



Integration of AI-Assisted Development Tools into Modern Software Delivery Workflows and Their Impact on Engineering Productivity

Michael Rainesh

East Gwillimbury, Canada
Director of Engineering, Portside, Inc.
mikrain@gmail.com

Received: 25 Aug 2026; Received in revised form: 20 Sep 2026; Accepted: 26 Sep 2026; Available online: 01 Oct 2026

Abstract— *AI-assisted development tools now enter routine software delivery through code completion, repository-aware coding assistants, automated pull request (PR) comments, and AI-supported testing. Their effect on productivity depends on the workflow stage in which engineers use them, the quality controls around generated output, and the way senior developers preserve ownership of design decisions. This article develops an analytical model for integrating AI code generation and AI-driven pull request review into modern software delivery. The study uses comparative source analysis, conceptual synthesis, typologization, and analytical generalization based on ten recent academic and industry-facing publications on large language models in software engineering, GitHub Copilot, automated review, hallucinations, package risks, and security attacks. The article separates measured evidence from practitioner-reported operational estimates. It identifies where AI shortens implementation work, where it adds review burden, and which governance mechanisms protect code quality, onboarding, mentoring, dependency control, and release stability.*

Keywords— *AI-assisted software development, code generation, pull request review, engineering productivity, software delivery workflows, GitHub Copilot, large language models, code quality, software governance, developer experience*

I. INTRODUCTION

AI-assisted development has become part of everyday engineering work. Developers use code completion, conversational coding, repository-aware assistants, AI-generated tests, and automated pull request comments inside delivery pipelines that already contain IDE automation, linters, static analysis, peer review, CI/CD checks, and security scanning. Productivity in such environments cannot be reduced to the speed with which one developer writes a function. Engineering teams judge productivity through cycle time, defect leakage, review latency, release stability, rework, onboarding speed, and the ability to scale delivery without losing architectural control.

The topic matters because AI tools change the location of effort inside the software delivery chain. A coding assistant can shorten the time between task description and first working draft. The same draft still has

to pass tests, review, merge, deployment, and operational monitoring. A pull request reviewer can flag a missing check or an unsafe pattern before a senior engineer reads the change. The same reviewer can fill the thread with weak comments that slow the merge. The productivity question therefore belongs to the workflow, not to the coding session alone.

The aim of this article is to develop an analytical model for integrating AI-assisted development tools into modern software delivery workflows and evaluating their influence on engineering productivity without treating practitioner estimates as controlled research results.

Three research objectives guide the argument. The first objective is to distinguish two layers of AI use in the software development lifecycle: AI code generation and AI-driven pull request review. The second objective is to examine how these layers affect coding speed, review

capacity, integration time, quality control, onboarding, mentoring, and senior engineering work. The third objective is to define governance principles for adoption, including measurement, risk control, human ownership, mentoring routines, and decision rules for high-risk code.

The novelty of the article lies in its workflow-centered interpretation of AI-assisted development. Existing studies often examine developer speed, user perception, review comments, hallucinations, or security exposure as separate topics. The present article connects those findings into one delivery model that places AI between developer cognition, repository conventions, quality gates, and organizational learning.

The hypothesis is that AI-assisted development improves engineering productivity when teams integrate code generation and pull request review as governed workflow components. Ungoverned adoption can raise local coding speed while shifting hidden costs into review queues, integration delay, security exposure, mentoring gaps, and long-term maintainability.

II. MATERIALS AND METHODS

The material base was formed through targeted literature selection in ACM Digital Library, arXiv, USENIX, Microsoft Research, SSRN, and scholarly indexing services. Search phrases combined “AI coding assistants,” “GitHub Copilot productivity,” “large language models for software engineering,” “automated code review,” “pull request review LLM,” “AI-generated code hallucination,” “package hallucination,” and “code completion security.” Inclusion criteria covered publication date from 2023 to 2026, direct relevance to software delivery workflows, focus on code generation or pull request review, presence of empirical evidence or systematic synthesis, and sufficient methodological transparency. Exclusion criteria removed generic AI management papers, vendor marketing materials without reproducible claims, articles outside software engineering, and publications where productivity claims had no connection to software delivery mechanisms. The initial pool contained 42 works. Screening of titles, abstracts, methodology descriptions, and relevance to the three objectives reduced the corpus to 10 sources. The final corpus combines controlled productivity research, field experiments, public-sector deployment, systematic review literature, ACM-published evidence on AI-assisted code review, peer-reviewed research on LLM-supported code-quality remediation, user-study evidence on LLM-assisted programming, and security research on hallucinations, package risks, and code completion attacks (Cui et al., 2026, Hou et al., 2024, Ng et al., 2024, Patcas & Motogna, 2026,

Peng et al., 2023, Spracklen et al., 2025, Vijayvergiya et al., 2024, Weber et al., 2024, Yan et al., 2024, Zhang et al., 2025). Practitioner-reported engineering experience was used as illustrative boundary evidence for adoption patterns, especially where engineering leadership experience concerns AI development strategy, generative AI code review, and a business-logic AI proof of concept. These materials were not treated as experimental measurements.

The article uses comparative analysis to separate AI code generation from AI-assisted pull request review. Source analysis supports evaluation of findings from recent studies. Conceptual synthesis connects productivity with SDLC stages. Typologization helps define adoption patterns. Analytical generalization supports governance principles. These methods correspond to the research objectives: comparison clarifies the two AI layers, synthesis explains productivity effects across workflow stages, and typologization supports the implementation model.

III. RESULTS

Recent research on AI-assisted development first requires a distinction that managerial language often blurs. AI code generation and AI-driven pull request review enter the software delivery chain at different points. Code generation shapes the moment when a developer turns intent into code. Pull request review shapes the moment when a proposed change becomes visible to teammates, maintainers, CI systems, and release governance. Treating both layers as a generic AI productivity tool weakens evaluation because each layer changes a different bottleneck.

Controlled evidence on GitHub Copilot links AI pair-programming with shorter completion time under bounded conditions. In an experiment where developers implemented an HTTP server in JavaScript, the Copilot group completed the task 55.8% faster than the control group (Peng et al., 2023). The finding matters for the first objective because it isolates the effect of AI code generation on implementation speed. The task design also limits the scope of inference. A bounded programming task differs from enterprise work where a developer must preserve domain rules, backward compatibility, security constraints, deployment behavior, and cross-team ownership. The result supports AI generation for well-scoped implementation work, while enterprise delivery still requires validation across the full pipeline.

Field experiments extend this interpretation. Evidence from software development settings at Microsoft, Accenture, and a Fortune 100 company connects generative AI use with productivity changes in high-skilled software work (Cui et al., 2026). The value of this evidence lies in

the organizational setting. Developers worked inside company environments with existing codebases, workplace routines, and productivity measures defined by the organization. For the second objective, this strengthens the claim that teams should read productivity across the workflow, including developer experience, task allocation, review practices, and adoption conditions.

User-study evidence on LLM-assisted programming gives a more cautious reading of coding acceleration. Weber et al. (2024) compare programming conditions with and without AI support and show that LLM-based assistance can reduce task completion time in controlled programming work. The same line of evidence keeps the evaluation tied to correctness, maintainability, and verification behavior. For software delivery, this distinction matters: a faster draft has value only when developers still check whether the generated code satisfies requirements, follows local conventions, and fits the surrounding codebase (Weber et al., 2024).

Public-sector evidence brings onboarding and developer satisfaction into the same discussion. A GovTech Singapore study of GitHub Copilot adoption reports higher coding or task speed, strong perceived developer satisfaction, and particular gains for junior developers (Ng et al., 2024). This source links AI adoption with learning conditions, not only task velocity. Junior engineers can benefit from immediate examples, syntax support, and faster exploration of unfamiliar APIs. The same mechanism can weaken learning when engineers accept generated solutions without understanding them. In workflow terms, AI supports onboarding when code reading, review feedback, and explicit discussion of design reasoning remain part of the process. It becomes risky when the tool replaces the mentoring conversation that connects a task to system architecture.

A systematic literature review of large language models for software engineering maps LLM use across requirements, code generation, testing, maintenance, and software analysis (Hou et al., 2024). This review supports the first objective because AI-assisted development reaches beyond code completion. Software delivery contains several transformation points. Requirements become design choices, design choices become implementation, implementation becomes reviewed change, reviewed change becomes deployable artifact, and deployed artifact becomes operational behavior. AI can enter each point, but engineers can trust output only when local repository knowledge, acceptance criteria, and engineering standards are available for validation.

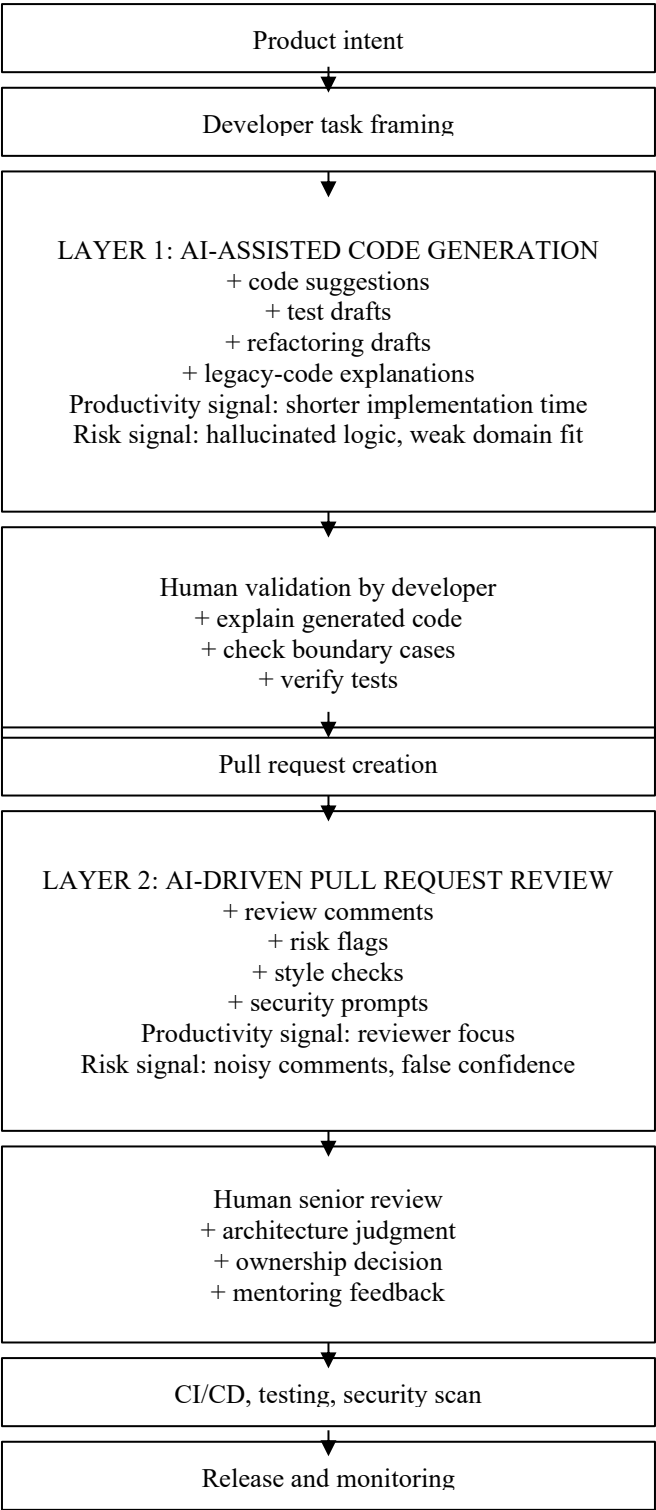
The first layer of the proposed model concerns AI code generation. It covers prompt-to-code transformation,

completion of repeated patterns, generation of unit tests, explanation of legacy code, refactoring suggestions, and creation of scaffolding. Its productivity contribution is strongest where the task has a recognizable pattern and clear acceptance criteria. Examples include DTO mapping, boilerplate service methods, test skeletons, small UI state changes, migration scripts, and API client code. Risk rises when code generation touches domain-specific pricing logic, financial calculations, authorization, data retention, concurrency, or infrastructure configuration. These areas require repository-specific and business-specific reasoning that general-purpose models can imitate without controlling the underlying constraints.

The second layer concerns AI-driven pull request review and adjacent code-quality checks. ACM-published industrial evidence on AutoCommenter shows that an LLM-backed system can learn and enforce coding best practices across several programming languages in a large code-review environment (Vijayvergiya et al., 2024). This result supports the second objective because it treats AI review as part of an established engineering workflow, not as a standalone comment generator. Code review in a delivery workflow is a judgment process involving defect detection, maintainability, architecture, team norms, and release risk. If an AI reviewer increases comment volume without improving signal quality, human reviewers must classify, accept, reject, or explain the tool's suggestions. That work reduces the practical value of automation.

Peer-reviewed evidence on LLM-supported code-quality remediation strengthens this filter-based interpretation. Patcas and Motogna (2026) evaluate several state-of-the-art LLMs on issues identified by SonarQube and report that generated fixes can reduce static-analysis findings, although effectiveness varies by model, issue type, and semantic preservation. For the workflow model, the implication is narrow but useful: AI can support reviewers by addressing routine quality findings before senior engineers spend time on architecture, hidden coupling, risk trade-offs, and long-term maintainability (Patcas & Motogna, 2026). Figure 1 presents the two-layer model of AI-assisted delivery integration. It shows AI code generation as an upstream acceleration layer and AI pull request review as a downstream quality-assistance layer. The figure is a conceptual map linking tool placement to productivity indicators and governance gates.

Figure 1. Two-layer model for integrating AI-assisted tools into software delivery workflows (compiled by the author based on Hou et al., 2024, Vijayvergiya et al., 2024, and Patcas & Motogna, 2026)



The figure separates acceleration from containment. AI generation increases the pace at which code enters the repository. AI review attempts to improve the quality of that inflow before merge. Without governance, the second layer becomes a compensation mechanism for risks created by the first layer. With governance, the layers complement each other: developers use AI to draft and explore, while PR-level assistance helps reviewers concentrate on non-trivial judgment.

Code quality drift follows from this model. It appears when generated code works locally but gradually changes the texture of the codebase. Naming conventions lose consistency. Abstractions multiply. Test coverage becomes shallow. Dependency choices become opportunistic. Domain concepts lose precision. The systematic review literature confirms that LLMs now cover many software engineering tasks, but broad coverage does not guarantee repository-level coherence (Hou et al., 2024). Repository coherence depends on architectural memory, review habits, test design, and local conventions. AI-generated code respects those conventions only when prompts, context windows, retrieval mechanisms, and review procedures expose them.

Hallucination research gives this concern a technical basis. Repository-level code generation studies identify hallucinations as a practical problem in complex development settings where models must handle dependencies, project context, and non-local constraints (Zhang et al., 2025). This matters for both code generation and review. In generation, hallucination can appear as invented methods, incorrect API use, missing imports, unreachable control flow, or plausible code that violates business rules. In review, hallucination can appear as criticism of a defect that is not present or advice that misunderstands surrounding code. The workflow cost differs in each case. Generated hallucination creates defects. Review hallucination creates noise and reviewer fatigue.

Package hallucinations make the supply-chain risk concrete. Research on code-generating LLMs found hallucinated package names across programming languages and model types, creating a package confusion threat where attackers can register invented packages that AI systems have suggested (Spracklen et al., 2025). This finding matters for the third objective because it moves the discussion beyond code correctness. AI-assisted development can affect dependency governance, build security, and open-source risk. A team that allows AI-generated dependency suggestions without package allowlists, internal mirrors, SBOM review, or security scanning creates a risk that unit tests may miss.

Security research on code completion models extends the issue further. Backdoor attack studies show that attackers can manipulate code completion systems through poisoned fine-tuning data or disguised vulnerabilities that evade detection (Yan et al., 2024). This finding affects organizations exploring private fine-tuning, internal code assistants, or repository-specific models. Local AI customization can improve relevance, but it expands the attack surface. Strong governance therefore has to go beyond rules about secrets in prompts. It needs model

provenance, dependency scanning, prompt-injection awareness, secure configuration, and auditability of generated code paths.

The relationship between AI tools and senior engineers requires separate attention. Studies on AI-assisted review and code-quality remediation show that LLM-based support can help teams address routine findings and apply coding practices when engineers use it as a filter (Patcas & Motogna, 2026, Vijayvergiya et al., 2024). Senior engineers therefore remain central to the delivery chain. Their work shifts. They evaluate generated work, maintain architectural coherence, design review standards, and mentor junior developers in the interpretation of AI output. Seniority becomes less tied to manual production of every solution and more tied to system judgment.

This shift affects onboarding. AI assistants can shorten the distance between a new developer and the first meaningful contribution by explaining code, generating examples, and reducing syntax friction (Ng et al., 2024, Peng et al., 2023). Over-reliance creates another pattern: a junior developer can learn to accept plausible solutions before learning why a design fits the product. AI-assisted onboarding works best when tool use is paired with mentoring rituals: explain the accepted suggestion, compare it with the repository standard, write a test that proves the behavior, and document the reason for the chosen implementation. Productivity then includes learning velocity, not only ticket velocity.

The boundary between developer tooling AI and business-logic AI should stay visible. Practitioner-reported engineering experience describes an AI proof of concept for a discount system where a small team trained a model to interpret submitted project details and suggest eligible discounts from historical information. This case illustrates another AI layer. It concerns business decision automation, data warehouse readiness, and organizational data maturity. It should not serve as evidence that code generation tools automatically improve SDLC productivity. Its value for this article is contrastive. Developer tooling AI changes how engineers produce and review software. Business-logic AI changes what the software decides for customers. Both require governance, but the validation object differs. In the first case, engineers validate code, tests, review, and delivery metrics. In the second case, the organization validates data quality, eligibility logic, business accountability, model drift, and decision auditability.

The comparative reading of productivity, review, and security literature supports the hypothesis with a necessary qualification. AI-assisted development can improve engineering productivity when organizations define which workflow stage the tool serves, what output

counts as acceptable, who owns the final decision, and which metrics reveal hidden costs. Productivity evidence from controlled, field, and user-study research supports acceleration in coding activity under defined conditions (Cui et al., 2026, Peng et al., 2023, Weber et al., 2024). Code-review and code-quality research shows that AI assistance works best when teams connect it to existing review rules, static analysis, and human validation (Patcas & Motogna, 2026, Vijayvergiya et al., 2024). Security and hallucination research shows that the cost of unguided AI output may accumulate in dependency risk, false confidence, and maintainability loss (Spracklen et al., 2025, Yan et al., 2024, Zhang et al., 2025).

Organizations sometimes report internal improvements such as faster delivery or fewer release issues after AI adoption. Academic writing should frame such figures as practitioner-reported operational evidence. These estimates are useful because they reflect managerial experience in real teams. They should not be converted into experimental findings unless the organization provides baseline definitions, sampling logic, measurement windows, and comparable control groups. A reported 40% or 50% improvement in delivery speed can inform a measurement plan that tracks cycle time, PR size, review latency, escaped defects, rework, and developer satisfaction before and after rollout.

The results indicate that AI-assisted software delivery works as a governed socio-technical arrangement. The technical layer contains models, IDEs, PR bots, static analysis, CI pipelines, tests, and dependency controls. The social layer contains review culture, senior mentoring, team norms, trust calibration, and leadership decisions. Productivity emerges from the interaction between these layers. Two organizations can adopt the same tool and receive different outcomes because the surrounding workflow differs.

IV. DISCUSSION

A restrained positive position follows from the results: engineering teams should integrate AI-assisted development tools as bounded workflow components. Tools such as GitHub Copilot, Cursor, Claude Code, and PR review bots already help with many routine tasks. The decisive managerial question concerns placement, ownership, and verification. A coding assistant inside an IDE accelerates a developer's movement through implementation. A review assistant inside the pull request pipeline changes the information available to reviewers. These interventions require separate adoption rules.

The first implementation principle is task-envelope design. Teams should encourage AI code

generation for repetitive, low-risk, pattern-heavy work where acceptance criteria are clear. Examples include scaffolding, test drafts, small refactors, interface adapters, documentation snippets, and local transformations. Teams should require additional review for authorization logic, pricing logic, financial workflows, cryptography, infrastructure permissions, data deletion, and production incident fixes. Broad AI policies often fail because they give developers abstract permission. Useful policies name tasks, risks, and gates.

The second principle is human ownership. Every AI-generated fragment entering the repository needs a human owner who can explain the behavior, test boundary cases, and defend the design during review. The responsible developer cannot treat the assistant as an external author. In delivery terms, code ownership remains indivisible. A tool proposes. An engineer commits.

The third principle concerns senior engineers. AI adoption raises the value of senior judgment because the volume of plausible code grows. Senior engineers should spend less time policing formatting and obvious patterns, but more time on architecture, hidden coupling, non-functional requirements, test sufficiency, and mentoring. Engineering leaders should reflect this shift in workload planning. If AI tools produce more pull requests, reviewer capacity must grow or the organization must narrow the scope of automated generation. Otherwise, gains at the coding level become review queues. Table 1 compares adoption decisions for the two layers of AI-assisted development. Its purpose is to separate tool function, benefit, risk, and control mechanism so engineering leaders do not treat all AI tooling as one intervention.

Table 1. Decision logic for adopting AI-assisted development tools in software delivery workflows (compiled by the author based on Peng et al., 2023, Hou et al., 2024, Vijayvergiya et al., 2024, and Patcas & Motogna, 2026)

Adoption layer	Appropriate use	Expected productivity effect	Main risk	Governance response
AI code generation	Boilerplate, tests, local refactoring, API examples, code explanation	Shorter implementation time and faster exploration of unfamiliar code	Plausible code that misses repository conventions or business rules	Human explanation requirement, test coverage rule, restricted use for high-risk domains
AI-driven PR review	Routine comments, style issues, suspicious patterns, missing checks, preliminary security prompts	Reduced reviewer load and earlier defect discovery when comment quality is high	Noisy comments, faulty corrections, longer PR closure	Signal-quality monitoring, reviewer override, comment taxonomy, periodic sampling
AI-assisted onboarding	Guided code reading, examples, first-task scaffolding, local explanations	Faster entry into codebase and lower syntax friction	Shallow learning and uncritical acceptance	Mentored AI use, explanation logs, pair review, task retrospectives
AI in business logic	Discount suggestions, document classification, customer-facing decision support	Process automation outside developer tooling	Data drift, accountability gaps, decision opacity	Model validation, data governance, audit trails, human approval for sensitive decisions

The table supports conditional adoption. AI code generation is most defensible when the team can specify expected behavior and verify it through tests or review. AI PR review has value when the assistant improves signal density, not when it floods the thread with generic comments. AI-assisted onboarding should accelerate learning while keeping mentoring in place. Business-logic AI belongs to another governance regime because it affects organizational decisions, not only engineering throughput.

A staged implementation model follows from this logic. The first stage is limited enablement. A team allows

AI coding tools for low-risk tasks and collects baseline metrics for cycle time, PR size, review time, rework rate, escaped defects, and developer satisfaction. The second stage introduces validation rules: the committer must understand generated code, tests must cover intended behavior, and every new dependency must pass verification. The third stage adds PR-level AI review after the team defines comment categories and measures whether automated comments are actionable. The fourth stage scales adoption through governance dashboards, training, and periodic audits.

Measurement needs care. Many engineering organizations already track velocity, but velocity alone is too blunt for AI-assisted delivery. A team can finish more tickets while increasing rework, weakening tests, or moving risk into production. Metrics should combine speed, quality, and learning. A practical dashboard should include median cycle time, review latency, PR closure time, reopened work, defect leakage, test coverage movement, dependency

additions, security findings, review comment acceptance, and developer self-reported cognitive load. The aim is not bureaucratic surveillance. The aim is to detect whether productivity gains are real or displaced. Table 2 proposes a monitoring matrix for AI-assisted development adoption. Its purpose is to connect observable indicators with managerial interpretation and intervention.

Table 2. Monitoring matrix for AI-assisted development productivity and risk (compiled by the author based on Weber et al., 2024, Ng et al., 2024, Zhang et al., 2025, Spracklen et al., 2025, and Yan et al., 2024)

Monitoring area	Indicator	Interpretation	Corrective action
Delivery speed	Cycle time from task start to merge	Detects whether coding acceleration reaches the delivery pipeline	Compare by task type and PR complexity
Review burden	PR closure time and reviewer comment load	Detects whether AI creates integration friction	Limit bot comments, tune prompts, refine review categories
Code quality	Escaped defects and rework rate	Detects hidden cost after merge	Strengthen tests, require design review for AI-heavy changes
Security	New dependencies, vulnerability alerts, suspicious generated patterns	Detects package and code completion risk	Enforce allowlists, SBOM review, dependency scanning
Learning	Junior developer task completion and explanation quality	Detects whether AI supports onboarding or masks weak understanding	Use pair review and explanation requirements
Senior workload	Time spent on architecture, mentoring, and repeated AI-output corrections	Detects whether senior attention shifts productively	Separate routine checks from expert judgment and adjust staffing

The matrix protects adoption from two errors. One error appears when managers treat faster coding as proof of organizational improvement. The other appears when teams reject useful tools because hallucinations and security risks exist. A measured approach avoids both. It treats AI as a delivery capability whose value has to be maintained through feedback loops.

Governance should include prompt and context discipline. Developers need training in framing tasks with repository constraints, expected behavior, test expectations, and non-functional limits. Review bots should focus on categories where they help: missing null checks, unsafe dependency additions, inconsistent error handling, missing tests, potential security problems, and style deviations already defined in team standards. Generic comments should be suppressed. AI review earns trust when it respects the team's definition of useful feedback.

Engineering governance should separate confidence from correctness. AI tools often produce fluent explanations that create an impression of certainty. Teams should counter this tendency through evidence requirements. A suggestion that changes business behavior

needs tests. A suggestion that adds a package needs dependency verification. A change touching security needs security review. A modification of shared architecture needs senior approval. This is ordinary software engineering discipline applied to a faster source of proposed change.

Practitioner-reported engineering leadership experience fits this model when handled with care. Reported improvements such as faster delivery, fewer release issues, or higher team output can signal operational potential. They become stronger when paired with transparent measurement design. For an engineering leader, the most defensible claim is that AI adoption, standardized SDLC workflows, and PR-level review automation can improve delivery conditions when the organization implements governance. The claim becomes weak when estimated percentages appear as research findings without data structure.

The discount-system proof of concept gives a useful boundary case. A model intended to reduce manual input in discount eligibility can expose the need for centralized data collection, historical data quality, and

cross-unit data governance. That lesson transfers indirectly to developer tooling. AI coding tools expose whether a company has clear architecture standards, coherent documentation, stable tests, and reliable review practices. AI does more than automate a workflow. It reveals where the workflow relied on informal knowledge.

Engineering managers therefore need a cautious acceleration sequence. Start with low-risk generation. Document acceptable use. Gather baseline metrics. Add PR-level AI review only where it improves reviewer signal. Train developers to explain AI output. Formalize security gates for dependencies and sensitive code paths. Senior engineers should lead these rules because they know where local productivity gains can damage long-term maintainability.

V. CONCLUSION

AI-assisted development tools improve software delivery most convincingly when teams treat code generation and pull request review as separate SDLC layers. Code generation mainly accelerates implementation and exploration. AI-driven review affects validation, integration, and senior engineering attention. Their combination can increase delivery capacity, but each layer requires different controls because each creates different risks.

The productivity impact is multidimensional. Evidence supports faster completion of bounded coding tasks and positive developer experience. Studies of LLM-assisted programming, code review, and code-quality remediation identify the need for verification, signal control, and human judgment around generated output. The strongest interpretation is workflow-based: AI can increase local speed, but organizational productivity depends on review, testing, security checks, merge decisions, and release stability.

The hypothesis is confirmed with qualification. AI-assisted development improves engineering productivity when teams integrate it through governance, measurement, and human ownership. Without these conditions, gains can become hidden costs: code quality drift, hallucinated logic, insecure dependencies, senior-review overload, and weaker mentoring. A sustainable implementation model pairs cautious enablement with validation rules, monitoring metrics, secure dependency control, and senior-led review standards.

REFERENCES

- [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*.
<https://doi.org/10.1287/mnsc.2025.00535>
- [2] Hou, X., Zhao, Y., Liu, Y., Yang, Z., Wang, K., Li, L., Luo, X., Lo, D., Grundy, J., & Wang, H. (2024). Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology*, 33(8), 1–79. <https://doi.org/10.1145/3695988>
- [3] Ng, K. K., Fauzi, L., Leow, L., & Ng, J. (2024). Harnessing the potential of Gen-AI coding assistants in public sector software development. *arXiv*.
<https://doi.org/10.48550/ARXIV.2409.17434>
- [4] Patcas, R., & Motogna, S. (2026). An evaluation study of large language models for addressing code quality issues. *Empirical Software Engineering*, 31(5), Article 118. <https://doi.org/10.1007/s10664-026-10858-8>
- [5] Peng, S., Kalliamvakou, E., Cihon, P., & Demirer, M. (2023). The impact of AI on developer productivity: Evidence from GitHub Copilot. *arXiv*.
<https://doi.org/10.48550/ARXIV.2302.06590>
- [6] Spracklen, J., Wijewickrama, R., Sakib, A. H. M. N., Maiti, A., & Viswanath, B. (2025). We have a package for you! A comprehensive analysis of package hallucinations by code generating LLMs. In *34th USENIX Security Symposium (USENIX Security 25)* (pp. 3687–3706). USENIX Association.
<https://www.usenix.org/conference/usenixsecurity25/presentation/spracklen>
- [7] Vijayvergiya, M., Salawa, M., Budiselić, I., Zheng, D., Lamblin, P., Ivanković, M., Carin, J., Lewko, M., Andonov, J., Petrović, G., Tarlow, D., Maniatis, P., & Just, R. (2024). AI-assisted assessment of coding practices in modern code review. In *Proceedings of the 1st ACM International Conference on AI-Powered Software* (pp. 85–93). <https://doi.org/10.1145/3664646.3665664>
- [8] Weber, T., Brandmaier, M., Schmidt, A., & Mayer, S. (2024). Significant productivity gains through programming with large language models. *Proceedings of the ACM on Human-Computer Interaction*, 8(EICS), 1–29. <https://doi.org/10.1145/3661145>
- [9] Yan, S., Wang, S., Duan, Y., Hong, H., Lee, K., Kim, D., & Hong, Y. (2024). An LLM-assisted easy-to-trigger backdoor attack on code completion models: Injecting disguised vulnerabilities against strong detection. In *33rd USENIX Security Symposium (USENIX Security 24)* (pp. 1795–1812). USENIX Association.
<https://www.usenix.org/conference/usenixsecurity24/presentation/yan>
- [10] Zhang, Z., Wang, C., Wang, Y., Shi, E., Ma, Y., Zhong, W., Chen, J., Mao, M., & Zheng, Z. (2025). LLM hallucinations in practical code generation: Phenomena, mechanism, and mitigation. *Proceedings of the ACM on Software Engineering*, 2(ISSTA), 481–503. <https://doi.org/10.1145/3728894>