Speed at the Cost of Quality? The Impact of LLM Agent Assistance on Software Development

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

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

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

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

Bogdan Vasilescu
Carnegie Mellon University
Pittsburgh, Pennsylvania, 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 their application to software development, with practitioners claiming a multifold productivity increase 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 significant, large, but transient increase in project-level development velocity, along with a significant and persistent increase in static analysis warnings and code complexity. Further panel generalized method of moments estimation reveals that the increase in static analysis warnings and code complexity acts as a major factor causing long-term velocity slowdown. Our study carries implications for software engineering practitioners, LLM agent assistant designers, and researchers.

ACM Reference Format:
Hao He, Courtney Miller, Shyam Agarwal, Christian Kästner, and Bogdan Vasilescu. 2025. Speed at the Cost of Quality? The Impact of LLM Agent Assistance on Software Development. In *Proceedings of arXiv Preprint*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Large language models (LLMs) have demonstrated remarkable capabilities in code generation, achieving near-human performance across various software engineering tasks [47, 62]. 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 [9], a popular LLM agent assistant, has generated considerable enthusiasm among practitioners, with developers self-reporting multi-fold productivity increases and claiming transformative workflow impacts [11, 15].

However, substantial concerns persist regarding LLM-generated code quality and the long-term consequences of AI-heavy development workflows. Studies have documented that AI coding assistants can produce code with security vulnerabilities [92, 94], performance issues [73], code smells [101], and increased complexity [76]. Yet, these findings, derived from evaluations of early Codex models in controlled experiments [94], completion-based tools like early GitHub Copilot [73, 92, 101], or chat-based interfaces like ChatGPT [76], may not generalize to modern LLM agent assistants.

The distinction represents a qualitative shift in architecture and integration, not simply incremental improvement. Completion tools suggest individual lines in response to immediate context; chat-based assistants require context-switching to formulate queries and integrate responses. Both operate at the development workflow's periphery, with developers remaining primary agents and maintaining oversight of AI-generated code at a granular level.

Agentic 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 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 substantially larger scale with different developer oversight patterns.

This gap between tool generations is underscored by contradictory signals in recent research. 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

ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

earlier-generation AI coding assistants [40, 45, 61, 64, 90, 93, 102, 105, 107, 110, 112], which generally found modest velocity improvements from code completion tools like early GitHub Copilot.

Recent work by Watanabe et al. [109] takes an important step toward understanding modern agentic tools by examining 567 pull requests generated by Claude Code, finding 83.8% are accepted and merged by maintainers. However, their analysis focuses on PR acceptance rates and task types rather than longitudinal, project-level effects of tool adoption on development velocity and code quality over time. 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 software quality?*

We focus on Cursor, one of the most widely adopted LLM agent assistants [3], as our empirical case. We employ 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.

By scanning Cursor configuration files (e.g., `.cursorrules`) in GitHub repositories, we identify 807 repositories that adopted Cursor between January 2024 and March 2025. To construct a comparable control group, we use propensity score matching [25] to select 1,380 similar repositories from those never adopting Cursor during observation. Our matching model incorporates 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. We estimate treatment effects using the Borusyak et al. [28] estimator, a modern DiD approach designed for staggered adoption that avoids biases inherent in traditional two-way fixed effects models. We estimate Cursor adoption's impact on two velocity outcomes (commits, lines added) and three quality outcomes (static analysis warnings, code complexity, duplicate line density). To test temporal interactions between outcomes, we estimate software quality changes' impact on future development velocity (and vice versa) using panel generalized method of moments (GMM) models [22].

Our findings reveal a concerning picture among Cursor-adopting GitHub open-source projects. 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 code complexity increases even when models control for project velocity dynamics.

These findings carry important implications for research and practice. Our longitudinal evidence of how LLM agent assistants affect real-world software projects reveals complex temporal dynamics between AI-augmented velocity gains and quality outcomes, warranting further investigation. For practitioners, our results suggest that deliberate process adaptations—those scaling 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, pointing to possible directions for improvement in tool design and model training.

2 Related Work: The Impact of LLMs on Software Engineering

The human-level performance of recent LLMs enables their practical applications to various software engineering tasks, such as code completion [63], code review [77], and testing [99] (see also the two surveys by Fan et al. [47] and Hou et al. [62]). 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 [40, 45, 61, 64, 90, 93, 102, 105, 107, 110, 112], with only a few execeptions [71, 91]. Evidence from small-scale, constrained randomized controlled experiments demonstrates a productivity increase ranging from 21% [102] to 56% [93], as measured by task completion time. Field experiments conducted at Microsoft, Accenture, and Cisco report similar numbers (from 22% [40] to 36% [90]). The productivity increase estimated from open-source projects on 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 [110]—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 [102]. Studies point to various mechanisms causing the productivity increase, such as how LLM adoption increases work autonomy [61] and helps iterative development tasks (e.g., bug fixing) [110].

In addition to the promising productivity gain, there is also increasing concern about the trustworthiness of LLM-generated code. For example, it is well-known that LLMs may generate code with security vulnerabilities [21, 51, 68, 76, 92], performance regressions [73], code smells [101], and outdated APIs [67, 108]. On the other hand, evidence regarding the complexity of LLM-generated code compared to humans is inconclusive [38, 79, 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, 98], and studies often suggest heterogeneous treatment effects of LLMs on developers of different skill levels [40, 42, 93, 102]. While prior works point to many mechanisms regarding how adopting LLMs may affect software quality, their findings are usually obtained from benchmark analyses [e.g., 92] or developer opinions [e.g., 79]. We are unaware of any prior studies that systematically investigated project-level quality outcomes in the wild after LLM adoption, let alone that did so using causal inference techniques (the closest being the Yeverechyahu et al. [110] study 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 [59, 65, 75]. 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 the prompt given by the developer. At the time of writing, there are several production-ready code editors with built-in LLM agent assistants, such as Cursor [9], VSCode [19], Windsurf [20], Tabnine [17], and Cline [5], with technology beginning to shift away from IDEs entirely with tools such as Claude Code [4] and OpenHands [13]. These agentic IDEs are seeing rapid adoption among developers, as evidenced also in our data for Cursor in Figure 1. From the gray literature, we see extremely optimistic estimates on the productivity boost brought by 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 on prior LLM tools. However, a recent controlled study with human participants shows that developers may be overoptimistic and the adoption of LLM agent assistants does not make them faster in real 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 to this emerging literature is two-fold. First, our DiD design looks at the additional project-level productivity gain, if any, from using an LLM *agent* 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 LLM agent assistance with Cursor on software quality, which is the first to the best of our knowledge, and highlight potential velocity-quality trade-offs and their complex interactions in the case of Cursor adoption.

3 Methods

In this study, we estimate the causal effects of adopting the Cursor agentic IDE on *development velocity* and *software quality*. Development velocity is a commonly measured outcome in software projects [102, 110] and represents an important dimension of productivity [50, 88]. For causal inference, we use a difference-in-differences (DiD) design with staggered adoption [28], taking advantage of the fact that repositories adopted Cursor at different times to compare outcome trajectories before and after adoption while accounting for underlying trends.

3.1 Data Collection

3.1.1 The Cursor IDE. Cursor [9] is an AI-powered IDE built as a fork of VS Code with agentic capabilities integrated into the development workflow. Unlike code completion tools that suggest individual lines, Cursor's Agent mode enables autonomous, goal-directed behavior: navigating entire codebases, understanding project architecture across multiple files, making multi-file edits, running terminal commands, executing tests, and iteratively debugging code with minimal human supervision. Developers can use Cursor for feature implementation, refactoring, test generation, documentation, and bug fixing within their native development environment. They are able to choose between frontier models from OpenAI, Anthropic, and Google, either through Cursor's built-in service or through their own API keys.

We study Cursor 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 allows optional configuration files (e.g., `.cursorrules`) that direct the AI agent's behavior [8]. When developers commit these files to version control, we observe a clear, timestamped adoption event in the git history.[1]

3.1.2 Identifying GitHub Projects Adopting Cursor. We identify Cursor-adopting repositories and track adoption dates using configuration files in git history. Using 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 split into 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 [58, 66], we follow prior work [57, 60, 103] by requiring at least 10 stars at collection time—a threshold achieving 97% precision in identifying engineered projects [83]. This yields 807 repositories with adoption dates between January 2024 and March 2025.

As expected, the dataset is highly skewed across many repository-level metrics (Table 1), and adoption time is highly staggered, with adoption growing over time (Figure 1). 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 (358 repositories), Python (123 repositories), JavaScript (57 repositories), Go (35 repositories), and Rust (22 repositories).

3.2 Metrics

For each repository in our sample (both treatment and control), we collect monthly outcome metrics and time-varying covariates from January 2024 to August 2025, providing sufficient observations before and after Cursor adoption. This forms an unbalanced panel dataset with staggered adoptions, common in DiD designs [39, 97].

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.

[1]While scalable, our identification strategy also has limitations that we address through robustness checks detailed in Section 3.5.

Table 1: Descriptive statistics of the 807 repositories using Cursor, collected at the time of data collection (April 2025).

	Mean	Min	25%	Median	75%	Max
Age (days)	915.7	209	284.5	440	1055.5	6134
Stars	1440.7	10	20.0	50	226.5	122280
Forks	215.9	0	3.0	9	36.5	51745
Contributors	19.1	1*	1.0	3	10.0	461
Commits	1801.1	1	47.5	203	896.5	86954
Issues	1073.6	0	3.0	30	230.0	100614
Pull Requests	720.1	0	1.0	17	157.5	72015

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

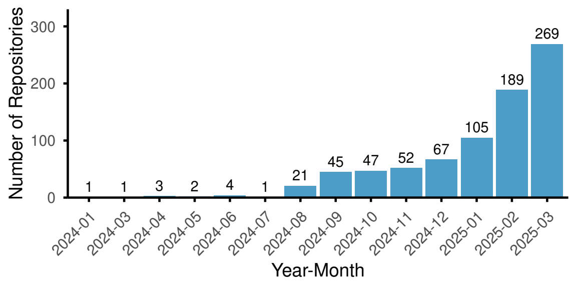

Figure 1: The Cursor adoption time of the 807 repositories in our study, which all have ≥10 stars and Cursor configuration files at the time of data collection (April 2025).

Both metrics have been used as productivity proxies [72, 82, 100], and tend to have moderate-to-strong correlation with managers' perceived productivity [88].

Software quality is multi-faceted and difficult to capture with a single metric [35, 69, 89]. Quality can be pivoted on defect density [53], specification rigor [49], user satisfaction [44], or technical debt [46]. However, many metrics cannot be reliably and universally collected from version control data. In this study, we take the technical debt perspective [78] and test three source code maintainability metrics reasonably estimable from 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. 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 [54]; this metric is best viewed as an estimate of actual issues.
- 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 [80].

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 Causal Inference Strategy

3.3.1 Difference-in-Differences (DiD) Framework. DiD is an established econometric technique for causal inference in observational data [32, 41], with growing adoption in software engineering [33, 48, 85]. The key idea is to compare outcome changes in treated repositories (those adopting Cursor) to changes in control repositories (those not yet adopting) over the same observation periods. This "difference in differences" isolates the tool's effect from other factors affecting all repositories similarly.

DiD relies on the *parallel trends assumption*: absent treatment, treated and control repositories would follow similar outcome trajectories. While untestable directly, we: (1) examine pre-treatment trends to verify parallel trajectories before adoption; (2) control for observable differences through matching and covariate adjustment; (3) take advantage of timing variation in when repositories adopted.

Staggered adoption is particularly valuable [23]. Rather than simultaneous adoption, we observe adoptions across months, providing multiple natural experiments. Repositories adopting in month $t + 1$ or later serve as additional controls for those adopting in month t, since they remain untreated during that period. This pattern also enables estimating how effects evolve over time while flexibly controlling for calendar effects.

Our design supports causal interpretation through: (1) *Temporal ordering*: We observe repositories before and after discrete adoption events, establishing temporal precedence. (2) *Never-treated comparison group*: Repositories never adopting Cursor provide counterfactuals. Matching ensures these controls are similar on observables. (3) *Removal of time-invariant confounders*: By comparing changes within repositories over time, we difference out time-invariant characteristics (team culture, domain, language) that might correlate with both adoption and outcomes. (4) *Control for common trends*: Comparing treated repositories to contemporaneous controls accounts for industry trends, platform changes, and seasonal patterns.

3.3.2 Propensity Score Matching. Repositories adopting AI coding tools likely differ systematically from non-adopters. More active projects, larger communities, or rapidly growing repositories may be more likely to experiment with new tools. If these characteristics affect velocity or quality, simple comparisons would be confounded.

To address selection bias [81], we implement propensity score matching [25, 29]. The propensity score is the predicted probability of adoption conditional on pre-treatment covariate trajectories. This method finds control repositories similarly likely to adopt Cursor, satisfying the conditional independence assumption for the establishment of quasi-experimental settings [96].

We define the population as all GitHub repositories with ≥10 stars at collection time (matching our inclusion threshold). For each month with major adoption (August 2024–March 2025), we collect monthly time series from GHArchive [1] for repositories with at least one event: 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_t denote repository age at month t and X_t denote remaining covariates. We estimate propensity scores via logistic regression:

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

where Γ_i and Θ are parameter vectors. This effectively captures: (1) *Repository maturity* (T_{t-1}): older repositories may have different adoption patterns. (2) *Recent dynamics* ($\sum_{i=1}^{6} \Gamma_i X_{t-i}$): month-by-month evolution over six months captures trends and growth. (3) *Historical baseline* ($\Theta \sum_{j=7}^{\infty} X_{t-j}$): cumulative history provides context on overall project scale and activity.

By including lags, we ensure 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—characteristics correlating with both adoption and outcomes.

Since candidate repositories outnumber adopters by orders of magnitude, we sample at most 10,000 candidates per month to avoid extreme imbalance and improve fit. This yields AUC values of 0.83–0.91, indicating high discriminative power.

For each treated 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, 106]. This yields 1,380 matched controls with similar propensity score distributions (replication package) and pre-adoption trajectories.

3.3.3 Difference-in-Differences Estimation. With our matched sample, we estimate treatment effects using panel data regression models. Specifically, the average treatment effect (ATE) can be estimated from the β parameter in the following model:

$$Y_{it} = \alpha + \beta D_{it} + \mu_i + \lambda_t + X'_{it}\Gamma + \epsilon_{it} \quad (2)$$

where Y_{it} is the outcome for repository i at month t; D_{it} indicates whether i has adopted Cursor by t; μ_i and λ_t—repository and time fixed effects; X'_{it}—time-varying controls; and ϵ_{it}—error term.

Beyond average effects, we estimate *dynamic effects* (event studies) showing how treatment effects evolve:

$$Y_{it} = \alpha + \sum_{j=2}^{J} \beta_j (\text{Lead } j)_{it} + \sum_{k=1}^{K} \gamma_k (\text{Lag } k)_{it} + \mu_i + \lambda_t + X'_{it}\Gamma + \epsilon_{it} \quad (3)$$

where Lead j and Lag k are dummies supporting estimation of effects J periods before and K periods after adoption. Pre-treatment Lead j coefficients serve as *placebo tests*: reliable estimation should show no significant effects before intervention, supporting parallel trends. Lead 1 is omitted as the counterfactual baseline.

It is worth noting that traditional two-way fixed effects (TWFE) estimators for the above models can produce biased estimates with staggered adoption when treatment effects are heterogeneous over time [23, 43, 52]. Violated treatment effect homogeneity leads to "forbidden comparisons" where already-treated units serve as controls for newly-treated units, biasing parameters. To address this, we use the **Borusyak et al. [28] imputation estimator**, designed explicitly for staggered adoption, with this two-step process:

Step 1: Impute counterfactual outcomes. For each repository-month post-adoption, the estimator first predict what the outcome *would have been* without adoption by estimating an outcome regression using only not-yet-treated observations (pre-adoption for treated repositories; all periods for never-treated controls):

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

where $\hat{\mu}_i$, $\hat{\lambda}_t$ represent repository and time fixed effects; X_{it} includes time-varying covariates, and ϵ_{it} is the error term. Critically, this uses only untreated observations, ensuring counterfactual predictions are not contaminated by treatment effects.

Step 2: Compare actual to counterfactual. For each treated repository-month, the estimator computes the ATE (i.e., β in Equation 2) as the mean of individual effects across treated observations:

$$\beta = \frac{1}{N_{\text{treated}}} \sum_{(i,t) \in \text{treated}} (Y_{it} - \hat{Y}_{it}(0)) \quad (5)$$

The post-treatment dynamic effects (i.e., γ_k in Equation 3) are given by averaging post-treatment periods; the pre-treatment dynamic effects (i.e., β_j in Equation 3) are given by an alternative TWFE model with only untreated observations and lead dummies.

3.4 Testing Velocity & Quality Interactions

While DiD estimates treatment effects on individual outcomes, it doesn't 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 suggests 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 such dynamic relationships and bidirectional causality, we employ *generalized method of moments* (GMM) [56], providing consistent estimates when variables are potentially endogenous (correlated with unobserved errors). The key insight uses *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:

$$Y_{it} = \alpha + \rho Y_{i,t-1} + \beta D_{it} + \gamma X_{it} + \mu_i + \lambda_t + Z'_{it}\Theta + \epsilon_{it} \tag{6}$$

where $Y_{i,t-1}$ captures outcome persistence; X_{it} represents potentially endogenous regressors; Z'_{it} are additional time-varying controls; remaining terms follow Equation 2. During estimation, historical values of X_{it} (e.g., $X_{i,t-2}$) serve as instrumental variables.

Specifically, we test these 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 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. Our identification strategy has important limitations that we address through robustness checks.

Observable adoption through committed configuration files. We observe only repositories committing Cursor configuration files to version control. Developers can use Cursor without committing such files, so our sample represents repositories with observable, committed adoption rather than all users. This creates potential selection: repositories committing configuration may be more committed to systematic integration, have more formal processes, or differ in unobservable ways. If selection exists, our estimates reflect treatment effects for repositories with observable, formal adoption. 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 (more likely to carefully review AI-generated code), estimates could be conservative. We view our sample as capturing repositories where adoption represents deliberate, visible practice change—precisely the population where long-term effects are most policy-relevant.

Uncertainty about usage intensity and persistence. Even observing committed configuration files, we don't know how intensively or persistently the tool was used. Repositories have multiple contributors, each with their own environment—committing `.cursorrules` indicates *someone* experimented, not that *all* contributors used it continuously throughout post-adoption observation. Contributors might 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. Our main estimates therefore represent intent-to-treat (ITT) effects: impact of adopting Cursor as measured by committing configuration, averaging over heterogeneous usage patterns.

To test whether findings are driven by repositories with genuine, sustained usage versus minimal engagement, we conduct two heterogeneity analyses splitting by usage intensity confidence:

(1) Continued engagement: We identify repositories where contributors continued modifying `.cursorrules` files post-adoption. For each post-adoption month, we classify as high-confidence if any configuration file was modified, low-confidence otherwise. Repositories iteratively refining rules month-by-month likely represent more sustained, systematic integration than those committing once and never touching files again.

(2) Adoption breadth: For each repository, we identify contributors modifying Cursor files (indicating experimentation) and calculate their fraction of total repository commits during observation. We split into high-adoption repositories (Cursor users contributed ≥80% of commits, suggesting tool use for most work) versus lower-adoption (Cursor users responsible for lower activity fraction).

Section 4 shows findings are robust and amplified in subsamples with higher sustained usage confidence. Negative quality effects are 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 main ITT estimates likely understate effects among repositories with intensive, sustained usage.

Model and version heterogeneity. Our dataset lacks information about which Cursor version or LLM backend individual repositories used, but this doesn't compromise validity. Our research question focuses on system-level effects of adopting agentic AI coding tools as integrated development practice, not effects of particular model architectures. What matters for causal identification is the discrete adoption event—when repositories begin using Cursor's agentic capabilities—not specific model versions invoked in each session. Architectural characteristics distinguishing agentic tools from earlier generations (persistent codebase awareness, autonomous multi-file operations, iterative debugging) are consistent across models available during our observation period. Moreover, to the extent different repositories use different model backends or developers switch models for different tasks, this heterogeneity increases external validity: our estimates reflect average treatment effects of adopting Cursor as actually used in practice, with all model diversity, rather than effects of single controlled LLM configurations. Any systematic quality differences between models would bias estimates toward null if better models are more commonly used (reducing observed quality degradation), making significant quality decline findings conservative.

Confounding from concomitant AI coding tools. Repositories may use multiple AI coding tools simultaneously. We identify concurrent use of other tools, namely GitHub Copilot (345), Claude (63), Windsurf (37), Cline (13), and OpenHands (2), and conduct robustness checks. Section 4 shows main findings are robust: no significant differences in effect magnitude or significance between solo Cursor users and potential multi-tool users.

Imperfect matching. While propensity score matching achieved strong performance (AUC 0.83–0.91), it remains subject to untestable unobserved confounders. Developer expertise, team practices, project complexity, or organizational culture may affect both adoption and outcomes. Although we include numerous covariates hoping to latently control many factors, perfect matching is generally impossible [25], leaving residual systematic differences. Our DiD design addresses this by incorporating time-invariant unobserved confounders through repository fixed effects (μ_i in Eq. 2, 3), controlling

Table 2: The Borusyak et al. [28] estimated average treatment effects (i.e., β in Eq 2) post Cursor adoption. All outcome variables are log-transformed to facilitate easy comparison of treatment effects across outcome variables (i.e., all treatment effect βs can be interpreted as a percentage change of $100(e^{\beta}-1)\%$ on the outcome variable after Cursor adoption).

Outcome	Estimate	Std. Error
Commits	0.0336	(0.0425)
Lines Added	0.2512^{*}	(0.1070)
Static Analysis Warnings	0.2598^{***}	(0.0509)
Duplicate Line Density	0.0782	(0.0426)
Code Complexity	0.3415^{***}	(0.0526)

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

broader temporal trends through time fixed effects (λ_t), and controlling remaining time-varying confounders using covariates.

Contamination in never-treated controls. Even if DiD leads to unbiased estimates regarding Cursor adoption impact in studied repositories, interpreting specific results remains challenging. Control groups are likely contaminated with LLM-based tools, especially earlier ones like GitHub Copilot and ChatGPT [2], so estimates would be smaller than true LLM agent assistant impact compared to using no LLM at all. Because of this contamination, estimating true impact is likely impossible in observational data; instead, our study provides evidence regarding LLM agent assistant adoption impact with respect to current state-of-the-practice. We discuss interpretation challenges further in Section 5.

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 investigations and continuous monitoring of state-of-the-art LLM coding tools as they roll out.

4 Results

Our main results (Borusyak et al. [28] estimator) are summarized in Table 2 and Figure 2 (Row 1; recall that variables are log-transformed and interpreting the coefficients requires exponentiation). See replication package for TWFE [41] estimations and Callaway and Sant'Anna [30] estimations, included as an additional robustness check; the latter is a modern, robust estimator similar to Borusyak et al. [28], but less statistically powerful (among other drawbacks) with relatively small samples like ours. Our main conclusions (directions of effects) are consistent.

4.1 Development Velocity

On average, Cursor adoption demonstrates 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 commit volume.

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. Contemporaneous velocity V_{it} instrumented with lags 2-3 to address endogeneity. Sargan p>0.05 indicates valid instruments. AR(1) p<0.05 is expected with first-difference transformation. AR(2) p>0.05 indicates no serial correlation.

The dynamic treatment effect estimations (Figure 2, Row 1) reveal important temporal patterns that explain these differences: *the only significant development velocity gain is in the first two months post Cursor adoption.* Specifically, we 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. Overall, the dynamic effect estimation is highly consistent across all three estimators (see replication package).

Finding 1: The DiD models suggest that the adoption of Cursor only leads to a significant and large velocity gain in the short term (i.e., first two months) in open-source projects.

4.2 Software Quality

In contrast to the transient velocity gains, Cursor adoption shows more sustained patterns across static analysis warnings and code complexity, with evidence of sustained technical debt accumulation. On average (Table 2), static analysis warnings increase significantly by 29.7% and code complexity also rises significantly by 40.7%. The effect on duplicate line density is insignificant across all three estimators. The dynamic treatment effect estimations (Figure 2, Row 1) reveal that code quality degradation, unlike development velocity gains, persists beyond the initial adoption period.

Finding 2: The DiD models suggest that the adoption of Cursor leads to a sustained accumulation of static analysis warnings and a sustained increase in code complexity.

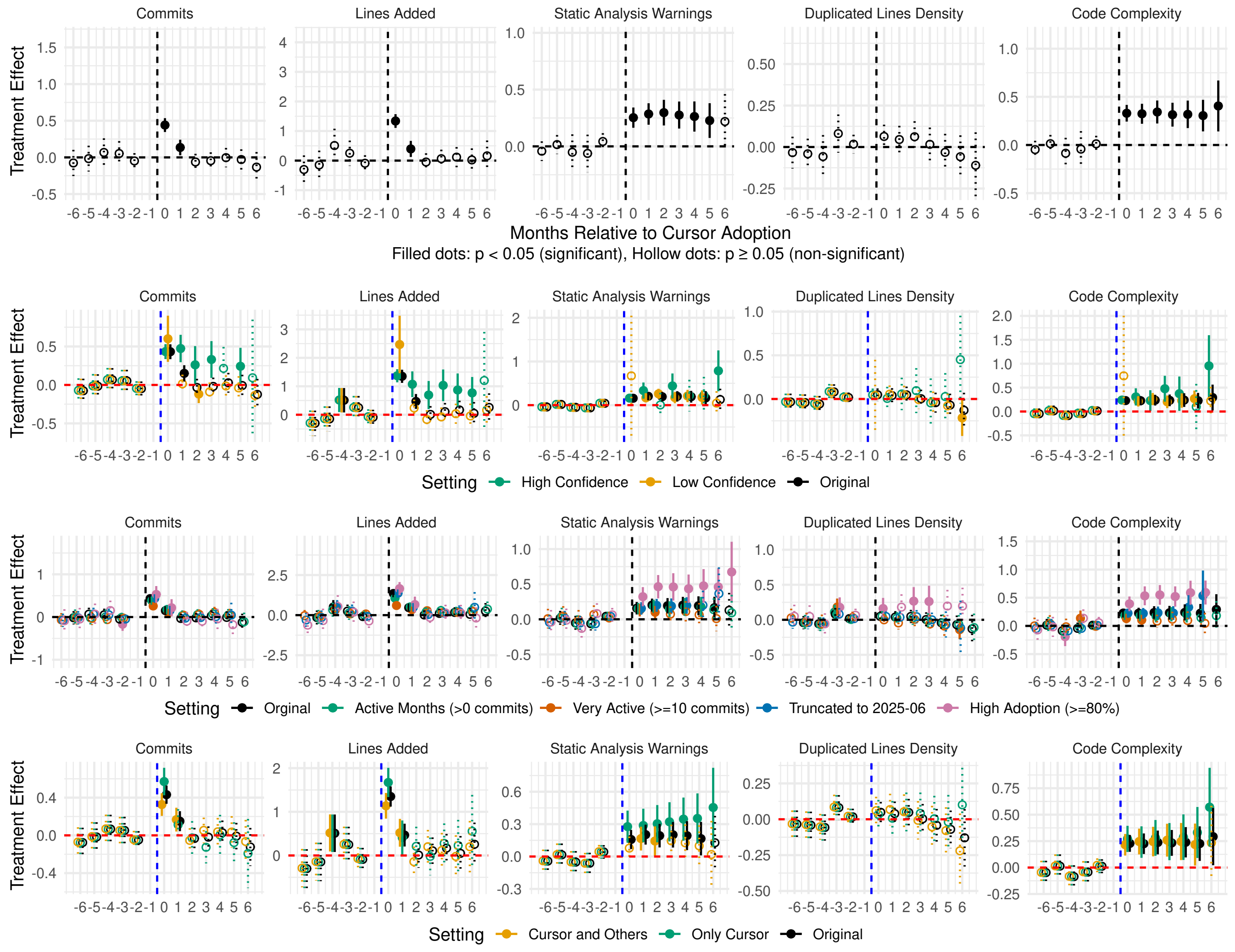

Figure 2: *Row 1*: The estimated dynamic treatment effects -6 to +6 months before or after adoption. All outcome variables are log-transformed same as the estimated average treatment effects in Table 2. Note that -1 month was deliberately removed from the models to serve as a counterfactual baseline (for TWFE; regression results in replication package) or avoid potential anticipation effects (for Borusyak—Table 2 and Callaway)—replication package. *Row 2:* Robustness check (Section 3.5), repositories with high-confidence Cursor usage show stronger effects. *Row 3:* Robustness check (Section 3.5), repositories where Cursor-file-tinkerers account for the majority of commit activity show slightly stronger effects. *Row 4:* Robustness check (Section 3.5), repositories using Cursor show similar effects as those using Cursor and other agentic AI tools.

4.3 Velocity & Quality Interactions

To distangle the temporal interactions between velocity and quality, we fit dynamic panel GMM models to test causal paths specified in Equation 7 (Table 3). All models pass the Sargan, AR(1), and AR(2) tests, forming a solid ground for causal interpretation [22].

The first two models show that, on average, 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 9.0% baseline increase on average compared to projects in similar dynamics but not using Cursor.

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 4.94x increase in static analysis warnings or a 3.28x increase in code complexity, according to the dynamic panel GMM estimations.

Finding 3: The dynamic panel GMM models suggest that: (1) the adoption of Cursor leads to an inherently more complex codebase; (2) the accumulation of static analysis warnings and code complexity decreases development velocity in the future.

4.4 Pre-trend and Robustness Checks

For pre-trend (placebo) tests, we use heteroscedasticity- and cluster-robust Wald tests [28] to test the joint null hypothesis that $\beta_j = 0$ for $j \in 2...6$ in Equation 3. Most models pass the pretrend test at the 0.05 level except the code complexity model estimated with TWFE: This violation likely stems from "forbidden comparisons" inherent to the TWFE estimator [43], as Borusyak and Callaway show no significant pretrends for this outcome.

For robustness checks, we consider several conditions discussed in Section 3.5, with results visualized across Rows 2–4 in Figure 2. First, slicing the sample by high- vs low-confidence of continued Cursor use (Row 2), we observe that the effects are stronger for the high-confidence slice, suggesting that indeed it is the use of Cursor that causes the velocity and code quality changes we record.

Second, we slice the sample by the activity levels of the repositories, of the developers responsible for commits to the Cursor files, and other factors: (1) removing inactive repository months without any commits; (2) retaining only extremely active repository months with at least 10 commits; (3) retaining only heavy Cursor adoptors for the treatment group, in which developers modifying the Cursor rule files contributed at least 80% of commits inside the repository; (4) removing months after June 2025, since which other LLM coding agents, notably OpenAI Codex [6] and Claude Code [4], have been rapidly rising. (5) comparing repositories with evidence of Cursor-only use to those with traces of other AI coding assistants. Each robustness check addresses a specific validity concern: (1) ensures results are not driven by developers selectively becoming inactive post-adoption; (2) confirms effects persist even during periods of high activity when measurement is most precise; (3) demonstrates a dose-response relationship where treatment intensity predicts effect magnitude; (4) and (5) rule out contamination from competing AI coding assistants. These results, available across Rows 3–4 in Figure 2, do not deviate much from our original setting except that some treatment effects are stronger for heavy Cursor adopters.

In general, these checks provide reassurance about our causal interpretation against potential selection bias, confounding from rival tools, or problematic treatment indicators (i.e., if developers modifying Cursor config files were not genuinely using Cursor, we would not observe systematically stronger effects in repositories where these developers contribute a larger share of commits).

5 Discussion

5.1 Theoretical Implications

Our study contributes to the rapidly growing literature regarding the impact of AI assistance on developer productivity [26, 40, 45, 61, 64, 71, 90, 93, 102, 105, 107, 110, 112]. 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, 70, 84, 104]. In this section, we connect our findings with prior research and discuss our theory around how LLM agent assistants may impact software development (Figure 3).

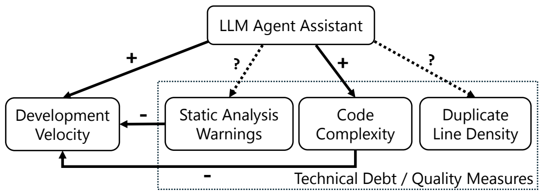

Figure 3: Our theory around how LLM agent assistants may impact software development. Solid lines show causal relationships supported by existing evidence, and dashed lines indicate relationships not fully supported in our data.

5.1.1 *The Transient Velocity Gains and Possible Causes Behind It.*

Our first longitudinal finding—that velocity gains out of AI adoption are concentrated in the initial one or two months before returning to a baseline level—diverges from the productivity improvements reported in controlled experiments [e.g., 91, 93]. One reason for such divergence 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 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 both supported in 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 3.28x increase in code complexity or a 4.94x 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—albeit not directly testable from our data—is that open-source developers may experience an excitement-frustration-abandonment cycle while they adopt LLM agent assistants. 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 (recall, not even touching config files guarantees intense usage). This interpretation aligns with the emerging qualitative research documenting developer challenges with AI-assisted coding [26, 36, 74] and anecdotal evidence from Cursor users [e.g., 10, 12].

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 [7]. 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 under the same velocity. This proportional relationship has important practical implications (see Section 5.2)

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 LLM agents may be inherently more complex than human-written code. This effectively creates a "complexity debt" in AI-heavy projects, which may amplify frustration and maintenance costs when LLM fails later on more complex codebases, possibly forming another mechanism explaining the transient velocity gain after Cursor adoption.

While the adoption of Cursor leads to no significant changes in duplicate line density in the entire study sample (Table 2, Figure 2), heavy Cursor adopters may exhibit modest increases (Figure 2, Row 3). Future research is necessary to gather evidence around 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 frustration-abandonment pathway while potentially dampening the quality feedback loop. In enterprise settings, organizational mandates, sunk training costs, and managerial oversight might sustain AI tool usage despite frustration, potentially leading to different temporal patterns (as shown in a recent study [71]). Similarly, enterprise quality assurance processes—mandatory code review, automated testing requirements, dedicated QA teams—might prevent proportional technical debt accumulation by catching issues before they accumulate. 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 agents should focus on process adaptation that scales quality assurance with velocity. The proportional technical debt accumulation we observe (Finding 2), combined with its velocity-dampening effects (Finding 3), creates a self-reinforcing cycle that needs to be addressed at a 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 journey toward an unmaintainable codebase.

To support the above process adaptation, AI coding tools need explicit design to support quality assurance alongside code generation. Current LLM agents are generation-first, leaving quality maintenance as an afterthought. Next-generation assistants should suggest tests alongside code, flag unnecessary complexities 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 tool incentives 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% code complexity increase we observe (Table 3) represent 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 [55, 111]. 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, 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 807 Cursor-adopting repositories with 1,380 matched controls, 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 during 2024-2025. LLM capabilities, agent architectures, and developer practices are improving at unprecedented rates—future tools will likely 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 multidimensional nature of code quality in AI-augmented 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 the abandonment dynamics and quality concerns compared to enterprise settings with mandatory and 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, including all the datasets and scripts to replicate findings presented in the study, at:

https://github.com/hehao98/CursorStudy

Acknowledgments

He's and Kästner's work was supported in part by the National Science Foundation (award 2206859). Miller's work was 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 (award 2317168) and research awards from Google and the Digital Infrastructure Fund. We would also like to thank Narayan Ramasubbu for providing valuable methodological feedback at the early stage of this research, and 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. *Codex | OpenAI.* Retrieved Sep 24, 2025 from https://openai.com/codex/
[7] 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
[8] 2025. *Cursor - Rules.* Retrieved Apr 22, 2025 from https://docs.cursor.com/context/rules/
[9] 2025. *Cursor - The AI Code Editor.* Retrieved Apr 21, 2025 from https://www.cursor.com/
[10] 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
[11] 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/
[12] 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
[13] 2025. *OpenHands — the leading open source AI coding agent.* Retrieved Oct 21, 2025 from https://openhands.dev
[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] DAvD 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] Damian Clarke and Kathya Tapia-Schythe. 2021. Implementing the panel event study. *The Stata Journal* 21, 4 (2021), 853–884.
[40] 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).
[41] Scott Cunningham. 2021. *Causal Inference: The Mixtape.* Yale University Press.
[42] 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.

[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] 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.
[51] 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).
[52] Andrew Goodman-Bacon. 2021. Difference-in-differences with variation in treatment timing. *J. Econom.* 225, 2 (2021), 254–277.
[53] Robert B. Grady. 1993. Practical Results from Measuring Software Quality. *CACM* 36, 11 (1993), 62–68.
[54] 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.
[55] Srishti Gureja, Elena Tommasone, Jingyi He, Sara Hooker, Matthias Gallé, and Marzieh Fadaee. 2025. Verification Limits Code LLM Training. *CoRR* abs/2509.20837 (2025).
[56] Lars Peter Hansen. 1982. Large sample properties of generalized method of moments estimators. *Econometrica* (1982), 1029–1054.
[57] 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.
[58] 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).
[59] 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.
[60] 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.
[61] Manuel Hoffmann, Sam Boysel, Frank Nagle, Sida Peng, and Kevin Xu. 2024. *Generative AI and the Nature of Work.* Technical Report. CESifo Working Paper.
[62] 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.
[63] 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.
[64] Saki Imai. 2022. Is GitHub Copilot a Substitute for Human Pair-programming? An Empirical Study. In *ICSE*. ACM, 319–321.
[65] 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).
[66] 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.
[67] 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).
[68] Raphaël Khoury, Anderson R. Avila, Jacob Brunelle, and Baba Mamadou Camara. 2023. How Secure is Code Generated by ChatGPT?. In *SMC*. IEEE, 2445–2451.
[69] Barbara A. Kitchenham and Shari Lawrence Pfleeger. 1996. Software Quality: The Elusive Target. *IEEE Softw.* 13, 1 (1996), 12–21.
[70] Philippe Kruchten, Robert L Nord, and Ipek Ozkaya. 2012. Technical debt: From metaphor to theory and practice. *IEEE Softw.* 29, 6 (2012), 18–21.
[71] 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).
[72] M. J. Lawrence. 1981. Programming methodology, organizational environment, and programming productivity. *JSS* 2, 3 (1981), 257–269.
[73] 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.
[74] 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.
[75] 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).
[76] 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.
[77] 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.
[78] Robert C Martin. 2009. *Clean Code: A Handbook of Agile Software Craftsmanship.* Pearson Education.
[79] Boris Martinović and Robert Rozić. 2024. Impact of AI tools on software development code quality. In *ICDTEAI*. Springer, 241–256.
[80] Thomas J. McCabe. 1976. A Complexity Measure. *TSE* 2, 4 (1976), 308–320.
[81] Douglas L Miller. 2023. An introductory guide to event study models. *JEP* 37, 2 (2023), 203–230.
[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] Paul R Rosenbaum and Donald B Rubin. 1983. The central role of the propensity score in observational studies for causal effects. *Biometrika* 70, 1 (1983), 41–55.
[97] Jonathan Roth, Pedro HC Sant'Anna, Alyssa Bilinski, and John Poe. 2023. What's trending in difference-in-differences? A synthesis of the recent econometrics literature. *J. Econom.* 235, 2 (2023), 2218–2244.
[98] 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.

[99] 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.

[100] 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.

[101] 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.

[102] 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).

[103] César Soto-Valero, Thomas Durieux, and Benoit Baudry. 2021. A longitudinal analysis of bloated Java dependencies. In *FSE*. ACM, 1021–1031.

[104] Margaret-Anne Storey, Brian Houck, and Thomas Zimmermann. 2022. How developers and managers define and trade productivity for quality. In *CHASE*. 26–35.

[105] 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).

[106] 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).

[107] 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.

[108] 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.

[109] 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).

[110] 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.

[111] 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).

[112] 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.