The Impact of AI Coding Assistants on Code Quality in Open-Source Software: An Empirical Mining Study

Lukas Westholt
Leipzig University
Leipzig, Germany
ed30edel@studserv.uni-leipzig.de

Abstract

AI-powered coding assistants such as GitHub Copilot, Cursor, and Claude Code have been widely adopted in open-source software development since 2022. Prior work has documented productivity gains and correctness risks in controlled settings; recent longitudinal studies address agentic tools, but repository-level, language-stratified evidence across mainstream assistants remains sparse.

We present an empirical mining study comparing 272 open-source repositories that detectably adopted AI coding assistance (113 Python, 159 TypeScript) against 299 control repositories satisfying the same selection criteria. Adoption is identified through a tiered signal hierarchy; this produces an individual treatment date per repository, enabling a staggered interrupted time series design with four per-repository snapshots (T_{-6}, T_{-1}, T_{+1}, T_{+6}) and a difference-in-differences estimator against the control group to separate adoption effects from secular trends in software engineering practice.

Fifteen static quality metrics are computed per snapshot, including average cyclomatic complexity, bug-fix (commit) ratio, lint warning density, and test density. For the research questions we detect *no* adoption effect: the difference-in-differences is -0.04 (95% confidence interval $[-0.13, 0.04]$) for average cyclomatic complexity and 0.03 ($[-0.02, 0.09]$) for the bug-fix ratio, both spanning zero, with no variation by language or activity. Exploratory analyses reveal robust but practically modest associations (every Cliff's δ negligible-to-small, largest 0.27): adopting repositories increase test density, test count, and (in Python) docstring coverage, while reducing parameters per function. A concurrent rise in *worst-case* cyclomatic complexity appears to track codebase growth rather than genuine complexity growth, as the normalized share of complex functions is unchanged. These associations are largely stable across signal-tier and shifted-treatment checks, but we read them as hypothesis-generating: for several, the arms were already diverging before adoption.

1 Introduction

AI-powered coding assistants have changed how developers write code. GitHub Copilot, built on a large language model fine-tuned on public code [9], drew more than 1.2 million developers during its technical preview and became generally available in June 2022 [12]. Tools such as Cursor and Claude Code have since brought similar AI assistance to everyday coding workflows. By generating completions and full function implementations from natural language descriptions, these tools reduce the time spent on routine coding tasks.

Early evaluations focused primarily on productivity and correctness in controlled settings. Peng et al. [21] found that in a controlled experiment Copilot users completed a programming task 55.8% faster than a control group. However, faster code generation does not necessarily translate into higher-quality code. Prior work has documented that AI-generated suggestions frequently contain security vulnerabilities [20] and that output quality degrades on complex algorithmic tasks [19]. These findings raise an empirical question about how observable code quality indicators are changing in practice as AI assistance becomes widespread in open-source development.

Open-source repositories on GitHub are well suited to this analysis. Their full commit histories span the pre- and post-adoption period, capturing real developer behavior on non-trivial software systems. Many repositories also leave detectable traces of AI tool adoption (tool-specific configuration files, AI-attributed commit co-authorship, or explicit references in commit messages), which allow the adoption event to be localized at the project level instead of being assumed from a global tool release date.

This paper presents an empirical mining study designed to answer the following research questions:

- **RQ1:** How did the average cyclomatic complexity of functions change in active open-source software (OSS) repositories after the first detectable adoption of an AI coding assistant?
- **RQ2:** How did the bug-fix commit ratio change before versus after AI coding assistant adoption?
- **RQ3:** How do observed metric differences vary by primary programming language or repository activity level?

Our contributions are: (i) a tiered signal hierarchy for detecting AI coding assistant adoption from publicly available repository metadata; (ii) a curated dataset of 272 treated and 299 control repositories with individually assigned treatment dates and per-snapshot static quality metrics; and (iii) a staggered interrupted time series (ITS) analysis with a difference-in-differences (DiD) estimator that detects no adoption effect on the research questions, alongside exploratory evidence that adoption is associated with more testing and documentation.

2 Background and Related Work

Three strands of prior work bear on this study: AI coding assistants, the static metrics that operationalize code quality, and the empirical studies that have begun to measure AI's effect on it.

2.1 AI Coding Assistants

Since the June 2022 release of GitHub Copilot [12], competing tools such as Amazon Q Developer, Cursor, and Gemini Code Assist have reached broad availability, and AI-assisted code generation is now

common across commercial and open-source projects. The field has moved from *assisted* development (autocomplete, linters, static analyzers) to *generative* development, in which a model authors substantial blocks of code from surrounding context or natural language prompts.

2.2 Code Quality Metrics in Open-Source Software

Empirical software engineering operationalizes code quality through static analysis metrics. McCabe's cyclomatic complexity (CC) [16] is a widely accepted proxy for maintainability and testability, and complexity metrics such as CC are standard static-code inputs to defect-prediction models [17]. Cross-language tools such as Lizard [26] enable consistent measurement across Python and TypeScript; language-specific alternatives such as radon or eslint-complexity would break this comparability.

At the commit level, the fraction of commits addressing bug fixes, identified through keyword matching in commit messages [18], is a standard proxy for corrective (defect-fixing) activity. Changes in test density (ratio of test-file to total lines of code, LOC) and lint warning density (static-analysis warnings per thousand LOC, or kLOC) provide complementary indicators of changing quality and coding-practice standards.

2.3 Empirical Studies on the Impact of AI Coding Assistants

Prior work on the effects of AI coding tools falls into three categories. *Productivity studies* [21] focus on output speed and throughput rather than artifact quality. *Correctness and security evaluations* [19, 20] assess the quality of AI-generated code in isolation, identifying CWE violations and correctness gaps in suggested snippets; these studies examine AI output directly, not its downstream effect on a codebase after developer review and integration. *Repository mining work* has characterized developer practices and discourse around AI coding tools [27] (the languages, IDEs, and perceived benefits and limitations) but draws on self-reported developer experience rather than quality indicators measured directly in production codebases at scale. Recent large-scale longitudinal work has begun to measure downstream quality outcomes, primarily for Cursor and autonomous coding agents [1, 14], finding lasting increases in code complexity and static-analysis warnings, while productivity gains proved short-lived.

As in recent adoption-based studies [1, 14], we detect adoption per repository rather than anchoring to a global release date, combining a staggered ITS [15] with a DiD estimator [8] against a control group; our contribution is to extend this staggered-adoption design from a single tool to a tiered signal hierarchy spanning many assistants and both languages, with the bug-fix (commit) ratio added as an outcome, a combination that prior mining work has not examined together.

3 Methodology

We answered the research questions with a per-repository design: we detected each repository's own adoption date, took four snapshots around it, and estimated difference-in-differences against a control arm over fifteen static metrics.

3.1 Study Design

We employed a quasi-experimental longitudinal design based on staggered ITS [15]. The observation window spanned the June 2022 general availability of GitHub Copilot [12] through December 2025, covering the pre- and post-adoption history of each repository. We extended the window past the originally planned 2024 cutoff because most strong adoption signals fall in 2024–2025; signals in 2026 were excluded because they lack the 180 days of post-history the design requires, which is the binding constraint on the size of the treated arm. We detected an individual treatment date per repository (Section 3.2). For each treated repository, we computed four snapshots: T_{-6} (last commit at least 180 days before treatment), T_{-1} (last commit before treatment), T_{+1} (first commit on or after treatment), and T_{+6} (first commit at least 180 days after treatment). The primary paired contrast is $T_{-6} \to T_{+6}$, which captures the accumulated effect six months on either side of adoption; $T_{-1} \to T_{+1}$ is a secondary, immediate contrast. Missing snapshots were recorded as `NaN`; the repository remained in the sample.

A control group of 299 repositories without detectable AI adoption signals enabled a DiD estimator [2, 7, 8]: DiD = Δ(treated: post−pre) − Δ(control: post − pre). The per-language median treatment date across treated repositories was the synthetic cutoff for control repositories. The 2×2 DiD already differences out any *time-invariant* baseline gap between the arms, including the age and popularity imbalance of Section 4.2. As a robustness check we also fitted a covariate-adjusted regression DiD (adding repository age, log-stars, contributor count, and language, with repository-clustered standard errors); because these covariates are time-invariant, the adjusted DiD coincided with the 2×2 estimate, serving only as a consistency check. Neither estimator addresses *time-varying* confounding or substitutes for propensity score matching, which the coarse covariates do not support. Throughout, we treated a metric as showing a robust treated-vs-control effect only when its DiD 95% confidence interval (CI) excluded zero. As a check on the DiD identifying assumption, we computed a placebo DiD on the pre-treatment interval $T_{-6} \to T_{-1}$; under parallel trends this should be near zero, and a 95% CI excluding zero flags pre-adoption divergence.

3.2 AI Adoption Signal Detection

We operationalized AI tool adoption through a tiered signal hierarchy applied to each repository's commit history and file tree. S1 (strong) is one of 39 tool-specific marker files spanning 17 assistants (e.g., `CLAUDE.md`, Copilot's `.github/copilot-instructions.md`, or `AGENTS.md`); S2 (medium) is an AI co-authorship commit trailer from any of seven tools (`Co-Authored-By` attribution to Claude, Copilot, Aider, and others); S3 (weak) is a commit-message keyword. The treatment date is the earliest S1 or S2 signal, taken from the commit author date (which rebasing does not rewrite); a repository is excluded if that earliest signal falls outside the observation window or coincides with the creation date (no pre-adoption baseline). We applied an *asymmetric* S3 policy: an S3 keyword match never defines a treatment date in the main specification, but the control-contamination screen does use S1, S2, and S3, so any repository showing even a weak AI hint is removed from the control arm. S3 keywords were matched whole-word and case-insensitively; the

Table 1: Participant flow from 19,174 discovered candidates to the 272 treated + 299 control analyzed sample. Size > 400 MB is the only measurement-stage exclusion.

Stage	*n*
Discovered candidates	19,174
excluded: below min stars	14,611
excluded: language	1,425
excluded: created after cutoff	999
excluded: archived	163
excluded: not found	22
Included (eligibility pool)	1,954
not put through detection (by design)	433
Signal-detected	1,521
Treated (census of adopters): S1/S2 signal	330
excluded: size > 400 MB	58
analyzed	**272**
S3-only (sensitivity arm, not headline)	10
Control (capped sample): no valid signal	1,181
excluded: size > 400 MB	40
not sampled (control cap, target 300)	841
lost: pipeline failure	1
analyzed	**299**

authoritative set is three patterns: `copilot`; `ai-generated` (hyphen or space); and `generated by`/`with` followed by `ai`, `claude`, `copilot`, or `cursor`. S3-only repositories appear only in a dedicated S3 sensitivity analysis; we also reported an S1-only subset alongside the main S1+S2 sample. Because inline completion (e.g., GitHub Copilot autocomplete) leaves neither a configuration marker nor a commit trailer, and the trailers we detected come mainly from agentic assistants and bots [3, 4, 13], our signals indicate agent-mediated use, not in-editor completion. We estimated effects on the pooled treated arm and reported the tool mix descriptively instead of attributing DiD estimates to any single tool.

3.3 Repository Selection

We seeded candidates from marker- and trailer-based code search, star-bucketed repository search, and a manual seed list, then expanded by snowball sampling [5] over dependency (pip), fork, and contributor links. All candidates were filtered against the following criteria.

Inclusion: created before January 1, 2024; at least 10 GitHub stars; primary language Python or TypeScript (which excludes documentation- and data-only repositories); not archived; and, for treated repositories, at least one S1 or S2 adoption signal inside the observation window.

Exclusion: repositories where the AI signal date equals the creation date (no pre-signal baseline available).

Deviations from the planned protocol: The planned design required creation before 2020 and at least 20 stars, with activity, contributor-count, and 1,000-LOC gates; because repositories carrying an AI marker are young and often small, these criteria left too few eligible repositories, so we relaxed them to the thresholds above. We further capped measurement at repositories of at most 400 MB to bound clone and analysis cost; larger repositories still remain classified as treated or control (Table 1) but are excluded from the analyzed sample.

3.4 Metrics

We computed fifteen static quality metrics per snapshot, spanning complexity and size, documentation, typing and duplication, testing, and process; lower is better for the complexity, size, duplication, and process metrics, higher for documentation, typing, and testing. Full definitions are in the companion repository. Every tool ran at a pinned version against fixed, project-independent configurations, so a repository's own config cannot influence a metric; those in the pinned container image additionally ran with the repository mounted read-only and networking disabled. Complexity and size came from Lizard [26], lines of code from `cloc` [11], duplication from `jscpd` (strict mode, 50-token minimum), typed-parameter and docstring coverage from the Python abstract syntax tree, churn from `git diff`, and lint from `ruff` (Python) and ESLint (TypeScript), each with a fixed ruleset. A commit counts as a bug fix if its message contains a whole-word `fix`, `bug`, `patch`, or `resolve` [18]; the generic `issue` and `error` count only with a fixing verb (e.g., `fixed issue`), suppressing false positives like "open an issue". We validated this classifier on a stratified random sample of 200 commits, double-labeled by two raters who agreed on 179 (the 21 disagreements adjudicated by discussion; Cohen's $\kappa = 0.74$): its precision is 0.57, recall 0.88 (re-weighted from the 50/50 validation strata to the cohort's true bug-fix base rate), and F1 0.69. We discuss what this modest precision implies for the bug-fix ratio in Section 5. Test files were identified by language-specific naming conventions (patterns in the companion repository). The bug-fix ratio and code churn are window metrics, computed over the pre-window ($T_{-6}\ldots T_{-1}$) and post-window ($T_{+1}\ldots T_{+6}$); all others are point-in-time at each snapshot.

3.5 Statistical Analysis

For each metric we compared pre- and post-adoption distributions using the Wilcoxon signed-rank test [25] (paired by repository, two-tailed, $\alpha = 0.05$) and reported effect sizes with Cliff's δ [10], its magnitude labeled by the Romano thresholds [22] (negligible < 0.147, small < 0.33, medium < 0.474). The Benjamini-Hochberg procedure [6] controlled the false discovery rate (FDR) across the metric family within each contrast and stratum. Pairs with a missing snapshot on either side were dropped (pairwise-complete deletion), so the reported n varied by metric. For RQ3, we stratified the analysis by primary language and by activity level (upper vs. lower quartile by commit frequency) and compared strata using the Mann-Whitney U test. Bootstrapped 95% confidence intervals (10,000 resamples, seed 42) were reported for all effect size estimates and for the 2×2 DiD. We ran no a priori power analysis; instead we reported a confidence interval for every effect (around 200 paired observations for cross-language metrics, fewer for Python-only and lint), so the reader can tell tightly bounded nulls from imprecise ones. Because lint warning rates are not comparable across Python and TypeScript toolchains, the lint metric was analyzed within language strata only.

4 Results

We report the cohort and its baseline imbalances, then answer each research question, and close with the exploratory findings and sensitivity analyses.

Table 2: Per-metric results, primary contrast $T_{-6} \rightarrow T_{+6}$ (treated pre vs. post). n is the paired-repository count per metric; Pre/Post are treated medians; Cliff's δ is the *within-treated* pre-vs-post effect size with its bootstrap 95% CI; p_{adj} is Benjamini-Hochberg adjusted; DiD is the 2×2 difference-in-differences of per-repository means vs. control with its 95% CI. A metric shows a robust treated-vs-control effect only when its DiD CI excludes zero (rows in bold). †Churn is on a far larger scale than the other metrics; read that row qualitatively, not as a precise bound.

Metric	n	Pre	Post	Cliff's δ [95% CI]	p_{adj}	DiD [95% CI]
Avg. cyclomatic complexity (RQ1)	231	1.95	1.97	0.02 [-0.02, 0.06]	0.005	-0.04 [-0.13, 0.04]
Max cyclomatic complexity	231	34.00	46.00	0.20 [0.15, 0.25]	<0.001	8.47 [4.22, 12.58]
Complex functions (CC>10) share	231	0.01	0.01	0.06 [0.02, 0.11]	0.005	0.00 [-0.00, 0.00]
Function length p95 (NLOC)	231	33.60	34.00	0.06 [0.01, 0.10]	<0.001	0.19 [-1.49, 2.02]
Avg. params / function	231	0.81	0.74	-0.06 [-0.10, -0.04]	<0.001	-0.08 [-0.14, -0.04]
Comment density (comments/code)	200	0.13	0.14	0.08 [0.03, 0.13]	0.447	0.01 [-0.01, 0.03]
Code duplication ratio	232	0.10	0.10	0.03 [-0.02, 0.08]	0.533	0.00 [-0.01, 0.01]
Typed-param ratio (Py)	93	0.57	0.61	0.10 [0.04, 0.16]	<0.001	0.02 [-0.03, 0.06]
Docstring coverage (Py)	93	0.22	0.33	0.27 [0.18, 0.37]	<0.001	0.09 [0.04, 0.13]
Lint warnings / kLOC (Py)	91	7.08	5.11	-0.13 [-0.23, -0.04]	0.001	-4.78 [-13.01, 1.23]
Test density	200	0.17	0.28	0.17 [0.12, 0.22]	<0.001	0.05 [0.03, 0.07]
Test-file ratio	237	0.09	0.14	0.08 [0.03, 0.13]	<0.001	0.02 [-0.00, 0.03]
Test count / kLOC	200	6.82	10.95	0.17 [0.12, 0.23]	<0.001	3.07 [1.69, 4.50]
Bug-fix commit ratio (RQ2)	242	0.28	0.30	0.10 [0.05, 0.16]	0.002	0.03 [-0.02, 0.09]
Code churn / commit (median, ex-lockfiles)†	242	22.00	28.00	0.16 [0.10, 0.23]	<0.001	-524 [-1,780, 968]

4.1 Cohort and participant flow

From 19,174 discovered candidates, 1,954 passed the inexpensive inclusion checks. This pool is not a rejection funnel: the treated arm is a census of measurable in-window adopters (272), the control arm a capped sample of the no-signal pool (target 300, 299 analyzed). Most of the remaining pool is eligible-but-unsampled controls; the rest were dropped for other reasons, which Table 1 lists. The 400 MB measurement cap barely cuts down the analyzed sample itself: median repository size is 15 MB, and only 9 of 571 repositories exceed 360 MB. The treated arm divides into 113 Python and 159 TypeScript repositories (against 179 Python and 120 TypeScript controls) and is split almost evenly between strong S1 file-marker signals ($n = 140$, 51%) and S2 co-authorship trailers ($n = 132$, 49%). The most frequently attributed tools are GitHub Copilot (110 repositories), Claude Code (56), and Cursor (55), followed by a long tail of agentic assistants such as Devin, Cline, Windsurf, and Aider.

4.2 Baseline characteristics

Treated repositories are far more popular than controls, carry far more contributors (median 47 and 30 for Python and TypeScript against 7 and 8 for controls), and are larger (median T_{-6} LOC 15,556 against 4,686 over the paired subset); they are also younger: median creation year 2021 against 2019 (Mann-Whitney $p = 0.0002$). Because this gap is time-invariant, the DiD differences it out; we revisit the residual, time-varying part in the Discussion.

4.3 RQ1: average cyclomatic complexity

We find no robust effect on average cyclomatic complexity. The within-treated change is negligible (Cliff's $\delta = 0.02$), and although the FDR-adjusted Wilcoxon test is significant ($p_{\text{adj}} = 0.005$) by virtue of the large paired sample, the treated-vs-control DiD is -0.04 with a 95% CI of $[-0.13, 0.04]$ that includes zero (Table 2). The within-group test is misleading because average complexity is essentially flat in treated repositories while it drifts *upward* in controls (Figure 1, left), so the raw pre/post movement does not survive differencing against the control trend. RQ1's answer is a null: adoption is not associated with a robust change in average function complexity.

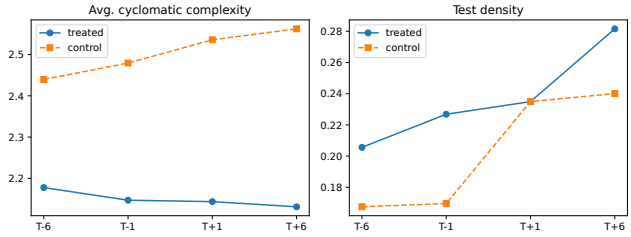

Figure 1: Mean metric across the four snapshots, treated vs. control. Average cyclomatic complexity (left) is flat for treated repositories while controls drift upward, yielding a null DiD. Test density (right) rises in both arms but earlier and further for treated repositories.

4.4 RQ2: bug-fix commit ratio

RQ2 is likewise null at the DiD level. The bug-fix ratio rises slightly within treated repositories (0.28 to 0.30; $\delta = 0.10$, negligible), but the DiD is 0.03 with CI $[-0.02, 0.09]$ spanning zero (Table 2). The keyword classifier's low precision (Section 3.4) means this null is best read as a noisy-measure null that could mask a small effect; we return to it in the Discussion.

4.5 RQ3: language and activity strata

Stratified Mann-Whitney tests on the per-repository change scores show no systematic moderation by language or activity. Three of the 24 comparisons across the language and activity axes reach or touch nominal significance before correction: max cyclomatic complexity by activity level ($p = 0.012$), code duplication by language ($p = 0.042$), and test density by activity level ($p = 0.050$). This is close to the ≈ 1.2 hits expected by chance at $\alpha = 0.05$ across 24 tests, and none survive multiple-comparison correction. We therefore find no convincing evidence that adoption effects differ between Python and TypeScript or between high- and low-activity repositories,

which also lends partial support to pooling the two languages in the primary DiD.

4.6 Exploratory findings

Beyond the research questions, four metrics show a robust and interpretable DiD (CI excluding zero): adopting repositories increase test density (DiD 0.05; Figure 1, right), test count per kLOC (3.07), and Python docstring coverage (0.09), and reduce parameters per function (−0.08). Max cyclomatic complexity also moves robustly (rising 8.47 while the average does not), but we report it for completeness only, not as a quality change: the maximum is an order statistic that tends to climb as a codebase adds functions, even when per-function complexity is unchanged. Since treated codebases grow faster than controls over the window (total-LOC DiD +10,013, CI [3,587, 16,211]) and the size-normalized share of complex functions (CC>10) has a null DiD. This rise most likely reflects codebase size, not genuine complexity growth (Table 2 gives each row's CI). We stress that these four effects, though statistically robust, are *practically modest*: every Cliff's δ is negligible-to-small on the Romano scale, the largest being docstring coverage at 0.27. The picture is one of small shifts in testing and documentation discipline, not a broad change in code quality. The two robust testing metrics are, moreover, near-redundant with each other and with the test-file ratio (Spearman 0.86–0.94), so they evidence a single shift in testing discipline, not independent corroboration. A further caution applies to three of these metrics, whose pre-treatment placebo DiD already excludes zero (Section 4.7). Because none of these metrics are among the research questions, we report them all as exploratory and hypothesis-generating. Two further metrics move within treated repositories but not against controls: Python typed-parameter ratio and lint warnings per kLOC both improve within the treated arm (FDR-adjusted $p \leq 0.001$) yet have DiDs whose CIs cover zero (0.02 $[-0.03, 0.06]$ and −4.78 $[-13.01, 1.23]$); like RQ1, these are within-treated shifts that do not survive differencing against controls. The TypeScript lint stratum (ESLint) is also null (DiD +2.58 $[-1.04, 6.66]$, $n = 109$). The code-churn metric is heavy-tailed across repositories, so its DiD (−524, bootstrap interval $[-1{,}780, 968]$) is highly unstable.

4.7 Sensitivity analyses

The exploratory effects are largely insensitive to the signal definition and treatment timing, with one exception. Restricting the treated arm to strong S1 markers only, admitting the 10 keyword-only S3 repositories, and re-anchoring every treatment date 30, 60, or 90 days earlier leave the direction and significance of three of the four robust metrics unchanged. The exception is the reduction in parameters per function, which loses significance in the S1-only subset (DiD −0.05, CI $[-0.12, 0.00]$), so we do not claim it holds across signal tiers. The two research-question nulls remain null under every perturbation.

We also probed the DiD parallel-trends assumption with a pre-treatment placebo, computing the DiD over $T_{-6} \to T_{-1}$ when both snapshots still precede adoption. For three of the four exploratory metrics this placebo excludes zero: docstring coverage ($n = 274$; 0.04, CI $[0.02, 0.06]$), test density ($n = 517$; 0.02, $[0.01, 0.03]$), and test count per kLOC ($n = 517$; 0.94, $[0.12, 1.76]$). The arms were therefore already diverging before adoption, so we treat those three associations as suggestive, not causal. Average cyclomatic complexity (RQ1) and parameters per function pass the placebo, with a CI covering zero; the bug-fix ratio (RQ2) is a window metric, so no pre-treatment placebo is defined for it. Because the placebo needs only the two pre-adoption snapshots, it pools treated and control repositories over larger samples than the primary contrast (n from 273 to 566, lowest for the Python-only metrics), so these are not low-power non-findings.

5 Discussion

Both primary results are null. Neither average cyclomatic complexity (RQ1) nor the bug-fix ratio (RQ2) shows a difference-in-differences distinguishable from zero, so on both outcomes we find no evidence that AI adoption changes code quality at the repository level. A precise null is itself informative: the confidence intervals are narrow and sit around zero, so any effect they leave room for is small. We did not pre-specify a smallest effect of interest or run a formal equivalence test, so we bound rather than formally exclude such an effect. The exploratory metrics that move (more tests and documentation, fewer parameters) tell a coherent secondary story, but must be read with restraint: effect sizes are negligible-to-small, they were not research questions, and for three the arms were already diverging before adoption. The reduction in parameters per function is the only exploratory effect whose placebo covers zero, making it the least fragile of the four; even so it loses significance in the S1-only subset, so we read it as suggestive, not confirmed. A reasonable reading is that adoption coincides with modest tooling-discipline improvements, not a transformation of the codebase.

Our nulls stand in contrast to recent longitudinal work, which reports that adopting Cursor or autonomous coding agents is followed by lasting increases in code complexity and static-analysis warnings [1, 14]. At the repository level we do not reproduce that pattern for *average* complexity, which shows no robust difference-in-differences, although the size-confounded *worst-case* maximum does rise. Several differences may explain the divergence: our cohort mixes many assistants rather than one agentic tool, our unit is the whole repository (not individual edits), a repository-level average masks tail effects, and those studies track *cognitive* complexity (SonarQube) whereas we measure *cyclomatic* complexity (Lizard). Whether AI raises complexity may thus depend on the tool, the complexity construct, and the granularity measured. A construct-validity caveat compounds this, as He et al. [14] put it: "[C]omplexity 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".

These findings carry cautious implications. For practitioners they are reassuring: across fifteen metrics we detect no quality degradation, the size-confounded maximum aside. The reassurance stops there, since nothing shows AI improving quality on its own; even the testing and documentation gains predate adoption (Section 4.7). For researchers, the null average beside a size-driven worst-case argues for finer-grained, size-normalized, and tail-sensitive outcomes

over longer horizons, ideally at the level of individual AI-attributed changes rather than whole repositories.

The remaining threats fall under four validity types [23].

Construct validity. The adoption signal marks the onset of *detectable*, agent-mediated use, not full or individual adoption: it shows that *some* contributors used AI, not which ones or at what rate, so we measure a population-level association, and pre-signal informal use is invisible. Our shifted-treatment sensitivity analysis (treatment moved 30, 60, and 90 days earlier) leaves the direction and significance of every robust metric unchanged, bounding the effect of undetected pre-signal use.

On the outcome side, the bug-fix ratio rests on a keyword classifier whose precision is only 0.57 (200 commits, two raters, Cohen's $\kappa = 0.74$); it recovers most genuine fixes but admits many false positives, so the absolute ratio overstates the defect rate. This misclassification is non-differential at best and would *attenuate* a true effect toward zero rather than cancel, so RQ2 is a noisy-measure null in which a real effect could be masked, not a confident no-effect. The outcome proxies are also imperfect: testing metrics count volume, not coverage, and docstring density can rise merely from assistant-emitted text. Pinned tool versions and a fixed configuration ensure a metric change reflects the code, not linter drift.

Internal validity. Repositories that adopt AI tools may differ from controls in expertise, tooling maturity, or prior quality trajectory. The arms are in fact imbalanced: treated repositories are younger and more popular (median creation year 2021 against 2019, Mann-Whitney $p = 0.0002$). Differencing removes this time-invariant gap; time-varying confounders remain and no propensity score matching was applied.

Two forms of selection are more troubling. First, adopting a repository-level AI marker (an S1 file) plausibly selects tooling-mature teams that already mandate tests and documentation, so part of the exploratory testing and documentation signal may reflect who adopts rather than an effect of adoption; consistent with this, the pre-treatment placebo shows the arms already diverging on docstring coverage, test density, and test count before adoption (Section 4.7), which DiD cannot remove. Second, general improvements in OSS practice over 2022–2025 could explain part of any change; the control group and DiD are our primary guard, yet residual secular confounding cannot be fully excluded, and controls anchored to a single synthetic cutoff sit at a later project-lifecycle stage than the treated windows, a further time-varying confound that differencing does not remove.

A distinct and more consequential concern is control contamination. Our signals detect only marker files, co-author trailers, and keywords; repositories whose contributors use AI solely through editor completion or web browsers leave none of these traces and remain in the control arm. Such undetected adopters make the control trend partly an adoption trend, biasing every difference-in-differences toward zero. The two research-question nulls are the results most exposed to this: they are consistent with no effect, but also with a real effect diluted by silent AI use among controls, which reinforces reading them as no *detected* effect rather than evidence of absence. Because a missing T_{+6} marks a repository that fell silent after adoption, the pairwise-complete deletion, applied per metric, conditions each estimate on continued post-adoption activity, an attrition that is plausibly informative (missing not at random, MNAR) and is distinct from the whole-repository survivorship noted below.

External validity. Snowball sampling (<14% of either arm) from repositories with visible AI adoption favors well-connected, higher-visibility projects. Because only repositories active throughout the window are observed, projects that went inactive after adoption are missed, biasing the treated arm's metrics upward. With the restriction to Python and TypeScript repositories of at least 10 stars and at most 400 MB, generalization is bounded to moderately popular projects in these languages.

Conclusion validity. To obtain a workable sample we relaxed the planned creation-date, star, activity, contributor, and LOC criteria, extended the window to 2025, and capped measurement at 400 MB (Section 3.3), shifting the cohort toward younger, smaller projects. The four exploratory metrics were, moreover, selected after inspecting the data. Benjamini-Hochberg correction applies to the Wilcoxon p-values; for the DiD, all five robust rows still exclude zero after correcting the fifteen-metric family for multiple comparisons (Bonferroni $\alpha = 0.05/15$, or a sup-t simultaneous band), so multiple testing does not explain them. We rely on the sensitivity and placebo checks rather than nominal p-values, having flagged where they fail: parameters per function under the S1-only subset, and three exploratory metrics under the parallel-trends placebo.

6 Conclusion

We presented an empirical mining study of the association between AI coding assistant adoption and static code quality across 272 treated and 299 control open-source repositories, using per-repository adoption detection, a staggered interrupted time series, and a difference-in-differences estimator over fifteen metrics. On the research questions we detect no adoption effect: neither average cyclomatic complexity (RQ1) nor the bug-fix ratio (RQ2) shifts under difference-in-differences, and these (non-)effects do not vary by language or activity (RQ3). Exploratory analyses associate adoption with modest increases in testing and documentation, none large in magnitude and several already diverging before adoption; an apparent rise in worst-case complexity most likely reflects codebase size. The contributions that outlast these particular numbers are the tiered signal hierarchy for detecting adoption from repository metadata and the reproducible, container-pinned measurement pipeline. Natural next steps are to confirm the exploratory signals, to measure finer-grained and tail-sensitive outcomes over longer horizons, and to move from repository-level association toward the individual AI-attributed changes that drive it.

Data Availability

The curated repository list, the data collection pipeline, and the computed per-snapshot metrics, tables, and figures are made publicly available at https://github.com/LukasWestholt/ai-impact-code-quality; the pinned measurement image (all tool versions) can be pulled by digest [24]. All randomized procedures use a fixed seed so that the reported confidence intervals reproduce exactly.

Acknowledgments

The author thanks Lukas Ruminski for serving as the second rater in the manual validation of the bug-fix commit classifier. Parts of the analysis pipeline were written with the assistance of Claude Code and reviewed by the author.

References

[1] Shyam Agarwal, Hao He, and Bogdan Vasilescu. 2026. AI IDEs or Autonomous Agents? Measuring the Impact of Coding Agents on Software Development. In *Proceedings of the 23rd International Conference on Mining Software Repositories (MSR)*. arXiv:2601.13597

[2] Joshua D. Angrist and Jörn-Steffen Pischke. 2009. *Mostly Harmless Econometrics: An Empiricist's Companion*. Princeton University Press, Princeton.

[3] Anthropic. 2025. Claude 3.7 Sonnet and Claude Code. https://www.anthropic.com/news/claude-3-7-sonnet. Claude Code announced in research preview 2025-02-24; accessed 2026-06-30.

[4] Anysphere. 2024. Cursor Changelog. https://cursor.com/changelog. Agent mode released 2024-12; accessed 2026-06-30.

[5] Sebastian Baltes and Paul Ralph. 2022. Sampling in software engineering research: a critical review and guidelines. *Empirical Software Engineering* 27, 4 (2022), 94. doi:10.1007/s10664-021-10072-8

[6] Yoav Benjamini and Yosef Hochberg. 1995. Controlling the False Discovery Rate: A Practical and Powerful Approach to Multiple Testing. *Journal of the Royal Statistical Society: Series B (Methodological)* 57, 1 (1995), 289–300. doi:10.1111/j.2517-6161.1995.tb02031.x

[7] Brantly Callaway and Pedro H. C. Sant'Anna. 2021. Difference-in-Differences with Multiple Time Periods. *Journal of Econometrics* 225, 2 (2021), 200–230. doi:10.1016/j.jeconom.2020.12.001

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

[9] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374

[10] Norman Cliff. 1993. Dominance Statistics: Ordinal Analyses to Answer Ordinal Questions. *Psychological Bulletin* 114, 3 (1993), 494–509. doi:10.1037/0033-2909.114.3.494

[11] Al Danial. 2025. cloc: Count Lines of Code. https://doi.org/10.5281/zenodo.15734241. doi:10.5281/zenodo.15734241 Version v2.06.

[12] GitHub. 2022. GitHub Copilot is generally available to all developers. https://github.blog/2022-06-21-github-copilot-is-generally-available-to-all-developers/. Accessed: 2026-06-07.

[13] GitHub. 2025. GitHub Copilot: Meet the New Coding Agent. https://github.blog/news-insights/product-news/github-copilot-meet-the-new-coding-agent/. Announced 2025-05-19; accessed 2026-06-30.

[14] Hao He, Courtney Miller, Shyam Agarwal, Christian Kästner, and Bogdan Vasilescu. 2025. Speed at the Cost of Quality: How Cursor AI Increases Short-Term Velocity and Long-Term Complexity in Open-Source Projects. arXiv:2511.04427

[15] James Lopez Bernal, Steven Cummins, and Antonio Gasparrini. 2017. Interrupted time series regression for the evaluation of public health interventions: a tutorial. *International Journal of Epidemiology* 46, 1 (2017), 348–355. doi:10.1093/ije/dyw098

[16] Thomas J. McCabe. 1976. A Complexity Measure. *IEEE Transactions on Software Engineering* SE-2, 4 (1976), 308–320.

[17] Tim Menzies, Jeremy Greenwald, and Art Frank. 2007. Data Mining Static Code Attributes to Learn Defect Predictors. *IEEE Transactions on Software Engineering* 33, 1 (2007), 2–13. doi:10.1109/TSE.2007.256941

[18] Audris Mockus and Lawrence G. Votta. 2000. Identifying Reasons for Software Changes Using Historic Databases. In *Proceedings of the International Conference on Software Maintenance (ICSM)*. IEEE, San Jose, CA, USA, 120–130.

[19] Arghavan Moradi Dakhel, Vahid Majdinasab, Amin Nikanjam, Foutse Khomh, Michel C. Desmarais, and Zhen Ming Jiang. 2023. GitHub Copilot AI pair programmer: Asset or liability? *Journal of Systems and Software* 203 (2023), 111734.

[20] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2022. Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, San Francisco, CA, USA, 754–768.

[21] Sida Peng, Eirini Kalliamvakou, Peter Cihon, and Mert Demirer. 2023. The Impact of AI on Developer Productivity: Evidence from GitHub Copilot. arXiv:2302.06590

[22] Jeanine Romano, Jeffrey D. Kromrey, Jesse Coraggio, and Jeff Skowronek. 2006. Appropriate Statistics for Ordinal Level Data: Should We Really Be Using t-test and Cohen's d for Evaluating Group Differences on the NSSE and Other Surveys?. In *Annual Meeting of the Florida Association of Institutional Research*. 1–33.

[23] William R. Shadish, Thomas D. Cook, and Donald T. Campbell. 2002. *Experimental and Quasi-Experimental Designs for Generalized Causal Inference*. Houghton Mifflin, Boston.

[24] Lukas Westholt. 2026. mining-tools: Pinned Measurement Container Image (All Analysis Tool Versions). Container image, pull by digest `ghcr.io/lukaswestholt/mining-tools@sha256:d92700d6511bf9c7060ec3b2c787f889e8fd4c324cc19fe7930a1e863eb87464`; package page https://github.com/LukasWestholt/ai-impact-code-quality/pkgs/container/mining-tools/versions.

[25] Frank Wilcoxon. 1945. Individual Comparisons by Ranking Methods. *Biometrics Bulletin* 1, 6 (1945), 80–83. doi:10.2307/3001968

[26] Terry Yin. 2026. Lizard: A Code Complexity Analyzer Supporting Multiple Languages. https://github.com/terryyin/lizard. Accessed: 2026-06-07.

[27] Beiqi Zhang, Peng Liang, Xiyu Zhou, Aakash Ahmad, and Muhammad Waseem. 2023. Demystifying Practices, Challenges and Expected Features of Using GitHub Copilot. *International Journal of Software Engineering and Knowledge Engineering* 33, 11n12 (2023), 1653–1672. doi:10.1142/S0218194023410048