GenAI and Test Co-Evolution in Software Projects: An Empirical Study

Charles Miranda
Federal University of Piauí
Teresina, Piauí, Brazil
charlesmiranda@ufpi.edu.br

Guilherme Avelino
Federal University of Piauí
Teresina, Piauí, Brazil
gaa@ufpi.edu.br

Pedro Santos Neto
Federal University of Piauí
Teresina, Piauí, Brazil
pasn@ufpi.edu.br

Roberto Felaço
Senior Sistemas
Indaial, Santa Catarina, Brazil
roberto.felaco@senior.com.br

Diogo Cristofolini
Senior Sistemas
Blumenau, Santa Catarina, Brazil
diogocri@gmail.com

Abstract

Generative Artificial Intelligence (GenAI) tools such as GitHub Copilot are increasingly used in software development, yet their relationship with testing practices remains poorly understood. In particular, it is unclear whether GenAI is associated not only with testing activity but also with the synchronization between production and test code over time. This study investigates the relationship between GenAI adoption, code coverage, and test co-evolution in industrial software projects. Web-oriented backend and frontend repositories are analyzed using repository mining, SonarQube data, and a practitioner survey. A quasi-experimental design combining Interrupted Time Series (ITS), Regression Discontinuity Design (RDD), and CausalImpact is adopted to compare pre- and post-adoption periods. The results indicate a general increase in testing activity and code coverage after adoption, with more evident gains in medium-term windows within the first post-adoption year. In contrast, test co-evolution remains highly variable and does not show a consistent pattern of sustained improvement; under a counterfactual perspective, post-adoption deviations are often negative. Survey responses suggest that participants perceive production and test updates as closely synchronized, but this perception is only partially aligned with the longitudinal evidence. Overall, the findings suggest that, in the studied context and observed period, no consistent evidence suggests that GenAI substantially changes the underlying nature of test co-evolution; instead, it is more clearly associated with increased testing activity and broadening coverage than with transforming synchronization between production and test evolution.

Keywords

Generative Artificial Intelligence, Software Testing, Test Co-Evolution, Mining Software Repositories

1 Introduction

Software development is inherently continuous and evolving [16, 17], making quality assurance increasingly challenging as systems adapt to new features, bug fixes, and changing requirements [22]. In this context, testing remains central to preventing regressions, validating functionality, and supporting maintainability [21]. Yet quality depends not only on writing tests, but also on keeping them aligned with production code over time. This interdependent evolution, or test co-evolution, is critical to preserving testing effectiveness [25, 34]; when synchronization fails, projects may accumulate outdated validations, weaker fault detection, and additional maintenance effort [37].

Prior work shows that synchronization between production and test code is desirable but difficult to sustain [18, 38], with large-scale studies reporting substantial project-level variation and recurring evolutionary lag, where production changes are not accompanied by corresponding test updates [24, 38].

This challenge is now being reshaped by Generative Artificial Intelligence (GenAI) in software development. Tools such as GitHub Copilot[1] can generate code, documentation, and test cases in real time, potentially improving productivity while also changing how developers create and maintain both production and test artifacts [41]. Even so, there is still limited empirical evidence on how GenAI is associated with the co-evolution of production and test code: it may be associated with increased testing activity, but it may also introduce inconsistencies or weaken alignment over time.

Against this backdrop, this study investigated the relationship between GitHub Copilot adoption and the co-evolution of production and test code. Web-oriented industrial projects, including backend and frontend systems, were analyzed using repository mining and time-series techniques. The scope reflects the availability of comparable workflows, testing pipelines, and longitudinal data across repositories, and should be interpreted accordingly. This study addressed the following research question: *How does the use of Generative AI tools relate to the co-evolution between production and test code?*

To answer this question, the analysis combines repository mining, SonarQube coverage data, and a complementary practitioner survey in a longitudinal quasi-experimental design. GitHub Copilot adoption is examined through code coverage and test co-evolution using Interrupted Time Series (ITS), Regression Discontinuity Design (RDD), and CausalImpact across multiple temporal windows.

As a result, the evidence points to a nuanced pattern. Code coverage tends to increase after adoption, especially over medium-term windows within the first post-adoption year, whereas test co-evolution remains unstable and does not exhibit consistent improvement. This suggests that, in the studied context and observed period, no consistent evidence suggests that GenAI substantially

[1] https://github.com/features/copilot

changes the underlying nature of test co-evolution; instead, it is more clearly associated with accelerating test generation and coverage expansion than with transforming synchronization between production and test evolution.

The main contributions of this work are: 1) an empirical investigation of the association between GenAI adoption, test co-evolution, and code coverage in real-world software projects; 2) a multi-method analytical approach combining ITS, RDD, and CausalImpact; and 3) evidence that higher code coverage does not necessarily correspond to stable co-evolution between production and test code over time.

2 Related Work

Generative Artificial Intelligence (GenAI), through tools such as GitHub Copilot, ChatGPT, and CodeWhisperer, has attracted increasing attention in software engineering. Recent studies have investigated its relationship with software development from multiple perspectives, particularly regarding the quality of generated code. Some works report positive outcomes, such as improvements in functional correctness [30], while others highlight limitations, suggesting that generated code may be inferior to human-written solutions [14]. Other studies emphasize both the potential and the risks associated with these tools, pointing to still inconclusive results [1, 36]. Overall, this diversity of findings suggests that the effectiveness of GenAI-based programming assistants is highly dependent on context and usage.

Beyond code quality, several studies have examined the practical challenges associated with adopting these tools. Prior work has identified limitations in Copilot's effectiveness as a development assistant [10], as well as issues related to code complexity and understandability [26]. Other studies report that developers may spend additional time reviewing and validating generated suggestions [6]. Research has also highlighted challenges inherent to applying large language models to programming tasks [31], reinforcing the need for a deeper understanding of their implications in real-world development workflows.

More recently, the literature has explored broader aspects such as developer experience and productivity. Some studies suggest that tools like Copilot can serve as a useful starting point, even if they do not necessarily reduce overall development time [4]. Other works analyze usage patterns and practices in real-world environments [29, 39], as well as how different developer profiles interact with these tools [32]. In parallel, there is growing interest in the performance of generated code, with studies identifying inefficiencies and proposing optimization techniques [9, 11, 13, 19, 20].

Taken together, prior studies provide relevant evidence on code quality, developer productivity, and usage patterns of GenAI assistants. Most analyses focus on code generation quality or short-term developer interactions, with limited attention to longitudinal associations with testing dynamics and maintenance-oriented practices. In particular, the literature still lacks a convergent explanation of how GenAI adoption is associated with the coordination between production and test evolution over time.

This study builds on prior work on test co-evolution [23, 24]. In contrast to prior efforts, it addresses this gap by shifting the analysis from isolated code-generation outcomes to the temporal relationship between production and test code in industrial settings. Repository mining is combined with a longitudinal quasi-experimental design, and evidence is triangulated through ITS, RDD, and CausalImpact across multiple temporal windows. This design allows the analysis to assess not only whether testing activity changes after adoption, but also whether such changes are consistently reflected in test co-evolution behavior.

3 Study Design

The study was conducted through dataset construction, metric extraction and time series analysis, statistical evaluation around Copilot adoption, and a complementary survey-based analysis. Figure 1 presents the workflow of the proposed approach.

3.1 Dataset Construction

This study uses data from a private Brazilian software company that develops enterprise solutions in domains such as ERP, HCM, logistics, sales, and access/security. Project selection followed convenience sampling: the initial set of 25 active projects comprised repositories with authorized source-code access, corresponding task-management data, and available SonarQube data.

The 25 projects were screened using four inclusion criteria: (i) complete repositories with development history available in the repository itself, (ii) implementation in TypeScript or Java, (iii) active use in production, and (iv) non-trivial projects maintained by a development team rather than isolated or short-lived efforts. A second filtering stage ensured analytical comparability. First, 5 repositories created only after organizational GenAI adoption were removed because they lacked a comparable pre-adoption period. Second, 6 repositories with sparse, irregular, or discontinuous testing histories were excluded because they could not support stable monthly series for code coverage and test co-evolution. The final dataset therefore comprises 14 projects.

The filtering was therefore guided by analytical completeness rather than by an arbitrary single-value cutoff. In summary, the dataset construction progressed from 25 initially eligible projects to 20 projects with a comparable pre-adoption period, and finally to 14 projects with sufficiently consistent longitudinal testing data for the proposed analyses.

The scope is therefore limited to web-oriented industrial repositories (backend and frontend systems), selected due to data availability and comparable workflows; consequently, the findings may not generalize to mobile, embedded, or substantially different development contexts. Nevertheless, the selected projects are mature systems with established practices, including automated testing, continuous integration, and rich historical data. As shown in Table 1, they exhibit substantial development activity, involve multiple developers, share a project age of 7 years, and have a median of 25% test files.

The final dataset comprises 7 Java backend microservices, 6 Angular frontend applications, and 1 Java backend library, reflecting different architectural layers within the organization. The analyzed tests are predominantly unit-oriented: unit and repository tests in Java projects, commonly implemented with JUnit-based stacks and

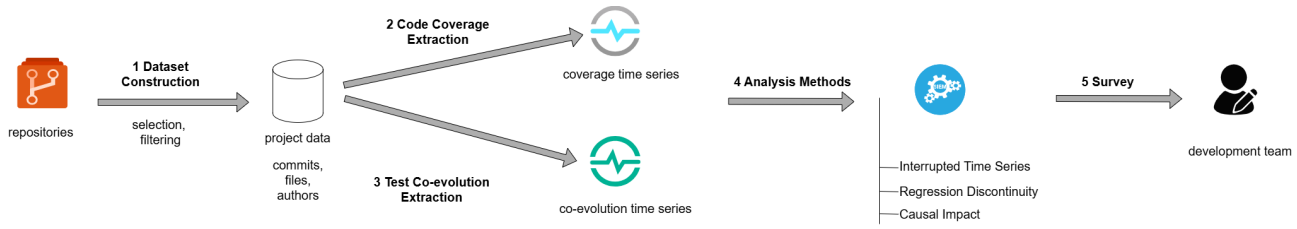


Figure 1: Workflow of the research method

Table 1: Distribution of key characteristics in the final dataset

	Commits	Devs	Age (Years)	Test Files (%)
Q1	2025	56	7	14
Q2	2434	73	7	25
Q3	3758	90	7	30
IQR	1733	33	0	16

Mockito, and component-level unit tests in Angular projects, typically executed with Jasmine/Karma and Angular TestBed. Therefore, the analysis primarily reflects unit, repository, and component testing practices rather than broader end-to-end or system-level testing.

3.2 Code Coverage Extraction

Code coverage data were extracted from project logs generated by SonarQube [8], which is integrated into the company's development pipeline and consolidates coverage reports produced by each project's testing toolchain. In the Java backend services and library, coverage is primarily computed from JaCoCo reports, which instrument test execution and record which lines and conditions are exercised. For Angular frontend applications, SonarQube ingests the coverage reports generated by the standard Angular unit-testing stack used in the projects.

In this study, the coverage metric represents the overall coverage of each analyzed project, rather than the coverage of only the code modified in a given change. This choice was made because SonarQube provides a consistent longitudinal project-level metric across repositories and technologies, whereas change-level coverage was not uniformly available in the analyzed pipelines. It is defined as the ratio between covered elements (executed lines and conditions) and the total number of executable elements, resulting in a percentage that reflects testing completeness.

Coverage data were collected automatically via the SonarQube REST API using a Python-based extract, transform, and load (ETL) pipeline. The extracted data were stored in JavaScript Object Notation (JSON) format and aggregated monthly by project component, computing intra-month averages. Thus, the monthly coverage of a project corresponds to the arithmetic mean of the coverage percentages of its active components in a given month. For example, if the 14 active components of a project in January 2025 sum to 565.7 percentage points of coverage, the monthly mean is $565.7/14 = 40.4\%$. This aggregation gives the same weight to all components and is therefore sensitive to changes in component composition over time.

3.3 Test Co-Evolution Extraction

The extraction procedure builds on a previously proposed tool-supported method for measuring co-evolution between production and test code [23]. Whereas the original approach focuses on aggregated temporal trends, the present study adapts it to a commit-level temporal coupling perspective and extends the underlying tool to support scalable local execution across multiple repositories. Commit-level analysis was adopted because it is the standard granularity used in much of the software co-evolution literature and can be consistently obtained from local Git repositories. We acknowledge that pull request-level analysis could better capture complete development tasks and review cycles, potentially affecting co-evolution estimates, and identify this as an important direction for future work. The framework used to classify production and test files is therefore grounded in prior studies on test co-evolution and repository mining [38], rather than being defined ad hoc for this study.

Furthermore, the original method is extended with a file-level, pairwise co-evolution analysis that associates each production file with its corresponding test file based on common naming conventions (e.g., `*Test`, `*.spec`), as illustrated in Figure 2. Although this heuristic-based pairing may introduce imprecision due to the absence of explicit traceability links, such approaches are widely adopted in mining software repositories studies to relate production and test artifacts. In particular, the use of naming and structural cues is consistent with prior work on understanding and facilitating test co-evolution, as well as with studies that leverage co-evolution patterns for downstream tasks such as automated test case recommendation [33, 35].

Test co-evolution is defined as the degree to which related production and test files evolve synchronously, operationalized as temporally coupled changes across commits. The analysis was restricted to source files aligned with the dominant technology of each repository (i.e., `.java` for backend and `.js`/`.ts` for frontend). To increase confidence in the automated pairing process, a manual inspection of a diverse sample of production–test pairs was conducted, verifying the plausibility of the identified matches according to repository structure and testing conventions. All proposed extensions are included in the research artifacts.

For each repository, changes in production files are identified and, for each production–test file pair, it is verified whether the corresponding test file was also modified within a 7-day window before or after the production change. In this study, file changes include file insertions and file modifications; deletions and renamings are not considered in the co-evolution rate. This decision prioritizes

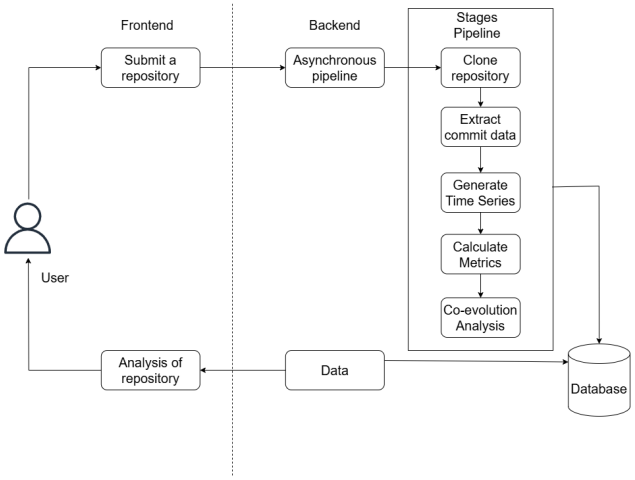


Figure 2: Overview of the refined tool-based approach for extracting test co-evolution based on a prior tool-supported approach [23]

measurement reliability. Exploratory analyses suggested that including file deletions and inferred renames introduced substantial artificial drops in the co-evolution metric during large repository restructurings. Deletions cannot be reliably paired once a file is removed, and Git detects renames heuristically rather than storing them explicitly, which may introduce false positives or inconsistent matches. Therefore, the analysis considers only additions and modifications, for which production–test pairing and temporal coupling are more stable. This choice may exclude legitimate synchronized deletions or renames and may underestimate co-evolution during refactorings or migrations, but it reduces noise and improves the reliability of the reported metric. In other words, a pair is considered co-evolving when the test file is committed up to 7 days earlier or up to 7 days later than the corresponding production file. In this study, the ±7-day temporal window was defined based on the workflow adopted by the company and the development team under analysis. In this context, development demands are usually broken down into smaller tasks that are expected to be implemented, tested, and released to customers within a cycle of up to 7 days. Therefore, this window reflects the short-term coordination expected between production and test changes in the industrial setting studied. Moreover, a 7-day interval is also consistent with the duration of a regular work week, making it appropriate for capturing related development and testing activities that may not occur in the same commit or on the same day, but still belong to the same delivery cycle.

We acknowledge that the ±7-day window is an operational choice based on the context of this study and that alternative definitions of a development work unit could also be adopted. To assess the robustness of the results, sensitivity analyses were conducted using ±7-, ±15-, and ±30-day windows, obtaining consistent conclusions across these configurations. Other operationalizations, such as same-commit or pull request-level coupling, are also possible but capture different aspects of production–test synchronization and remain an important direction for future work. This approach captures the synchronization between development and testing activities over time.

The co-evolution rate is defined as the proportion of production changes whose corresponding test files were modified within this ±7-day temporal window. Formally, for a given month m, the metric is computed as

$$CoEvoRate_m = \frac{CoEvolvingChanges_m * 100}{ProductionChanges_m},$$

where $CoEvolvingChanges_m$ denotes the number of production changes in month m whose paired test file was also changed within ±7 days, and $ProductionChanges_m$ denotes the total number of production changes observed in the same month. This metric reflects the extent to which test code evolves alongside production code. The unit of analysis is defined in three steps. First, the metric is computed independently for each repository on a monthly basis. Second, each repository-level time series is aligned by relative month with respect to the intervention date. Third, the aligned repository-level series are aggregated by arithmetic mean, producing one cross-repository series for each metric and bandwidth. For monthly aggregation within each repository, all co-evolving production changes across the active components of a project are summed and this value is divided by the total number of production changes in the same month. For example, if in January 2023 a project has 32 co-evolving changes out of 101 production changes across its components, the monthly co-evolution rate is $(32 \times 100)/101 = 31.68\%$. Unlike the monthly coverage mean, this aggregation weights components according to their volume of changes and is therefore more sensitive to the distribution of development activity across components.

This approach is consistent with prior empirical studies on test co-evolution, which model co-evolution as a temporal and structural relationship between production and test artifacts [24, 38].

3.4 Analysis Methods

To assess the association between Generative AI adoption and co-evolution behavior, three complementary statistical approaches are applied: Interrupted Time Series (ITS), Regression Discontinuity Design (RDD), and CausalImpact. ITS estimates changes in level and trend, RDD focuses on local discontinuities around the cutoff, and CausalImpact compares the observed post-adoption trajectory with a counterfactual baseline; using the same temporal windows across methods improves comparability and supports triangulation. January 2025 is defined as the intervention point, corresponding to the organizational proxy date for recurrent developer-level adoption of GitHub Copilot in the analyzed projects. Since licenses were introduced gradually throughout 2024 and no single adoption date applies uniformly across developers, January 2025 represents consolidated day-to-day usage. To reduce transition-period biases, such as gradual onboarding, learning effects, and partial adoption, the intervention month is excluded from the analysis [40].

For each metric (code coverage and test co-evolution), one monthly time series per repository is constructed (14 series per metric) and aligned relative to the January 2025 cutoff, which represents the consolidated organizational adoption period rather than a directly observed usage event in every repository or commit. Monthly aggregation reduces day-to-day noise while preserving medium-term

variation, consistent with quasi-experimental Software Engineering studies that use monthly observations to evaluate process interventions [40]. Sensitivity analysis is performed over symmetric 6-, 9-, and 12-month bandwidths. For each relative month [40] and bandwidth, repository-level series are aggregated by arithmetic mean, capturing organizational-level trends rather than separate repository-specific patterns.

Interrupted Time Series (ITS) analysis is used [5, 28] to evaluate changes in both level and trend before and after the intervention. The time series is modeled using segmented regression, where the pre-intervention period establishes a baseline trend and the post-intervention period captures potential deviations associated with the adoption of GitHub Copilot. This approach enables the identification of both immediate deviations, reflected as level changes at the intervention point, and gradual patterns, reflected as changes in trend over time. To complement this longitudinal perspective, Regression Discontinuity Design (RDD) is applied [12, 40] to estimate local discontinuities around the intervention point. By focusing on observations close to the cutoff, RDD isolates short-term discontinuities and minimizes the influence of long-term trends.

Finally, CausalImpact is used [3, 7] to estimate the counterfactual behavior of the time series, modeling what would have occurred in the absence of the intervention. This method relies on Bayesian structural time series to predict expected post-intervention trajectories based on pre-intervention data, which are then compared with observed values. By providing probabilistic estimates and confidence intervals, CausalImpact enables a more nuanced interpretation of changes, capturing cumulative differences over time and offering a complementary perspective to ITS and RDD.

In the model specification, ITS and RDD use Ordinary Least Squares (OLS) regression with HC3 robust standard errors, while CausalImpact employs a Bayesian Structural Time Series (BSTS) model.

3.5 Survey

To better understand the mechanisms behind the observed quantitative results, a survey was conducted with both closed-ended and open-ended questions focusing on practitioners' perceptions of GitHub Copilot adoption. The questionnaire was designed to triangulate the evidence on code coverage and test co-evolution by collecting information about respondents' profile, timing of recurrent Copilot adoption, intensity of use in test-related tasks, and perceived changes in test maintenance and synchronization between production and test code. The survey was administered to members of the development teams involved in the analyzed projects, including developers, quality assurance (QA) professionals, and team leads.

The questionnaire combined structured and open-ended items. Closed-ended questions were used to characterize participants and summarize recurrent perceptions about Copilot use, review effort, test synchronization, and perceived coverage changes, while open-ended questions were used to capture examples and possible confounding factors affecting the observed results. Responses were analyzed qualitatively to identify recurring themes and explanatory mechanisms, especially regarding the increase in code coverage alongside unstable co-evolution. Participation was voluntary and anonymous, all participants provided informed consent, and the authors did not include their own responses in the analysis.

4 Results

The results are reported by contrasting two complementary dimensions of testing practice: testing expansion (code coverage) and maintenance synchronization (test co-evolution). For each dimension, evidence is triangulated from ITS, RDD, and CausalImpact across multiple bandwidths. Practitioner perceptions from the survey are then used to interpret convergences and divergences between observed behavior and developer experience.

4.1 Code Coverage Patterns After Adoption

Across the analytical perspectives, code coverage shows a modest post-adoption increase, although the strength of this pattern varies by method and temporal window.

The distributional comparison in Figure 3 shows a modest upward shift in central tendency after adoption, with the median increasing from 51.98% to 54.57% (+2.59 pp) and the mean from 51.43% to 55.19% (+3.75 pp, +7.3%). However, this improvement is not uniform over time, with more pronounced increases emerging in the later post-adoption period. Starting around August 2025, the boxplots exhibit a clearer upward shift, with higher medians and means (frequently approaching or exceeding 56–59%), as well as an upward displacement of the upper quartiles. At the same time, the variability appears relatively stable over time, suggesting that the improvement is not solely driven by isolated peaks but reflects a broader elevation of coverage levels across projects. This temporal pattern suggests that coverage gains may become more evident in the medium to longer term rather than immediately after adoption.

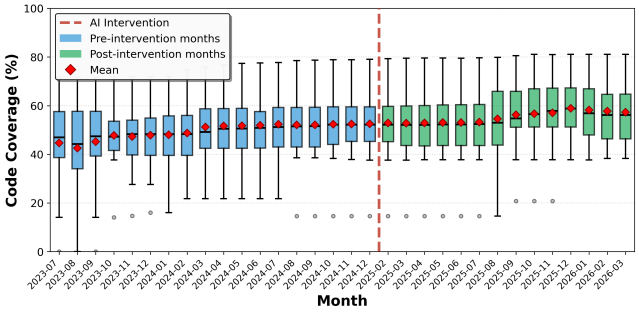


Figure 3: Code coverage distribution showing a modest post-adoption increase, more evident in later months

The Interrupted Time Series results in Figure 4 provide additional insight into the temporal dynamics observed in the distributional analysis. Across the bandwidth configurations, as indicated by the window limits in the figure, coverage follows a relatively stable upward trend before the intervention, with no clear visual evidence of a structural break at the cutoff point. Instead, post-adoption behavior appears as a continuation of the existing trend, with more noticeable increases emerging several months later, particularly from mid-2025 onwards. These observations reinforce the absence of a consistent immediate shift and support the interpretation of gradual improvement over time.

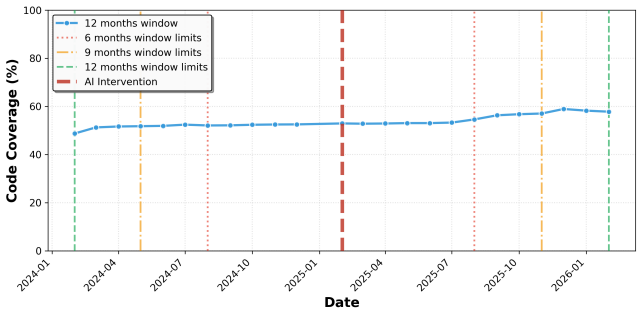


Figure 4: ITS results showing no clear structural break, but gradual post-adoption increase in coverage

Similarly, the Regression Discontinuity Design estimates presented in Table 2 do not indicate clear positive local discontinuities at the intervention cutoff. Across all bandwidths, the estimated local discontinuities are negative, small in the 6- and 9-month windows, and statistically significant only in the 12-month window. These findings suggest that coverage gains are not expressed as a consistent immediate local effect at the intervention cutoff.

Table 2: RDD results for code coverage across bandwidths

Bandwidth	Coef.	95% CI	p-value
6 months	-0.125	[-1.099, 0.850]	0.802
9 months	-0.435	[-1.456, 0.586]	0.404
12 months	-2.259	[-4.190, -0.327]	0.022

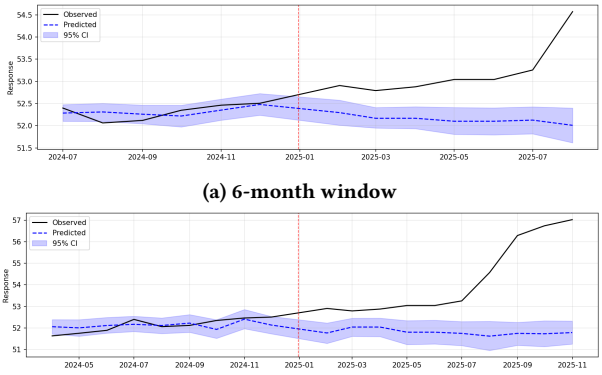
In contrast, the counterfactual analysis in Figure 5 consistently indicates positive post-adoption deviations across all bandwidths. Observed coverage values remain above the predicted baseline trajectory, producing consistent positive cumulative deviations over time across all analyzed windows, with the divergence becoming more pronounced in the later post-adoption period.

Table 3 consolidates this interpretation across methods. It shows progressively larger pre/post differences as the observation window widens (52.32% → 53.21%, 52.13% → 54.25%, and 51.43% → 55.19%), together with statistically significant positive CausalImpact estimates in all three windows ($p = 0.001$). Read alongside the ITS and RDD results, this synthesis suggests that coverage gains become more visible over medium-term horizons than as an immediate discontinuity at the intervention cutoff.

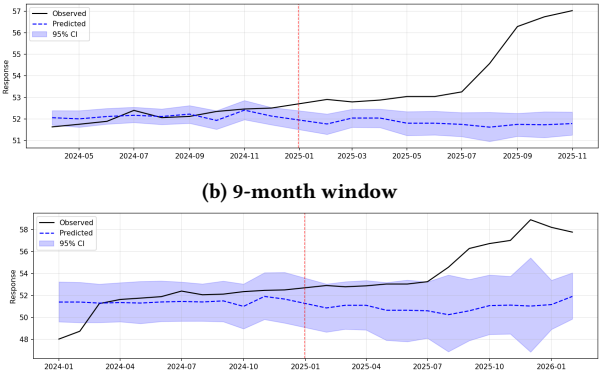
Taken together, the evidence indicates that GitHub Copilot adoption is associated with higher code coverage, primarily through

Table 3: Code coverage before and after GitHub Copilot adoption

Result	6-mo. window	9-mo. window	12-mo. window
Pre/Post	52.32 → 53.21	52.13 → 54.25	51.43 → 55.19 ↑
ITS	n.s. (p=0.519)	↓ **Sig.** (p=0.004)	n.s. (p=0.053)
RDD	n.s. (p=0.802)	n.s. (p=0.404)	↓ **Sig.** (p=0.022)
CausalImpact	↑ **Sig.** (p=0.001)	↑ **Sig.** (p=0.001)	↑ **Sig.** (p=0.001)



(a) 6-month window



(b) 9-month window

(c) 12-month window

Figure 5: CausalImpact results indicating consistent positive post-adoption deviations in coverage

gradual improvements that become more evident over medium-term horizons rather than through immediate structural changes. While the overall pattern is positive, its magnitude and detectability vary across analytical methods, highlighting the importance of adopting multiple perspectives when assessing associations with Generative AI adoption in software development.

4.2 Test Co-Evolution Patterns After Adoption

In contrast to code coverage, test co-evolution does not show consistent improvement after adoption. Across the analytical perspectives, the observed changes remain small, unstable, and difficult to interpret as evidence of a clear directional gain.

The distributional comparison in Figure 6 reinforces this interpretation. Pre- and post-adoption periods exhibit substantial overlap, with no clear and consistent shift in central tendency across time. Although some post-adoption months present higher central values, these increases are not sustained, and are interspersed with periods of lower co-evolution rates. The spread of the distributions remains similar across periods, indicating that co-evolution behavior is highly variable and does not exhibit a stable or systematic pattern of improvement after adoption.

The Interrupted Time Series results in Figure 7 further support this pattern. Across the bandwidth configurations, as indicated by the window limits in the figure, the time series exhibits substantial fluctuations throughout the analyzed period, without a sustained upward trend in the post-intervention phase. Although some post-adoption months present higher values, these increases are not maintained over time. No clear or consistent structural break is observed at the intervention cutoff, suggesting that adoption is not

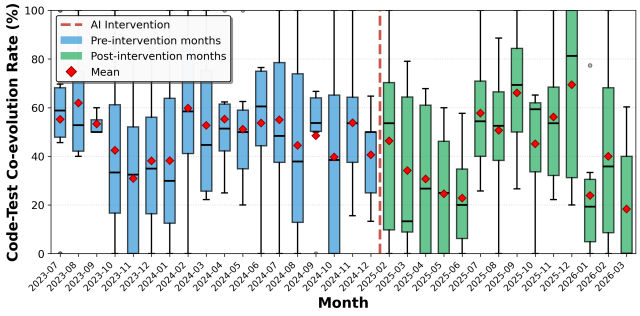


Figure 6: Test co-evolution distribution showing high variability and no clear post-adoption shift

associated with a stable or systematic improvement in synchronization behavior.

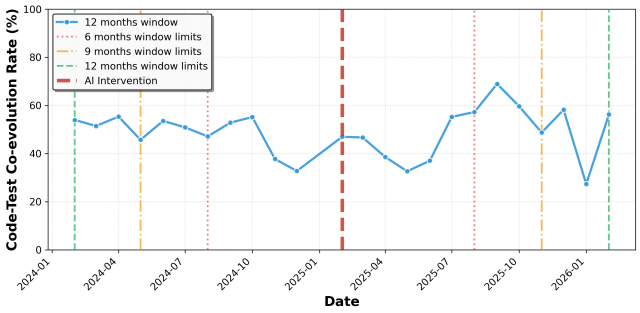


Figure 7: ITS results showing no consistent trend or structural change after adoption

Similarly, the Regression Discontinuity Design results summarized in Table 4 do not indicate clear discontinuities around the intervention cutoff. Across the three analyzed bandwidths, the estimated local discontinuities are consistently negative but highly imprecise, with wide confidence intervals that include zero and non-significant p-values. Overall, these results reinforce the absence of a consistent short-term association with synchronization behavior.

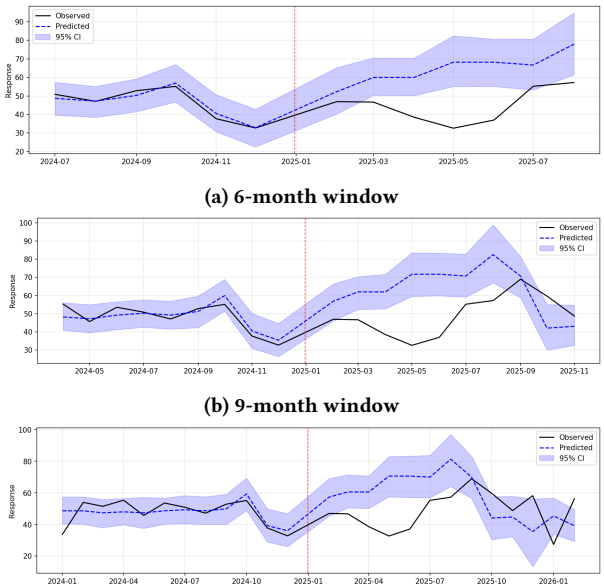
Table 4: RDD results for test co-evolution rate across bandwidths

Bandwidth	Coef.	95% CI	p-value
6 months	-0.692	[-26.536, 25.151]	0.958
9 months	-6.039	[-29.598, 17.519]	0.615
12 months	-6.653	[-30.335, 17.028]	0.582

In contrast, the counterfactual analysis in Figure 8 suggests predominantly negative post-adoption deviations, particularly in the shorter time windows. Observed values often remain below the predicted baseline trajectory, indicating negative relative deviations in several periods. However, this pattern is not consistent across all bandwidths, with some intervals showing mixed or overlapping behavior. Overall, these results suggest that, relative to the expected trajectory, co-evolution does not exhibit a consistent improvement after adoption and may, in some cases, underperform the counterfactual expectation.

Table 5: Test co-evolution before and after GitHub Copilot adoption

Result	6-mo. window	9-mo. window	12-mo. window
Pre/Post	46.05 → 44.88	47.87 → 49.15	47.49 → 48.70
ITS	n.s. (p=0.266)	n.s. (p=0.095)	n.s. (p=0.478)
RDD	n.s. (p=0.958)	n.s. (p=0.615)	n.s. (p=0.582)
CausalImpact	↓ **Sig.** (p=0.001)	↓ **Sig.** (p=0.001)	↓ **Sig.** (p=0.006)



(a) 6-month window

(b) 9-month window

(c) 12-month window

Figure 8: CausalImpact results suggesting negative or mixed post-adoption deviations in co-evolution

Table 5 brings these results together in a single view. The pre/post comparisons remain small and unstable across bandwidths, shifting from 46.05% to 44.88% in the 6-month window, from 47.87% to 49.15% in the 9-month window, and from 47.49% to 48.70% in the 12-month window. At the same time, ITS and RDD remain non-significant throughout, whereas CausalImpact indicates statistically significant negative deviations in all three windows. Taken together, this synthesis reinforces the absence of a clear directional improvement in co-evolution after adoption.

Taken together, the evidence does not support a consistent improvement in test co-evolution after GitHub Copilot adoption. While coverage tends to increase, co-evolution remains highly variable over time, without a stable or sustained pattern of improvement.

Under the counterfactual perspective, the results suggest predominantly negative or mixed deviations relative to the expected baseline, indicating that co-evolution does not consistently improve in the post-adoption period.

4.3 Survey Results

To complement the quantitative analysis, a survey was conducted with members of the development teams involved in the analyzed projects. A total of 9 responses were received out of 15 invitations (60% response rate), including eight developers and one Quality Assurance (QA) professional, with varied levels of experience.

The responses indicate that GitHub Copilot is actively used in test-related activities. Most participants report frequent or constant use of the tool for generating, completing, and reviewing automated tests, with 55.6% indicating "frequently", 33.3% "always", and 11.1% "sometimes". This suggests that Copilot is integrated into day-to-day testing workflows.

Participants also perceive strong synchronization between production and test updates. Specifically, 77.8% report that tests are updated together with production code "always", while 22.2% report this occurs "frequently". Regarding coverage, among the eight respondents who addressed this aspect, 62.5% perceived an increase after adoption and 37.5% reported stability, with no respondents indicating a decrease.

However, these perceptions contrast with the quantitative findings. While participants report high synchronization, the longitudinal analysis does not show consistent improvement in test co-evolution. This divergence suggests that perceived alignment at the task level may not translate into sustained synchronization over time.

Qualitative responses provide further insight into this gap. Participants consistently describe Copilot as effective for accelerating test creation and reducing boilerplate, but emphasize that generated tests still require manual review and often lack project-specific context. For example, one respondent noted that generated tests "require review, especially for new tests, where there is little contextual information for the AI", while another observed that in some projects "it can get lost".

Several respondents also highlighted that the effectiveness of generated tests depends on the surrounding development context, including project configuration and available tooling. Issues such as incorrect or unnecessary mocks were mentioned, reinforcing that Copilot-generated tests are not always aligned with project-specific testing practices.

These examples suggest that while Copilot reduces the effort required to produce tests, it does not eliminate the need for developer intervention and contextual understanding, particularly in more complex scenarios.

These survey results should be interpreted cautiously due to the small sample size, self-reported bias, and context specificity; thus, they serve only as complementary evidence to contextualize quantitative findings, not as a standalone measure of synchronization behavior.

5 Discussion

This section discusses the implications of the observed divergence between testing expansion and testing synchronization after GitHub Copilot adoption. It first interprets this divergence by contrasting coverage and co-evolution behaviors, then explores potential explanations grounded in repository evidence and qualitative responses, and finally discusses implications for practice and research.

5.1 Interpreting Code Coverage and Co-Evolution Results

The results of this study suggest a nuanced and, in some aspects, counterintuitive relationship between Generative AI adoption and testing practices. While code coverage exhibits a general upward tendency, test co-evolution does not show consistent improvement and, in some configurations, presents negative deviations. This divergence suggests that increased testing activity does not necessarily translate into better synchronization between production and test code.

For code coverage, the distributional view in Figure 3 shows substantial overlap between the pre and post-adoption periods, with a slight upward shift in post-adoption values. This indicates that, although coverage tends to increase, the magnitude of this change is modest. From a temporal perspective, the ITS analysis in Figure 4 does not reveal a consistent structural change, while the RDD estimates summarized in Table 2 show null discontinuities in the 6- and 9-month windows and a significant negative local discontinuity in the 12-month window. In contrast, the counterfactual analysis (Figure 5) consistently shows positive deviations, indicating that coverage is higher than expected under the baseline trajectory. This triangulation, also summarized in Table 3, suggests that coverage increases, but not as a strong structural shift.

This pattern is consistent with recent studies showing that AI-assisted test generation can improve, or at least achieve, non-trivial levels of code coverage across different contexts. Altmayer Pizzorno et al. reported a median per-module coverage of 80% compared to 47% in a baseline setting, as well as total coverage of 89% versus 77% in another comparison [2]. Pan et al. found that combining LLMs with static analysis increased line coverage by 26.4% in Java EE applications [27]. Likewise, Jain and Goues reported an average line coverage of approximately 44.4% with TestForge-generated test suites [15]. Taken together, these results suggest that GenAI-based approaches are often associated with broader test execution and increased coverage, particularly in controlled test generation scenarios.

In contrast, test co-evolution presents a different pattern. Figure 6, Figure 7, and the RDD estimates summarized in Table 4 indicate high variability and lack of consistent trends, while the counterfactual analysis (Figure 8) shows negative deviations across all bandwidths. As summarized in Table 5, these results do not provide consistent evidence of improvement in co-evolution and, in some analyses, are compatible with deterioration relative to its expected trajectory.

The variability measures further reinforce this interpretation. In the 12-month window, coverage increased from 51.43% to 55.19% despite higher post-adoption dispersion (SD: 1.48 to 2.35; IQR: 0.82

to 3.98). In contrast, co-evolution changed only slightly in mean values (47.49% to 48.70%) but exhibited substantially greater variability after adoption (SD: 8.32 to 12.09; IQR: 9.95 to 18.70). This pattern indicates that the post-adoption period is better characterized by heterogeneous and unstable behavior than by a uniform shift across the analyzed observations.

In this sense, separate frontend and backend analyses would be valuable, since these contexts may exhibit different testing and co-evolution dynamics. However, splitting the dataset of only 14 repositories would substantially reduce statistical power and the stability of ITS, RDD, and CausalImpact estimates. Therefore, the study adopts an organizational-level analysis, acknowledges this limitation, and leaves stratified analyses with larger frontend and backend samples as future work.

5.2 Explaining the Divergence

A key insight emerges when these quantitative results are contrasted with practitioner perceptions. Survey responses indicate that developers perceive strong synchronization between production and test updates, with most reporting that tests are "always" or "frequently" updated together with code changes. However, the empirical results do not clearly confirm this perception at a broader temporal scale, suggesting a gap between perceived and observed co-evolution.

Part of this gap may stem from how synchronization is conceptualized in practice. The co-evolution metric captures longitudinal alignment across commits and delivery cycles, rather than local task-level coordination. Consequently, activities that appear synchronized within individual tasks may still be reflected as weakly coupled in the historical signal captured by the metric.

This interpretation is reinforced by an exploratory manual inspection of the repositories that most negatively affected the co-evolution metric. To better understand the observed decline, an exploratory sample were traced back to their corresponding pull requests and associated tasks. Although these cases were not systematically ranked, the inspection frequently pointed to recurring situations in which one-to-one production–test synchronization is naturally weak or not expected. This pattern was particularly evident in frontend-related work, where artifacts such as certain user interface (UI) components do not always have a directly associated test file, as well as in modifications to continuous integration and continuous delivery (CI/CD) scripts and utility files (e.g., `.sh` and `.py`), which primarily support infrastructure and pipeline activities rather than production behavior.

Other recurrent scenarios point in the same direction. The inspected cases included migrations to newer versions of the Java platform and behavior-preserving refactorings in already covered code; for example, removing a Spring annotation to avoid dependency-injection conflicts may affect several production files without requiring updates to existing tests. Test-only maintenance activities were also identified, such as updating libraries used exclusively by the test suite, which naturally have no corresponding production changes. Taken together, these cases suggest that lower co-evolution rates do not necessarily indicate weaker testing discipline; rather, they may partially reflect the composition of maintenance work captured by the metric. This interpretation is also consistent with periods of concentrated technical activity, especially around January 2026, when the joint release of three projects made these patterns more visible in the time series.

This divergence also matters substantively for how testing quality is interpreted in practice. Higher coverage indicates broader test execution and can reflect faster test production, but it does not guarantee that tests remain aligned with the parts of the system that are changing. When coverage increases while co-evolution remains unstable, organizations may obtain more tests without strengthening the synchronization needed to preserve fault-detection capability, reduce maintenance friction, and sustain confidence that the test suite still reflects current production behavior. In this sense, coverage captures testing expansion, whereas co-evolution captures maintenance synchronization; considering both dimensions helps distinguish having more tests from maintaining more effectively aligned tests.

Qualitative responses are also compatible with this explanation. Participants often reported that generated tests still require review, especially in complex or context-specific scenarios, as highlighted by statements such as *"require review [...] where there is little contextual information for the AI"* and *"it can get lost"* in projects with particular characteristics. At the same time, they emphasized productivity gains, noting that Copilot *"speeds up test creation"* and enables higher coverage levels. This combination helps explain why coverage can increase even when co-evolution remains unstable: GenAI seems to make it easier to produce tests and expand coverage, but not necessarily to preserve systematic synchronization between production and test evolution over time. This observation suggests that GenAI tools may facilitate test generation more readily than sustained test maintenance over time.

5.3 Implications for Practice and Research

Taken together, these findings may reflect a partial shift from a maintenance-driven testing process to a more generation-driven process, in which tests are created more opportunistically. This helps explain the observed divergence: coverage increases because test generation becomes easier, while co-evolution remains unstable because synchronization is not reinforced with the same consistency over time. In other words, in the studied context, GenAI does not appear to change the underlying nature of test co-evolution in a sustained way; rather, it primarily seems to accelerate test generation and coverage expansion.

From a practical perspective, increasing code coverage alone is not sufficient to ensure effective testing practices. Organizations adopting GenAI tools should complement them with practices and metrics that explicitly monitor the alignment between production and test code over time.

From a research perspective, the gap between perceived and observed synchronization reinforces the importance of combining quantitative and qualitative evidence. It also highlights the need for more comprehensive evaluation frameworks that consider not only artifact generation, but also maintenance dynamics over time in AI-assisted software development.

6 Threats to Validity

This study faces typical limitations of observational and industrial empirical research, which are mitigated through methodological triangulation, sensitivity analyses, and complementary qualitative evidence.

Internal Validity: This study relies on observational data and a quasi-experimental design, limiting causal inference despite the use of ITS, RDD, and CausalImpact. Unobserved confounders may have influenced the results, including AI-assisted code review, new pipelines, and supporting tools reported in the survey, so observed changes should not be attributed exclusively to Copilot. Adoption was also gradual, and no per-developer telemetry or per-commit Copilot-usage signal was available; therefore, eligible commits were analyzed regardless of whether they were directly assisted by GenAI. To mitigate this limitation, two survey questions captured when participants started using Copilot regularly and how frequently they used it for automated-test tasks. A common organizational proxy date was then defined, the transition month was excluded, and observation windows were standardized. Nevertheless, heterogeneous adoption may attenuate estimates or introduce exposure misclassification, so the results should be interpreted as associations with the organizational adoption period rather than commit-level effects of confirmed Copilot use. The year 2024 was retained as the closest pre-consolidation baseline, balancing contamination risk from early GenAI use against temporal comparability with the post-adoption period.

Construct Validity: The results depend on methodological choices, especially the temporal configuration and the operationalization of co-evolution. The post-adoption period covers approximately 12 months, supporting short- to medium-term observations but not strong claims about long-term GenAI-related changes. To reduce temporal sensitivity, sensitivity analyses used 6-, 9-, and 12-month bandwidths aligned with organizational release cycles. Co-evolution is measured with a ±7-day window reflecting the studied workflow, although delayed or asynchronous coordination may be missed. Additional ±15- and ±30-day analyses led to consistent conclusions, but stricter or alternative operationalizations, such as same-commit or pull request-level coupling, could produce different estimates. Test-file and production–test pairing rely on heuristics, which may introduce noise; to reduce this risk, a diverse sample of pairs was manually inspected and auxiliary artifacts, such as infrastructure scripts and unrelated test types, were excluded. Even so, the metric remains an approximation of synchronization behavior. Finally, although the main CausalImpact structure is specified, some hyperparameters rely on library defaults, which may limit reproducibility and methodological transparency.

Conclusion Validity: The aggregation of metrics across repositories using arithmetic means may obscure variability between projects. Arithmetic means summarize organizational-level trends but may mask variability across repositories; therefore, mean-based results should not be interpreted as uniform patterns across all projects. To mitigate this limitation, aggregated results are complemented with distributional analyses and variability measures, including boxplots, standard deviations, and interquartile ranges. The relatively small sample size (14 projects) may also limit statistical power. However, this limitation is partially addressed through the use of multiple complementary analytical methods, which consistently evaluate the results from different perspectives, increasing confidence in the findings.

External Validity: The dataset is derived from a single private company and focuses on web-based systems, which may limit generalizability to other domains. Project selection was based on convenience, constrained by data availability, potentially introducing selection bias. However, this is common in industrial studies. This limitation is partially mitigated by including multiple projects across backend and frontend systems, increasing diversity. Additionally, the survey is subject to response bias and subjective interpretation; therefore, it is used to contextualize quantitative results rather than as primary evidence.

7 Conclusion and Future Work

This study investigated the relationship between GenAI adoption, specifically GitHub Copilot, and testing practices in industrial software projects, focusing on code coverage and test co-evolution. Based on repository mining, SonarQube coverage data, a complementary practitioner survey, and longitudinal quasi-experimental analyses across backend and frontend systems, pre- and post-adoption periods were compared. The results indicate that, within the observed post-adoption period, GenAI adoption is more strongly associated with increased testing activity and higher code coverage than with sustained improvements in test co-evolution. More specifically, the evidence suggests that, in the studied context and within the first post-adoption year, no consistent evidence suggests that GenAI substantially changes the underlying nature of test co-evolution; instead, it is more clearly associated with accelerating test generation than with transforming synchronization between production and test evolution. Although participants often perceived production and test updates as closely synchronized, this perception was only partially aligned with the longitudinal evidence.

Future work should further investigate test co-evolution as a heterogeneous phenomenon, analyzing how generative AI is associated with different test evolution patterns (e.g., Progressive, Stable, Deferred). It is also important to explore the relationship between generative AI, co-evolution, and maintenance types (feature development versus bug fixing), as well as extend the analysis to longer observation periods and open-source projects to understand how collaboration and project context influence alignment between test code and production code over time.

Artifact Availability

The research artifacts supporting the findings of this study are available at https://doi.org/10.5281/zenodo.21313548.

Acknowledgements

This research was supported by CNPq (process 403304/2025-3).

References

[1] Naser Al Madi. 2022. How readable is model-generated code? examining readability and visual inspection of github copilot. In *Proceedings of the 37th IEEE/ACM international conference on automated software engineering*. 1–5.

[2] Juan Altmayer Pizzorno and Emery D Berger. 2025. Coverup: Effective high coverage test generation for python. *Proceedings of the ACM on Software Engineering* 2, FSE (2025), 2897–2919.

[3] Susan Athey, Mohsen Bayati, Nikolay Doudchenko, Guido Imbens, and Khashayar Khosravi. 2021. Matrix completion methods for causal panel data models. *J. Amer. Statist. Assoc.* 116, 536 (2021), 1716–1730.

[4] Shraddha Barke, Michael B James, and Nadia Polikarpova. 2023. Grounded copilot: How programmers interact with code-generating models. *Proceedings of the ACM on Programming Languages* 7, OOPSLA1 (2023), 85–111.

[5] 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.

[6] Christian Bird, Denae Ford, Thomas Zimmermann, Nicole Forsgren, Eirini Kalliamvakou, Travis Lowdermilk, and Idan Gazit. 2022. Taking Flight with Copilot: Early insights and opportunities of AI-powered pair-programming tools. *Queue* 20, 6 (2022), 35–57.

[7] Kay H Brodersen, Fabian Gallusser, Jim Koehler, Nicolas Remy, and Steven L Scott. 2015. Inferring causal impact using Bayesian structural time-series models. (2015).

[8] G Ann Campbell and Patroklos P Papapetrou. 2013. *SonarQube in action*. Manning Publications Co.

[9] Tristan Coignion, Clément Quinton, and Romain Rouvoy. 2024. A performance study of llm-generated code on leetcode. In *Proceedings of the 28th international conference on evaluation and assessment in software engineering*. 79–89.

[10] 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? *Journal of Systems and Software* 203 (2023), 111734.

[11] Spandan Garg, Roshanak Zilouchian Moghaddam, and Neel Sundaresan. 2023. Rapgen: An approach for fixing code inefficiencies in zero-shot. *arXiv preprint arXiv:2306.17077* (2023).

[12] Jinyong Hahn, Petra Todd, and Wilbert Van der Klaauw. 2001. Identification and estimation of treatment effects with a regression-discontinuity design. *Econometrica* 69, 1 (2001), 201–209.

[13] Wenpin Hou and Zhicheng Ji. 2024. A systematic evaluation of large language models for generating programming code. *arXiv preprint arXiv:2403.00894* (2024).

[14] Saki Imai. 2022. Is github copilot a substitute for human pair-programming? an empirical study. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*. 319–321.

[15] Kush Jain and Claire Le Goues. 2025. Testforge: Feedback-driven, agentic test suite generation. *arXiv preprint arXiv:2503.14713* (2025).

[16] MM Lehman and FN Parr. 1976. Program evolution and its impact on software engineering. In *Proceedings of the 2nd international conference on Software engineering*. 350–357.

[17] Meir M Lehman. 1979. On understanding laws, evolution, and conservation in the large-program life cycle. *Journal of Systems and Software* 1 (1979), 213–221.

[18] Stanislav Levin and Amiram Yehudai. 2017. The co-evolution of test maintenance and code maintenance through the lens of fine-grained semantic changes. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 35–46.

[19] 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 *2024 IEEE 35th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 216–227.

[20] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems* 36 (2023), 21558–21572.

[21] Cosmin Marsavina, Daniele Romano, and Andy Zaidman. 2014. Studying fine-grained co-evolution patterns of production and test code. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*. IEEE, 195–204.

[22] Tom Mens, Michel Wermelinger, Stéphane Ducasse, Serge Demeyer, Robert Hirschfeld, and Mehdi Jazayeri. 2005. Challenges in software evolution. In *Eighth International Workshop on Principles of Software Evolution (IWPSE'05)*. IEEE, 13–22.

[23] Charles Miranda, Guilherme Avelino, and Pedro Santos Neto. 2025. Highlight Test Code: Visualizing the Co-Evolution of Test and Production Code in Software Repositories. In *Simpósio Brasileiro de Engenharia de Software (SBES)*. SBC, 893–899.

[24] Charles Miranda, Guilherme Avelino, and Pedro Santos Neto. 2025. Test Co-Evolution in Software Projects: A Large-Scale Empirical Study. *Journal of Software: Evolution and Process* 37, 7 (2025), e70035.

[25] Glenford J Myers, Tom Badgett, Todd M Thomas, and Corey Sandler. 2004. *The art of software testing*. Vol. 2. Wiley Online Library.

[26] Nhan Nguyen and Sarah Nadi. 2022. An empirical evaluation of GitHub copilot's code suggestions. In *Proceedings of the 19th International Conference on Mining Software Repositories*. 1–5.

[27] Rangeet Pan, Myeongsoo Kim, Rahul Krishna, Raju Pavuluri, and Saurabh Sinha. 2025. Aster: Natural and multi-language unit test generation with llms. In *2025 IEEE/ACM 47th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 413–424.

[28] Robert B Penfold and Fang Zhang. 2013. Use of interrupted time series analysis in evaluating health care quality improvements. *Academic pediatrics* 13, 6 (2013), S38–S44.

[29] Alan Peslak and Lisa Kovalchick. 2024. AI For coders: An analysis of the usage of ChatGPT and GitHub CoPilot. *Issues in Information Systems* 25, 4 (2024), 252–260.

[30] Gustavo Sandoval, Hammond Pearce, Teo Nys, Ramesh Karri, Brendan Dolan-Gavitt, and Siddharth Garg. 2022. Security implications of large language model code assistants: A user study. *arXiv preprint arXiv:2208.09727* (2022).

[31] Advait Sarkar, Andrew D Gordon, Carina Negreanu, Christian Poelitz, Sruti Srinivasa Ragavan, and Ben Zorn. 2022. What is it like to program with artificial intelligence? *arXiv preprint arXiv:2208.06213* (2022).

[32] Anshul Shah, Anya Chernova, Elena Tomson, Leo Porter, William G Griswold, and Adalbert Gerald Soosai Raj. 2025. Students' Use of GitHub Copilot for Working with Large Code Bases. In *Proceedings of the 56th ACM Technical Symposium on Computer Science Education V. 1*. 1050–1056.

[33] Samiha Shimmi and Mona Rahimi. 2022. Leveraging code-test co-evolution patterns for automated test case recommendation. In *Proceedings of the 3rd ACM/IEEE International Conference on Automation of Software Test*. 65–76.

[34] Ian Sommerville. 2011. Software engineering 9th Edition. *ISBN-10* 137035152 (2011), 18.

[35] Sinan Wang, Ming Wen, Yepang Liu, Ying Wang, and Rongxin Wu. 2021. Understanding and facilitating the co-evolution of production and test code. In *2021 IEEE international conference on software analysis, evolution and reengineering (SANER)*. IEEE, 272–283.

[36] Burak Yetistiren, Isik Ozsoy, and Eray Tuzun. 2022. Assessing the quality of GitHub copilot's code generation. In *Proceedings of the 18th international conference on predictive models and data analytics in software engineering*. 62–71.

[37] Andy Zaidman, Bart Van Rompaey, Arie van Deursen, and Serge Demeyer. 2008. Mining software repositories to study co-evolution of production & test code. (2008), 220–229.

[38] Andy Zaidman, Bart Van Rompaey, Arie van Deursen, and Serge Demeyer. 2011. Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. *Empirical Software Engineering* 16, 3 (2011), 325–364.

[39] Beiqi Zhang, Peng Liang, Xiyu Zhou, Aakash Ahmad, and Muhammad Waseem. 2023. Practices and challenges of using github copilot: An empirical study. *arXiv preprint arXiv:2303.08733* (2023).

[40] Yangyang Zhao, Alexander Serebrenik, Yuming Zhou, Vladimir Filkov, and Bogdan Vasilescu. 2017. The impact of continuous integration on other software development practices: a large-scale empirical study. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 60–71.

[41] Albert Ziegler, Eirini Kalliamvakou, X Alice Li, Andrew Rice, Devon Rifkin, Shawn Simister, Ganesh Sittampalam, and Edward Aftandilian. 2022. Productivity assessment of neural code completion. In *Proceedings of the 6th ACM SIGPLAN international symposium on machine programming*. 21–29.