), warranting further investigation. For practitioners, our results suggest that deliberate process adaptations—those that scale quality assurance with AI-era velocity—are necessary to realize sustained benefits from the use of LLM agent assistants. Our findings also highlight the need for quality assurance as a first-class design citizen in AI-driven development tools and workflows, suggesting directions for improvement in tool design and model training.

2 Related Work

The human-level performance of recent LLMs enables their practical applications to various software engineering tasks, such as code completion [64], code review [78], and testing [98] (see also the two surveys by Fan et al. [47] and Hou et al. [63]). The 2024 Stack Overflow Developer Survey shows that 76% of all respondents are using or planning to use LLM tools in their development process [2]. This wide adoption raises two main questions for researchers: (1) To what extent do LLMs improve developer productivity? (2) To what extent should we trust the code generated by LLMs?

A large body of prior research on the productivity impact of LLMs focuses on *code completion tools*—mostly the pre-agentic GitHub Copilot [39, 45, 62, 65, 90, 93, 101, 104, 106, 109, 111], with only a few execeptions [72, 91]. Evidence from small-scale, constrained randomized controlled experiments demonstrates a productivity increase ranging from 21% [101] to 56% [93], as measured by task completion time. Field experiments conducted at Microsoft, Accenture, and Cisco report similar numbers (from 22% [39] to 36% [90]). The productivity increase estimated from open-source projects on

[1] Earlier Cursor versions offered code completion and chat-based code generation. Cursor rule files were first released in mid-2024, roughly at the same time as the Composer feature, which provides a conversational interface for multi-file code generation. The agentic features of Composer were released in November 2024 and made default in February 2025 [97]. Thus, precisely speaking, our identification captures the adoption of modern Cursor with a mix of Composer's multi-file editing and subsequent transition to the default agentic mode. We call this modern version simply "Cursor" for brevity, and we show in the Appendix that limiting our analyses to adoption cohorts before/after agent release/agent made default does not change our main results.

observational data is similar and sometimes lower: a DiD design comparing Python and R packages estimates a 17.82% increase in new releases among Python packages after Copilot availability [109]—without clear knowledge of which packages used Copilot. Another study of proprietary Copilot backend data estimates only a 6.5% increase in project-level productivity, as measured by the number of accepted pull requests [101]. A more general study using a neural classifier to identify AI-generated code on GitHub shows moving to 30% AI use raises quarterly commits by 2.4% [42]. Studies point to various mechanisms causing the productivity increase, such as how LLM adoption increases work autonomy [62] and helps iterative development tasks (e.g., bug fixing) [109].

Although the productivity gains are promising, there are also increasing concerns around the trustworthiness of LLM-generated code. For example, it is well-known that LLMs may generate code with security vulnerabilities [21, 52, 69, 77, 92], performance regressions [74], code smells [100], and outdated APIs [68, 107]. On the other hand, evidence regarding the complexity of LLM-generated code compared to humans is inconclusive [38, 80, 86]. The LLM trustworthiness problem becomes more complicated with humans in the loop. For example, prior controlled experiments report mixed results on whether developers write more or less secure code with the help of LLMs [87, 94, 96], and studies often suggest heterogeneous treatment effects of LLMs on developers of different skill levels [39, 41, 93, 101]. While prior work points to many mechanisms by which adopting LLMs may affect software quality, their findings are typically derived from benchmark analyses [e.g., 92] or developer opinions [e.g., 80]. We are unaware of any prior studies that systematically investigated project-level quality outcomes in the wild after LLM adoption, let alone those that used rigorous causal inference techniques (the closest being the study by Yeverechyahu et al. [109] discussed above).

Recently, there has been an increasing interest in the application of *LLM agents*—LLMs with the capability to autonomously utilize external resources and tools—to software engineering [60, 66, 76]. A popular application scenario is an *LLM agent assistant* within a code editor, in which LLMs are allowed to inspect/edit project files, conduct web searches, and execute shell commands to fulfill prompts provided by developers. At the time of writing, there are several production-ready code editors with built-in LLM agent assistants, such as Cursor [8], VS Code [19], Windsurf [20], Tabnine [17], and Cline [5], with the extreme beginning to shift away from IDEs entirely and switching to command-line or web interfaces, such as Claude Code [4] and OpenHands [12]. These agentic tools are seeing rapid adoption among developers, as evidenced also in our data for Cursor in Figure 2. From the gray literature, we see extremely optimistic estimates of the productivity boost from LLM agent assistants: for example, developers self-report multi-fold productivity increases in a Reddit post [15], orders of magnitude larger than any empirical estimates for prior LLM tools. However, a recent controlled study with human participants shows that developers may be overoptimistic and that adopting LLM agent assistants does not make them faster in real-world open-source development tasks [26]. To the best of our knowledge, empirical evidence regarding the impact of LLM agent assistants on *long-term project-level outcomes*, especially *software quality outcomes*, is still lacking.

Our contribution is two-fold. First, our DiD design looks at the additional project-level productivity gain, if any, from using a modern *agentic* coding assistant (Cursor) *relative to the state-of-the-practice* (likely a mixture of human-written code and code generated by earlier-generation AI tools). Second, we provide a comprehensive analysis of the impact of adopting Cursor on code quality, which is the first to the best of our knowledge, and highlight potential velocity-quality trade-offs and their complex interactions.

3 Research Design and Methods

We estimate the *causal* effects of adopting Cursor on *development velocity* and *code quality*, both of which are considered important project outcomes, are commonly measured in prior research, and are closely tied to perceived overall project productivity [37, 51, 88]. We start by building a dataset with: (1) repositories adopting Cursor at different times and (2) comparable repositories that never adopted Cursor (Section 3.1). Then, we define our specific outcomes of interest and additional covariates (Section 3.2). The estimation of Cursor's causal effect on these outcomes is enabled by a difference-in-differences (DiD) design with staggered adoption [28] (Section 3.3): Under the assumption that similar repositories would, on average, evolve similarly in the absence of Cursor adoption (i.e., the *parallel trend assumption*), later-adopters and never-adopters can effectively serve as a quasi-experimental comparison group for earlier-adopters while accounting for observable covariates and macro trends (e.g., open-source repositories overall getting more or less active over time). Finally, since the results from DiD suggest *interactions* between development velocity and software quality, as also indicated in prior work [27, 37], we fit dynamic panel generalized method of moments (GMM) models [22] to support our interpretation of the DiD results (Section 3.4).

3.1 Data Collection

3.1.1 The Cursor IDE. Cursor [8] is an AI-powered IDE built as a VS Code fork with agentic capabilities integrated into the development workflow. Unlike earlier code completion tools, Cursor's agentic mode enables an autonomous, goal-directed AI workflow: The agent can navigate entire codebases, infer project architecture across multiple files, make multi-file edits, run terminal commands, execute tests, and iteratively debug code, with humans mainly serving as supervisors rather than traditional coders. Using this workflow, developers can rely entirely on AI for feature implementation, refactoring, test generation, documentation, and bug fixing within their native development environment. They can choose between frontier models from OpenAI, Anthropic, and Google, either via Cursor's built-in service or their own API keys.

We chose Cursor as our main study case for two reasons. First, our preliminary exploration found widespread and growing adoption compared to competing tools, providing sufficient statistical power for causal inference. Second, Cursor is among the earliest to allow optional configuration files (e.g., `.cursorrules`) in the git repository to direct AI behavior [7], leading to an adoption event proxy timestamp in the version control history and a scalable identification strategy based on these files. However, this identification strategy also has important limitations (Section 3.5), which we will address through robustness checks (Section 4.3).

Table 1: Descriptive statistics of the 806 repositories using Cursor, collected at the time of data collection (April 2025).

	Mean	Min	25%	Median	75%	Max
Age (days)	1,002.3	283	361.5	521.5	1164.2	6,208
Stars	1,475.3	10	20.0	51.0	242.0	122,280
Forks	215.9	0	3.0	9.0	37.0	51,745
Contributors	19.2	0*	1.0	3.0	10.0	461
Commits	1,816.6	1	49.0	209.0	951.8	86,954
Issues	1,070.3	0	3.0	31.5	232.5	100,614
Pull Requests	719.6	0	1.0	18.0	161.8	72,015

∗ The GitHub API can return zero contributors for a repository if none of its commits can be mapped back to a GitHub user.

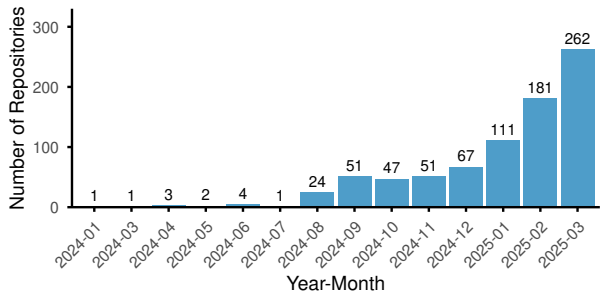


Figure 2: The Cursor adoption time of the 806 repositories in our study, which all have ≥10 stars and Cursor configuration files at the time of data collection (April 2025).

3.1.2 Identifying GitHub Projects Adopting Cursor. We identify Cursor-adopting repositories and track adoption dates through configuration files in the git history. In the GitHub code search API [14], we query for repositories with `.cursorrules` files or `.cursor` folders. Since the API limits results to 1,000 per query, we implement an adaptive partitioning algorithm based on file sizes: For each query with size interval $[a, b]$, we create two queries $[a, (a+b)/2)$ and $[(a+b)/2, b]$ until results fall below 1,000. This discovered 23,308 Cursor files across 3,306 non-fork repositories as of March 2025.

To filter non-software, educational, toy, and spam repositories [59, 67], we follow prior work [58, 61, 102] by requiring at least 10 stars at collection time—a threshold achieving 97% precision in identifying engineered projects [83]. This yields 806 repositories with adoption dates between January 2024 and March 2025 that are still available on GitHub at the time of data analysis (August 2025).

As expected, the dataset is highly skewed across many repository-level metrics (Table 1), and adoption time is staggered, with adoptions growing over time (Figure 2). These dataset characteristics motivate us to adopt a DiD design with staggered adoption and a matched control group, as we will discuss in the remainder of this section. While we did not filter based on activity levels here, activity-based subsets will be used as part of our robustness checks. The top five primary programming languages in our dataset are: TypeScript (366 repositories), Python (118 repositories), JavaScript (60 repositories), Go (36 repositories), and Rust (24 repositories).

3.1.3 Building a Control Group via Propensity Score Matching. A staggered DiD design usually requires a "never-treated" group to serve as comparison units, which, in our case, means repositories that never adopted Cursor (i.e., *never-adopters*). This comparison allows for a causal interpretation under the parallel trend assumption: Repositories that adopted Cursor should, on average, evolve similarly in the absence of Cursor adoption compared to the control group. However, this assumption is unlikely to hold if there are systematic differences between Cursor-adopting repositories and the control group. Repositories adopting Cursor may be more active, more rapidly growing, and have larger communities compared to random never-adopter repositories, leading to biased estimates.

To address these potential confounders and build a comparable quasi-experimental control group, we use *propensity score matching* [25, 29]. The high-level idea is to estimate propensity scores (i.e., the probability of adoption conditional on pre-adoption covariates) for each Cursor-adopting repository and a large population of other GitHub repositories, and retain only never-adopter repositories with propensity scores similar to those of the control group. Specifically, we define this population as all GitHub repositories with ≥10 stars (matching our inclusion threshold for Cursor adopters). For each month with major Cursor adoptions (August 2024—March 2025), we collect monthly time series from GHArchive [1] for all repositories in the population: age, active users, stars, forks, releases, pull requests, issues, comments, and total events.

Rather than static snapshots, we fit propensity score models to capture *dynamics*: repositories experiencing rapid growth or changing patterns may be more likely to adopt new tools. Let T_{it} denote repository age and X_{it} denote remaining covariates for repository i at month t. We estimate propensity scores $P(\text{treat}|t, T_i, X_i)$ (i.e., the probability of getting treated given t and repository-level covarates T_i, X_i) via the following logistic regression:

$$\log \frac{P(\text{treat}|...)}{1-P(\text{treat}|...)} = \alpha + \beta T_{i,t-1} + \sum_{j=1}^{6} \Gamma_j X_{i,t-j} + \Theta \sum_{j=7}^{\infty} X_{i,t-j} + \epsilon_i \quad (1)$$

where α, β, Γ_j and Θ are regression parameters. This effectively captures: (1) *repository maturity* ($T_{i,t-1}$), as older repositories may have different adoption patterns, (2) *recent dynamics* ($\sum_{j=1}^{6} \Gamma_j X_{i,t-j}$), i.e., month-by-month evolution over six months captures trends and growth, and (3) *historical baselines* ($\Theta \sum_{j=7}^{\infty} X_{i,t-j}$), i.e., cumulative history providing context on overall project scale and activity. By including lags, we ensure that propensity scores reflect both activity *level* and *trajectory*—two repositories with identical July 2024 pull requests may differ if one is growing while the other declines; such longitudinal characteristics may correlate with both Cursor adoption and long-term velocity and quality outcomes.

Since candidate repositories outnumber Cursor adopters by orders of magnitude, we sample at most 10,000 candidates per Cursor adoption month to avoid extreme imbalance and improve fit. This yields AUC values ranging from 0.83 to 0.91, indicating high discriminative power in the fitted logistic regression models. For each Cursor-adopting repository, we perform 1:3 nearest-neighbor matching (three controls per treated unit). While 1:1 or 1:2 is most common [24, 95], many adopters matched the same control during 1:1 matching, so 1:3 provides higher control group diversity. We additionally match only repositories with the same primary language

(queried from GitHub API, unavailable in GHArchive), controlling for language-specific LLM performance differences [34, 105]. This yields a matched "never-treated" group with similar propensity score distributions and pre-adoption characteristics (see Appendix). Following standard quasi-experimental terminology, we refer to the sample of repositories adopting Cursor as the *treatment group* and the matched "never-treated" repositories not using Cursor as the *control group* in the remainder of this paper.

3.2 Metrics

For each repository in the treatment and control group, we collect monthly outcome metrics and time-varying covariates from January 2024 to August 2025. This ensures that there are at least six months of observations pre- and post-adoption for the treatment group and abundant comparison observations from the control group in each month, providing statistical power for DiD-based causal inference. Note that the dataset is an *unbalanced panel*, as not all repositories have observations over the entire observation period.

3.2.1 Outcomes. For repository i at month t, we collect two outcome metrics related to *development velocity*, a key dimension of software engineering productivity [37, 84]:

- Commits$_{it}$: Number of commits in repository i at month t;
- Lines Added$_{it}$: Total lines added, summed over all commits in repository i at month t.

Both have been used as productivity proxies [73, 82, 99] with moderate-to-strong correlation with perceived productivity [88].

Software quality is multi-faceted and difficult to capture with a single metric [35, 50, 70, 89]. Quality can be pivoted on defect density [54], specification rigor [49], user satisfaction [44], or technical debt [46]. However, many metrics cannot be reliably and scalably collected from version control data. In this study, we take the technical debt perspective [79] and test three source code maintainability metrics that can be reasonably estimated with static analysis: *static analysis warnings*, *duplicate line density*, and *code complexity*. All three are arguably positively correlated with project-level technical debt and negatively correlated with perceived code quality. Specifically, we use a local SonarQube Community server [16] to compute these outcome metrics for repository i at month t:

- Static Analysis Warnings$_{it}$: Total number of reliability, maintainability, and security issues for repository i at month t, as detected by SonarQube's static analysis. We refer to them as warnings, since static analysis can generate false positives [55]; this metric should be viewed as an estimate of the effort required to review potential issues in a project.
- Duplicate Line Density$_{it}$: Percentage of duplicated lines in codebase for repository i at month t. SonarQube's definition varies across programming languages, but usually requires at least 10 consecutive duplicate statements or 100 duplicate tokens to mark a block as duplicate [18].
- Code Complexity$_{it}$: Overall cognitive complexity [31] of codebase for repository i at month t. Per SonarQube [31], this metric quantifies code understandability and aligns better with modern coding practices than classic cyclomatic complexity [81].

3.2.2 Time-Varying Covariates. We control for the following time-varying covariates in our models for all treatment and control repositories over the entire observation period (Jan 2024 to Aug 2025): lines of code, age (days), number of contributors at month t, number of stars received at month t, number of issues opened at month t, and number of issue comments added at month t. Lines of code is collected from SonarQube [16] along with outcome metrics; number of contributors is estimated from version control history; remaining covariates are estimated from GHArchive event data [1]. Multi-collinearity analysis reveals that number of issues opened and number of issue comments added are highly correlated (Pearson's $\rho > 0.7$), so we exclude issue comments from subsequent modeling.

3.3 Difference-in-Differences

3.3.1 Background. DiD is an established econometric technique for causal inference in observational data [32, 40], with growing adoption in software engineering [33, 48, 85]. The key idea is to compare outcome changes in a treatment group to those in a control group (i.e., those not-yet-treated or never-treated) over the same observation periods. The name "difference-in-differences" originates from the fact that temporal changes are first differenced before differencing the outcome changes between the two groups, effectively isolating the effect of an intervention from other factors that affect all repositories similarly over the same period.

A DiD design critically relies on the *parallel trends assumption* for a causal interpretation: Absent treatment, the treatment and control group should, on average, follow similar outcome trajectories over the same period. While this assumption is generally not directly testable (would need a time machine), *pre-trend tests* are often used for assessing the *plausibility* of this assumption: If the model predicts similar outcomes for the treatment and control groups *before the adoption*, it is more plausible that the treatment group would evolve similarly if they were not exposed to the treatment. Apart from pre-trend tests, a matching process that controls for observable differences pre-adoption (e.g., Section 3.1.3) also strengthens the plausibility of this assumption in a specific research context.

In a DiD design, a *staggered adoption* setting comes with both promises and perils [23, 28]. In this setting, treatments occur at different times across cohorts rather than simultaneously (as in our case, Figure 2), and each treatment unit has repeated observations both before and after treatment. This setting enables repeated, multiple natural experiments: Later-adopters in the treatment group can serve as additional controls for those adopting before them, since they remain untreated during earlier periods. However, a staggered adoption setting also presents significant mathematical challenges to achieve consistent, efficient, and unbiased estimation of the causal effect [23, 43, 53] with ongoing active research [28, 30].

3.3.2 The Estimation Targets. In a DiD design, researchers are often interested in estimating the following two types of causal effects: (1) ATT, the average treatment effect on treated, and (2) ATT_h, the "horizon-average" treatment effect in a specific horizon h (i.e., the effect in h periods since the treatment). We define the two estimation targets mathematically in this section.

Let $\Omega = \{it\}$ denote the set of all observations from repository i and month t, Ω_1 denote the set of treated observations, and Ω_0 denote the set of untreated (i.e., never-treated and not-yet-treated) observations. Let Y_{it} denote the *actual* outcome of interest for repository i at month t, and $Y_{it}(0)$ denote the *potential* outcome for

repository i at month t if it is never treated. ATT is defined as the average of this causal treatment effect on all treated observations:

$$ATT = \frac{1}{|\Omega_1|} \sum_{it \in \Omega_1} \mathbb{E}[Y_{it} - Y_{it}(0)] \tag{2}$$

Let $\Omega_{1,h}$ denote the set of all treated observations h time periods after the treatment; ATT_h is defined as the average of the causal treatment effect in that specific horizon h:

$$ATT_h = \frac{1}{|\Omega_{1,h}|} \sum_{it \in \Omega_{1,h}} \mathbb{E}[Y_{it} - Y_{it}(0)] \tag{3}$$

3.3.3 The Borusyak et al. [28] Estimator. There are many possible methods to estimate ATT and ATT_h defined in Section 3.3.2, among which the two-way fixed effects (TWFE) estimator is most commonly used in early econometric studies. However, recent research shows that TWFE may produce biased estimates in the staggered adoption setting if treatment effects are heterogeneous over time [23, 43, 53]. To address this known limitation, we use the *Borusyak et al. [28] imputation estimator*, a state-of-the-art estimator designed explicitly for robust and efficient estimation in the staggered adoption setting, with this two-step process:

Step 1: Impute counterfactual outcomes. The estimator fits a counterfactual outcome regression model, *using only untreated observations* Ω_0 (i.e., pre-adoption observations for treated repositories and all observations for never-treated controls):

$$\hat{Y}_{it}(0) = \hat{\mu}_i + \hat{\lambda}_t + \hat{\Gamma}' Z_{it} + \epsilon_{it} \tag{4}$$

where $\hat{\mu}_i$, $\hat{\lambda}_t$ represent per-repository and per-month fixed effects; Z_{it} includes time-varying covariates (Section 3.2.2), and $\epsilon_{it} = Y_{it} - \hat{Y}_{it}$, $it \in \Omega_0$ is the error term when fitting this model on Ω_0. The use of only Ω_0 in this step ensures that counterfactual predictions are not contaminated by treatment effects as in the TWFE estimator. It is also important to note that all repository-invariant confounders (e.g., team culture, domain, language) are effectively controlled by per-repository fixed terms $\hat{\mu}_i$ and all time-invariant confounders (e.g., industry trends, platform changes, and seasonal patterns) are effectively controlled by per-month fixed terms $\hat{\lambda}_t$ in this step.

Step 2: Compare actual to counterfactual. For each repository-month post-adoption ($it \in \Omega_1$), the estimator predicts the potential outcome from the counterfactual outcome model: $\hat{Y}_{it}(0) = \hat{\mu}_i + \hat{\lambda}_t + \hat{\Gamma}' X_{it}$. We replace the $Y_{it}(0)$ in Equations 2 and 3 with the estimated $\hat{Y}_{it}(0)$, to get the final ATT and ATT_h estimations.

Finally, to assess the plausibility of the parallel trend assumption, Borusyak et al. [28] advises fitting an alternative model of Y_{it} for untreated observations Ω_0 with additional observables. In our paper, we follow the most typical convention and fit the following model with additional dummies before the onset of treatment:

$$Y_{it} = \hat{\mu}_i + \hat{\lambda}_t + \hat{\Gamma}' Z_{it} + \sum_{h=-k}^{-2} \hat{\tau}_h \mathbf{1}[t = E_i + h] + \epsilon_{it} \tag{5}$$

where E_i means the time of treatment for repository i and $\mathbf{1}[t = E_i + h]$ is equal to 1 if and only if the current time t is h months away from treatment (0 otherwise). Then, we use heteroscedasticity- and cluster-robust Wald tests [28] to test the joint null hypothesis that $\hat{\tau}_h = 0$ for $h = -k, ..., -2$. We drop $h = -1$ due to potential anticipation concerns (the developer may try use Cursor before adding Cursor rule files in the immediate month before). One way of viewing the above pre-trend testing procedure is a placebo test, in which τ_h estimates ATT_h for $h < 0$, which should be zero if the treatment has not yet happened. The estimated $\hat{\tau}_h$ for $h < 0$ and ATT_h for $h > 0$ are often combined into *event study plots* (e.g., Figure 3), in which the dynamic effect of treatment is visualized.

Note that the Borusyak et al. [28] imputation estimator has many alternative specifications with varying assumptions, and we merely describe the version we used in our paper. We refer interested readers to the original paper [28] regarding alternative specifications and the mathematical assumptions behind them. The Borusyak et al. [28] estimator is also not the only option, and we present results from alternative DiD estimators (TWFE and Callaway and Sant'Anna [30]) in the Appendix as additional robustness checks.

3.4 Testing Velocity & Quality Interactions

While DiD estimates treatment effects on individual outcomes, it does not capture temporal dynamics *between* outcomes. However, it is known that velocity and quality outcomes interact in our setting [27, 37]. Plus, our DiD results (Section 4) also suggest interactions, showing that the adoption of Cursor leads to non-sustained velocity increases and sustained quality declines: Development velocity increases may cause rapid technical debt accumulation, which may subsequently decrease velocity.

To test for dynamic relationships and bidirectional causality, we use the *generalized method of moments* (GMM) [57] to obtain consistent estimates when variables are potentially endogenous (correlated with unobserved errors). The key insight is to use *instrumental variables*—correlated with endogenous regressors but uncorrelated with errors—to identify causal effects. In panel data, lagged values serve as natural instruments, assuming past values influence current values but are uncorrelated with current shocks [22].

In our study, we use Arellano-Bond dynamic panel GMM [22], suited for: (1) dynamic dependence (current outcomes depend on past); (2) potential bidirectional causality; (3) short time series with many entities. To test a causality direction $X_t \rightarrow Y_t$ while accounting for Cursor adoption D, we estimate the following regression:

$$Y_{it} = \hat{\mu}_i + \hat{\lambda}_t + \hat{\rho} Y_{i,t-1} + \hat{\beta} D_{it} + \hat{\gamma} X_{it} + \hat{\Gamma}' Z_{it} + \epsilon_{it} \tag{6}$$

where $Y_{i,t-1}$ captures outcome persistence; D_{it} is a dummy representing Cursor adoption; X_{it} represents the potentially endogenous regressor of interest; the remainig follows Equation 4. During estimation, historical values of X_{it} (e.g., a linear combination of $X_{i,t-2}$ and $X_{i,t-3}$) are used as instrumental variables.

Specifically, we test the following temporal interactions:

$$\begin{aligned} \text{Lines Added}_{it} &\rightarrow \text{Static Analysis Warnings}_{it} \\ \text{Lines Added}_{it} &\rightarrow \text{Code Complexity}_{it} \\ \text{Static Analysis Warnings}_{it} &\rightarrow \text{Lines Added}_{i,t+1} \\ \text{Code Complexity}_{it} &\rightarrow \text{Lines Added}_{i,t+1} \end{aligned} \tag{7}$$

These models complement DiD by decomposing the mechanisms through which Cursor affects long-term outcomes, revealing whether quality degradation leads to subsequent velocity declines.

3.5 Limitations and Threats to Validity

3.5.1 Internal Validity. We discuss several important limitations of our identification strategy and advise readers to interpret our results in the context of this experimental setup and its limitations.

Observable adoption through committed configuration files. Our treatment group only includes repositories committing Cursor configuration files to their git system, but developers can use Cursor without committing such files. Thus, our sample represents repositories with observable Cursor adoption rather than all possible Cursor-adopting repositories. This creates potential selection bias: Repositories committing configuration files may be more committed to systematic integration, have more formal processes, or differ in unobservable ways. To the extent committed adopters use Cursor more systematically, our estimates may represent an upper bound on average effects across all users. However, if committed adopters are more quality-conscious (e.g., more likely to review AI-generated code carefully), estimates could also be conservative. In general, we view our sample as capturing repositories where adoption represents deliberate, visible practice change—precisely the population where long-term effects are most relevant.

Uncertainty about usage intensity and persistence. Even with observed configuration files, we do not know how intensively or persistently Cursor was used in each repository. If a repository has multiple contributors with their own development environment, the presence of `.cursorrules` only indicates *someone* experimented, not that *all* contributors used it continuously throughout post-adoption observations. Contributors may use Cursor heavily for one feature, then revert to traditional development, without leaving visible traces. Unless we observe explicit configuration removal (rare), we assume continued usage, but this approximates actual engagement. Therefore, our main estimates represent intent-to-treat (ITT) effects: the impact of adopting Cursor as measured by committing configuration, averaging over heterogeneous usage patterns.

Model and version heterogeneity. Our dataset lacks information on which Cursor version or LLM backend each repository used; regardless, we still argue that this does not compromise validity. Our research question focuses on the system-level effects of adopting an agentic coding assistant as an integrated development practice, rather than the effects of particular model architectures. Moreover, to the extent that different repositories use different model backends or developers switch models for different tasks, this heterogeneity increases external validity: Our estimates reflect the average treatment effects of adopting Cursor as it is actually used in practice on adopting repositories, across all model diversity, rather than the effects of single, fixed LLM configurations.

Imperfect matching. Even if our propensity score matching process achieved strong performance (AUC 0.83–0.91), it still remains subject to untestable unobserved confounders. For example, factors like developer expertise, team practices, project complexity, or organizational culture may affect both Cursor adoption and the post-adoption outcomes. Although we include numerous covariates hoping to control these factors latently, we believe perfect matching is generally impossible in our setting [25]. The matching process is intended to strengthen the plausibility of our DiD design, particularly the parallel trend assumption, rather than creating two directly comparable groups. The DiD design further addresses imperfect matching through incorporating per-repository and per-period fixed effects (Equation 4) and time-varying covariates (Section 3.2.2) into the counterfactual outcome model.

Contamination from alternative AI coding tools. Another particular concern for our study setting is that repositories in both the treatment and control groups may be using alternative AI tools before and throughout the observation period, with or without visible traces. For example, many developers may use early versions of GitHub Copilot or chat-based interfaces like the ChatGPT web portal [2] without leaving any visible traces in the git repository. Therefore, our estimates should be interpreted as the impact of systematic Cursor adoption compared to the current state-of-the-practice, in which code completion tools and chat-based AI interfaces may be prevalently used, not the impact of using Cursor with respect to no AI usage at all (the latter is generally not estimable in our observational dataset). However, we identify several observable AI coding tools in our dataset (e.g., Claude Code with `.claude` folders) and present robustness checks in Section 4.3.

3.5.2 External Validity. Our results may not generalize to other LLM agent assistants, proprietary software projects, and programming languages beyond the three dominant ones in our dataset (JavaScript, TypeScript, Python)—adoption patterns and impacts may differ substantially in these contexts. Importantly, our study period coincides with rapid evolution in LLM capabilities, agent tooling, and developer adoption patterns. Results observed may not persist as LLM agent assistants mature and developer workflows adapt. We encourage future replications and additional investigations of state-of-the-art LLM coding tools as they roll out.

Table 2: The Borusyak et al. [28] estimated average treatment effects on treated (ATT in Equation 2) post Cursor adoption. We log-transform all outcome variables to address skewness and facilitate comparison of treatment effects across outcome variables; after log-transformation, all estimated ATTs can be interpreted as a percentage change of $100(e^{ATT}-1)\%$.

Outcome	Estimate (Std. Error)	Percentage Change
Commits	0.0260 (0.0429)	+2.63% (±4.40%)
Lines Added	0.2514* (0.1063)	+28.58% (±13.7%)
Static Analysis Warnings	0.2644*** (0.0511)	+30.26% (±6.66%)
Duplicated Lines Density	0.0679 (0.0448)	+7.03% (±4.79%)
Code Complexity	0.3481*** (0.0538)	+41.64% (±7.62%)

Note: $^{*}p < 0.05$; $^{**}p < 0.01$; $^{***}p < 0.001$

4 Results

4.1 Difference-in-Differences

We summarize the Borusyak et al. [28] estimated average treatment effects ATT and horizon-average treatment effects ATT_h (see definitions in Section 3.3) in Table 2 and Figure 3 for the five velocity and quality outcome variables (see definitions in Section 3.2.1). All outcome variables pass the heteroscedasticity- and cluster-robust Wald tests [28] at the 0.05 level (i.e., passing pre-trend tests).

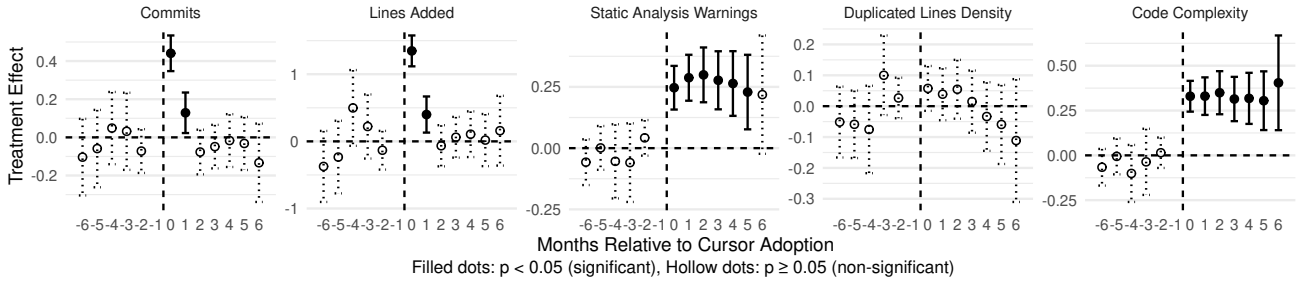


Figure 3: The estimated "horizon-average" treatment effects (ATT_h, Equation 3) 0 to +6 months after adoption, plus the "placebo" pre-adoption treatment effect estimates ($\hat{\tau}_h$ for testing the parallel trend assumption, Equation 5) -6 to -2 months before adoption. All outcome variables are log-transformed same as the estimated average treatment effects in Table 2.

4.1.1 Development Velocity. On average, Cursor adoption has a modestly significant positive impact on development velocity, particularly in terms of code production volume: Lines added increase by about 28.6% (Table 2). There is no statistically significant effect for the volume of commits. The horizon-average treatment effect estimations (Figure 3) reveal important temporal patterns that explain these differences: *The only significant development velocity gain is in the first two months post Cursor adoption.* The models estimate a 55.4% increase in commits in the first month, a 14.5% increase in commits in the second month, a 281.3% increase in lines added in the first month, and a 48.4% increase in lines added in the second month, respectively.

4.1.2 Software Quality. In contrast to the transient velocity gains, Cursor adopters show sustained patterns across static analysis warnings and code complexity. On average (Table 2), static analysis warnings increase significantly by 30.3%, and code complexity increases by 41.6%. The effect on duplicate line density is insignificant. The horizon-average treatment effect estimates (Figure 3) reveal that the increase in the two outcome metrics persists beyond the initial adoption period, contrasting the transient velocity gains.

4.2 Velocity & Quality Interactions

To distangle the temporal interactions between velocity and quality, we summarize dynamic panel GMM models testing causal paths specified in Equation 7, in Table 3. All models pass the Sargan test (confirming instrument validity) and AR(2) test (confirming no serial correlation in the original errors), validating the moment conditions required for causal interpretation [22].

The first two models show that, on average (across all Cursor-adopters and non-adopters in our dataset), and *holding all other temporal dynamic factors constant*: (1) An increase in development velocity does not produce a significant effect on static analysis warnings and code complexity. (2) Cursor adoption does not have a significant effect on static analysis warnings. Notably, increases in codebase size are a major determinant of increases in static analysis warnings and code complexity, and absorb most variance in the two outcome variables. However, even with strong controls for codebase size dynamics, the adoption of Cursor still has a significant effect on code complexity, leading to a 9% baseline increase on average compared to projects in similar dynamics but not using Cursor.

Table 3: The dynamic panel GMM estimates testing temporal interactions between velocity and quality attributes. L, W, and C stand for lines added, static analysis warnings, and code complexity, respectively (see Equation 7). The estimates for the remaining covariates are omitted here for brevity.

	$L_{it} \rightarrow W_{it}$	$L_{it} \rightarrow C_{it}$	$C_{it} \rightarrow L_{i,t+1}$	$W_{it} \rightarrow L_{i,t+1}$
Main Effect	−0.000	−0.006	−0.718***	−0.588***
	(0.015)	(0.016)	(0.098)	(0.092)
Cursor	−0.011	0.086**	1.044***	1.048***
	(0.033)	(0.030)	(0.124)	(0.124)
Lines of Code	0.845***	0.852***	0.869***	0.851***
	(0.073)	(0.059)	(0.153)	(0.155)
Num. Obs.	14,755	14,755	14,755	14,755
Sargan p	0.248	0.141	0.633	0.639
AR(1) p	<0.001	<0.001	<0.001	<0.001
AR(2) p	0.734	0.438	0.393	0.330

Notes: Two-way fixed effects (repository + month), two-step GMM with first-difference transformation. Robust standard errors in parentheses. ***p<0.001, **p<0.01, *p<0.05. The contemporaneous variables L_{it}, C_{it}, and W_{it} are instrumented with lags 2-3 to address endogeneity. Sargan p>0.05 indicates valid instruments. AR(1) p<0.05 is expected with the first-difference transformation. AR(2) p>0.05 indicates no serial correlation in the original errors. Sargan p>0.05 and AR(2) p>0.05 validate the moment conditions required for causal interpretation [22].

The last two models show that, on average, and *holding all other temporal dynamic factors constant*: (1) A 100% increase in code complexity and static analysis warnings causes a 64.5% and 50.3% decrease in development velocity as measured by lines added, respectively. (2) The adoption of Cursor results in a 1.84x baseline increase in lines added post adoption. Thus, the velocity gain from Cursor adoption would be fully cancelled out by a ∼5x increase in static analysis warnings or a ∼3x increase in code complexity, according to the dynamic panel GMM estimations.

To summarize, the dynamic panel GMM models suggest that: (1) *The adoption of Cursor leads to an inherently more complex codebase*; and (2) *The accumulation of static analysis warnings and code complexity decreases development velocity in the future.*

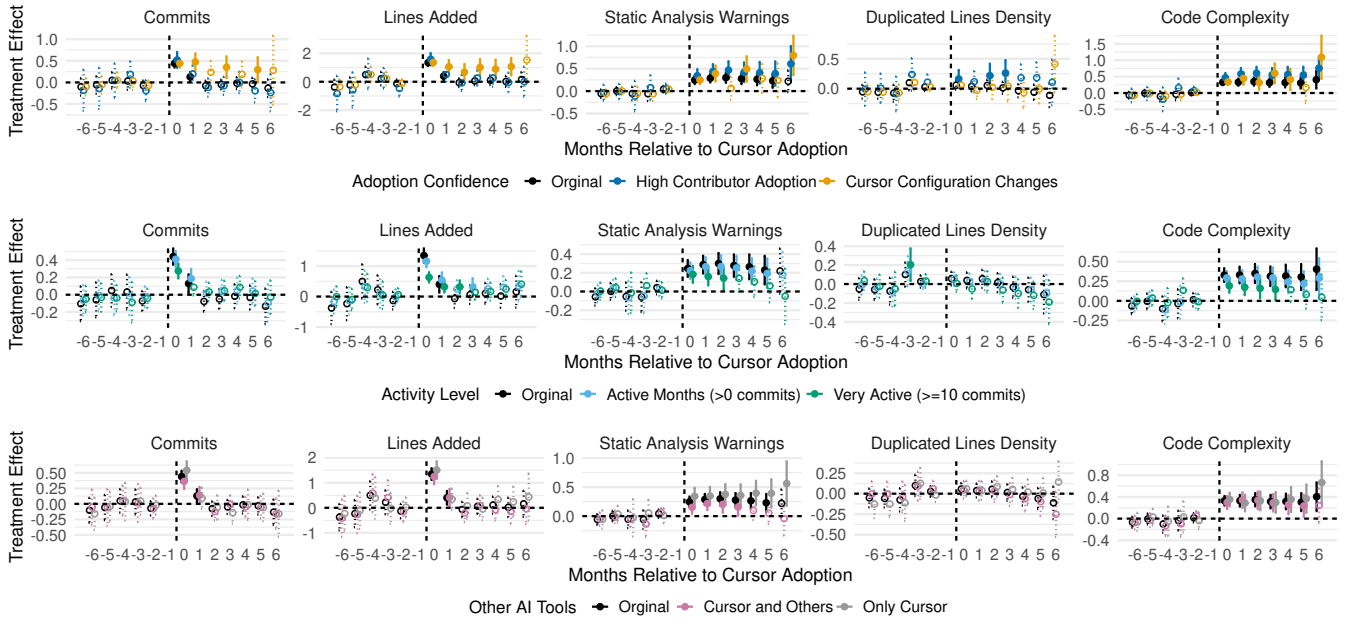


Figure 4: The estimated "horizon-average" treatment effects in alternative dataset settings as robustness checks. *Row 1*: Robustness check on subsets with higher Cursor adoption confidence, showing generally stronger effects in high-confidence adoption observations. *Row 2*: Robustness check on subsets of different activity levels, showing the persistence of our findings even in very active repositories. *Row 3*: Robustness check between repositories with/without observable use of other AI tools, showing that the potential use of other AI tools weakens Cursor effect estimations, but only by a small margin.

4.3 Robustness Checks

The above overall findings and the internal validity concerns behind them (Section 3.5) present significant interpretation challenges. In this section, we present additional robustness checks to explore and rule out alternative explanations for our observed results.

Recall that one limitation of our identification strategy is that it merely displays an "intent-to-treat" (ITT) signal without us knowing how Cursor was actually used in each treatment repository (Section 3.5). To test whether our findings are driven by repositories with genuine, sustained usage (versus minimal engagement), we repeat the same DiD modeling process on two subsets of data based on two alternative measurements of usage intensity:

- *High Contributor Adoption:* For each repository, we identify contributors modifying Cursor files (indicating experimentation) and calculate their fraction of total repository commits during observation. This subset keeps only "high-adoption" repositories, in which those modifying Cursor files also contributed ≥80% of commits, along with their own matched controls.
- *Cursor Configuration Changes:* For each repository, we identify all commits where `.cursorrules` files were modified post-adoption. This subset keeps only post-adoption observations with at least one commit modifying Cursor files during that period, representing sustained usage and ruling out potential post-adoption abandonments.

Results from the two subsets (Figure 4, Row 1) show that our quality-related findings are robust and amplified in subsamples with higher sustained usage: The accumulation of static analysis warnings and code complexity is stronger—not weaker—among repositories with continued configuration refinement and where Cursor users dominated activity. This strengthens causal interpretation: Effects are attributable to Cursor adoption rather than spurious correlation with other changes, and our main ITT estimates likely understate effects among repositories with intensive, sustained usage. Interestingly, while the velocity gain remains transient on average across repositories with high contributor adoption, we still observe a velocity increase in observations where developers are actively modifying Cursor files. This indicates that abandonments of Cursor post-adoption likely nullify at least part of the velocity gains (more discussions in Section 5).

Another alternative explanation for the transient velocity gain is that our DiD setting still does not sufficiently control for the macro-trend that many open-source repositories become inactive very fast (e.g., within two months). To test against this alternative explanation, we build two subsets of observations: The first keeps only repository-months with at least one commit, and the second keeps only repository-months with at least 10 commits. The results (Figure 4, Row 2) show that our main findings persist even in very active repositories, albeit weaker than our main ITT estimates, which is expected because it would be more difficult to achieve the same percentage change in the measured outcomes in repositories with higher baseline sizes and activity levels.

Finally, we assess the impact of dataset contamination from alternative AI coding tools in our study setting (Section 3.5). For this purpose, we take an overly conservative approach and identify all

repositories where alternative AI coding tools may have been used based on repository files (e.g., those with `.vscode` folder may have used GitHub Copilot). We find 382 repositories from the treatment group that may use other AI tools during our observation period: 345 with GitHub Copilot, 63 with Claude Code, 37 with Windsurf, 13 with Cline (13), and 2 with OpenHands (note that they overlap heavily). We build a *Cursor and Other* subset for these repositories and an *Only Cursor* subset for the remaining repositories, along with their own matched controls. The results (Figure 4, Row 3) show that while contamination weakens our main ITT estimates (e.g., some repositories may be using GitHub Copilot before Cursor), all of our main findings are persistent through all three settings and amplified in settings where prior AI tool usage is less likely.

To summarize, the above robustness checks reassure our main causal findings against concerns from potential non-compliance (i.e., treatment repositories not actually using Cursor), selection bias (i.e., treatment repositories generally becoming inactive fast), and confounding from other AI tools (i.e., the presence of other AI tools nullifying estimates on the adoption of Cursor).

5 Discussion

5.1 Theoretical Implications

Our study contributes to the rapidly growing literature regarding the impact of AI assistance on developer productivity [26, 39, 45, 62, 65, 72, 90, 93, 101, 104, 106, 109, 111]. More importantly, our study provides a novel longitudinal lens into project-level macro-outcomes, effectively connecting our findings to the existing software engineering literature around development velocity and software quality [27, 37, 71, 84, 103]. In this section, we connect our findings with prior research and discuss our theory around how LLM agent assistants may impact software development (Figure 1).

5.1.1 The Transient Velocity Gains and Possible Causes Behind It. Our first longitudinal finding—that the project-level velocity gains from adopting Cursor are concentrated in the initial one or two months before returning to a baseline level—contrasts task-level productivity improvements reported in controlled experiments [e.g., 91, 93]. One reason for such contrast likely stems from the temporal dynamics between development velocity and software quality (which is only observable in a longitudinal study setting): While LLM agent assistants increase development velocity, the increase in development velocity itself may increase codebase size and cause accumulation of technical debt; the latter would consequently decrease development velocity in the future. This negative effect of technical debt is supported by both the prior literature [27, 37] and our panel GMM models (Table 3). However, this mechanism alone likely does not fully explain why the development velocity gain vanishes after two months: A ~3x increase in code complexity or a ~5x increase in static analysis warnings would be necessary to fully cancel out the effect of Cursor adoption according to our models (Table 3), which is unlikely.

Another highly plausible explanation, as indicated in Section 4.3, is that open-source developers experience an *excitement-frustration-abandonment cycle* while they adopt Cursor. For example, during the initial adoption phase, developers may experience novelty effects and actively experiment on tasks where AI excels (e.g., rapid prototyping), contributing to the immediate velocity spike post-adoption. However, as developers encounter scenarios where AI is still limited (e.g., debugging intricate logic, understanding existing codebases, handling edge cases), frustration may accumulate. This frustration, combined with the cognitive overhead of verifying and debugging AI-generated suggestions, could lead to reduced usage or complete abandonment. This interpretation aligns with the emerging qualitative research documenting developer challenges with AI-assisted coding [26, 36, 75] and anecdotal evidence from Cursor users [e.g., 9, 11].

5.1.2 The Accumulation of Technical Debt and Code Complexity. Our findings reveal a nuanced relationship between velocity and quality that challenges simplistic narratives about AI coding degrading code quality [6]. While the absolute levels of static analysis warnings increase post adoption (Finding 2), a large part of this observed effect can be attributed to the causal path of increased velocity → increased code base size → increased technical debt (Table 3). In other words, LLM agent assistants amplify existing velocity-quality dynamics by enabling faster code production, but may not necessarily introduce more code quality issues than non-adopting projects moving with the same velocity. This proportional relationship has an important practical implication that quality assurance needs to scale with AI-era velocity (see Section 5.2). After all, the use of AI does not change the fact that all code is a liability and the asset lies only in the code's capabilities [13].

The substantial average increase in code complexity (25.1%, Table 2) warrants particular attention, as code complexity represents a distinct quality dimension from code quality issues. That code complexity increases even after accounting for velocity dynamics (Table 3) gives strong evidence that code generated with Cursor in our study sample is inherently more complex than human-written code. This effectively creates a "complexity debt" in projects that use AI heavily, which may amplify frustration and maintenance costs when the AI fails on more complex codebases later, possibly providing another mechanism explaining the transient velocity gain we observe after Cursor adoption in our dataset.

While the adoption of Cursor leads to no significant changes in duplicate line density in the entire study sample (Table 2, Figure 3), heavy Cursor adopters may exhibit modest increases (Figure 4). Future research is needed to gather evidence on code duplication concerns in high AI usage scenarios.

5.1.3 Contextual Factors in Open-Source Settings. Our findings should be interpreted within the specific context of open-source software development, which differs from enterprise settings in ways that likely influence the patterns we observe. Open-source projects typically feature: (1) voluntary participation with low switching costs, enabling easy abandonment when tools prove frustrating, (2) distributed collaboration with varying levels of coordination, potentially reducing systematic code review that might catch AI-introduced defects; (3) intrinsic motivation and learning goals, where experimenting with AI tools provides value beyond pure productivity; and (4) resource constraints that may limit comprehensive testing and quality assurance regardless of development velocity. These contextual factors likely amplify the excitement-frustration-abandonment cycle while potentially dampening the

quality feedback loop. In enterprise settings, organizational mandates, sunk training costs, and managerial oversight might sustain AI tool use despite user frustration, potentially leading to distinct temporal patterns (as indicated in a recent study [72]). Similarly, enterprise quality assurance processes—mandatory code review, automated testing requirements, dedicated QA teams, and even dedicated agents to do maintenance work—might prevent proportional technical debt accumulation by catching issues before they escalate. Future research should examine whether the transient gains and proportional debt patterns we observe generalize to enterprise contexts or represent open-source-specific phenomena.

5.2 Practical Implications

To overcome the technical debt accumulation ratchet, software projects using LLM agent assistants should focus on process adaptation that scales quality assurance with AI-era velocity. The proportional technical debt accumulation we observe (Section 4.1.2), combined with its velocity-dampening effects (Section 4.2), creates a self-reinforcing cycle that needs to be addressed at the project level. To overcome this, AI-adopting teams may consider refactoring sprints triggered by code quality metrics, mandating test coverage requirements that scale with lines of code added, or prompt engineering (e.g., engineered Cursor rules) to enforce rigid quality standards for LLM agents. Without such adaptations, the initial productivity surge may accelerate the project toward an unmaintainable end state.

To support the above process adaptation, AI coding tools need explicit design to support quality assurance alongside code generation. The LLM agents (at the time of study) are generation-first, leaving quality maintenance as an afterthought. Future assistants should suggest tests alongside code, flag unnecessary complexity in real time, and proactively recommend refactoring when code quality degrades—essentially becoming "pair programmers" for quality, not just velocity. More provocatively, tools might implement self-throttling: automatically reducing suggestion volume or aggressiveness when project-level complexity or debt exceeds healthy thresholds, forcing developers to consolidate before generating more code. Such features would align tools with long-term project health rather than short-term code production.

The potential overcomplication in AI-generated code warrants further research and improvement. The 25% increase in code complexity we observe (Table 3) represents a distinct quality dimension beyond code quality issues—a "comprehension tax" that persists regardless of functional correctness. This suggests LLMs may be generating structurally valid but semantically opaque code, perhaps because training objectives prioritize passing tests over non-functional requirements such as human readability [56, 110]. Unless future development workflows allow fully automated AI development without any human code reviews, code readability will remain an important dimension to pursue in AI-generated code. Addressing this requires both technical innovation (e.g., readability-aware fine-tuning, post-hoc simplification passes) and empirical investigation into what specifically makes LLMs generate overly complicated implementations. Until these complexities are addressed, software project teams should treat AI-generated code as requiring extra scrutiny during review, with particular attention to whether simpler implementations exist that achieve the same functionality.

6 Conclusion

This study presents the first large-scale empirical investigation of how LLM agent assistants impact real-world software development projects. Through a rigorous difference-in-differences design comparing Cursor-adopting repositories with matched control group repositories, complemented by dynamic panel GMM analysis, we provide evidence that challenges both unbridled optimism and categorical pessimism surrounding AI-assisted coding: Cursor adoption produces substantial but transient velocity gains alongside persistent increases in technical debt; such technical debt accumulation subsequently dampens future development velocity. Ultimately, our results suggest a self-reinforcing cycle where initial productivity surges give way to maintenance burdens.

However, several considerations suggest this picture may not be as bleak as it initially appears. First, our study captures a snapshot of rapidly evolving technology from mid-2024 to mid-2025, when LLM capabilities, agent designs, and developer practices are improving at unprecedented rates—future tools might be able to address the quality concerns we observed. Second, our quality metrics, while well-established in software engineering research, may not fully capture the multi-dimensional nature of code quality in AI-driven development. For example, complexity metrics were designed for human-written code; whether they appropriately penalize AI-generated patterns that are mechanically verifiable yet syntactically complex remains an open question. Third, the open-source context of our study may amplify both abandonment dynamics and quality concerns relative to enterprise settings with mandatory, dedicated quality assurance processes.

Looking forward, our findings point to clear research and practice directions (Section 5). Ultimately, this study demonstrates that realizing the promise of AI-assisted software development requires a holistic understanding of how AI assistance reshapes the fundamental trade-offs between development velocity, code quality, and long-term project sustainability. The age of AI coding has arrived—our challenge now is to harness it wisely.

Data Availability

We provide a replication package for this paper at:

https://doi.org/10.5281/zenodo.18368661

The Appendix is available in the latest arXiv version of this paper at: https://arxiv.org/abs/2511.04427.

Acknowledgments

He's, Kästner's, and Miller's work was supported in part by the National Science Foundation (award 2206859). Miller's work was also supported by the National Science Foundation Graduate Research Fellowship Program under Grant Number DGE214073. Vasilescu's and Agarwal's work was supported in part by the National Science Foundation (awards 2317168 and 2120323) and research awards from Google and the Digital Infrastructure Fund. We would like to thank Narayan Ramasubbu and Alexandros Kapravelos for providing valuable methodological feedback at earlier stages of this research. We would also like to thank all S3C2 Quarterly Meeting attendees for their insightful discussions around the early results of this study. Finally, we would like to thank Google Cloud for offering research credits to cover BigQuery-based analysis in this research.

References

[1] 2011. *GHArchive.* Retrieved Sep 19, 2024 from https://www.gharchive.org/

[2] 2025. *AI | 2024 Stack Overflow Developer Survey.* Retrieved Apr 20, 2025 from https://survey.stackoverflow.co/2024/ai

[3] 2025. *AI Global: Global Sector Trends on Generative AI.* Retrieved November 5, 2025 from https://www.similarweb.com/corp/wp-content/uploads/2025/07/attachment-Global-AI-Tracker-17.pdf

[4] 2025. *Claude Code | Claude.* Retrieved Sep 24, 2025 from https://claude.com/product/claude-code

[5] 2025. *Cline | AI Autonmous Coding Agent for VS Code.* Retrieved Apr 21, 2025 from https://cline.bot/

[6] 2025. *Coding on Copilot: 2023 Data Suggests Downward Pressure on Code Quality.* Retrieved Oct 1, 2025 from https://www.gitclear.com/coding_on_copilot_data_shows_ais_downward_pressure_on_code_quality

[7] 2025. *Cursor - Rules.* Retrieved Apr 22, 2025 from https://docs.cursor.com/context/rules/

[8] 2025. *Cursor - The AI Code Editor.* Retrieved Apr 21, 2025 from https://www.cursor.com/

[9] 2025. *Cursor AI Was Everyone's Favourite AI IDE. Until Devs Turned on It.* Retrieved Oct 1, 2025 from https://dev.to/abdulbasithh/cursor-ai-was-everyones-favourite-ai-ide-until-devs-turned-on-it-37d

[10] 2025. *How Cursor AI Can Make Developers 10x More Productive.* Retrieved Oct 16, 2025 from https://brianchristner.io/how-cursor-ai-can-make-developers-10x-more-productive/

[11] 2025. *The Love-Hate Relationship with Cursor: Why Some Devs Think It's Getting Worse.* Retrieved Oct 1, 2025 from https://www.arsturn.com/blog/the-love-hate-relationship-with-cursor-why-some-devs-think-its-getting-worse

[12] 2025. *OpenHands — the leading open source AI coding agent.* Retrieved Oct 21, 2025 from https://openhands.dev

[13] 2025. *Pluralistic: Code is a liability (not an asset).* Retrieved Jan 22, 2026 from https://pluralistic.net/2026/01/06/1000x-liability/

[14] 2025. *REST API endpoints for search - GitHub Docs.* Retrieved Apr 23, 2025 from https://docs.github.com/en/rest/search/search

[15] 2025. *Senior Devs Survey - Productivity Boost: r/cursor.* Retrieved Apr 21, 2025 from https://www.reddit.com/r/cursor/comments/1i3x7ul/senior_devs_survey_productivity_boost/

[16] 2025. *SonarQube Community Build Documentation.* Retrieved May 8, 2025 from https://docs.sonarsource.com/sonarqube-community-build/

[17] 2025. *Tabnine AI Code Assistant | private, personalized, protected.* Retrieved Apr 21, 2025 from https://www.tabnine.com/

[18] 2025. *Understanding Measures and Metrics | SonarQube Server Documentation.* Retrieved May 8, 2025 from https://docs.sonarsource.com/sonarqube-server/latest/user-guide/code-metrics/metrics-definition/

[19] 2025. *Visual Studio Code - The open source AI code editor.* Retrieved Oct 21, 2025 from https://code.visualstudio.com

[20] 2025. *Windsurf - Where developers are doing their best work.* Retrieved Oct 21, 2025 from https://windsurf.com

[21] Sri Haritha Ambati, Norah Ridley, Enrico Branca, and Natalia Stakhanova. 2024. Navigating (in)Security of AI-Generated Code. In *CSR*. IEEE, 1–8.

[22] Manuel Arellano and Stephen Bond. 1991. Some tests of specification for panel data: Monte Carlo evidence and an application to employment equations. *REStud* 58, 2 (1991), 277–297.

[23] Susan Athey and Guido W Imbens. 2022. Design-based analysis in difference-in-differences settings with staggered adoption. *J. Econom.* 226, 1 (2022), 62–79.

[24] Peter C Austin. 2010. Statistical criteria for selecting the optimal number of untreated subjects matched to each treated subject when using many-to-one matching on the propensity score. *AJE* 172, 9 (2010), 1092–1097.

[25] Peter C Austin. 2011. An introduction to propensity score methods for reducing the effects of confounding in observational studies. *Multivar. Behav. Res.* 46, 3 (2011), 399–424.

[26] Joel Becker, Nate Rush, Elizabeth Barnes, and David Rein. 2025. Measuring the Impact of Early-2025 AI on Experienced Open-Source Developer Productivity. *CoRR* abs/2507.09089 (2025).

[27] Terese Besker, Antonio Martini, and Jan Bosch. 2018. Technical debt cripples software developer productivity: A longitudinal study on developers' daily software development work. In *TechDebt*. ACM, 105–114.

[28] Kirill Borusyak, Xavier Jaravel, and Jann Spiess. 2024. Revisiting event-study designs: Robust and efficient estimation. *REStud* 91, 6 (2024), 3253–3285.

[29] Marco Caliendo and Sabine Kopeinig. 2008. Some practical guidance for the implementation of propensity score matching. *J. Econ. Surv.* 22, 1 (2008), 31–72.

[30] Brantly Callaway and Pedro HC Sant'Anna. 2021. Difference-in-differences with multiple time periods. *J. Econom.* 225, 2 (2021), 200–230.

[31] G. Ann Campbell. 2018. Cognitive complexity: An overview and evaluation. In *TechDebt@ICSE*. ACM, 57–58.

[32] David Card and Alan B Krueger. 1994. Minimum Wages and Employment: A Case Study of the Fast-Food Industry in New Jersey and Pennsylvania. *AER* 84, 4 (1994), 772–793.

[33] Annalí Casanueva, Davide Rossi, Stefano Zacchiroli, and Théo Zimmermann. 2025. The impact of the COVID-19 pandemic on women's contribution to public code. *EMSE* 30, 1 (2025), 25.

[34] Federico Cassano, John Gouwar, Francesca Lucchetti, Claire Schlesinger, Anders Freeman, Carolyn Jane Anderson, Molly Q. Feldman, Michael Greenberg, Abhinav Jangda, and Arjun Guha. 2024. Knowledge Transfer from High-Resource to Low-Resource Programming Languages for Code LLMs. *Proc. ACM Program. Lang.* 8, OOPSLA2 (2024), 677–708.

[35] Joseph P. Cavano and James A. McCall. 1978. A framework for the measurement of software quality. *SIGMETRICS Perform. Evaluation Rev.* 7, 3-4 (1978), 133–139.

[36] Valerie Chen, Ameet Talwalkar, Robert Brennan, and Graham Neubig. 2025. Code with Me or for Me? How Increasing AI Automation Transforms Developer Workflows. *CoRR* abs/2507.08149 (2025).

[37] Lan Cheng, Emerson R. Murphy-Hill, Mark Canning, Ciera Jaspan, Collin Green, Andrea Knight, Nan Zhang, and Elizabeth Kammer. 2022. What improves developer productivity at Google? Code quality. In *FSE*. ACM, 1302–1313.

[38] Chun Jie Chong, Zhihao Yao, and Iulian Neamtiu. 2024. Artificial-Intelligence Generated Code Considered Harmful: A Road Map for Secure and High-Quality Code Generation. *CoRR* abs/2409.19182 (2024).

[39] Zheyuan Kevin Cui, Mert Demirer, Sonia Jaffe, Leon Musolff, Sida Peng, and Tobias Salz. 2024. The effects of generative AI on high skilled work: Evidence from three field experiments with software developers. *Available at SSRN 4945566* (2024).

[40] Scott Cunningham. 2021. *Causal Inference: The Mixtape.* Yale University Press.

[41] Arghavan Moradi Dakhel, Vahid Majdinasab, Amin Nikanjam, Foutse Khomh, Michel C. Desmarais, and Zhen Ming (Jack) Jiang. 2023. GitHub Copilot AI pair programmer: Asset or Liability? *JSS* 203 (2023), 111734.

[42] Simone Daniotti, Johannes Wachs, Xiangnan Feng, and Frank Neffke. 2025. Who is using AI to code? Global diffusion and impact of generative AI. *CoRR* abs/2506.08945 (2025).

[43] Clément De Chaisemartin and Xavier d'Haultfoeuille. 2020. Two-way fixed effects estimators with heterogeneous treatment effects. *AER* 110, 9 (2020), 2964–2996.

[44] Peter J Denning. 1992. What is software quality? *CACM* 35, 1 (1992), 13–15.

[45] Thomas Dohmke, Marco Iansiti, and Greg Richards. 2023. Sea change in software development: Economic and productivity analysis of the AI-powered developer lifecycle. *CoRR* abs/2306.15033 (2023).

[46] Neil A. Ernst, Stephany Bellomo, Ipek Ozkaya, Robert L. Nord, and Ian Gorton. 2015. Measure it? Manage it? Ignore it? Software practitioners and technical debt. In *FSE*. ACM, 50–60.

[47] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M. Zhang. 2023. Large Language Models for Software Engineering: Survey and Open Problems. In *ICSE-FoSE*. IEEE, 31–53.

[48] Hongbo Fang, Hemank Lamba, James Herbsleb, and Bogdan Vasilescu. 2022. "This is damn slick!" Estimating the impact of tweets on open source project popularity and new contributors. In *ICSE*. 2116–2129.

[49] B Farbey. 1990. Software quality metrics: considerations about requirements and requirement specifications. *IST* 32, 1 (1990), 60–64.

[50] Norman Fenton and James Bieman. 2014. *Software Metrics: A Rigorous and Practical Approach.* CRC Press.

[51] Nicole Forsgren, Margaret-Anne D. Storey, Chandra Shekhar Maddila, Thomas Zimmermann, Brian Houck, and Jenna L. Butler. 2021. The SPACE of Developer Productivity: There's more to it than you think. *ACM Queue* 19, 1 (2021), 20–48.

[52] Yujia Fu, Peng Liang, Amjed Tahir, Zengyang Li, Mojtaba Shahin, and Jiaxin Yu. 2023. Security Weaknesses of Copilot Generated Code in GitHub. *CoRR* abs/2310.02059 (2023).

[53] Andrew Goodman-Bacon. 2021. Difference-in-differences with variation in treatment timing. *J. Econom.* 225, 2 (2021), 254–277.

[54] Robert B. Grady. 1993. Practical Results from Measuring Software Quality. *CACM* 36, 11 (1993), 62–68.

[55] Zhaoqiang Guo, Tingting Tan, Shiran Liu, Xutong Liu, Wei Lai, Yibiao Yang, Yanhui Li, Lin Chen, Wei Dong, and Yuming Zhou. 2023. Mitigating False Positive Static Analysis Warnings: Progress, Challenges, and Opportunities. *TSE* 49, 12 (2023), 5154–5188.

[56] Srishti Gureja, Elena Tommasone, Jingyi He, Sara Hooker, Matthias Gallé, and Marzieh Fadaee. 2025. Verification Limits Code LLM Training. *CoRR* abs/2509.20837 (2025).

[57] Lars Peter Hansen. 1982. Large sample properties of generalized method of moments estimators. *Econometrica* (1982), 1029–1054.

[58] Hao He, Bogdan Vasilescu, and Christian Kästner. 2025. Pinning Is Futile: You Need More Than Local Dependency Versioning to Defend against Supply Chain Attacks. *PACMSE* 2, FSE (2025), 266–289.

[59] Hao He, Haoqin Yang, Philipp Burckhardt, Alexandros Kapravelos, Bogdan Vasilescu, and Christian Kästner. 2024. 4.5 Million (Suspected) Fake Stars in GitHub: A Growing Spiral of Popularity Contests, Scams, and Malware. *CoRR* abs/2412.13459 (2024).

[60] Junda He, Christoph Treude, and David Lo. 2025. LLM-Based Multi-Agent Systems for Software Engineering: Literature Review, Vision, and the Road

Ahead. *TOSEM* 34, 5 (2025), 1–30.
[61] Runzhi He, Hao He, Yuxia Zhang, and Minghui Zhou. 2023. Automating Dependency Updates in Practice: An Exploratory Study on GitHub Dependabot. *TSE* 49, 8 (2023), 4004–4022.
[62] Manuel Hoffmann, Sam Boysel, Frank Nagle, Sida Peng, and Kevin Xu. 2024. *Generative AI and the Nature of Work*. Technical Report. CESifo Working Paper.
[63] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2024. Large Language Models for Software Engineering: A Systematic Literature Review. *TOSEM* 33, 8 (2024), 220:1–220:79.
[64] Rasha Ahmad Husein, Hala Aburajouh, and Cagatay Catal. 2025. Large language models for code completion: A systematic literature review. *Comput. Stand. Interfaces* 92 (2025), 103917.
[65] Saki Imai. 2022. Is GitHub Copilot a Substitute for Human Pair-programming? An Empirical Study. In *ICSE*. ACM, 319–321.
[66] Haolin Jin, Linghan Huang, Haipeng Cai, Jun Yan, Bo Li, and Huaming Chen. 2024. From LLMs to LLM-based Agents for Software Engineering: A Survey of Current, Challenges and Future. *CoRR* abs/2408.02479 (2024).
[67] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. Germán, and Daniela E. Damian. 2016. An in-depth study of the promises and perils of mining GitHub. *EMSE* 21, 5 (2016), 2035–2071.
[68] Mohammed Kharma, Soohyeon Choi, Mohammed AlKhanafseh, and David Mohaisen. 2025. Security and Quality in LLM-Generated Code: A Multi-Language, Multi-Model Analysis. *CoRR* abs/2502.01853 (2025).
[69] Raphaël Khoury, Anderson R. Avila, Jacob Brunelle, and Baba Mamadou Camara. 2023. How Secure is Code Generated by ChatGPT?. In *SMC*. IEEE, 2445–2451.
[70] Barbara A. Kitchenham and Shari Lawrence Pfleeger. 1996. Software Quality: The Elusive Target. *IEEE Softw.* 13, 1 (1996), 12–21.
[71] Philippe Kruchten, Robert L Nord, and Ipek Ozkaya. 2012. Technical debt: From metaphor to theory and practice. *IEEE Softw.* 29, 6 (2012), 18–21.
[72] Anand Kumar, Vishal Khare, Deepak Sharma, Satyam Kumar, Vijay Saini, Anshul Yadav, Sachendra Jain, Ankit Rana, Pratham Verma, Vaibhav Meena, et al. 2025. Intuition to Evidence: Measuring AI's True Impact on Developer Productivity. *CoRR* abs/2509.19708 (2025).
[73] M. J. Lawrence. 1981. Programming methodology, organizational environment, and programming productivity. *JSS* 2, 3 (1981), 257–269.
[74] Shuang Li, Yuntao Cheng, Jinfu Chen, Jifeng Xuan, Sen He, and Weiyi Shang. 2024. Assessing the Performance of AI-Generated Code: A Case Study on GitHub Copilot. In *ISSRE*. IEEE, 216–227.
[75] Jenny T. Liang, Chenyang Yang, and Brad A. Myers. 2024. A Large-Scale Survey on the Usability of AI Programming Assistants: Successes and Challenges. In *ICSE*. ACM, 52:1–52:13.
[76] Junwei Liu, Kaixin Wang, Yixuan Chen, Xin Peng, Zhenpeng Chen, Lingming Zhang, and Yiling Lou. 2024. Large Language Model-Based Agents for Software Engineering: A Survey. *CoRR* abs/2409.02977 (2024).
[77] Zhijie Liu, Yutian Tang, Xiapu Luo, Yuming Zhou, and Liang Feng Zhang. 2024. No Need to Lift a Finger Anymore? Assessing the Quality of Code Generation by ChatGPT. *TSE* 50, 6 (2024), 1548–1584.
[78] Junyi Lu, Lei Yu, Xiaojia Li, Li Yang, and Chun Zuo. 2023. LLaMA-Reviewer: Advancing Code Review Automation with Large Language Models through Parameter-Efficient Fine-Tuning. In *ISSRE*. IEEE, 647–658.
[79] Robert C Martin. 2009. *Clean Code: A Handbook of Agile Software Craftsmanship*. Pearson Education.
[80] Boris Martinović and Robert Rozić. 2024. Impact of AI tools on software development code quality. In *ICDTEAI*. Springer, 241–256.
[81] Thomas J. McCabe. 1976. A Complexity Measure. *TSE* 2, 4 (1976), 308–320.
[82] Audris Mockus, Roy T. Fielding, and James D. Herbsleb. 2002. Two case studies of open source software development: Apache and Mozilla. *TOSEM* 11, 3 (2002), 309–346.
[83] Nuthan Munaiah, Steven Kroh, Craig Cabrey, and Meiyappan Nagappan. 2017. Curating GitHub for engineered software projects. *EMSE* 22, 6 (2017), 3219–3253.
[84] Emerson R. Murphy-Hill, Ciera Jaspan, Caitlin Sadowski, David C. Shepherd, Michael Phillips, Collin Winter, Andrea Knight, Edward K. Smith, and Matthew Jorde. 2021. What Predicts Software Developers' Productivity? *TSE* 47, 3 (2021), 582–594.
[85] Keitaro Nakasai, Hideaki Hata, and Kenichi Matsumoto. 2018. Are donation badges appealing?: A case study of developer responses to Eclipse bug reports. *IEEE Software* 36, 3 (2018), 22–27.
[86] Nhan Nguyen and Sarah Nadi. 2022. An Empirical Evaluation of GitHub Copilot's Code Suggestions. In *MSR*. ACM, 1–5.
[87] Sanghak Oh, Kiho Lee, Seonhye Park, Doowon Kim, and Hyoungshick Kim. 2024. Poisoned ChatGPT Finds Work for Idle Hands: Exploring Developers' Coding Practices with Insecure Suggestions from Poisoned AI Models. In *S&P*. IEEE, 1141–1159.
[88] Edson Oliveira, Eduardo Fernandes, Igor Steinmacher, Marco Cristo, Tayana Conte, and Alessandro Garcia. 2020. Code and commit metrics of developer productivity: A study on team leaders perceptions. *EMSE* 25, 4 (2020), 2519–2549.
[89] Leon J. Osterweil. 1996. Strategic Directions in Software Quality. *ACM Comput. Surv.* 28, 4 (1996), 738–750.
[90] Ruchika Pandey, Prabhat Singh, Raymond Wei, and Shaila Shankar. 2024. Transforming Software Development: Evaluating the Efficiency and Challenges of GitHub Copilot in Real-World Projects. *CoRR* abs/2406.17910 (2024).
[91] Elise Paradis, Kate Grey, Quinn Madison, Daye Nam, Andrew Macvean, Vahid Meimand, Nan Zhang, Ben Ferrari-Church, and Satish Chandra. 2025. How much does AI impact development speed? An enterprise-based randomized controlled trial. In *ICSE-SEIP*. IEEE, 618–629.
[92] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2022. Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions. In *S&P*. IEEE, 754–768.
[93] Sida Peng, Eirini Kalliamvakou, Peter Cihon, and Mert Demirer. 2023. The Impact of AI on Developer Productivity: Evidence from GitHub Copilot. *CoRR* abs/2302.06590 (2023).
[94] Neil Perry, Megha Srivastava, Deepak Kumar, and Dan Boneh. 2023. Do Users Write More Insecure Code with AI Assistants?. In *CCS*. ACM, 2785–2799.
[95] Jeremy A Rassen, Abhi A Shelat, Jessica Myers, Robert J Glynn, Kenneth J Rothman, and Sebastian Schneeweiss. 2012. One-to-many propensity score matching in cohort studies. *Pharmacoepidemiol. Drug Saf.* 21 (2012), 69–80.
[96] Gustavo Sandoval, Hammond Pearce, Teo Nys, Ramesh Karri, Siddharth Garg, and Brendan Dolan-Gavitt. 2023. Lost at C: A User Study on the Security Implications of Large Language Model Code Assistants. In *USENIX Security*. USENIX, 2205–2222.
[97] Suproteem K Sarkar. 2025. Ai agents, productivity, and higher-order thinking: Early evidence from software development. *Available at SSRN 5713646* (2025).
[98] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2024. An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation. *TSE* 50, 1 (2024), 85–105.
[99] Ingo Scholtes, Pavlin Mavrodiev, and Frank Schweitzer. 2016. From Aristotle to Ringelmann: a large-scale analysis of team productivity and coordination in Open Source Software projects. *EMSE* 21, 2 (2016), 642–683.
[100] Mohammed Latif Siddiq, Shafayat H. Majumder, Maisha R. Mim, Sourov Jajodia, and Joanna C. S. Santos. 2022. An Empirical Study of Code Smells in Transformer-based Code Generation Techniques. In *SCAM*. IEEE, 71–82.
[101] Fangchen Song, Ashish Agarwal, and Wen Wen. 2024. The Impact of Generative AI on Collaborative Open-Source Software Development: Evidence from GitHub Copilot. *CoRR* abs/2410.02091 (2024).
[102] César Soto-Valero, Thomas Durieux, and Benoit Baudry. 2021. A longitudinal analysis of bloated Java dependencies. In *FSE*. ACM, 1021–1031.
[103] Margaret-Anne Storey, Brian Houck, and Thomas Zimmermann. 2022. How developers and managers define and trade productivity for quality. In *CHASE*. 26–35.
[104] Viktoria Stray, Elias Goldmann Brandtzæg, Viggo Tellefsen Wivestad, Astri Barbala, and Nils Brede Moe. 2025. Developer Productivity With and Without GitHub Copilot: A Longitudinal Mixed-Methods Case Study. *CoRR* abs/2509.20353 (2025).
[105] Lukas Twist, Jie M. Zhang, Mark Harman, Don Syme, Joost Noppen, and Detlef D. Nauck. 2025. LLMs Love Python: A Study of LLMs' Bias for Programming Languages and Libraries. *CoRR* abs/2503.17181 (2025).
[106] Priyan Vaithilingam, Tianyi Zhang, and Elena L. Glassman. 2022. Expectation vs. Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models. In *CHI Extended Abstracts*. ACM, 332:1–332:7.
[107] Chong Wang, Kaifeng Huang, Jian Zhang, Yebo Feng, Lyuye Zhang, Yang Liu, and Xin Peng. 2025. LLMs Meet Library Evolution: Evaluating Deprecated API Usage in LLM-based Code Completion. In *ICSE*. IEEE, 781–781.
[108] Miku Watanabe, Hao Li, Yutaro Kashiwa, Brittany Reid, Hajimu Iida, and Ahmed E Hassan. 2025. On the use of agentic coding: An empirical study of pull requests on GitHub. *CoRR* abs/2509.14745 (2025).
[109] Doron Yeverechyahu, Raveesh Mayya, and Gal Oestreicher-Singer. 2024. The Impact of Large Language Models on Open-source Innovation: Evidence from GitHub Copilot. In *ICIS*. AIS.
[110] Jiasheng Zheng, Boxi Cao, Zhengzhao Ma, Ruotong Pan, Hongyu Lin, Yaojie Lu, Xianpei Han, and Le Sun. 2024. Beyond correctness: Benchmarking multi-dimensional code generation for large language models. *CoRR* abs/2407.11470 (2024).
[111] Albert Ziegler, Eirini Kalliamvakou, X. Alice Li, Andrew Rice, Devon Rifkin, Shawn Simister, Ganesh Sittampalam, and Edward Aftandilian. 2022. Productivity assessment of neural code completion. In *MAPS@PLDI*. ACM, 21–29.