Nexus Science Review
INTERNATIONAL JOURNAL OF INTERDISCIPLINARY RESEARCH

Received: 19/08/2026 - **Accepted:** 28/09/2026
Vol: 01.05
DOI: 10.66104/z67gjz21
Pages: 1-21

Generative AI Assistants in Software Development: Productivity Gains, Code Quality, and Security Trade-offs

Rikelmy Rodrigues Menezes

Faculdade Presidente Antônio Carlos, Brazil

Corresponding author: rikelmym33@gmail.com

Orcid ID: https://orcid.org/0009-0004-1805-2789

Abstract

Generative artificial intelligence (AI) coding assistants promise to shorten software-development tasks, yet task speed alone cannot establish an improvement in engineering productivity. This critical narrative review asks whether assistants help developers complete work faster without increasing functional defects, maintenance burden, or security weaknesses. The final analytical corpus comprises 20 principal sources available through September 27, 2026, including experiments, field trials, repository studies, a benchmark, methodological work, and security guidance. Findings remain heterogeneous: a bounded JavaScript task was completed 55.8% faster with Copilot; pooled workplace trials found 26.08% more completed tasks; an observational Phase 1 maintainability result showed a 30.7% shorter median initial-task time; and experienced maintainers in mature repositories took 19% longer when AI was permitted. A 2026 controlled interface study found shorter task times while directly recording verification-related interactions. A later METR update suggests tool-generation change but cannot yield a precise productivity estimate because of selection and measurement problems. We propose Quality-Adjusted Development Productivity (QADP) as an author-developed decision framework for measuring the complete task lifecycle under independent quality gates, not as a validated score. The evidence supports conditional gains, with task, developer, interface, verification burden, and downstream effects defining the applicable boundary.

Keywords: generative artificial intelligence; software engineering; coding assistants; developer productivity; software security

CC BY

1. Introduction

Generative AI systems now suggest code, explain repositories, draft tests, and propose changes to existing projects. These activities differ substantially: completing a local function, repairing a defect in a mature codebase, and approving a production change require different context, verification, and accountability. A tool that reduces typing time may still increase review effort or future maintenance; conversely, a slower drafting process might improve tests or clarity. The SPACE framework therefore cautions against treating a single activity measure as developer productivity (Forsgren et al., 2021; Jimenez et al., 2024).

The empirical picture contains apparently contradictory results. In a controlled experiment, participants implementing a JavaScript HTTP server with GitHub Copilot completed the task 55.8% faster than controls. In three workplace experiments, combined access to a coding assistant increased completed tasks by 26.08%, although individual experiments were noisy. Yet an experiment in mature open-source repositories found that experienced maintainers took 19% longer when allowed to use early-2025 AI tools. These findings have different participants, tools, tasks, comparison groups, and outcome measures; none by itself estimates a universal effect (Becker et al., 2025; Cui et al., 2026; Peng et al., 2023).

Quality and security make the question more demanding. Controlled vendor research reports gains on unit tests and modest improvements in reviewer ratings for one Python assignment, while security-focused user studies and analyses of generated snippets demonstrate that passing basic tests cannot establish freedom from exploitable weaknesses. Later maintenance work must also be considered because a change can be easy to produce but costly for another developer to understand or extend (Borg et al., 2026; GitHub, 2024; Perry et al., 2023; Sandoval et al., 2023).

The research question is: Under what conditions do generative AI assistants reduce end-to-end software-development time without increasing defects, security vulnerabilities, or subsequent maintenance effort? Four subsidiary questions concern task and developer heterogeneity, how quality is measured, whether observed security risks are causal, and which verification practices allow an organization to retain genuine productivity gains. This review contributes a comparison of incompatible outcome measures, a proposed quality-adjusted evaluation framework, and a practical research design. It does not report a new experiment or claim that the proposed framework has been empirically validated (Becker et al., 2025; Forsgren et al., 2021; National Institute of Standards and Technology [NIST], 2022).

CC BY

2. Conceptual Background

2.1 Assistance modes and task boundaries

An inline completion system predicts code in the editor; a conversational assistant answers questions and drafts solutions; a repository-aware agent may search files, execute tools, and create a patch. Tool capability and exposure differ because an agent can encounter untrusted repository text and make broader changes than a local suggestion. A fair comparison must record the model, access mode, permissions, context supplied, available tools, and time of evaluation, rather than pooling unlike products under the single label “AI coding” (Jimenez et al., 2024; OWASP, 2025).

Task boundaries likewise matter. A benchmark may end when hidden tests pass, a workplace study may count completed tasks or pull requests, and a maintainer may regard work as complete only after design review, integration, and follow-up fixes. An issue can pass a supplied test suite yet violate an unstated requirement or damage another component. SWE-bench introduced 2,294 real issue and pull-request pairs from 12 Python repositories, illustrating a more contextual evaluation than isolated functions, but benchmark resolution remains different from a securely shipped feature maintained over time (Jimenez et al., 2024; NIST, 2022).

2.2 A multidimensional productivity construct

The SPACE framework organizes satisfaction and well-being, performance, activity, communication and collaboration, and efficiency and flow. It does not imply that every dimension can be reduced to a single scalar. In AI-assisted development, prompts, model waiting, code reading, tests, and discussion with reviewers are inputs to the task, while correct and maintainable software is its valued output. More code or more accepted suggestions can indicate automation activity without proving additional customer value (Forsgren et al., 2021; Borg et al., 2026).

This distinction helps interpret studies that measure different endpoints. Peng et al. (2023) measured elapsed time on a constrained implementation; Cui et al. (2026) measured completed tasks in workplace experiments; Becker et al. (2025) timed work in mature repositories. A percent change in time is not the same estimand as a percent change in throughput. The denominator, task completion definition, unfinished work, and baseline quality must be specified before numbers are compared. Studies of developer perceptions are useful for understanding experience but cannot substitute for logged time and independent assessment of outputs (Becker et al., 2025; Cui et al., 2026; Peng et al., 2023).

CC BY

2.3 Quality is more than test passing

Functional correctness asks whether a change satisfies specified behavior; reliability asks whether it behaves consistently under realistic conditions; maintainability asks whether later developers can safely understand and evolve it; security asks whether adversarial inputs or environments can cause harm. These dimensions overlap but cannot be inferred from one another. A passing unit test speaks to the tested cases, not to untested authorization rules, input handling, secrets, concurrency, or deployment configuration (Borg et al., 2026; GitHub, 2024; NIST, 2022).

Security studies also use different units. A generated completion, a participant's finished program, and a repository commit are not interchangeable observations. A scanner finding can be a useful warning, but it is not automatically an exploitable vulnerability; a clean scan does not prove safety. Human adjudication, realistic threat models, and negative controls improve interpretation, especially where detection tools have incomplete coverage or false positives (Fu et al., 2025; NIST, 2022; Sandoval et al., 2023).

3. Methodology and Scope

3.1 Review design

This is a critical narrative and thematic review, not a systematic review or meta-analysis. The last targeted search and source verification occurred on September 27, 2026. Sources were located through searches of scholarly publication and conference pages, institutional research pages, official security guidance, and backward citation checking. Representative query blocks included ("AI coding assistant" OR "generative AI software development") AND ("developer productivity" OR "task completion" OR "verification burden"); ("AI-generated code" OR "coding agent") AND ("maintainability" OR "code quality" OR "security"); and ("AI coding" OR "Copilot") AND ("randomized trial" OR "field experiment"). These are representative retrospective descriptions, not a preserved line-by-line database search log. Direct reproducible exports from ACM Digital Library, IEEE Xplore, Scopus, and Web of Science were not retained, so a structured supplementary search and record counts cannot be claimed. Publisher and official project pages were checked for bibliographic status. No complete identification, duplicate, screening, or exclusion log exists; PRISMA counts would therefore be fictitious (Page et al., 2021).

The final analytical corpus comprises 20 principal sources: 13 empirical studies or original research reports (including three preprints and one vendor-conducted report), one benchmark paper, three methodological or research-design sources (SPACE, PRISMA, and the METR follow-up), and three security standards or guidance sources. These categories describe each item’s primary role in this review; empirical research may also inform methodology. Peer-

CC BY

reviewed empirical studies were preferred where available. Becker et al. (2025), Liu et al. (2026), and Vella and Blincoe (2026) are identified as preprints. GitHub (2024) is vendor-conducted. METR (Becker et al., 2026) is an institutional update, not a peer-reviewed trial report. Standards and guidance inform safeguards rather than estimate assistant effects.

For each focal empirical study the synthesis records tool and vintage, participants, task, setting, comparator, outcome, main finding, and transfer limit. Evidence is interpreted qualitatively rather than pooled because denominators, tool generations, participant experience, and definitions of quality differ. One author performed selection, extraction, and interpretation. There was no independent dual screening or inter-rater reliability assessment. The evidence map is selective and decision-oriented, so omission of a study does not imply irrelevance or poor quality. A supplementary structured database search should be undertaken for any future systematic extension of this work (Borg et al., 2026; Forsgren et al., 2021; Page et al., 2021).

3.2 Appraisal of evidence

The highest confidence for a specific task effect comes from randomized access with objectively measured work and comparable quality checks, although an experiment can still lack external validity. Repository mining has stronger ecological reach but faces selection, attribution, and confounding problems. Security prompt studies identify feasible failures under chosen scenarios but do not estimate the production vulnerability rate. Vendor evaluations may be informative primary evidence while requiring independent replication and careful inspection of design and reporting (Becker et al., 2025; Cui et al., 2026; Fu et al., 2025; GitHub, 2024).

This appraisal is narrative, not a formal GRADE rating, numerical evidence index, or causal meta-analysis. It distinguishes directly observed outcomes from the author's inferences. A study that finds more scanner warnings after an AI-attributed commit can motivate review, but without an appropriate non-AI counterfactual it cannot establish that AI caused more warnings than human-written changes under the same conditions. Conversely, absence of a detected difference in one language and task does not establish general equivalence (Liu et al., 2026; Sandoval et al., 2023).

4. Evidence on Development Speed

4.1 Bounded tasks and controlled experiments

Peng et al. (2023) randomly assigned recruited developers to implement an HTTP server in JavaScript with or without GitHub Copilot. Assisted participants completed the task 55.8% faster. Randomization supports a causal interpretation for that task and tool, but the experiment did not demonstrate an identical gain for large existing repositories, security-sensitive

CC BY

architecture, or long-term ownership. It also cannot establish that every minute saved in initial implementation survives review, deployment, and maintenance (Peng et al., 2023).

A controlled vendor study of a Python web-server assignment found a greater likelihood of passing all ten unit tests for Copilot-access participants and small improvements in blind reviewer ratings of readability, reliability, maintainability, and conciseness. The favorable evidence deserves attention because the experiment observed concrete outputs rather than relying solely on self-reports. Its narrow assignment, vendor sponsorship, finite tests, and small differences in subjective quality ratings restrict the conclusion to the studied conditions. The published report should not be described as independent proof of better production security (GitHub, 2024).

Borg et al. (2026) reported a 30.7% shorter median initial-task time for AI users in observational Phase 1, where AI use was not randomized. The study involved 151 participants across two phases. In randomized Phase 2, new developers manually evolved the resulting code without AI assistance; the study found no clear systematic difference in subsequent evolution time or code quality under the studied conditions. Thus the 30.7% figure is not a randomized causal speed estimate, while the randomized result concerns a different endpoint: later maintainability. This separation is necessary whenever the initial result is cited (Borg et al., 2026).

Fan et al. (2026) randomized 60 participants to AI-assisted or no-AI conditions across three bounded Python tasks and compared inline, chat, and structured interfaces while holding the underlying model fixed. AI assistance reduced mean task time by 22% relative to control (25.0 versus 32.1 minutes) and improved correctness in the reported analysis; interaction logs also captured compile/test failures, churn, pauses, and context switches as verification-load indicators. These measurements make verification effort visible but do not establish an identical benefit in mature repositories or over longer maintenance horizons. Interface effects should not be conflated with a change in model capability (Fan et al., 2026).

4.2 Field experiments and repository work

Cui et al. (2026) combined randomized workplace experiments at Microsoft, Accenture, and another large company. Across 4,867 developers, access to the assistant was associated with a 26.08% increase in completed tasks, with a standard error of 10.3%; individual experiments were noisy, and less experienced developers adopted the tool more and gained more. These are consequential field data, but “completed task” depends on organizational recording practices and should not be translated automatically into an equal improvement in defect-free delivered functionality or security (Cui et al., 2026).

CC BY

Becker et al. (2025) provide a meaningful counterexample. Sixteen experienced open-source developers completed 246 real tasks in mature projects they knew well, with permission to use early-2025 AI tools randomly assigned by task. Allowing AI increased completion time by 19%, even though participants anticipated a gain. The result demonstrates that the sign of an effect can reverse when developers must coordinate with complex context and meet repository-specific standards. It should not be generalized to novice programmers, new projects, different tools, or later model generations without replication (Becker et al., 2025).

The two field settings answer different questions. Workplace assistant access examines adoption and output under organizational workflows; maintainer experiments focus on elapsed time for experienced contributors in familiar open-source repositories. Tool use is permitted, not necessarily identical across participants, and assistance has context-dependent opportunity costs. A coherent synthesis therefore asks what type of work and what endpoint each estimate describes rather than averaging 26% more tasks with 19% more time as if they were directly comparable effect sizes. Table 1 records the distinct estimands and boundaries (Becker et al., 2025; Cui et al., 2026).

4.3 Why gains can disappear

Assistance can save effort by supplying a syntactic starting point, recalling a library call, or drafting repetitive scaffolding. It can also impose prompting, waiting, reading, debugging, adaptation, and review. In a mature system, understanding invariants, ownership, deployment requirements, and hidden dependencies may dominate typing time. A suggestion that is locally plausible but mismatched to project conventions can increase the burden on the person accountable for merging it. These mechanisms are plausible interpretations of observed heterogeneity, rather than independent causal findings of this review (Becker et al., 2025; Borg et al., 2026; Forsgren et al., 2021).

A developer may also value reduced frustration or faster exploration even when logged task time does not decrease. SPACE treats satisfaction as a legitimate outcome while keeping it distinct from performance. This distinction avoids dismissing positive user experience and also avoids presenting subjective relief as evidence of lower engineering cost. The proper unit of measurement depends on the decision: individual learning, team throughput, customer-facing delivery, or maintenance over a release cycle (Becker et al., 2025; Forsgren et al., 2021).

5. Functional Correctness and Reliability

5.1 What tests show

Unit tests offer a reproducible minimum for checking generated patches, but their conclusions depend on coverage and test quality. Passing all supplied tests means the implementation met those assertions; it does not prove that requirements were complete or that hidden,

CC BY

adversarial, or integration cases succeed. The GitHub controlled study offers evidence that assistant access can improve the probability of passing its selected tests. SWE-bench offers broader repository issues, yet success on its test harness still differs from successful production operation (GitHub, 2024; Jimenez et al., 2024).

Generated tests present a related circularity: an assistant may implement an interpretation of a requirement and then generate tests that repeat the same mistaken interpretation. Independent acceptance criteria, property-based or adversarial cases, mutation testing, and human specification review can reduce this risk. Research on LLM test generation uses mutation testing precisely because merely producing executable tests is weaker than demonstrating that tests detect faults. These measures should be evaluated for incremental detection yield and their own cost, not assumed effective because they are automated (NIST, 2022; Dakhel et al., 2024).

5.2 Hidden regressions and integration

A patch can be correct for a local function while changing API behavior, performance, error handling, or compatibility elsewhere. Repository-aware assessment must therefore include regression tests, dependency checks, configuration review, and a clear definition of the permitted change. A benchmark based on actual issue and patch pairs improves contextual realism, but issue descriptions can still omit tacit team knowledge and operational constraints. Independent reviewers should assess changed behavior as well as the number of tests that pass (Jimenez et al., 2024; NIST, 2022).

The most useful functional outcome is not the quantity of generated code. A smaller patch that satisfies requirements with fewer side effects can be preferable to a verbose solution, even if the latter appears faster to draft. Evaluations should report valid task completion, severity-weighted defects, rework after review, and change failure in addition to first-pass test success. This is a proposed measurement design grounded in the distinction between observable activity and performance, rather than a claim that existing studies have measured all of these endpoints together (Forsgren et al., 2021; GitHub, 2024; NIST, 2022).

6. Maintainability and Technical Debt

6.1 Downstream ownership

Maintainability concerns the effort required for another developer to comprehend, change, test, and integrate the artifact. Borg et al. (2026) separated observational Phase 1 initial development, in which AI users had a 30.7% shorter median task time, from randomized Phase 2 downstream manual evolution by another developer. Phase 2 found no clear systematic difference in subsequent evolution time or code quality under its bounded conditions. This

CC BY

behavioral test is stronger than inferring maintainability from code length or stylistic ratings alone; it still cannot measure technical debt emerging months or years later (Borg et al., 2026).

Repository evidence provides complementary signals. Liu et al. (2026), in a preprint analyzing verified AI-authored commits, reported numerous static-analysis issues and tracked whether identified issues persisted. This study examines actual repository trajectories, but attribution by commit, differences among projects, scanner limitations, and the lack of a matched non-AI causal comparator constrain its meaning. It documents a possible maintenance burden in selected AI-attributed changes; it does not by itself establish that an otherwise comparable human patch would have been cleaner (Liu et al., 2026).

6.2 Mechanisms of maintenance cost

Longer, duplicated, or poorly integrated code can make a future change harder even when the immediate feature works. Conversely, an assistant may help a developer document unfamiliar code or create tests that reduce future uncertainty. Both pathways are contingent on what the team accepts, reviews, and preserves. The initial intervention and the later artifact must be evaluated separately: a speed benefit to the first developer is not automatically a team benefit if another person bears the interpretation and repair cost (Borg et al., 2026; Forsgren et al., 2021).

A useful longitudinal design follows patches across subsequent releases, records reverts, defect fixes, reviewer comments, and the time needed to make a similar modification later. Normalization should account for task complexity, codebase size, language, team practices, and developer experience. Static smells are screening signals, not direct measures of money or future effort; later human maintenance tasks can provide a stronger behavioral test. These are proposed design requirements motivated by the differences between controlled downstream experiments and observational repository mining (Borg et al., 2026; Liu et al., 2026).

7. Security and Vulnerability Evidence

7.1 Generated snippets and adversarial tasks

Pearce et al. (2022) tested Copilot outputs in security-relevant scenarios derived from common weakness categories and assessed generated responses with automated and manual analysis. Fu et al. (2025) examined Copilot-generated snippets in GitHub projects and found security weaknesses in selected Python and JavaScript material. These studies show that plausible code can contain recognizable weakness patterns; their sampling and detection procedures should not be interpreted as the overall probability that an AI-assisted production change will be exploitable (Fu et al., 2025; Pearce et al., 2022).

CC BY

Perry et al. (2023) examined people solving security-related coding tasks with and without an AI assistant and reported less secure outputs, alongside greater confidence among assisted users that their solutions were secure. The important unit here is the human–tool interaction, not the model's isolated completion. Prompting, acceptance choices, and misplaced confidence can mediate the outcome. The study used a specific model and experimental tasks, so its findings establish a risk under those conditions rather than a permanent property of every coding assistant (Perry et al., 2023).

Sandoval et al. (2023) randomized participants working on low-level C tasks and reported a comparatively small security effect in that setting. This apparent tension with Perry et al. (2023) is informative: languages, tasks, models, interfaces, participants, and security endpoints differed. Neither study justifies saying that AI always increases vulnerabilities or that its security impact is generally negligible. Risk should be estimated for the actual task, language, adversarial environment, and workflow being adopted (Perry et al., 2023; Sandoval et al., 2023).

7.2 Threats beyond generated code

Repository-aware agents encounter instructions embedded in files, issue descriptions, external pages, and tool output. Prompt injection can induce an assistant to treat untrusted content as instructions; excessive permissions can amplify consequences. Secrets and personal data can be exposed if repository context is copied into an unsuitable service. These are distinct from whether the generated function contains a conventional weakness, and they require controls on data access, tool permissions, sandboxing, and review of actions (OWASP, 2025; NIST, 2024).

Supply-chain risks also remain when generated code introduces a dependency or imitates a nonexistent package. Dependency pinning, provenance checks, access controls, and existing secure software development procedures still apply. The NIST Secure Software Development Framework provides a lifecycle approach to preparing the organization, protecting software, producing well-secured software, and responding to vulnerabilities. Its generative-AI community profile addresses AI model development specifically; it should not be portrayed as experimental evidence that a particular coding assistant makes ordinary application code secure (NIST, 2022, 2024; OWASP, 2025).

7.3 Security review as measured work

A defensible AI-assisted workflow uses threat modeling for sensitive changes, checks of authentication and authorization, secret scanning, dependency review, static analysis, targeted tests, and human examination of code paths that tools miss. Automated findings need triage; human sign-off requires someone capable of understanding the threat and rejecting an

CC BY

attractive but unsafe suggestion. The relevant productivity outcome includes the time and expertise these safeguards require, as well as the defects they prevent (NIST, 2022; Perry et al., 2023; Sandoval et al., 2023).

Security analysis should distinguish a warning, a confirmed weakness, an exploitable vulnerability, and an incident. Results should report the denominator of generated samples or accepted changes, the severity and type of finding, and whether failures were detected before or after merge. Without those distinctions, a raw vulnerability percentage may combine different threat models or tool false positives and produce a misleading comparison. This review proposes that reporting discipline based on the heterogeneous units used in existing studies (Fu et al., 2025; Liu et al., 2026; NIST, 2022).

8. Quality-Adjusted Development Productivity Framework

8.1 Definition and boundaries

The Quality-Adjusted Development Productivity framework (QADP) is an author-developed analytical synthesis. It is not a validated numerical instrument, industry standard, or substitute for controlled evaluation. Its purpose is to make the review's decision rule explicit: a claimed gain should survive counting the full work cycle and testing that output quality has not deteriorated under a predeclared tolerance. This framing extends the multidimensional SPACE perspective and the secure-development lifecycle to a concrete AI-assistance decision (Forsgren et al., 2021; NIST, 2022).

The time boundary begins when the team receives a defined task and ends at a verified, integrated change; a second observation window captures follow-up fixes and maintenance. Total effort includes requirements clarification, context retrieval, prompting, waiting, draft review, testing, security checks, rework, code review, and integration. The output side records valid completion, defect severity, maintainability under later change, security findings, and delivery outcomes. Saved typing time alone does not satisfy the framework if net effort increases or an unacceptable security risk is introduced (Becker et al., 2025; Borg et al., 2026; Forsgren et al., 2021).

For a matched or sufficiently homogeneous task stratum, let T be aggregate person-minutes to acceptance plus follow-up repair, C the number of changes meeting independent functional criteria, and Q the predeclared quality and security gates. The descriptive rate C/T can be compared between AI and baseline groups only within comparable task strata and when Q is satisfied. Ten trivial patches cannot be equated with one difficult architectural change; materially heterogeneous tasks require stratification or justified weighting, with task-specific distributions reported alongside any summary. For illustration, Q could require all independently designed acceptance tests to pass, no critical regression, no unresolved high-

CC BY

severity security vulnerability after human triage, and later-change effort within a prespecified tolerance. These thresholds are hypothetical local decisions, not universal values. Severe authorization failures can be hard constraints rather than small penalties offset by time saved. A vector of time, correctness, maintainability, and security is often more informative than a scalar. This rule is an author proposal, not a validated composite measure (Forsgren et al., 2021; NIST, 2022).

8.2 Interpretation of comparisons

A tool is conditionally beneficial if it improves time or completed valid work under comparable tasks, people, and review requirements while keeping predeclared defect and security outcomes within acceptable bounds. “No statistically significant difference” in a small sample is not proof of noninferiority. A true noninferiority claim requires a justified margin, adequate precision, and sensitive outcomes. Rare but severe failures may require larger samples, adversarial tests, and postdeployment monitoring (Perry et al., 2023; Sandoval et al., 2023; NIST, 2022).

The framework also separates automation from augmentation. If a model drafts a patch but the developer independently understands and verifies it, the tool augments human judgment. If an agent modifies a repository and opens a pull request with limited oversight, the scope of automation and the necessary controls are greater. Task risk, data sensitivity, available context, reversibility, and reviewer expertise should determine the verification intensity, not a product's marketing category (NIST, 2022, 2024; OWASP, 2025).

9. Moderators of Benefit and Harm

9.1 Developer experience and project familiarity

An inexperienced developer may benefit from explanations and examples yet lack the expertise to recognize subtle failure modes. A highly experienced maintainer may already know a familiar project so well that reading and correcting model output costs more than writing directly. Cui et al. (2026) found greater adoption and gains among less experienced developers, whereas Becker et al. (2025) observed slowdown among experienced maintainers. These are not contradictory population estimates because work, setting, and tool differed (Becker et al., 2025; Cui et al., 2026).

Experience should be operationalized beyond job title: language proficiency, knowledge of the repository, security expertise, and prior use of an assistant can each moderate results. Comparing “junior” and “senior” groups without accounting for task assignment risks confounding. A study should record who can assess an output independently and whether AI changes the allocation of work to reviewers. This recommendation follows from the observed

CC BY

heterogeneity and the need to measure team rather than isolated drafting outcomes (Borg et al., 2026; Cui et al., 2026; Forsgren et al., 2021).

9.2 Task structure and verification cost

Small, well-specified tasks offer a clear target and cheap feedback; ambiguous architectural changes and security-critical integrations require contextual knowledge and stronger checks. This is a plausible mechanism consistent with the contrast between a bounded HTTP-server experiment and mature-repository work, rather than a causal threshold proved by those studies. An organization should stratify its evaluation by task class before concluding that an average benefit applies to every change (Becker et al., 2025; Peng et al., 2023).

The model's ability to access repository context can improve local relevance while introducing more data handling and injection exposure. Verification cost may rise with patch size, number of touched components, or uncertainty about requirements. Documented coding conventions, good automated tests, and clear review ownership can reduce these costs, but such practices should be measured as moderators rather than assumed to neutralize all risks. The QADP proposal therefore treats organizational capability as part of the intervention. Table 2 specifies the proposed domains and measures (Borg et al., 2026; NIST, 2022; OWASP, 2025).

9.3 Tool and model change

Coding systems change rapidly; an estimate tied to an early-2025 model is not timeless. In a February 2026 update, METR reported raw follow-up results suggestive of speedups with later tools, but its own assessment stressed that developers and tasks selected out of the experiment, payment changed, and time accounting became unreliable when agents ran concurrently. The update is therefore weak contextual evidence of temporal change, not a precise causal speedup estimate (Becker et al., 2026). Reports should record model and version or access date, assistant mode (inline completion, conversational support, repository-aware assistance, or autonomous/semi-autonomous agent), context access, permissions, enabled tools, and review workflow. Model changes and workflow changes should be distinguished (Becker et al., 2025, 2026; Jimenez et al., 2024).

A benchmark score can assist comparison among systems but should not be substituted for measured work by the target team's developers. Benchmark tasks may differ in language, age, test quality, data exposure, and opportunity for interaction. In particular, a model solving a recorded issue does not establish that a human team will merge, deploy, and maintain its patch safely. Benchmark evidence and field evidence should occupy distinct columns in an evidence table (Jimenez et al., 2024; Cui et al., 2026; NIST, 2022).

CC BY

10. Organizational Evaluation Protocol

10.1 Prospective comparison

Before deployment, a team can preregister task categories, acceptable quality thresholds, outcomes, and analysis rules. Comparable developers can be randomized to permitted AI access by task, with care for learning, crossover, and information leakage. A baseline group should use the normal development environment and receive equivalent tests and review. Task assignment, incomplete tasks, and non-use among the AI-access group must be recorded because access and actual use estimate different effects (Becker et al., 2025; Cui et al., 2026; Peng et al., 2023).

The primary outcome should be time to a change accepted after independent functional and security checks; secondary outcomes can include active time, review time, rework, defect escape, maintainability on later change, developer satisfaction, and team throughput. The analysis should report confidence intervals and variation across developers and task types, rather than only a favorable average. A baseline with weak tests or relaxed review would distort any comparison, so quality gates must be applied equally to AI and non-AI changes (Borg et al., 2026; Forsgren et al., 2021; NIST, 2022).

10.2 Follow-up and governance

Post-merge observation should follow defects, reverts, incidents, and maintenance effort across a stated horizon. Security events are infrequent and unevenly severe, so absence of an observed incident in a short pilot provides limited assurance. The team should record why a generated suggestion was rejected, modified, or accepted, including whether the developer understood it. Sensitive repositories need rules for data disclosure and tool permissions, while high-risk changes need reviewers with relevant expertise (NIST, 2022, 2024; OWASP, 2025).

Evaluation should distinguish process adherence from outcome. A team can log prompt use and scans without improving software; conversely, an assistant might be useful in a low-risk drafting role where a simple independent check is sufficient. Decisions should be revisited when the tool version, workflow, team, or codebase changes. The proposed protocol deliberately measures net organizational effects and does not assume that universal prohibition or universal adoption follows from the current mixed evidence (Becker et al., 2025; Cui et al., 2026; NIST, 2022).

11. Discussion

The central question has a conditional answer. Randomized experiments provide credible evidence that coding assistance can improve speed or throughput on selected tasks. A separate randomized study demonstrates that assistance can also slow experienced maintainers in familiar codebases. The difference is not resolved by choosing a preferred

CC BY

study: these experiments estimate effects for different participants, tools, tasks, periods, and endpoints. The evidence supports targeted evaluation of fit, not a universal “percent faster” claim (Becker et al., 2025; Cui et al., 2026; Peng et al., 2023).

The second half of the question—whether gains occur without more errors or vulnerabilities—has less complete evidence. The GitHub quality study suggests better test outcomes and modest reviewer-rated improvements for its assignment; Borg et al. (2026) extend inquiry to later maintainability. Security studies find plausible weaknesses and sometimes worse assisted-user outcomes, but one C-language trial reports a smaller effect. There is no single contemporary experiment that establishes across a broad set of production changes both a net time gain and security noninferiority over a sufficiently long follow-up (Borg et al., 2026; GitHub, 2024; Perry et al., 2023; Sandoval et al., 2023).

A useful synthesis is that AI may shift work from generating a candidate toward specifying, selecting, and validating one. In a longitudinal mixed-methods preprint, Vella and Blincoe (2026) describe supervisory engineering work involving direction, evaluation, and correction of AI output, based substantially on participants’ reported experience. Fan et al. (2026) measured verification-related interactions in bounded tasks. These sources empirically anchor the proposed mechanism, but do not establish a universal shift or a net gain for every workplace. Verification may erase initial speed gains or shift effort to reviewers and later releases (Becker et al., 2025; NIST, 2022).

The QADP framework avoids presenting one indicator as a complete answer. It prioritizes a valid change over text volume, recognizes severe security failures as constraints, and requires later maintainability to be observed rather than guessed from style. Its value is conceptual until independently tested against real adoption decisions and outcomes. An empirical validation could examine whether QADP-guided pilots better predict subsequent delivery and incident rates than pilots using acceptance rate or self-reported hours saved alone (Forsgren et al., 2021; Borg et al., 2026; NIST, 2022).

12. Limitations and Research Agenda

This review is selective and single-authored. It did not retain a full database search and screening record, assess every potentially eligible study, calculate pooled effects, or obtain a second independent reviewer. Its compact corpus concentrates on published English-language evidence and Python, JavaScript, Java, and C tasks; Rust, Go, C++, Kotlin, Swift, embedded systems, and safety-critical languages remain underrepresented. Findings from ordinary repository and web tasks cannot be generalized automatically to medical devices, aviation, automotive safety, financial infrastructure, or industrial control. Accessibility conditions, regions, and model generations also differ across settings. These limits preclude a

CC BY

comprehensive census or a claim that effects remain stable over time (Becker et al., 2025, 2026; Cui et al., 2026; Fan et al., 2026).

Some evidence comes from a vendor-conducted study, and repository studies face uncertainty over how AI involvement is attributed. Security scanners observe a subset of weakness classes, while experiments rarely have power to detect uncommon high-impact incidents. Downstream maintenance can take months or years to surface. These problems cannot be repaired by averaging available percentages; they call for independent replication, shared outcome definitions, and longitudinal tracking of matched changes (Borg et al., 2026; GitHub, 2024; Liu et al., 2026).

Future studies should preregister realistic task strata; randomize or otherwise construct credible counterfactuals; retain model and configuration details; and measure the same task through drafting, independent review, security validation, deployment, and later evolution. Reports should give denominators for suggestions, accepted patches, tested changes, confirmed weaknesses, and failures; distinguish scanner findings from exploitable vulnerabilities; and publish null and adverse results. Multi-site studies can test whether organizational review capacity and repository maturity moderate the effect (Becker et al., 2025; Cui et al., 2026; NIST, 2022).

Research should also examine whether generated tests independently detect errors, whether assistants improve or degrade understanding for novice developers, and whether agents with repository permissions create new attack surfaces. Outcomes should include the time required to verify an answer, not merely the time to generate it. Comparing assistants to strong non-AI baselines with existing IDE support and established secure-development processes would make any estimated incremental benefit more useful for decisions (Forsgren et al., 2021; NIST, 2022, 2024; OWASP, 2025).

13. Conclusion

Generative AI assistants can accelerate some bounded development tasks and increase measured output in some workplaces, but they can also slow experienced maintainers working in complex, familiar repositories. The current literature does not establish a general speed gain with no accompanying increase in defects, vulnerabilities, or maintenance cost. The defensible conclusion is conditional: evaluate the tool in the intended workflow, count verification and repair, apply the same quality and security gates to both groups, and follow the resulting code beyond its first passing test (Becker et al., 2025; Borg et al., 2026; Cui et al., 2026; NIST, 2022; Peng et al., 2023).

CC BY

Data Availability

No original participant or repository dataset was created for this narrative review. The cited sources and their public locations are listed in the references.

Declaration on Generative AI Use

Generative AI was used to assist with literature searches, reference filtering, content verification, and translation and editing into academic English. The author reviewed the manuscript, verified its claims and references against the cited sources, and takes full responsibility for its content, interpretation, and final submission.

CC BY

Table 1 Comparative Evidence on Developer Productivity and Quality

Study	Design and task	Reported finding	Main boundary
Peng et al. (2023)	Randomized JavaScript HTTP server task	55.8% faster completion with Copilot	One bounded task; limited maintenance follow-up
Cui et al. (2026)	Three workplace field experiments; 4,867 developers	26.08% more completed tasks, pooled	Task counts are not security or defect rates
Becker et al. (2025)	Randomized task access; 16 maintainers, 246 tasks	19% longer with early-2025 AI allowed	Experienced maintainers in familiar mature repositories; preprint
Borg et al. (2026)	Phase 1 observational initial task; Phase 2 randomized later manual evolution	Phase 1: 30.7% shorter median initial-task time; Phase 2: no clear systematic downstream time or code-quality difference	Do not treat Phase 1 as randomized or absence of a Phase 2 difference as proof of equivalence
GitHub (2024)	Vendor-run randomized Python assignment	More tests passed; modest reviewer-rated quality gains	Single assignment and vendor involvement
Perry et al. (2023)	Security-focused human–AI user study	Assisted users produced less secure solutions in studied tasks	Specific model and tasks; does not estimate production incidence
Sandoval et al. (2023)	Randomized study on low-level C tasks	Comparatively small security effect in its setting	Language- and task-specific
Fu et al. (2025)	Observational analysis of attributed snippets	Selected Python and JavaScript snippets showed weaknesses	No matched causal estimate of AI effect
Fan et al. (2026)	Controlled study; 60 participants, three Python tasks; fixed model, interface comparison	22% shorter average task time versus no-AI; verification interactions logged	Bounded tasks and interface effects; no long-term production follow-up
METR update (Becker et al., 2026)	Institutional follow-up with later tools and randomized task access	Raw results suggested speedups, but magnitude unreliable	Participation and task selection; concurrent-agent time measurement

Note. Settings and endpoints differ; percentages are not directly comparable.

CC BY

Table 2 Proposed Quality Adjusted Development Productivity Evaluation

Domain	Operational measure	Verification or interpretation
Net effort	Person-minutes from task assignment to accepted change, plus follow-up repair	Log prompting, review, testing, integration, and later fixes
Functional quality	Independent acceptance tests, regression failures, severity-weighted defects	Same requirements and test gates for both groups
Maintainability	Time and quality of later modification by another developer	Follow artifact beyond initial author and initial task
Security	Confirmed weaknesses, exploitability, incident severity and detection stage	Separate scanner flags from confirmed vulnerabilities
Developer experience	Satisfaction, learning, perceived workload	Report separately from elapsed time and throughput
Context	Task class, repository familiarity, language, model/version, reviewer skill	Stratify effects and report configuration and denominators
Illustrative gate Q	Independent acceptance tests pass; no critical regression or unresolved high-severity vulnerability; later-change effort within preset tolerance	Predeclare locally, apply equally to both arms, and report exceptions and follow-up horizon

Note. QADP is an author-developed analytical framework, not a validated score or formal standard. A serious security failure is a constraint, not a small penalty interchangeable with time saved.

CC BY

References

Becker, J., Rush, N., Barnes, E., & Rein, D. (2025). Measuring the impact of early-2025 AI on experienced open-source developer productivity [Preprint]. arXiv. https://arxiv.org/abs/2507.09089

Becker, J., Rush, N., Cunningham, T., Rein, D., & Mahamud, K. (2026, February 24). We are changing our developer productivity experiment design. METR. https://metr.org/blog/2026-02-24-uplift-update/

Borg, M., Hewett, D., Hagatulah, N., Couderc, N., Söderberg, E., Graham, D., Kini, U., & Farley, D. (2026). Echoes of AI: Investigating the downstream effects of AI assistants on software maintainability. Empirical Software Engineering, 31, Article 161. https://doi.org/10.1007/s10664-026-10889-1

Cui, K. Z., Demirer, M., Jaffe, S., Musolff, L., Peng, S., & Salz, T. (2026). The effects of generative AI on high-skilled work: Evidence from three field experiments with software developers. Management Science, advance online publication. https://doi.org/10.1287/mnsc.2025.00535

Dakhel, A. M., Nikanjam, A., Majdinasab, V., Khomh, F., & Desmarais, M. C. (2024). Effective test generation using pre-trained large language models and mutation testing. Information and Software Technology, 171, Article 107468. https://doi.org/10.1016/j.infsof.2024.107468

Fan, G., Liu, D., Pan, L., & Zhang, R. (2026). When help hurts: Verification load and fatigue with AI coding assistants. Proceedings of the 2026 CHI Conference on Human Factors in Computing Systems, Article 733, 1–25. https://doi.org/10.1145/3772318.3791176

Forsgren, N., Storey, M.-A., Maddila, C., Zimmermann, T., Houck, B., & Butler, J. (2021). The SPACE of developer productivity: There's more to it than you think. Queue, 19(1), 20–48. https://doi.org/10.1145/3454122.3454124

Fu, Y., Liang, P., Tahir, A., Li, Z., Shahin, M., Yu, J., & Chen, J. (2025). Security weaknesses of Copilot-generated code in GitHub projects: An empirical study. ACM Transactions on Software Engineering and Methodology, 34(8), 1–34. https://doi.org/10.1145/3716848

GitHub. (2024, November 18). Does GitHub Copilot improve code quality? Here's what the data says. https://github.blog/news-insights/research/does-github-copilot-improve-code-quality-heres-what-the-data-says/

Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., & Narasimhan, K. (2024). SWE-bench: Can language models resolve real-world GitHub issues? Proceedings of the International Conference on Learning Representations. https://proceedings.iclr.cc/paper_files/paper/2024/hash/edac78c3e300629acfe6cbe9ca88fb84-Abstract-Conference.html

Liu, Y., Widyasari, R., Zhao, Y., Irsan, I. C., & Lo, D. (2026). Debt behind the AI boom: A large-scale empirical study of AI-generated code in the wild [Preprint]. arXiv. https://arxiv.org/abs/2603.28592

National Institute of Standards and Technology. (2022). Secure software development framework (SSDF) version 1.1: Recommendations for mitigating the risk of software vulnerabilities (NIST SP 800-218). https://doi.org/10.6028/NIST.SP.800-218

National Institute of Standards and Technology. (2024). Secure software development practices for generative AI and dual-use foundation models: An SSDF community profile (NIST SP 800-218A). https://doi.org/10.6028/NIST.SP.800-218A

OWASP Foundation. (2025). OWASP top 10 for LLMs and generative AI applications 2025. https://genai.owasp.org/llm-top-10/

Page, M. J., McKenzie, J. E., Bossuyt, P. M., Boutron, I., Hoffmann, T. C., Mulrow, C. D., Shamseer, L., Tetzlaff, J. M., Akl, E. A., Brennan, S. E., Chou, R., Glanville, J., Grimshaw, J. M., Hróbjartsson, A., Lalu, M. M., Li, T., Loder, E. W., Mayo-Wilson, E., McDonald, S., . . . Moher,

D. (2021). The PRISMA 2020 statement: An updated guideline for reporting systematic reviews. BMJ, 372, Article n71. https://doi.org/10.1136/bmj.n71

Pearce, H., Ahmad, B., Tan, B., Dolan-Gavitt, B., & Karri, R. (2022). Asleep at the keyboard? Assessing the security of GitHub Copilot's code contributions. Proceedings of the 2022 IEEE Symposium on Security and Privacy, 754–768. https://doi.org/10.1109/SP46214.2022.9833571

Peng, S., Kalliamvakou, E., Cihon, P., & Demirer, M. (2023). The impact of AI on developer productivity: Evidence from GitHub Copilot [Preprint]. arXiv. https://arxiv.org/abs/2302.06590

Perry, N., Srivastava, M., Kumar, D., & Boneh, D. (2023). Do users write more insecure code with AI assistants? Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, 2785–2799. https://doi.org/10.1145/3576915.3623157

Sandoval, G., Pearce, H., Nys, T., Karri, R., Garg, S., & Dolan-Gavitt, B. (2023). Lost at C: A user study on the security implications of large language model code assistants. Proceedings of the 32nd USENIX Security Symposium, 2205–2222. https://www.usenix.org/conference/usenixsecurity23/presentation/sandoval

Vella, A., & Blincoe, K. (2026). The impact of AI coding assistants on software engineering: A longitudinal study [Preprint]. arXiv. https://arxiv.org/abs/2605.23135

CC BY