

MEASURING THE IMPACT OF GENERATIVE AI CODING ASSISTANTS ON SOFTWARE QUALITY, DEVELOPER PRODUCTIVITY, AND TECHNICAL DEBT

Wafa Hassan^{*1}, Anees Kumar Abbasi², Nazia Azim³, Asiya Ali Khan⁴, Hameed Hussain⁵

^{*1}Undergraduate Student, BS Information Technology, Department of Software Engineering, National University of Modern Languages, Islamabad

²Department of Computer Sciences & Information Technology, Women University of AJ&K Bagh

³Lecturer department of computer science Abdul Wali Khan University Mardan

⁴Bachelors in Electrical Engineering, School of Electrical Engineering and Computer Science (SEECs), National University of Sciences and Technology (NUST)

⁵Department of Computer Science, University of Buner, 19290, Khyber Pakhtunkhwa, Pakistan

¹wafahassanxyz@gmail.com, ²aneesqumar@wuajk.edu.pk, ³n.azim@awkum.edu.pk, ⁴aalikhan.bee22seecs@seecs.edu.pk⁴, ⁵dr.hameed@ubuner.edu.pk⁵

DOI: <https://doi.org/10.5281/zenodo.23052907>

Keywords

generative AI; coding assistants; GitHub Copilot; developer productivity; software quality; technical debt; maintainability; software security; empirical software engineering

Article History

Received: 01 June 2026

Accepted: 23 August 2026

Published: 30 September 2026

Copyright @Author

Corresponding Author: *

Wafa Hassan

Abstract

Generative artificial intelligence (AI) coding assistants are now embedded in professional software-development workflows, yet evidence about their engineering value remains fragmented across productivity experiments, code-quality trials, security studies, and repository mining. This article integrates recent empirical evidence to evaluate three connected outcomes: developer productivity, software quality, and technical debt. The evidence base includes controlled and field experiments involving professional developers, code-quality studies using automated tests and blinded review, security analyses of AI-generated code, and a 2026 large-scale study of approximately 302,600 verified AI-authored commits from 6,299 GitHub repositories. Productivity effects are heterogeneous: a controlled experiment reported 55.8% faster task completion with GitHub Copilot [1], three field experiments covering 4,867 developers reported a 26.08% increase in completed tasks [2], and an enterprise randomized trial estimated approximately 21% less time on task [3], whereas a randomized study of experienced open-source developers found that allowing frontier AI tools increased completion time by 19% [4]. Quality evidence is similarly multidimensional. A 2024 randomized study of 202 valid submissions found higher functional success and small but statistically significant improvements in readability, reliability, maintainability, and conciseness with Copilot [5], while security studies continue to identify important weaknesses in AI-generated code [6–8]. At repository scale, AI-authored commits can introduce persistent maintenance liabilities: 89.3% of identified AI-introduced issues were code smells and 22.7% of tracked issues remained at the latest revision [9]. The synthesis shows that generative AI does not produce a uniform productivity-quality trade-off. Its effects depend

on task context, developer expertise, evaluation horizon, and verification practices. Lifecycle-adjusted productivity provides a broader organizational measure by incorporating review, rework, security remediation, and technical-debt costs.

INTRODUCTION

Generative AI coding assistants have shifted from experimental autocomplete systems to interactive development tools capable of generating functions, tests, documentation, refactorings, explanations, and multi-file changes. Their rapid adoption has created a measurement problem for software engineering. An assistant may reduce the time required to produce an initial implementation while simultaneously changing review effort, defect risk, security exposure, code ownership, and future maintenance cost. Consequently, lines of code, suggestion acceptance, or elapsed implementation time alone cannot establish whether AI improves engineering performance.

Early causal evidence emphasized speed. Peng et al. reported that developers with GitHub Copilot completed a JavaScript HTTP-server task 55.8% faster than the control group [1]. More recent field evidence has broadened this result: Cui et al. combined randomized trials at Microsoft, Accenture, and another Fortune 100 company and estimated a 26.08% increase in completed tasks across 4,867 developers [2]. An enterprise randomized controlled trial with Google engineers estimated approximately 21% less time on a complex task [3]. However, the direction is not universal. Becker et al. randomized 246 tasks completed by experienced open-source developers working in mature repositories and found that allowing early-2025 AI tools increased completion time by 19% [4]. These results imply that the relevant question is not whether AI is inherently productive, but under which development conditions it changes total engineering effort.

Quality and maintainability evidence is equally nuanced. GitHub's randomized code-quality study recruited experienced Python developers and retained 202 valid submissions. Copilot access

increased the likelihood of passing all ten unit tests and produced small statistically significant improvements in blinded ratings of readability, reliability, maintainability, and conciseness [5]. At the same time, empirical security studies have identified insecure AI-generated snippets and broad Common Weakness Enumeration categories [6–8]. A 2026 study of downstream maintainability found no significant difference in later evolution time or code quality when new developers modified code initially produced with or without AI assistance [10].

Technical debt is therefore central to evaluating AI-assisted development. Technical debt captures technical choices that accelerate present delivery while creating future remediation or maintenance costs [11–14]. AI can reduce debt by generating tests, documentation, standard patterns, and refactorings; it can increase debt when rapid code generation produces duplicated logic, code smells, weak abstractions, insecure patterns, or code that developers do not fully understand. A 2026 repository-scale study provides direct evidence of this risk, identifying 484,366 distinct issues introduced by verified AI-authored commits and showing that a meaningful share persisted over time [9].

Taken together, these evidence streams frame AI coding-assistant performance as a lifecycle problem. The analysis makes three related contributions: it connects productivity, quality, and technical debt rather than treating them as separate outcomes; it distinguishes short-task experimental effects from enterprise and repository-scale evidence; and it develops a lifecycle-adjusted interpretation in which review, rework, remediation, and maintainability costs are part of productivity rather than external considerations.

2. Literature Review and Hypotheses

2.1 AI-Assisted Developer Productivity

Developer productivity is multidimensional. The SPACE framework cautions against reducing productivity to a single activity metric and emphasizes satisfaction, performance, activity, communication, and efficiency [15]. This warning is especially relevant to generative AI because assistance can increase output while changing cognitive load, review behavior, or understanding of the resulting code. Controlled and field experiments now provide stronger causal evidence than early perception surveys, but effect sizes differ substantially across contexts [1–4]. Less-experienced developers have shown larger gains in some field settings [2], while experienced maintainers working in familiar mature repositories may face verification and interaction overheads that offset generation speed [4].

H1: Generative AI coding assistance is associated with higher short-run developer productivity on bounded implementation tasks, but the magnitude and direction of the effect are moderated by task context and developer experience.

2.2 Software Quality, Correctness, and Security

Software quality cannot be represented by functional correctness alone. ISO/IEC 25010 treats quality as a family of characteristics that includes functional suitability, reliability, security, maintainability, performance efficiency, compatibility, usability, flexibility, and safety [16]. For source-code research, automated tests, static-analysis findings, complexity, duplication, review outcomes, and security weaknesses provide practical operational measures. Comparative code-generation studies have found substantial variation in correctness and technical-debt estimates across assistants [8,17,23], while studies

of Copilot suggestions show accuracy differences across languages and problem types [18].

Security deserves separate treatment because code may pass functional tests while remaining exploitable. Pearce et al. demonstrated security weaknesses in Copilot-generated solutions [6]. Fu et al. subsequently analyzed AI-generated snippets in GitHub projects and found security weaknesses affecting 29.5% of Python and 24.2% of JavaScript snippets in their sample, spanning 43 CWE categories [7]. These findings show why quality evaluation must combine functional and security-oriented measures.

H2: AI-assisted code quality is heterogeneous across functional, readability, maintainability, and security dimensions rather than uniformly better or worse than conventionally produced code.

2.3 Technical Debt and Code Ownership

The technical-debt metaphor originated with Cunningham and has developed into a broad research area covering code, design, architecture, tests, documentation, infrastructure, and other maintainability liabilities [11–14]. Static-analysis proxies such as code smells, duplication, complexity, reliability findings, and estimated remediation effort are imperfect but reproducible indicators of code-level debt. AI adds a further human factor: developers may accept generated code without developing the same degree of code ownership. A 2026 experiment found more complete task solutions with AI but a 12.5% reduction in participants' ability to answer technical questions about code they had implemented [19].

H3: The estimated engineering benefit of AI assistance differs when downstream quality-assurance, rework, remediation, code-comprehension, and persistent technical-debt costs are incorporated into the evaluation rather than measuring initial development effort alone.

2.4 Conceptual Framework

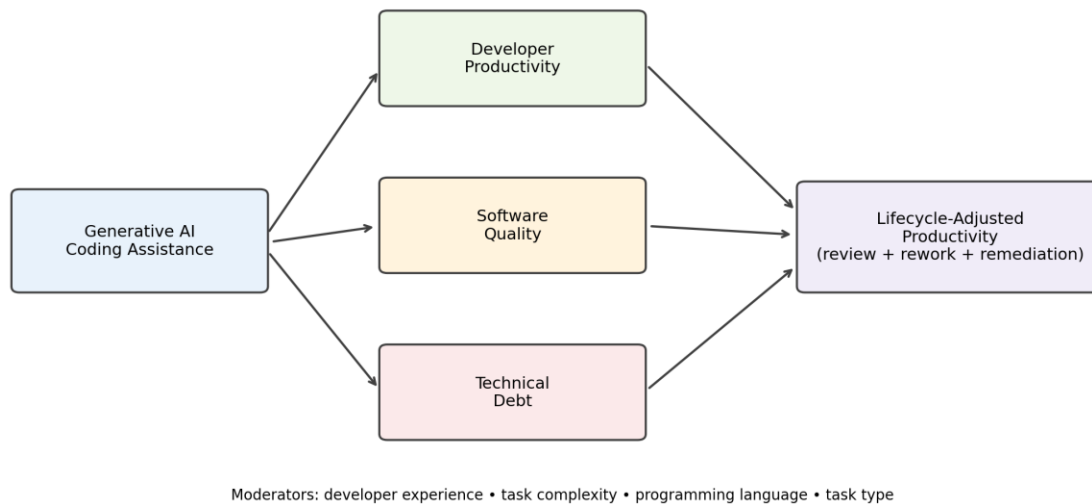


Figure 1. Integrated conceptual framework linking generative AI assistance to productivity, software quality, technical debt, and lifecycle-adjusted productivity.

Figure 1 formalizes the central argument. AI assistance can affect productivity, quality, and debt directly, while quality defects and technical debt generate downstream review, rework, and remediation. Developer experience, task complexity, language, and task type moderate these relationships. This structure avoids assuming that a faster first implementation is necessarily a lower-cost software outcome.

3. Methodology

3.1 Study Design and Evidence Base

The article uses an structured empirical evidence-synthesis design. The analysis synthesizes published quantitative evidence from studies that identify AI-assisted and non-assisted development through randomized assignment or verified AI authorship. The core evidence was selected to represent three levels of analysis: developer/task experiments, code/submission quality studies, and repository/commit-level technical-debt studies. Because outcome definitions differ, results are not mechanically pooled into a single meta-analytic effect. Instead, effect estimates are

compared within their original measurement scale and interpreted according to design and population.

3.2 Evidence Identification and Eligibility

Evidence was identified through a structured targeted search completed on 26 September 2026. Searches combined terms for generative AI coding assistants (for example, GitHub Copilot, AI coding assistant, generative AI and software development) with outcome terms covering developer productivity, completion time, software quality, correctness, maintainability, security, code ownership, code smells, and technical debt. Publisher and scholarly sources were prioritized, including ACM Digital Library, IEEE Xplore, SpringerLink, journal publisher pages, and arXiv for recent studies not yet available in a final journal version. Vendor research was retained only when it reported a clearly described empirical design and was treated as source-sensitive evidence rather than as equivalent to independent peer-reviewed evidence.

Studies were eligible when they (1) empirically compared AI-assisted with non-assisted

development, evaluated AI-generated code using reproducible quality or security measures, or identified AI-authored repository changes; (2) reported quantitative outcomes relevant to productivity, software quality, maintainability, security, code ownership, or technical debt; and (3) provided sufficient methodological detail to interpret the reported result. Purely conceptual articles, opinion pieces, marketing claims without an empirical design, and studies lacking a relevant engineering outcome were excluded from the core evidence table. Foundational software-engineering

sources were retained to define productivity, quality, technical debt, complexity, and repository-mining limitations.

This procedure is a structured evidence synthesis rather than a claim of exhaustive systematic-review coverage. No pooled meta-analysis was attempted because the included studies use materially different populations, tasks, treatments, and outcome scales. The synthesis therefore emphasizes design-aware triangulation, direction of effect, reported uncertainty, and consistency across evidence types.

3.3 Outcome Operationalization

Table 1. Core empirical evidence used in the integrated analysis.

Study	Design	Sample	Primary outcome	Exposure
Peng et al. [1]	Controlled experiment	95 professional developers	Task completion time / success	GitHub Copilot vs control
Cui et al. [2]	Three randomized field experiments	4,867 developers	Completed tasks	AI coding assistant access
Google enterprise RCT [3]	Randomized controlled trial	96 full-time engineers	Time on complex task	AI features vs control
Becker et al. [4]	Randomized task-level trial	16 experienced OSS developers; 246 tasks	Completion time	AI allowed vs disallowed
GitHub quality RCT [5]	Randomized trial + blind review	202 valid submissions; 1,293 reviews	Tests and quality ratings	Copilot vs control
Fu et al. [7]	Repository empirical study	733 AI-generated snippets	Security weaknesses	Static analysis
Liu et al. [9]	Large-scale repository study	approximately 302,600 AI-authored commits; 6,299 repositories	Introduced/persistent issues	Verified AI-authored commits
Echoes of AI [10]	Preregistered two-phase experiment	151 participants; 95% professionals	Downstream evolution time/quality	Prior code with vs without AI
More Code, Less Understanding [19]	Crossover experiment	69 participants	Completeness and code ownership	AI vs no AI

As summarized in Table 1, the evidence base spans controlled experiments, randomized field trials, blinded code-quality review, security analysis, repository mining, and downstream-

maintainability experiments. This breadth allows the synthesis to compare immediate developer outcomes with later quality and maintenance

consequences while preserving the design and population of each source study.

Table 2. Measurement framework for productivity, quality, and technical debt.

Domain	Operational indicators	Interpretation
Developer productivity	Completion time; completed tasks; task completeness	Higher output or lower time, interpreted in study-specific units
Functional quality	Unit-test success; correctness	Automated behavioral validation
Review quality	Readability; reliability; maintainability; conciseness; approval	Blind or structured review
Security	CWE/static-analysis findings	Security weaknesses by language/type
Technical debt	Code smells; bugs; security issues; persistence	Issues introduced and surviving over repository history
Code ownership	Technical comprehension of authored code	Ability to explain/answer questions about implementation
Lifecycle-adjusted productivity	Initial effort plus review, rework and remediation	Conceptual organizational outcome

Table 2 shows how the three principal outcome domains are operationalized across the included studies. It also distinguishes immediate productivity and functional quality from security, technical debt, code ownership, and the broader lifecycle-adjusted productivity construct used in the synthesis.

3.4 Statistical Interpretation

Where the underlying studies report randomized estimates, this article preserves the reported treatment contrast, confidence information, and p-values rather than recalculating participant-level models without raw data. Heterogeneity is assessed descriptively across populations and task settings. Quality outcomes from the GitHub randomized study are presented as reported mean differences. Repository debt is summarized using issue composition and persistence. Because these outcomes are measured on different scales, they are interpreted separately rather than combined into a common effect estimate. Interpretation of controlled evidence follows standard empirical software-engineering principles concerning treatment contrasts, validity, and uncertainty [40].

A mixed-effects specification with developer and task random intercepts provides an appropriate basis for subsequent replication studies, particularly when treatment effects are allowed to vary by experience and task complexity. Confidence intervals, multiplicity adjustment for secondary outcomes, and intention-to-treat and per-protocol sensitivity analyses would further strengthen inference. Technical debt should be operationalized using reproducible static-analysis rules such as code smells, duplicated code, cyclomatic or cognitive complexity, reliability/security findings, and estimated remediation effort.

4. Results

4.1 Developer Productivity

The productivity evidence supports substantial heterogeneity rather than a single universal effect. Peng et al. found 55.8% faster completion in a bounded JavaScript task [1]. Cui et al. reported a 26.08% increase in completed tasks when pooling three field experiments involving 4,867 developers, with larger gains among less-experienced developers [2]. The Google enterprise randomized trial estimated approximately 21%

less time on task [3]. In contrast, Becker et al. found that experienced open-source developers working on mature repositories took 19% longer

when AI tools were allowed [4]. Figure 2 displays these effects without pooling them because the outcome definitions differ.

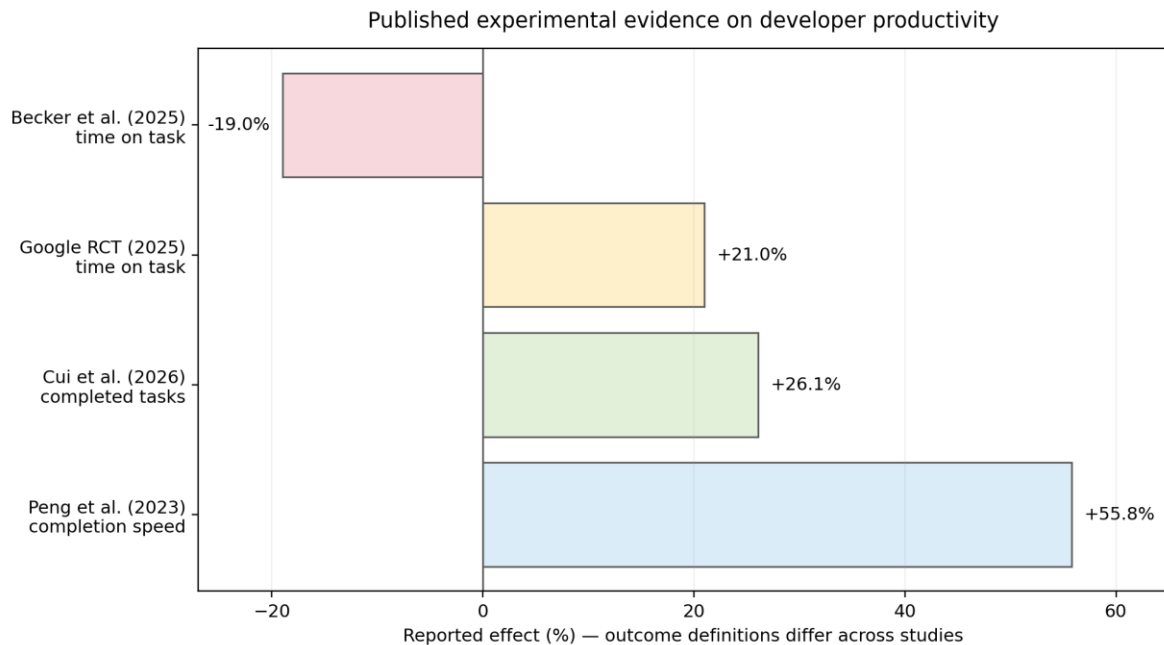


Figure 2. Reported experimental productivity effects. Positive values indicate reported improvement; the studies use different productivity metrics and are therefore not pooled [1–4].

Table 3. Productivity findings across controlled and field settings.

Evidence	Population	Reported effect	Statistical note
Peng et al. [1]	95 professional developers	55.8% faster completion	$p = .0017$; reported 95% CI for speed gain 21%–89%
Cui et al. [2]	4,867 developers	26.08% increase in completed tasks	SE = 10.3%
Google RCT [3]	96 engineers	~21% reduction in time on task	Confidence interval reported as wide
Becker et al. [4]	246 tasks; 16 experienced OSS developers	19% increase in completion time	AI slowed work in this context

Table 3 places the four productivity estimates alongside their study populations and statistical notes. Read together with Figure 2, it shows that favorable effects dominate several bounded and enterprise settings, while the experienced open-source-maintainer study reports a slowdown.

The evidence is consistent with H1, although the effect is strongly conditional on context. AI assistance frequently improves short-run productivity, but expertise and task ecology materially change the effect. The negative result among experienced maintainers is particularly important because it shows that generation speed

can be outweighed by prompting, verification, and integration costs in familiar, complex codebases.

4.2 Software Quality

The strongest randomized quality evidence is favorable but modest. In the GitHub study, Copilot users had a 53.2% greater likelihood of

passing all ten unit tests. Blind reviews also reported mean improvements of 3.62% in readability, 2.94% in reliability, 2.47% in maintainability, and 4.16% in conciseness; reported p-values were .003, .01, .041, and .002, respectively [5]. Reviewers were also 5% more likely to approve Copilot-authored code [5].

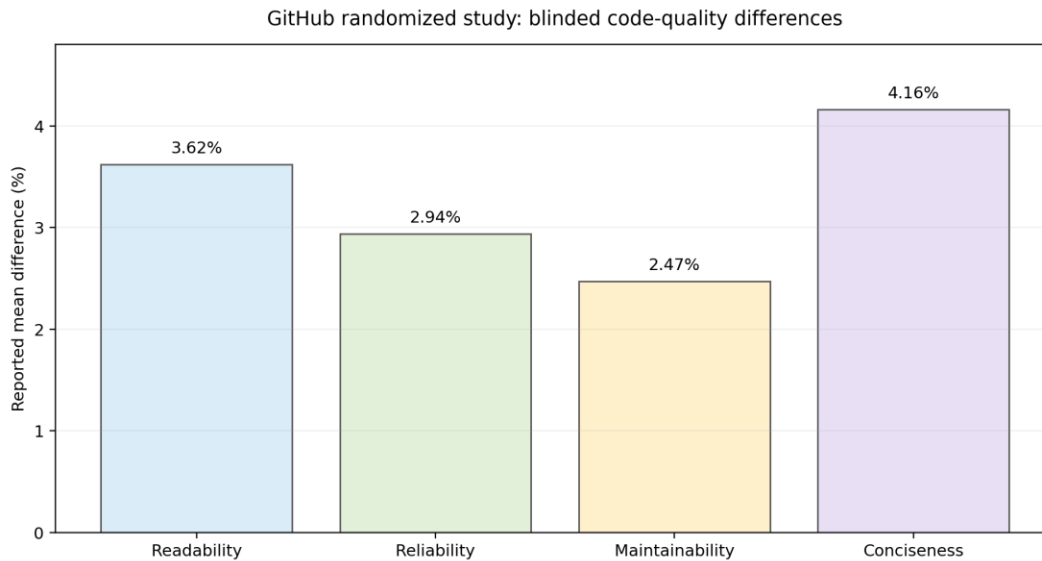


Figure 3. Reported mean quality differences in GitHub's randomized code-quality study [5].

Figure 3 visualizes the four blinded review dimensions for which the GitHub randomized study reported

mean percentage improvements with Copilot: readability, reliability, maintainability, and conciseness.

Table 4. Randomized evidence on functional and review-based code quality [5].

Quality outcome	Reported Copilot difference	Reported significance
Functional success	53.2% greater likelihood of passing all 10 unit tests	$p < .01$
Readability	+3.62%	$p = .003$
Reliability	+2.94%	$p = .010$
Maintainability	+2.47%	$p = .041$
Conciseness	+4.16%	$p = .002$
Approval	5% more likely to approve	$p = .014$

As detailed in Table 4, the randomized quality evidence combines functional test success with reviewer-assessed dimensions and approval. The table also makes clear that the reported

improvements are modest in magnitude even where statistical significance was reached. These findings do not eliminate quality risk. Security analyses show that functional code may remain vulnerable. Fu et al. found security

weaknesses in 29.5% of Python and 22.7% of JavaScript AI-generated snippets in their repository sample, spanning 43 CWE categories [7]. Earlier controlled security work likewise showed that Copilot can produce vulnerable code [6]. These findings are consistent with H2: quality effects are multidimensional, and improvements in readability or test success should not be generalized automatically to security.

4.3 Technical Debt and Downstream Maintainability

Repository-scale evidence indicates that AI-generated changes can create persistent

maintenance liabilities. Liu et al. analyzed approximately 302,600 verified AI-authored commits from 6,299 repositories and identified 484,366 distinct issues introduced by those commits [9]. Code smells represented 89.3% of the identified issues, and more than 15% of commits from every studied AI assistant introduced at least one issue. Of the tracked AI-introduced issues, 22.7% remained at the latest repository revision [9].

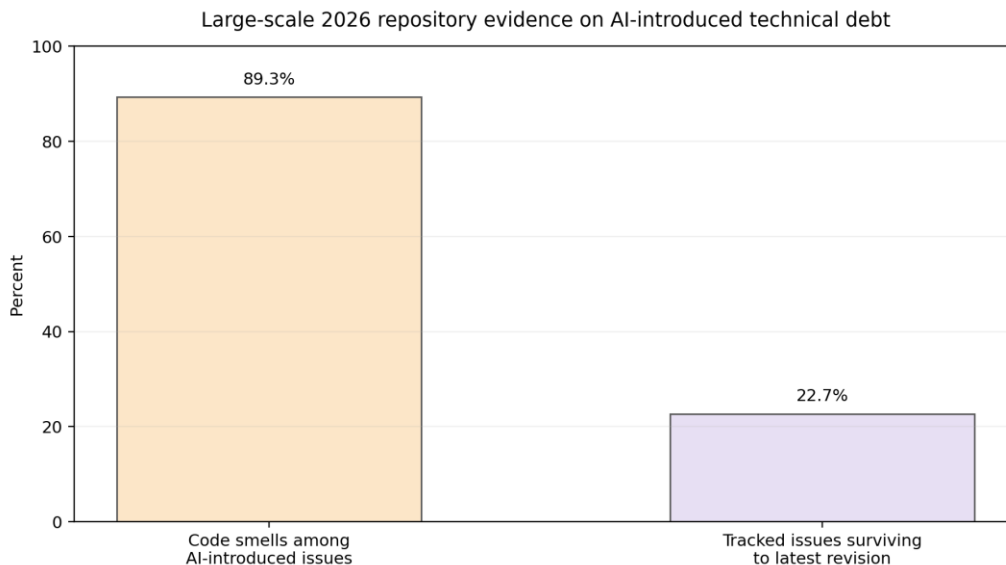


Figure 4. Technical-debt indicators reported in the 2026 large-scale study of verified AI-authored commits [9].

Figure 4 highlights two repository-level indicators from the 2026 technical-debt study: code smells account for 89.3% of identified AI-introduced

issues, while 22.7% of tracked issues remain at the latest repository revision.

Table 5. Large-scale evidence on AI-introduced issues and persistence [9].

Repository evidence	Reported value
Verified AI-authored commits	302,600
GitHub repositories	6,299
Distinct introduced issues	484,366
Code smells among issues	89.3%
Commits introducing ≥ 1 issue	>15% for every studied assistant

Tracked issues surviving to latest revision

22.7%

Table 5 provides the scale and persistence context behind Figure 4, including the number of verified AI-authored commits, repositories, distinct introduced issues, and the share of commits associated with at least one identified issue.

The downstream-maintainability experiment provides an important counterpoint. In a preregistered two-phase study involving 151 participants, later developers who evolved code initially produced with AI did not show significant differences in completion time or code quality,

and Bayesian analysis suggested that any downstream speed or quality benefits were at most small and uncertain [10]. Thus, AI-introduced static-analysis debt does not necessarily translate into a measurable maintenance penalty in every bounded evolution task. The evidence is consistent with H3 as a measurement proposition rather than a universal directional effect; downstream costs vary across settings and therefore require direct measurement.

4.4 Integrated Robustness and Heterogeneity Assessment

Table 6. Cross-study robustness and heterogeneity assessment.

Robustness dimension	Evidence variation	Implication
Population	Professional developers, enterprise engineers, experienced OSS maintainers, mixed-experience participants	Effects are not confined to student samples.
Task setting	Bounded greenfield tasks vs mature repository work	Productivity direction changes across settings.
Outcome definition	Time, completed tasks, tests, review quality, security, debt persistence	No single metric captures engineering value.
Experience	Larger gains for less-experienced developers in field experiments; slower result among highly experienced OSS maintainers	Experience is a plausible moderator.
Evaluation horizon	Immediate completion vs later code evolution vs repository persistence	Short-run speed can diverge from lifecycle outcomes.
Evidence source	Academic experiments, enterprise trials, repository studies, vendor RCT	Triangulation strengthens inference but vendor evidence requires source-aware interpretation.

Table 6 synthesizes the principal sources of heterogeneity across the evidence base. In particular, differences in population, task ecology, outcome definition, developer experience, evaluation horizon, and evidence source help explain why the reported effects should not be treated as a single universal AI effect.

5. Discussion

Across the included studies, two simple interpretations are not sustained: that AI coding assistants automatically make developers faster, and that faster AI-assisted coding necessarily lowers software quality. The observed effects are conditional on development context. AI assistance can produce large speed gains in bounded tasks

and meaningful throughput gains in enterprise settings [1–3], but those gains can disappear or reverse when experienced developers work on complex repositories they already know well [4]. This heterogeneity is theoretically plausible because AI changes both production and verification costs.

The quality results are similarly conditional. Randomized evidence demonstrates that AI-assisted code can perform better on tests and receive slightly higher blinded quality ratings [5]. Yet security studies show that generated code can contain important weaknesses [6–8]. Accordingly, software quality is better evaluated as a set of distinct dimensions rather than as a single undifferentiated outcome. Functional correctness, readability, maintainability, and security can move in different directions.

Technical debt provides the missing temporal dimension. The large-scale repository evidence shows that AI-authored commits can introduce code smells, bugs, and security issues that persist [9]. However, the preregistered maintainability experiment found no significant downstream penalty in a bounded evolution task [10]. Together, these findings suggest that static debt indicators and realized maintenance cost are related but not interchangeable. Both static debt indicators and realized maintenance costs therefore warrant measurement in organizational evaluations.

A further concern is code ownership. The 2026 experiment reporting lower technical comprehension after AI-assisted coding suggests that higher task completeness can coexist with weaker understanding of the implementation [19]. This matters because comprehension affects debugging, review, incident response, and long-term maintenance even when the initial code is correct. The result strengthens the case for lifecycle-adjusted productivity: engineering value should include the effort required to understand, verify, repair, and evolve code.

5.1 Theoretical Implications

The findings extend productivity theory in software engineering by positioning generative AI as a context-dependent complement rather than a uniform labor-saving technology. The SPACE framework [15] is useful because it separates activity from performance and efficiency. AI may increase activity or throughput without proportionally improving performance, and it may improve immediate efficiency while creating later verification costs. Technical-debt theory adds the intertemporal mechanism: short-term acceleration can create liabilities whose costs are realized later [11–14].

5.2 Practical Implications

Organizations evaluating coding assistants should avoid using suggestion acceptance, generated lines, or self-reported time savings as standalone success metrics. A defensible dashboard should combine cycle time or completed work with test outcomes, review effort, escaped defects, security findings, duplication, code smells, remediation effort, and rework. Teams should stratify results by developer experience and task type because aggregate averages can conceal groups for whom AI creates additional overhead. Security-sensitive code should retain independent static analysis and human review even when functional tests pass.

6. Limitations and Future Research

This article integrates heterogeneous studies rather than reanalyzing one common participant-level dataset. The structured search was designed to identify the most relevant empirical evidence streams, but it was not an exhaustive systematic review and therefore remains vulnerable to search and publication-selection effects. The included studies differ in tools, model generations, languages, tasks, populations, and outcome definitions; consequently, the reported percentages should not be interpreted as directly comparable effect sizes or pooled into a single estimate. Several prominent quality results come from vendor-sponsored research, which increases the importance of independent replication.

Repository studies also depend on the validity of AI-authorship identification and static-analysis rules. Finally, AI systems evolve rapidly, so results are partly version- and time-specific.

Future research should preregister multi-site experiments that measure productivity, functional quality, security, code ownership, and technical debt in the same participants and repositories. Longitudinal designs should track AI-assisted changes for months rather than hours, while causal mediation models can test whether quality and debt explain differences between immediate and lifecycle productivity. Replication packages should preserve prompts, model versions, IDE settings, task specifications, test suites, static-analysis configurations, and anonymized interaction logs.

7. Conclusion

Generative AI coding assistants can improve software-development productivity, but the effect is neither universal nor sufficient to establish engineering value. Across controlled and field experiments, reported productivity effects range from substantial acceleration to measurable slowdown depending on developer expertise and task context. Randomized evidence shows modest improvements in several code-quality dimensions, while security and repository studies demonstrate that AI-generated code can also introduce weaknesses and persistent technical debt. Taken together, the findings emphasize a methodological point: organizations and researchers should measure AI-assisted development across the full lifecycle. Productivity should be adjusted for review, rework, remediation, comprehension, and maintenance rather than inferred from initial coding speed alone. From this lifecycle perspective, generative AI functions as a context-sensitive engineering intervention whose benefits depend on how effectively teams verify and maintain the code it helps produce.

REFERENCES

- [1] S. Peng, E. Kalliamvakou, P. Cihon, and M. Demirer, "The Impact of AI on Developer Productivity: Evidence from GitHub Copilot," arXiv:2302.06590, 2023. doi:10.48550/arXiv.2302.06590.
- [2] K. Z. Cui, M. Demirer, S. Jaffe, L. Musolff, S. Peng, and T. Salz, "The Effects of Generative AI on High-Skilled Work: Evidence from Three Field Experiments with Software Developers," *Management Science*, published online 2026. doi:10.1287/mnsc.2025.00535.
- [3] N. Forsgren et al., "How Much Does AI Impact Development Speed? An Enterprise-Based Randomized Controlled Trial," 2025 IEEE/ACM 47th International Conference on Software Engineering: Software Engineering in Practice, 2025. doi:10.1109/ICSE-SEIP66354.2025.00060.
- [4] J. Becker, N. Rush, E. Barnes, and D. Rein, "Measuring the Impact of Early-2025 AI on Experienced Open-Source Developer Productivity," arXiv:2507.09089, 2025. doi:10.48550/arXiv.2507.09089.
- [5] J. Bauer, "Does GitHub Copilot improve code quality? Here's what the data says," GitHub Research, 2024, updated 2025.
- [6] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, "Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions," 2022 IEEE Symposium on Security and Privacy, 2022. doi:10.1109/SP46214.2022.9833571.
- [7] Y. Fu, P. Liang, A. Tahir, Z. Li, M. Shahin, J. Yu, and J. Chen, "Security Weaknesses of Copilot-Generated Code in GitHub Projects: An Empirical Study," *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 8, article 218, 2025. doi:10.1145/3716848.

- [8] B. Yetiştiren, I. Özsoy, M. Ayerdem, and E. Tüzün, "Evaluating the Code Quality of AI-Assisted Code Generation Tools: An Empirical Study on GitHub Copilot, Amazon CodeWhisperer, and ChatGPT," *arXiv:2304.10778*, 2023.
- [9] Y. Liu, R. Widyasari, Y. Zhao, I. C. Irsan, J. Chen, and D. Lo, "Debt Behind the AI Boom: A Large-Scale Empirical Study of AI-Generated Code in the Wild," *arXiv:2603.28592*, 2026.
- [10] M. Borg, D. Hewett, N. Hagatulah, N. Couderc, E. Söderberg, D. Graham, U. Kini, and D. Farley, "Echoes of AI: Investigating the Downstream Effects of AI Assistants on Software Maintainability," *Empirical Software Engineering*, vol. 31, article 161, 2026. doi:10.1007/s10664-026-10889-1.
- [11] W. Cunningham, "The WyCash Portfolio Management System," *OOPSLA '92 Experience Report*, 1992.
- [12] P. Kruchten, R. L. Nord, and I. Ozkaya, "Technical Debt: From Metaphor to Theory and Practice," *IEEE Software*, vol. 29, no. 6, pp. 18–21, 2012.
- [13] N. S. Rios, M. G. de Mendonça Neto, and R. O. Spínola, "A tertiary study on technical debt: Types, management strategies, research trends, and base information for practitioners," *Information and Software Technology*, vol. 102, pp. 117–145, 2018.
- [14] P. Avgeriou, P. Kruchten, I. Ozkaya, and C. Seaman, "Managing Technical Debt in Software Engineering," *Dagstuhl Reports*, 2016.
- [15] N. Forsgren, M.-A. Storey, C. Maddila, T. Zimmermann, B. Houck, and J. Butler, "The SPACE of Developer Productivity: There's more to it than you think," *ACM Queue*, vol. 19, no. 1, 2021.
- [16] ISO/IEC 25010:2023, *Systems and software engineering—Systems and software Quality Requirements and Evaluation (SQuaRE)—Product quality model*, 2023.
- [17] A. Moradi Dakhel, V. Majdinasab, A. Nikanjam, F. Khomh, M. C. Desmarais, and Z. M. (Jack) Jiang, "GitHub Copilot AI Pair Programmer: Asset or Liability?" *Journal of Systems and Software*, vol. 203, article 111734, 2023. doi:10.1016/j.jss.2023.111734.
- [18] R. Mo, D. Wang, W. Zhan, Y. Jiang, Y. Wang, Y. Zhao, Z. Li, and Y. Ma, "Assessing and Analyzing the Correctness of GitHub Copilot's Code Suggestions," *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 7, article 194, pp. 1–32, 2025. doi:10.1145/3715108.
- [19] A. Martin-Lopez, R. Tufano, E. Guglielmi, A. B. Sánchez, A. L. Sanz, S. Scalabrino, R. Oliveto, S. Segura, and G. Bavota, "More Code, Less Understanding? On the Impact of AI Assistants on Developers' Productivity and Code Ownership," *IEEE Transactions on Software Engineering*, vol. 52, no. 7, pp. 2173–2187, 2026. doi:10.1109/TSE.2026.3679627.
- [20] J. Butler, J. Suh, S. Haniyur, and C. Hadley, "Dear Diary: A Randomized Controlled Trial of Generative AI Coding Tools in the Workplace," *2025 IEEE/ACM 47th International Conference on Software Engineering: Software Engineering in Practice*, 2025. doi:10.1109/ICSE-SEIP66354.2025.00034.
- [21] GitHub, "Research: Quantifying GitHub Copilot's impact on developer productivity and happiness," *GitHub Research*, 2022.
- [22] GitHub, "Research: Quantifying GitHub Copilot's impact on code quality," *GitHub Research*, 2023.
- [23] B. Yetiştiren, I. Özsoy, and E. Tüzün, "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 (PROMISE 2022)*, pp. 62–71, 2022.

-
- [24] N. Nguyen and S. Nadi, "GitHub Copilot AI pair programmer: Asset or Liability?" *Journal of Systems and Software*, vol. 203, 111734, 2023. doi:10.1016/j.jss.2023.111734.
- [25] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, "The Promises and Perils of Mining GitHub," in *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR)*, pp. 92–101, 2014.
- [26] T. McCabe, "A Complexity Measure," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, 1976.
- [27] M. Fowler, *Refactoring: Improving the Design of Existing Code*, 2nd ed., Addison-Wesley, 2018.
- [28] R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*, Prentice Hall, 2008.
- [29] MITRE, "Common Weakness Enumeration (CWE)," software weakness classification resource.
- [30] SonarSource, "Clean Code and static analysis documentation," technical documentation.
- [31] GitHub, "CodeQL documentation," security analysis documentation.
- [32] Semgrep, "Semgrep documentation," static analysis documentation.
- [33] T. D. LaToza, G. Venolia, and R. DeLine, "Maintaining Mental Models: A Study of Developer Work Habits," *ICSE*, 2006.
- [34] M.-A. Storey, T. Zimmermann, C. Bird, J. Czerwinka, B. Murphy, and E. Kalliamvakou, "Towards a Theory of Software Developer Job Satisfaction and Perceived Productivity," *IEEE Transactions on Software Engineering*, 2019.
- [35] A. S. Meyer, T. Fritz, G. C. Murphy, and T. Zimmermann, "Software developers' perceptions of productivity," *FSE*, 2014.
- [36] B. Vasilescu, Y. Yu, H. Wang, P. T. Devanbu, and V. Filkov, "Quality and Productivity Outcomes Relating to Continuous Integration in GitHub," in *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, pp. 805–816, 2015. doi:10.1145/2786805.2786850.
- [37] D. Spinellis, "Global Software Development in the Era of GitHub," *IEEE Software*, 2012.
- [38] G. Gousios, M. Pinzger, and A. van Deursen, "An exploratory study of the pull-based software development model," *ICSE*, 2014.
- [39] M. Beller, G. Gousios, and A. Zaidman, "Oops, my tests broke the build: An explorative analysis of Travis CI with GitHub," *MSR*, 2017.
- [40] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering*. Berlin, Germany: Springer, 2012.