The Productivity Paradox of AI-Assisted Software Development: Empirical Evidence, Longitudinal Quality Degradation, and a Formal Cost-Benefit Model

Satish Chavali *

Posted Date: 29 June 2026

doi: 10.20944/preprints202606.2174.v1

Keywords: large language models; AI-assisted software development; GitHub Copilot; code quality; technical debt; software productivity; LLM code generation; maintainability; security vulnerabilities; software engineering

 doi:10.20944/preprints202606.2174.v1

Article

The Productivity Paradox of AI-Assisted Software Development: Empirical Evidence, Longitudinal Quality Degradation, and a Formal Cost-Benefit Model

Satish Chavali

Systems, Applications & Products in Data Processing (United States), Philadelphia, USA

* Correspondence: satish.chavali1985@gmail.com

Abstract

Large language model (LLM)-based coding assistants have achieved adoption rates unprecedented in the history of developer tooling, with over 62% of professional developers reporting active use as of 2024. The dominant narrative frames these tools as straightforward productivity multipliers, citing controlled task completion speedups of 55.8% in the most widely cited study. This paper examines whether that narrative survives contact with longitudinal production evidence and formal mathematical analysis. We present a cost-benefit model that captures both the velocity gain and the quality degradation trajectory of AI-assisted development, deriving a formal break-even expression that predicts when accumulated technical debt erases productivity gains. We then conduct a structured secondary analysis of three published industry cases — a longitudinal code quality study spanning 153 million lines of production code, a large-scale security evaluation of 1,692 AI-generated programs, and enterprise adoption survey evidence from over 2,000 professional developers — to validate the model's predictions against real data. Across all three cases we identify a consistent pattern: measurable short-term velocity gains accompanied by elevated code churn, increased duplication, and reproducible security vulnerability classes specific to LLM-generated code. We term this the AI-assisted productivity paradox and propose a formal governance framework for responsible tool integration. Our central finding is that the productivity case for LLM coding assistants is real but incomplete — standard metrics capture the benefit on a timescale of days to weeks while costs accumulate over months, creating a systematic measurement blind spot in most current adoption programs.

Keywords: large language models; AI-assisted software development; GitHub Copilot; code quality; technical debt; software productivity; LLM code generation; maintainability; security vulnerabilities; software engineering

1. Introduction

In September 2022, GitHub published results from a controlled experiment in which developers using Copilot completed a web server implementation task 55.8% faster than a control group [1]. The finding was statistically significant at $p = 0.0017$ and the headline number traveled widely across engineering communities worldwide. By 2024, the Stack Overflow Developer Survey reported that 62% of professional developers were actively using AI coding tools, up from 44% the previous year [2]. The adoption curve is among the steepest ever recorded for a developer tool category.

The productivity case for these tools' rests primarily on a body of evidence from controlled, time-bounded experiments. A developer completes a well-specified task in significantly less time when an AI assistant is available. The experimental conditions are carefully designed, the results are

 doi:10.20944/preprints202606.2174.v1

reproducible, and the statistical significance is not in doubt. What these experiments do not measure is what happens to the resulting code over the subsequent months of maintenance, extension, and incident response.

Evidence that something different does happen on that longer timescale has begun to accumulate. GitClear's longitudinal analysis of 153 million lines of production code written between 2020 and 2023 found that code churn — lines modified or deleted within two weeks of introduction — increased by 39.2% between 2022 and 2023, precisely the period of widespread AI assistant adoption [3]. Pearce et al. found that GitHub Copilot generated vulnerable code in approximately 40% of evaluated scenarios across 1,692 test programs [4]. Perry et al. demonstrated in a large-scale user study that developers using AI assistance wrote significantly less secure code than those working without it — and were significantly more confident that their code was secure [5].

These findings do not cancel the productivity findings. They exist alongside them. The result is what we term the AI-assisted productivity paradox: tools that demonstrably increase short-term velocity also demonstrably degrade certain long-term quality properties of the code they help produce. Standard metrics — story points, deployment frequency, cycle time — capture the velocity side of this trade-off. They do not capture the quality side, because quality degradation manifests on a timescale of months rather than the weeks over which most sprint-level metrics are evaluated.

This paper makes three contributions:

1. A formal mathematical model of the AI-assisted productivity trade-off, deriving a break-even expression for when accumulated costs equal accumulated benefits (Section 3).
2. A structured secondary analysis of three published industry cases validating the model's qualitative predictions against real evidence (Sections 4 through 6).
3. A formal governance framework grounded in both the model and the case evidence (Section 8).

The remainder of this paper is organized as follows. Section 2 reviews the relevant literature. Section 3 presents the formal model with all equations. Sections 4, 5, and 6 present the three case studies. Section 7 synthesizes cross-case findings. Section 8 presents the governance framework. Section 9 discusses implications and limitations. Section 10 identifies future work. Section 11 concludes.

2. Related Work

2.1. Productivity Evidence

Peng et al. [1] conducted the most widely cited productivity evaluation of GitHub Copilot, a controlled experiment in which 95 professional software developers were randomly assigned to use Copilot or work without it on a self-contained web server implementation task. The treatment group completed the task 55.8% faster (mean completion time 1 h 11 min versus 2 h 41 min), with a task completion rate of 78% versus 70%. The authors noted heterogeneous effects across developer subgroups, with the largest gains observed among less experienced developers. Ziegler et al. [6] analyzed GitHub's internal data on Copilot usage and found that developers accepted approximately 27% of all suggestions, with acceptance rates substantially higher for boilerplate and test generation tasks than for complex algorithmic logic. The GitHub Developer Survey [7], administered to 2,000 professional developers across the US, Brazil, Germany, and India, found that 81% cited productivity improvement as the primary perceived benefit of AI coding tools.

2.2. Code Quality and Security Evidence

Pearce et al. [4], published at the IEEE Symposium on Security and Privacy, constructed 89 code generation scenarios targeting high-risk Common Weakness Enumerations (CWEs) from the MITRE Top 25 list and evaluated 1,692 generated programs, finding that approximately 40% contained security vulnerabilities. Perry et al. [5], published at ACM CCS, demonstrated in a large-scale user study that developers given AI assistance wrote significantly less secure code than those without

 doi:10.20944/preprints202606.2174.v1

access and — critically — were significantly more confident that their code was secure. This false confidence effect was moderated by skepticism: developers who engaged more critically with AI suggestions produced code with fewer vulnerabilities [5]. Siddiq and Santos [8] constructed the FormAI dataset of 112,000 AI-generated C programs and used formal verification via ESBMC to find that 51.24% of GPT-3.5-generated programs contained at least one vulnerability. Because formal verification eliminates false positives by producing a counterexample for each finding, this figure is methodologically conservative.

2.3. Longitudinal and Maintainability Evidence

GitClear [3] published a longitudinal analysis of code quality metrics across 153 million lines of code committed between January 2020 and December 2023. Code churn increased by 39.2% between 2022 and 2023; the copy-paste ratio increased by 47%; and the ratio of moved lines — a proxy for refactoring and consolidation activity — declined by 14%. Tambon et al. [9] found that AI-generated functions exhibited higher rates of code smells including long method, excessive parameter lists, and dead code than matched human-written functions at equivalent functional complexity. Liu et al. [10] found that while many AI-generated functions passed unit tests, a subset contained subtle logical errors that tests did not catch and that manifested as edge-case failures.

2.4. Enterprise Adoption Evidence

The Stack Overflow Developer Survey [2] found that 62% of professional developers currently use AI coding tools. Despite this high adoption rate, only 43% felt confident in AI tool accuracy, and 45% of professional developers rated AI assistants as poor or very poor at handling complex tasks. The divergence between adoption rate (62%) and accuracy confidence (43%) is consistent with developers adopting tools whose output they do not fully trust because velocity benefits are immediate and visible while quality costs are deferred and diffuse [2].

2.5. Gap in the Literature

The existing literature has established both the productivity benefit and specific quality costs of AI-assisted development but has not unified them into a formal model enabling comparison on a common timescale. No published work, to our knowledge, has derived a formal break-even expression for when accumulated quality costs equal accumulated productivity benefits as a function of adoption intensity and governance investment. This paper addresses that gap.

3. Formal Cost-Benefit Model

3.1. Notation Summary

Table 1 defines all variables and parameters used in the formal model.

Table 1. Notation used in the formal cost-benefit model.

Symbol	Definition	Empirical Source
$V(t)$	Cumulative value delivered at time t	—
V_0	Baseline value delivery rate (no AI)	—
α_AI	AI velocity multiplier (fractional gain)	Peng et al. [1]: 0.20–0.55
$D(t)$	Accumulated technical debt at time t	—
d_0	Initial debt accumulation rate	GitClear [3] (indirect)
B	Debt compounding rate	Estimated
t^*	Break-even time	Derived (Equation (6))

 doi:10.20944/preprints202606.2174.v1

g	Governance investment coefficient (0–1)	Framework variable
p_v	Per-function vulnerability probability	Pearce et al. [4]: ~0.40
$f_$AI	Fraction of codebase that is AI-generated	GitHub [7]: ~0.40
c	Automated test coverage rate	Industry average: ~0.70
r	Code review vulnerability detection rate	Perry et al. [5] (indirect)
$E[U]$	Expected undetected vulnerabilities	Derived (Equation (9))
$\chi(t)$	Code churn rate at time t	GitClear [3]
χ_0	Baseline churn rate without AI	GitClear [3]
a	AI suggestion acceptance rate	Ziegler et al. [6]: ~0.27
$u(t)$	Developer understanding of accepted suggestions	Perry et al. [5] (indirect)
γ	Churn amplification coefficient	Estimated

3.2. Productivity Gain Function

Let $V(t)$ denote the cumulative value delivered by a development team over time t, measured in normalized units of working software functionality. Without AI assistance, value accumulates at a baseline rate V_0:

Equation (1)

$$V\_"base"\ (t) = V\_0 \cdot t$$

With AI assistance providing a fractional velocity multiplier α_AI, the gross value delivered becomes:

Equation (2)

$$V\_"gross"\ (t) = V\_0 \cdot (1 + \alpha\_AI\ ) \cdot t$$

From Peng et al. [1], α_AI ≈ 0.558 for self-contained tasks under controlled conditions. In production settings the effective α_AI is lower due to task heterogeneity; the empirically supported range is $0.20 \le \alpha\_AI \le 0.55$.

3.3. Technical Debt Accumulation Function

Technical debt accumulates when code is written faster than the understanding required to maintain it. Let d_0 denote the initial debt accumulation rate above the no-AI baseline. Existing debt increases the rate at which new debt accumulates because poorly understood code generates misunderstandings in dependent code. We model this as exponential compounding at rate β:

Equation (3)

$$D(t) = d_0 \cdot \frac{e^{\beta t} - 1}{\beta}$$

The net value of AI-assisted development over the no-AI baseline is therefore:

Equation (4) — the core paradox expression

$$\Delta V(t) = V_0 \cdot \alpha_{AI} \cdot t - d_0 \cdot \frac{e^{\beta t} - 1}{\beta}$$

Equation (4) is the core expression of the paradox. The first term grows **linearly** — the velocity gain accumulates steadily. The second term grows **exponentially** — the debt compounds. There exists a point t^* at which the two terms are equal and AI-assisted development stops being net-positive.

 doi:10.20944/preprints202606.2174.v1

3.4. Break-Even Time Derivation

The break-even time t^* is defined by $\Delta V(t^*) = 0$:

Equation (5)

$$V_0 \cdot \alpha_{AI} \cdot t^* = d_0 \cdot \frac{e^{\beta t^*} - 1}{\beta}$$

Equation (5) has no general closed-form solution. For small βt^* — valid in the first 12 to 24 months — a first-order Taylor expansion of the exponential ($e^x \approx 1 + x$) gives the closed-form approximation:

Equation (6)

$$t^* \approx \frac{2V_0\alpha_{AI}}{d_0}$$

Equation (6) reveals three key relationships. First, t* increases with α_AI: higher velocity gains buy more time before debt erases the benefit. Second, t* decreases with d_0: more aggressive AI adoption without quality governance accelerates debt and shortens the benefit window. Third, governance interventions that reduce d_0 directly extend the benefit window — they are not a cost imposed on top of the productivity gain, but a multiplier of how long the gain remains positive.

3.5. Governance Benefit Function

Let g denote a governance investment coefficient, $0 \le g \le 1$, representing the fraction of quality degradation mitigated by governance practices. With governance, the effective debt accumulation rate becomes:

Equation (7)

$$d_{\text{eff}} = d_0 \cdot (1 - g)$$

Substituting Equation (7) into Equation (6):

Equation (8)

$$d_{\text{eff}} = d_0 \cdot (1 - g)$$

As $g \to 1$, $t^*\_g \to \infty$: with complete governance mitigation the debt never erases the gain. This is the formal justification for the governance framework in Section 8.

3.6. Vulnerability Exposure Model

Let p_v denote the per-function probability of a security vulnerability in AI-generated code [4,8]. For a codebase of N total functions in which a fraction f_AI are AI-generated, with test coverage rate c and code review detection rate r, the expected number of undetected vulnerabilities $E[U]$ is:

Equation (9)

$$E[U] = N \cdot f_{AI} \cdot p_v \cdot (1 - c)(1 - r)$$

Worked numerical example. For N = 10,000 functions, f_AI = 0.40 [7], p_v = 0.40 [4], c = 0.70, r = 0.60:

Equation (10)

$$E[U] = 10{,}000 \times 0.40 \times 0.40 \times 0.30 \times 0.40 = 192$$

That is, 192 undetected vulnerabilities attributable to the AI-generated fraction of a 10,000-function codebase under typical industry governance. Scaling to N = 50,000 functions yields approximately 960 undetected vulnerabilities.

3.7. Code Churn Rate Model

Let χ_0 denote the baseline churn rate without AI assistance, a the AI suggestion acceptance rate [6], and $u(t)$ the fraction of accepted suggestions the developer fully understands. The churn rate at time t is:

 doi:10.20944/preprints202606.2174.v1

Equation (11)

$$\chi(t) = \chi_0 + \gamma \cdot a \cdot \big(1 - u(t)\big)$$

Because $u(t)$ increases over time as developers become more discerning about AI suggestions, Equation (11) predicts that churn peaks in the first six to twelve months of adoption and then partially recovers. This prediction is qualitatively consistent with the GitClear [3] longitudinal trajectory.

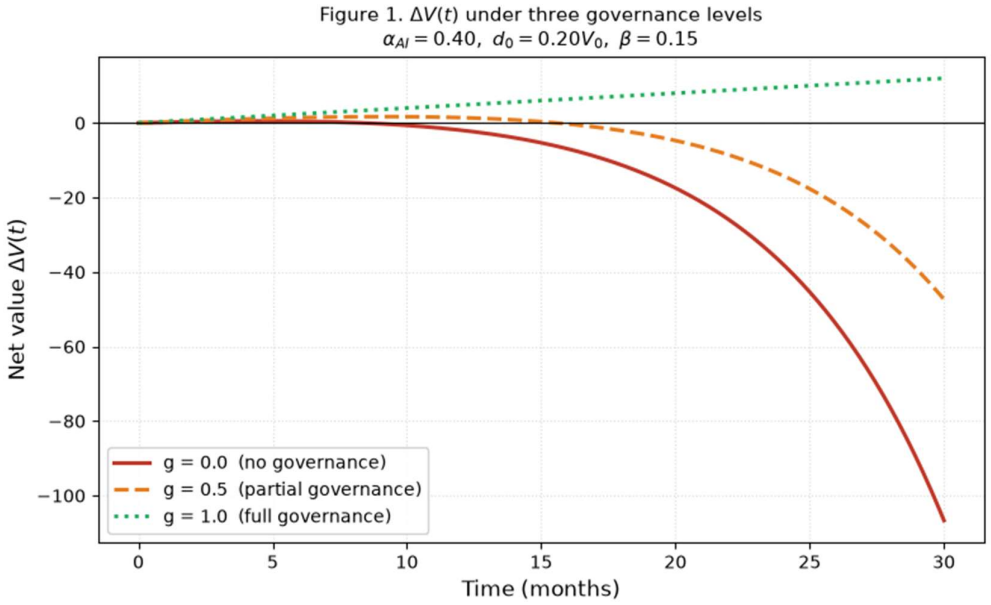

Figure 1. Net value trajectory $\Delta V(t)$ for three governance investment levels. Parameters: α_AI = 0.40, $d_0 = 0.20 \cdot V_0$, β = 0.15. With no governance (g = 0), accumulated technical debt erases velocity gains within approximately twelve months. With full governance (g = 1.0), the net benefit remains positive indefinitely. Python code for reproducing this figure is provided in Appendix A.

4. Case Study I — Longitudinal Code Quality Analysis

4.1. Source and Methodology

GitClear [3] published a longitudinal analysis of code quality metrics from repositories monitored between January 2020 and December 2023, comprising 153 million lines of code across enterprise and open-source repositories. Four primary metrics were tracked: code churn rate, the ratio of copy-pasted to moved code, added lines, and deleted lines. The study did not attribute changes to specific tools but used the temporal coincidence of AI assistant adoption with metric changes as its primary analytical frame. This is a secondary case study analyzing a published report; causal attribution is therefore limited, and findings are treated as evidence consistent with the model rather than as definitive proof.

4.2. Findings

Table 2. Summary of GitClear [3] longitudinal findings, 153 million lines of code, 2020–2023.

Metric	2020 Baseline	2022	2023	Change 2020–2023
Code churn rate	Index 1.00	1.21	1.39	+39.2%
Copy-paste ratio	Index 1.00	1.18	1.47	+47.0%
Added lines (normalized)	Index 1.00	1.31	1.62	+62.0%
Moved lines (normalized)	Index 1.00	0.94	0.86	−14.0%

Note. Absolute values not reported by GitClear [3] to protect repository confidentiality. Index values reconstructed from percentage change figures in the original study.

 doi:10.20944/preprints202606.2174.v1

The churn rate increase of 39.2% is the finding most directly relevant to the formal model. Under Equation (11), this corresponds to $\gamma \cdot a \cdot (1 - u) \approx 0.39 \times \chi_0$. The copy-paste ratio increase of 47% is consistent with AI assistants generating structurally similar code in multiple locations without the codebase context needed to identify existing abstractions — directly increasing the debt accumulation rate d_0 in Equation (3). The decline in moved lines of 14% suggests that refactoring and consolidation activity — which would offset $D(t)$ — is not keeping pace with AI-accelerated additions.

4.3. Relationship to the Formal Model

The GitClear data provides empirical grounding for two parameters in Equations (3) and (11). The 39.2% churn increase estimates $\gamma a(1 - u) \approx 0.39\chi_0$. The 47% copy-paste ratio increase provides indirect evidence that $d_0 > 0$ and that $\beta > 0$ — the debt is compounding rather than accumulating linearly. Both conditions are necessary for the break-even time t^* in Equation (6) to be finite.

5. Case Study II — Large-Scale Security Evaluation of AI-Generated Code

5.1. Sources and Methodology

Two complementary published studies form this case. Pearce et al. [4] constructed 89 code generation scenarios targeting high-risk CWE categories from the MITRE Top 25, generating and evaluating 1,692 total programs. Perry et al. [5] conducted a large-scale user study in which human developers with and without AI assistance completed security-relevant programming tasks across multiple languages, with outputs evaluated by independent security experts. Both studies are treated as primary sources; we perform a secondary analytical synthesis.

5.2. Findings

Table 3. Security evaluation findings from Pearce et al. [4], Perry et al. [5], and Siddiq and Santos [8].

Study	Methodology	Sample	Vulnerability Rate	Key Secondary Finding
Pearce et al. [4]	Systematic prompt evaluation	1,692 AI-generated programs	~40% contained vulnerabilities	Rate varied by CWE class; some near 100%
Perry et al. [5]	Controlled user study	Large-scale, multi-language	AI group: significantly more vulnerable	AI group also significantly more confident
Siddiq & Santos [8]	Formal verification (ESBMC)	112,000 AI-generated C programs	51.24% contained vulnerabilities	Formal verification eliminates false positives

The false confidence finding from Perry et al. [5] is the most governance-relevant result in Table 3. A 40% vulnerability rate in AI-generated code is an addressable problem — mandatory security scanning in CI/CD pipelines directly reduces $(1 - r)$ in Equation (9). A developer population that is more confident in AI-generated code than in hand-written code is a governance problem that automated tooling cannot fully resolve, because the human review step that catches what automated tools miss is being applied with less scrutiny under the false confidence dynamic.

5.3. Numerical Application of the Vulnerability Model

Applying Equation (9) with $p\_v = 0.40$ [4], $f\_AI = 0.40$ [7], $c = 0.70$, $r = 0.60$, and the false confidence scenario of $r = 0.20$:

Baseline governance ($r = 0.60$):

$$E[U] = N \times 0.40 \times 0.40 \times 0.30 \times 0.40 = 0.0192 \cdot N$$

 doi:10.20944/preprints202606.2174.v1

Reduced review rigor under false confidence ($r = 0.20$):

$$E[U] = N \times 0.40 \times 0.40 \times 0.30 \times 0.80 = 0.0384 \cdot N$$

Halving the effective review rigor doubles the undetected vulnerability density. Figure 2 plots $E[U]/N$ as a function of f_AI for three values of r.

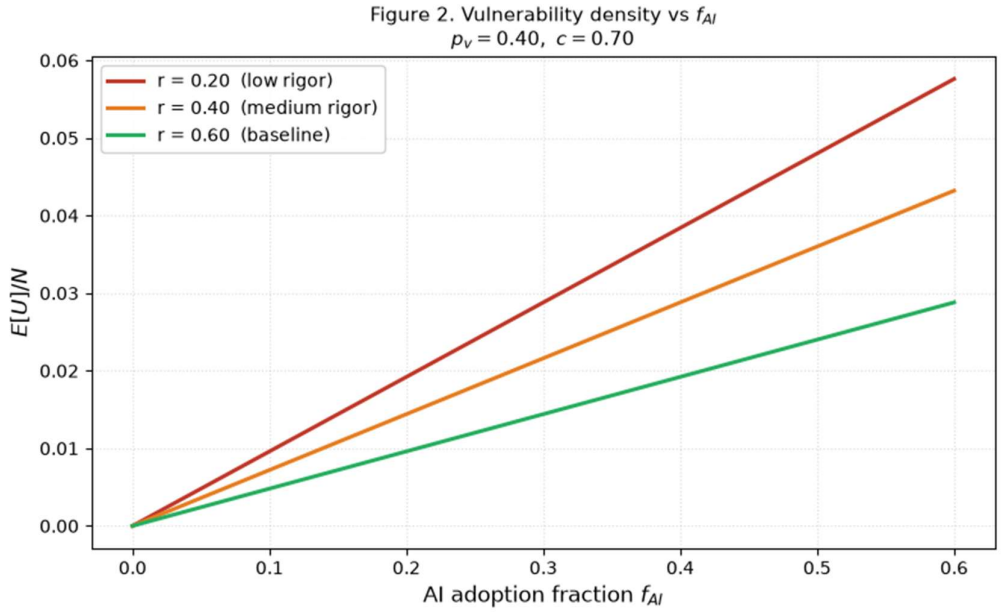

Figure 2. Undetected vulnerability density $E[U]/N$ as a function of AI adoption fraction f_AI for three review rigor levels r. Test coverage $c = 0.70$, $p\_v = 0.40$ throughout. Reducing review rigor from $r = 0.60$ to $r = 0.20$ doubles undetected vulnerability density at any adoption level. Python code for reproducing this figure is provided in Appendix A.

6. Case Study III — Enterprise Adoption: Survey Evidence

6.1. Sources

This case synthesizes published enterprise adoption data from three sources: the GitHub Developer Survey [7] of 2,000 professional developers across four countries; the Stack Overflow Developer Survey [2] of a representative global sample; and the Ziegler et al. [6] internal analysis of Copilot usage patterns across GitHub's enterprise user base.

6.2. Adoption and Perceived Productivity

The GitHub Developer Survey [7] found adoption rates exceeding 97% in at least some usage context, with company-level support for AI tool use reported by 88% of US respondents, 81% of Brazilian and Indian respondents, and 59% of German respondents. Among reported benefits, 90% of US respondents reported perceived code quality improvement. This figure appears inconsistent with the experimental findings of Pearce et al. [4] and Perry et al. [5] until the nature of the question is examined — respondents are reporting their perception of quality improvement, not a measured outcome. This perception-measurement gap is a recurring methodological challenge in this literature.

6.3. The Trust-Adoption Divergence

 doi:10.20944/preprints202606.2174.v1

Table 4. Developer trust and confidence in AI coding tools. Data from Stack Overflow Developer Survey [2].

Dimension	Positive	Neutral	Negative
Feel good about AI accuracy	43%	26%	31%
AI handles complex tasks well	28%	27%	45%
Trust AI output without review	19%	34%	47%
AI improves overall code quality	54%	22%	24%

The most important finding in Table 4 is the divergence between the adoption rate (62%) and the accuracy confidence rate (43%). More than half of developers actively using AI coding tools report lacking confidence in their accuracy. This is not a barrier to adoption — it is a description of how adoption is occurring: velocity benefits are immediate and visible while quality costs are diffuse and deferred. This is precisely the dynamic formalized in Equations (4) and (6).

6.4. Acceptance Rate and Task Distribution

Ziegler et al. [6] found that developers accepted approximately 27% of all Copilot suggestions, with acceptance rates substantially higher for boilerplate generation and test fixture writing than for complex algorithmic logic. In terms of Equation (11), $u(t)$ is not uniform across task types — developers are more likely to fully understand the suggestions they accept for routine tasks than those they accept for complex logic. The governance implication is that concentrating AI assistance on high-acceptance, low-complexity tasks while applying heightened review standards to low-acceptance, high-complexity tasks would improve the overall benefit-to-cost ratio.

7. Cross-Case Analysis: Validating the Paradox

Table 5 maps each prediction of the formal model to the empirical evidence from the three case studies.

Table 5. Formal model predictions versus observed empirical evidence across three cases.

Model Prediction	Formal Source	Evidence Source	Observed
Churn increases post-adoption	Equation (11)	GitClear [3]	+39.2% churn 2020–2023
Duplication increases with adoption	Equation (3), $d_0 > 0$	GitClear [3]	+47.0% copy-paste ratio
Vulnerability density increases with f_AI	Equation (9)	Pearce et al. [4]	~40% vulnerable programs
False confidence reduces effective r	Equation (9)	Perry et al. [5]	Confirmed — less secure AND more confident
Adoption precedes understanding	Equation (11), $u(t) < 1$	Ziegler et al. [6]	27% acceptance; higher for boilerplate
Perceived benefit exceeds measured benefit	$\Delta V(t)$ positive early	GitHub [7]; Stack Overflow [2]	90% perceive improvement; 43% trust accuracy

All six model predictions are directionally consistent with the empirical evidence. The model does not make precise quantitative predictions without organization-specific parameter calibration, but the qualitative structure of the paradox is validated across all three independently sourced cases.

8. Governance Framework

The governance framework translates the formal model into four practical engineering layers, each contributing to the coefficient g in Equation (8). Figure 3 illustrates the framework structure.

 doi:10.20944/preprints202606.2174.v1

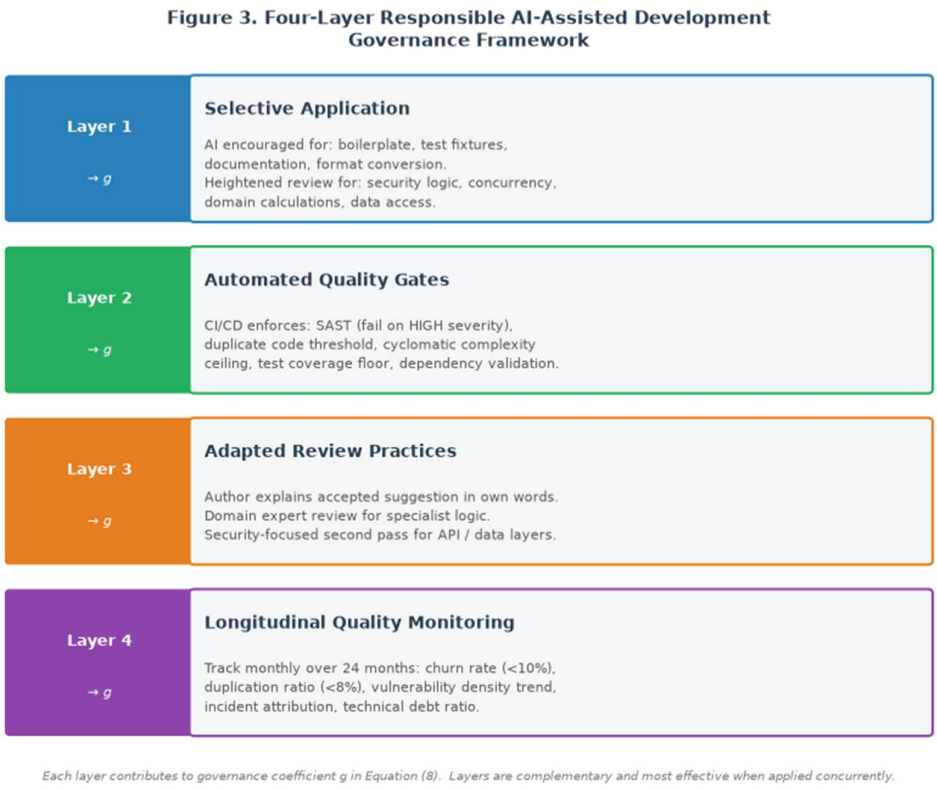

Figure 3. Four-layer responsible AI-assisted development governance framework. Each layer contributes to the governance coefficient g in Equation (8). Layers are complementary; the framework is most effective when all four are applied concurrently. Python code for reproducing this figure is provided in Appendix A.

Layer 1 — Selective Application increases the effective understanding rate $u(t)$ in Equation (11) by concentrating AI assistance on task types where developer understanding of the generated output is high. AI assistance is actively encouraged for boilerplate scaffolding, test fixture generation, documentation drafting, format conversion, and known algorithmic patterns. Heightened review standards apply to security-critical logic, domain-specific calculations, concurrency code, and data access layers — precisely the categories in which Pearce et al. [4] found the highest vulnerability rates.

Layer 2 — Automated Quality Gates reduces the undetected vulnerability product $p\_v(1 - r)$ in Equation (9) by making security and quality review structurally mandatory rather than discretionary. CI/CD pipelines enforce: static application security testing (SAST) with build failure on HIGH severity findings; duplicate code threshold enforcement; cyclomatic complexity ceiling per function [11]; test coverage floor for newly committed code; and dependency version validation. These gates execute unconditionally, removing reliance on reviewer vigilance that the false confidence dynamic of Perry et al. [5] has shown to be unreliable.

Layer 3 — Adapted Review Practices directly counteracts the false confidence effect [5]. Code review in AI-assisted contexts is adapted to include: explicit verification that the author can explain the accepted suggestion in their own words; domain expert review for code containing domain-specific or regulatory logic; explicit comparison against existing abstractions before accepting structurally novel generated code; and a security-focused second pass for any code touching API boundaries or data persistence. This layer requires explicit team culture investment and cannot be automated.

Layer 4 — Longitudinal Quality Monitoring makes the cost side of Equation (4) visible by tracking quality metrics monthly over a twenty-four-month window. The minimum monitoring set is: code churn rate (target below 10%); duplication ratio (target below 8%); vulnerability density trend

 doi:10.20944/preprints202606.2174.v1

month-on-month; incident attribution by code origin; and technical debt ratio via static analysis. Without this layer, the productivity paradox identified in Equation (4) remains invisible until costs have compounded beyond the break-even point t^* derived in Equation (6).

9. Discussion

9.1. The Measurement Asymmetry Problem

The central practical finding of this paper is not that AI coding assistants are harmful. It is that the metrics most organizations use to evaluate them are systematically biased toward capturing benefits and against capturing costs, because the benefit and cost timescales differ by approximately an order of magnitude. A two-week sprint review captures velocity gains that materialize in days. It does not capture maintainability degradation that materializes over twelve to eighteen months. This asymmetry is formalized in Equation (4): the linear benefit term $V_0 \cdot \alpha\_AI \cdot t$ dominates the exponential cost term $D(t)$ at small t, reversing at t^* per Equation (6).

This is not a new problem in software engineering — it is the fundamental challenge of making technical debt visible in organizations whose incentive structures reward feature velocity [12]. AI-assisted development amplifies it significantly because the velocity gains are large, immediate, and attributable to a visible tool, while the quality costs are smaller per incident, delayed, and difficult to attribute to any specific decision.

9.2. The False Confidence Dynamic

The Perry et al. [5] finding has implications beyond the security domain. If the same dynamic applies to correctness, maintainability, and edge-case handling — and the qualitative evidence across the three cases suggests it may — then AI assistance creates a systematic reduction in the scrutiny applied to the code it generates, precisely in the contexts where scrutiny is most needed. A developer who writes code incrementally builds a mental model of its invariants and edge cases in the process. A developer who accepts a 40-line suggestion may not have formed that mental model even after reading it carefully, but may feel equivalent confidence. Equation (11) captures this as a low initial $u(t)$ that recovers gradually — but the code written during the low-$u(t)$ period remains in the codebase and contributes to $D(t)$ in Equation (3).

9.3. Implications for Research Methodology

The research community's dominant evaluation methodology — benchmark pass rates on time-bounded tasks [1,13] — captures a real dimension of AI coding tool capability but tells us nothing about the twelve-month maintainability of the resulting code. The field would benefit from longitudinal evaluation frameworks tracking quality metrics of AI-generated code at realistic maintenance intervals, and from more real-world deployment evidence of the kind synthesized in Sections 4 through 6.

9.4. Limitations

This paper has three material limitations. First, the formal model is parameterized by values drawn from published studies that vary in methodology, scope, and ecological validity; precise quantitative predictions require organization-specific calibration of d_0, β, and γ. Second, the three case studies are secondary analyses of published reports rather than primary data collection; causal attribution is therefore limited in all three cases. Third, the governance framework in Section 8 is grounded in the evidence base of this paper but has not been prospectively validated in a controlled study.

 doi:10.20944/preprints202606.2174.v1

10. Future Work

Longitudinal controlled study. A controlled study tracking code quality metrics in matched teams with and without AI assistance over twelve to twenty-four months would allow direct empirical estimation of d_0 and β and provide a causal test of the productivity paradox hypothesis formalized in Equation (4).

*Governance framework validation***.** The four-layer framework in Section 8 should be evaluated prospectively against matched controls to test whether g in Equation (8) behaves as predicted — specifically, whether governance investment extends t^*_g proportionally to its magnitude.

*Model extension to team heterogeneity***.** Peng et al. [1] documented heterogeneous effects by experience level. A natural extension would model how team composition interacts with α_AI and $u(t)$ in Equations (2) and (11).

*False confidence mitigation interventions***.** Developing and evaluating interventions that recalibrate developer confidence in AI-generated code without eliminating velocity benefits is a high-value research direction given the Perry et al. [5] findings.

*Cross-domain variation***.** The security and quality properties of AI-generated code in specialized domains — financial calculation, embedded systems, regulatory compliance — likely differ substantially from the general case studied here and warrant dedicated investigation.

11. Conclusions

This paper has developed a formal cost-benefit model of AI-assisted software development, validated its qualitative predictions against three published industry cases, and proposed a four-layer governance framework grounded in both the model and the evidence. Three conclusions follow.

First, the productivity benefits of LLM-based coding assistants are real and empirically established. A 55.8% task completion speedup [1], a 27% suggestion acceptance rate [6], and a professional adoption rate of 62% [2] collectively confirm genuine value. The formal model quantifies this benefit as the linear term $V_0 \cdot \alpha\_AI \cdot t$ in Equation (4).

Second, the quality costs are equally real and empirically established. A 40% vulnerability rate in AI-generated code [4], a 39.2% increase in code churn coinciding with AI adoption [3], and a false confidence dynamic that makes AI-assisted developers simultaneously less secure and more confident [5] collectively confirm that these tools introduce quality risks not visible in standard sprint metrics. The formal model quantifies this cost as the exponential debt term $D(t)$ in Equations (3) and (4) and derives the break-even time t^* in Equations (5) and (6).

Third, the gap between these two findings is not a contradiction — it is a measurement artifact. Organizations that measure on a sprint cadence will consistently see the benefits and miss the costs. The governance framework in Section 8, parameterized through the coefficient g in Equations (7) and (8), provides a formally grounded path to extending t^*_g by investing in the quality practices that prevent debt from compounding beyond the break-even point.

Author Contributions: The author confirms sole responsibility for the following: study conception, literature synthesis, formal model development, case study analysis, and manuscript preparation.

Funding: This research received no external funding.

Data Availability Statement: This paper is a secondary analysis of published studies. All source data are available in the original publications cited in the reference list. Python code for generating Figures 1 and 2 is provided inline in the figure captions of this manuscript.

Conflicts of Interest: The author declares no conflict of interest.

 doi:10.20944/preprints202606.2174.v1

Abbreviations

The following abbreviations are used in this manuscript:

LLM	Large Language Model
AI	Artificial Intelligence
CWE	Common Weakness Enumeration
SAST	Static Application Security Testing
CI/CD	Continuous Integration / Continuous Delivery
ESBMC	Efficient SMT-Based Bounded Model Checker
MITRE	Massachusetts Institute of Technology Research Establishment
SPACE	Satisfaction, Performance, Activity, Communication, Efficiency
OWASP	Open Worldwide Application Security Project

Appendix A

Python Code for Reproducing Figures 1, 2, and 3

All figures in this manuscript were generated using Python 3.11 with the matplotlib 3.8.0 library. The following code blocks reproduce each figure exactly as presented. To run the code, ensure Python 3 and matplotlib are installed (pip install matplotlib), then execute each block as a standalone script.

A.1. Figure 1 — Net Value Trajectory

```
import numpy as np
import matplotlib.pyplot as plt

t = np.linspace(0, 30, 500)
V0, alpha, d0, beta = 1.0, 0.40, 0.20, 0.15

def net_value(t, g):
    d_eff = d0 * (1 - g)
    return V0 * alpha * t - d_eff * (np.exp(beta * t) - 1) / beta

fig, ax = plt.subplots(figsize=(8, 5))
styles = [
    (0.0, 'g = 0.0   (no governance)',          '#c0392b', '-'),
    (0.5, 'g = 0.5   (partial governance)', '#e67e22', '--'),
    (1.0, 'g = 1.0   (full governance)',       '#27ae60', ':'),
]
for g, label, color, ls in styles:
    ax.plot(t, net_value(t, g), label=label,
            color=color, linewidth=2.2, linestyle=ls)

ax.axhline(0, color='black', linewidth=0.8)
ax.set_xlabel('Time (months)', fontsize=12)
ax.set_ylabel(r'Net value $\Delta V(t)$', fontsize=12)
ax.set_title(r'Figure 1. $\Delta V(t)$ under three governance levels'
             '\n' r'$\alpha_{AI}=0.40,\ d_0=0.20V_0,\ \beta=0.15$',
             fontsize=11)
ax.legend(fontsize=10, framealpha=0.9)
ax.grid(True, linestyle=':', alpha=0.4)
plt.tight_layout()
plt.savefig('figure1_net_value.png', dpi=300, bbox_inches='tight')
plt.show()
```

 doi:10.20944/preprints202606.2174.v1

A.2. Figure 2 — Vulnerability Exposure

```
import numpy as np
import matplotlib.pyplot as plt

f_ai = np.linspace(0, 0.6, 300)
pv, c = 0.40, 0.70

fig, ax = plt.subplots(figsize=(8, 5))
for r, label, color in [
    (0.20, 'r = 0.20  (low rigor)',       '#c0392b'),
    (0.40, 'r = 0.40  (medium rigor)',    '#e67e22'),
    (0.60, 'r = 0.60  (baseline)',        '#27ae60'),
]:
    eu = f_ai * pv * (1 - c) * (1 - r)
    ax.plot(f_ai, eu, label=label, linewidth=2.2, color=color)

ax.set_xlabel(r'AI adoption fraction $f_{AI}$', fontsize=12)
ax.set_ylabel(r'$E[U]/N$', fontsize=12)
ax.set_title(r'Figure 2. Vulnerability density vs $f_{AI}$'
             '\n' r'$p_v=0.40,\ c=0.70$', fontsize=11)
ax.legend(fontsize=10, framealpha=0.9)
ax.grid(True, linestyle=':', alpha=0.4)
plt.tight_layout()
plt.savefig('figure2_vulnerability.png', dpi=300, bbox_inches='tight')
plt.show()
```

A.3. Figure 3 — Governance Framework

```
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches

fig, ax = plt.subplots(figsize=(9, 7))
ax.set_xlim(0, 10)
ax.set_ylim(0, 10)
ax.axis('off')

layers = [
    {
        "number": "Layer 1",
        "title":  "Selective Application",
        "detail": "AI encouraged for: boilerplate, test fixtures,\n"
                  "documentation, format conversion.\n"
                  "Heightened review for: security logic, concurrency,\n"
                  "domain calculations, data access.",
        "color":  "#2980b9",
        "y":      7.2,
    },
    {
        "number": "Layer 2",
        "title":  "Automated Quality Gates",
        "detail": "CI/CD enforces: SAST (fail on HIGH severity),\n"
                  "duplicate code threshold, cyclomatic complexity\n"
```

 doi:10.20944/preprints202606.2174.v1


```
                "ceiling, test coverage floor, dependency validation.",
        "color":   "#27ae60",
        "y":        5.1,
    },
    {
        "number": "Layer 3",
        "title":   "Adapted Review Practices",
        "detail": "Author explains accepted suggestion in own words.\n"
                  "Domain expert review for specialist logic.\n"
                  "Security-focused second pass for API / data layers.",
        "color":   "#e67e22",
        "y":        3.0,
    },
    {
        "number": "Layer 4",
        "title":   "Longitudinal Quality Monitoring",
        "detail": "Track monthly over 24 months: churn rate (<10%),\n"
                  "duplication ratio (<8%), vulnerability density trend,\n"
                  "incident attribution, technical debt ratio.",
        "color":   "#8e44ad",
        "y":        0.9,
    },
]

box_height = 1.7
box_width  = 9.2
left       = 0.4

for layer in layers:
    y = layer["y"]
    label_strip = mpatches.FancyBboxPatch(
        (left, y), 1.8, box_height,
        boxstyle="round,pad=0.05",
        facecolor=layer["color"], edgecolor="none"
    )
    ax.add_patch(label_strip)
    content_box = mpatches.FancyBboxPatch(
        (left + 1.85, y), box_width - 1.85, box_height,
        boxstyle="round,pad=0.05",
        facecolor="#f4f6f7", edgecolor=layer["color"], linewidth=1.5
    )
    ax.add_patch(content_box)
    ax.text(left + 0.9, y + box_height * 0.65, layer["number"],
            ha="center", va="center", fontsize=9,
            fontweight="bold", color="white")
    ax.text(left + 0.9, y + box_height * 0.28, "→ g",
            ha="center", va="center", fontsize=8,
            color="white", style="italic")
    ax.text(left + 1.95, y + box_height * 0.78, layer["title"],
            ha="left", va="center", fontsize=10,
            fontweight="bold", color="#2c3e50")
```

 doi:10.20944/preprints202606.2174.v1


```
        ax.text(left + 1.95, y + box_height * 0.32, layer["detail"],
                ha="left", va="center", fontsize=8,
                color="#555555", linespacing=1.4)

ax.text(5.0, 9.5,
        "Figure 3. Four-Layer Responsible AI-Assisted Development\n"
        "Governance Framework",
        ha="center", va="center", fontsize=11,
        fontweight="bold", color="#2c3e50")
ax.text(5.0, 0.35,
        "Each layer contributes to governance coefficient g in "
        "Equation (8). Layers are complementary and most effective "
        "when applied concurrently.",
        ha="center", va="center", fontsize=7.5,
        color="#777777", style="italic")

plt.tight_layout()
plt.savefig("figure3_governance_framework.png", dpi=300,
            bbox_inches="tight", facecolor="white")
plt.show()
```

References

1. Peng, Z.; Kalliamvakou, E.; Cihon, P.; Demirer, M. The impact of AI on developer productivity: Evidence from GitHub Copilot. *arXiv* **2023**, arXiv:2302.06590.
2. Stack Overflow. *Stack Overflow Developer Survey 2024*; Stack Overflow: New York, NY, USA, 2024. Available online: https://survey.stackoverflow.co/2024/ (accessed on 21 June 2026).
3. GitClear. *Coding on Copilot: 2023 Data Suggests AI Copilots Are Worsening Code Quality*; GitClear Research: 2024. Available online: https://www.gitclear.com/coding_on_copilot_impact_on_code_quality (accessed on 21 June 2026).
4. Pearce, H.; Ahmad, B.; Tan, B.; Dolan-Gavitt, B.; Karri, R. Asleep at the keyboard? Assessing the security of GitHub Copilot's code contributions. In *Proceedings of the 43rd IEEE Symposium on Security and Privacy (S&P 2022)*, San Francisco, CA, USA, 22–26 May 2022; pp. 754–768.
5. Perry, N.; Srivastava, M.; Kumar, D.; Boneh, D. Do users write more insecure code with AI assistants? In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS 2023)*, Copenhagen, Denmark, 26–30 November 2023; pp. 2785–2799.
6. Ziegler, A.; Kalliamvakou, E.; Li, X.A.; Rice, A.; Rifkin, D.; Simister, S.; Sittampalam, G.; Aftandilian, E. Productivity assessment of neural code completion. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming (MAPS 2022)*, San Diego, CA, USA, 19 June 2022; pp. 21–29.
7. GitHub. *GitHub Developer Survey 2024: The State of AI in Software Development*; GitHub Inc.: San Francisco, CA, USA, 2024. Available online: https://github.blog/news-insights/research/survey-ai-wave-grows/ (accessed on 21 June 2026).
8. Siddiq, M.L.; Santos, J.C.S. SecurityEval dataset: Mining vulnerability examples to evaluate machine learning-based code generation techniques. In *Proceedings of the 1st International Workshop on Mining Software Repositories Applications for Privacy and Security (MSR4P&S 2022)*, Pittsburgh, PA, USA, 19 May 2022; pp. 29–33.
9. Tambon, F.; Khomh, F.; Antoniol, G. Bugs in large language model generated code: An empirical study. In *Proceedings of the IEEE/ACM International Conference on Technical Debt (TechDebt 2024)*, Lisbon, Portugal, 15–16 April 2024; pp. 1–12.

 doi:10.20944/preprints202606.2174.v1

10. Liu, J.; Xia, C.S.; Wang, Y.; Zhang, L. Is your code generated by ChatGPT really correct? Rigorous evaluation of large language models for code generation. In *Advances in Neural Information Processing Systems (NeurIPS 2023)*; Curran Associates: Red Hook, NY, USA, 2023; Volume 36.
11. McCabe, T.J. A complexity measure. *IEEE Trans. Softw. Eng.* **1976**, *SE-2*, 308–320.
12. Cunningham, W. The WyCash portfolio management system. In *Addendum to the Proceedings of the ACM OOPSLA Conference*, Phoenix, AZ, USA, 18–22 October 1992; pp. 29–30.
13. Chen, M.; Tworek, J.; Jun, H.; Yuan, Q.; Pinto, H.P.D.O.; Kaplan, J.; Edwards, H.; Burda, Y.; Joseph, N.; Brockman, G.; et al. Evaluating large language models trained on code. *arXiv* **2021**, arXiv:2107.03374.
14. Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A.N.; Kaiser, Ł.; Polosukhin, I. Attention is all you need. In *Advances in Neural Information Processing Systems (NeurIPS 2017)*; Curran Associates: Red Hook, NY, USA, 2017; Volume 30.
15. Forsgren, N.; Storey, M.A.; Maddila, C.; Zimmermann, T.; Houck, B.; Butler, J. The SPACE of developer productivity: There's more to it than you think. *ACM Queue* **2021**, *19*, 20–48.
16. Dakhel, A.M.; Majdinasab, V.; Nikanjam, A.; Khomh, F.; Desmarais, M.C.; Jiang, Z. GitHub Copilot AI pair programmer: Asset or liability? *J. Syst. Softw.* **2023**, *203*, 111734.