International Journal of
INTELLIGENT SYSTEMS AND APPLICATIONS IN ENGINEERING
ISSN:2147-6799 www.ijisae.org Original Research Paper

Impact of AI Code Assistants on Audit Tool Development

Karishma Velisetty

Abstract: AI-assisted code development tools are increasingly being integrated into audit analytics and controls testing environments, where custom scripts in Python, SQL, R, and JavaScript support high-volume compliance testing, data transformations, and regulatory reporting. Such tools accelerate audit script synthesis by generating data extraction queries, automating standard audit procedures including Benford's Law testing and aging analyses, and producing documentation scaffolding. Nevertheless, the introduction of AI-generated code into SOX and ICFR testing processes creates material risks including SQL injection vulnerabilities, hard-coded credentials, compromised code security arising from uncritical developer acceptance of AI suggestions, and escalating verification debt caused by the mismatch between developer mental models and actual AI tool behavior. Effective mitigation demands mandatory peer review, automated security scanning using tools such as Bandit, SonarQube, and CodeQL, governance of prompt construction through version-controlled template libraries, and continuous integration pipelines enforcing quality standards via platforms such as Jenkins or GitLab CI. A hybrid development model combining AI-driven acceleration with rigorous human oversight, test-driven development practices, and comprehensive multi-dimensional software quality evaluation provides the most defensible path for maintaining compliance integrity in regulated audit environments.

Keywords: *AI Code Assistants Audit Analytics Automation Software Security Vulnerabilities, SOX Compliance Validation, Verification Debt, ICFR Testing*

1. Introduction

AI-assisted code development tools are fundamentally reshaping the practice of software engineering, introducing a paradigm in which large language models trained on extensive code corpora serve as real-time collaborators within integrated development environments (IDEs). These tools offer auto-completions, context-aware logic generation, and full function suggestions that augment developer workflows across programming languages and application frameworks [1]. For example, an audit developer working in Visual Studio Code or PyCharm can receive instant suggestions for a complete Python function that connects to an SAP HANA database, extracts a full population of journal entries for a given fiscal period, and formats the output for downstream analytical testing—all generated from a brief natural language comment or a partial function signature. Despite widespread adoption enthusiasm, however, the actual productivity impact of these tools on experienced professionals working within realistic project environments remains a subject of active empirical investigation. Recent controlled research studying the effect of AI coding assistants on experienced open-source developers working on real-world tasks within established repositories found that productivity outcomes were nuanced and did not uniformly confirm optimistic narratives, suggesting that the relationship between AI assistance and developer output is more complex than early adopter discourse implies [2].

The broader developer ecosystem is nonetheless undergoing structural transformation driven by AI integration. The ways in which developer communities share and review code, individual coding workflows, team collaboration patterns, and project management paradigms are all being reshaped by the emergence of AI-driven development environments [3]. From initial design and prototyping through testing and deployment, AI tools are establishing new expectations around development speed, code standardization, and collaborative human-AI systems across the full software development lifecycle. Consider, for instance, an internal audit team that previously maintained a shared repository of manually written SQL scripts for extracting accounts payable populations from Oracle ERP. With AI code assistants integrated into their IDE, the same team can now generate equivalent extraction scripts for different ERP platforms—such as SAP S/4HANA or Microsoft Dynamics 365—by providing contextual prompts describing required data fields, join logic, and filtering criteria. This ecosystem-level shift has direct implications for specialized domains like audit analytics, where development teams rely on custom scripts in Python, SQL, R, or JavaScript to perform high-volume testing of control populations,

Independent Researcher, USA

execute data transformations, and power compliance dashboards [4].

Within the audit environment, stakes are particularly high for control effectiveness, evidence reliability, and regulatory compliance. In audit analytics, scripts and code are integral to the testing of SOX and Internal Control over Financial Reporting (ICFR) controls, meaning that degradation of code quality, introduction of silent defects, or erosion of maintainability directly affects an organization's ability to achieve and demonstrate regulatory compliance. Research has investigated the impact of AI-generated code on software quality and developer productivity, finding that while productivity gains can arise from increased initial output and reductions in time spent on ordinary tasks, the longer-term impact on software quality, structural soundness, and maintainability varies substantially based on the degree to which AI-generated code is monitored and validated by a human reviewer [5]. A practical illustration of this risk emerges when an AI assistant generates a Python script for a three-way match between purchase orders, goods receipts, and invoices: the generated code may correctly perform matching logic on standard transactions while failing to account for tolerance thresholds specific to client policy, partial deliveries, or multi-currency conversions—edge cases that are critical for SOX control testing but unlikely to be well-represented in AI training data. This underscores a central tension for audit teams: the efficiency gains offered by AI coding tools are real and substantial, but reduced validation effort may introduce defects into scripts that underpin compliance evidence.

This article examines the impact of AI code assistants on the development of audit tools, including their application in rapidly developing audit analytics scripts, the risks and limitations they introduce, and best practices for validating AI-assisted code in SOX and ICFR testing environments. The remainder of this paper is structured as follows: Section 2 discusses the acceleration of audit analytics script development through AI assistance. Section 3 examines the risks of errors, bias, and security vulnerabilities in AI-generated code. Section 4 presents best practices for validating AI-assisted code in compliance-sensitive environments. Section 5 integrates these findings into a discussion of hybrid development models and quality frameworks. Section 6 concludes with recommendations for audit teams.

Aspect	Key Insight
IDE Integration	VS Code/PyCharm generate full functions from comments
ERP Cross-Platform	Prompts produce scripts for Oracle, SAP S/4HANA, Dynamics 365
Productivity Nuance	Gains are context-dependent, not uniformly positive
Domain Edge Cases	Three-way match misses tolerances, partial deliveries, FX logic
Quality-Speed Tension	Long-term quality varies with human oversight level
Compliance Sensitivity	SOX/ICFR scripts need validation beyond functional correctness

Table 1. Context and challenges of AI code assistants in audit environments [1–3]

2. Acceleration Of Audit Analytics Script Development

By accommodating repetitive and pattern-driven programming work that constitutes a significant portion of audit analytics development, code assistants can reduce the time and cognitive load required to write audit tools and analytics scripts. Researchers have examined productivity effects using evidence gathered from controlled studies and applied it to realistic professional settings. In one longitudinal mixed-methods case study exploring developer productivity with and without an AI coding assistant over a period of weeks in an industrial software development organization, productivity effects were context-dependent: developers experienced substantial benefits in certain categories of work, but insignificant and decaying effects in others, particularly when architectural reasoning or substantial domain knowledge was required [6]. This finding is directly relevant to audit analytics teams, where the mixture of routine scripting tasks and complex compliance logic means that AI-assisted acceleration will be unevenly distributed across the development workflow.

For example, an audit developer tasked with writing a SQL query to extract the complete population of revenue transactions from an Oracle database for a specific reporting period can receive a fully functional query from an AI assistant within seconds, including appropriate JOIN clauses, WHERE filters, and GROUP BY aggregations. However, when the same developer needs to implement a complex revenue recognition rule accounting for multi-element arrangements under ASC 606, the AI assistant's output is far more likely to require substantial manual correction and domain-

specific refinement. This asymmetry in AI utility—high for syntactically regular tasks, limited for semantically complex ones—is consistent with broader findings on AI capabilities in professional software development settings [7].

The degree to which software engineering teams adopt AI tools for code review is influenced not only by the performance of the generative AI tools themselves but also by organizational factors including team culture, management support, pre-existing code review processes, and the specific task at hand [8]. Teams with established quality assurance processes and collaborative review cultures tend to integrate AI assistants more effectively, extracting productivity gains while maintaining code quality standards. For audit analytics, this suggests that teams already operating within structured development frameworks—such as those using Git-based version control with pull request workflows, automated linting tools like Pylint or ESLint, and peer review checklists aligned with PCAOB standards—are best positioned to benefit from AI coding tools, as the tools amplify existing discipline rather than compensating for its absence [9].

Comparisons between individual AI coding assistants across task types and quality dimensions further illuminate trade-offs relevant to audit development. In a simulated comparison of assistant performance in web application development, multiple AI assistants were faster at generating initial code and functional prototypes, but differed with regard to code completeness, organization, adherence to best practices, and the degree of human adjustment required after initial code generation [10]. This underscores how purposeful selection and configuration of AI coding tools matters for audit teams, and the need to weigh potential acceleration benefits against the effort of verifying, cleaning, and integrating AI outputs into production-quality audit analytics pipelines.

Illustrative examples help clarify these dynamics. The first-digit distribution test (Benford's Law) and the routine for detecting duplicated payments are standard audit analytics tasks where AI assistance yields the greatest impact. For Benford's Law testing, an AI assistant can produce a Python script using the pandas and matplotlib libraries to extract leading digits, calculate observed frequencies, compare them with expected Benford distributions, and generate a visualization—requiring only minor adjustments. However, AI-generated code for duplicate payment detection may rely on basic exact matching and omit fuzzy matching logic—as provided by the fuzzywuzzy or recordlinkage Python packages—for identifying near-duplicates such as vendors differing only by minor spelling variations or invoice formatting differences, which are critical for effective audit testing [11].

Aspect	Key Insight
Routine Tasks	SQL extractions with JOINs/WHEREs can be generated in seconds
Complex Logic Limit	ASC 606 rules need heavy manual correction
Organizational Fit	Git workflows, Pylint/ESLint, PCAOB checklists amplify gains
Benford's Law	pandas/matplotlib scripts need only minor adjustments
Duplicate Detection	Misses fuzzy matching via fuzzywuzzy/recordlinkage
Downstream Rework	Speed gains offset by refinement and integration effort

Table 2. Task-specific gains and constraints in AI-assisted audit analytics [4–6]

3. Risks Of Errors And Bias In Ai-Generated Code

3.1. Structural and Security Risks

AI-generated code is not immune to quality and security issues, and rigorous empirical evidence confirms that the risks are more prevalent than many practitioners assume. A systematic investigation into the security properties of code produced by an AI code generation model evaluated behavior across 89 different scenarios designed to elicit completions relevant to high-risk cybersecurity weaknesses drawn from MITRE's Top 25 Common Weakness Enumeration (CWE) list [12]. The study produced 1,689 programs across these scenarios and found that approximately 40% were vulnerable, with weaknesses spanning categories including out-of-bounds writes, SQL injection, OS command injection, path traversal, use-after-free errors, NULL pointer dereference, hard-coded credentials, and insufficiently protected credentials. The security quality of generated code varied substantially depending on the programming language, the nature of the prompt, and the presence of secure or insecure patterns in the surrounding context, with certain categories—such as path traversal—producing vulnerable top-scoring suggestions across all tested scenarios.

These findings are particularly concerning for audit tool development, where scripts routinely interact with sensitive financial databases, process personally identifiable information, and produce outputs that serve as direct audit evidence in regulatory filings. For example, an AI assistant asked to generate a Python

function querying an audit database for employee expense records based on a user-provided department name may produce code that constructs the SQL query through string concatenation rather than using parameterized queries—a classic SQL injection vulnerability. If deployed in an audit analytics dashboard built with frameworks such as Flask or Django, this vulnerability could allow unauthorized access to underlying financial data, compromising both the security of the audit evidence and the integrity of the control testing process [13].

3.2. Quality, Maintainability, and Bias Risks

Beyond isolated points of vulnerability, systemic concerns arise from the confidence developers place in AI-generated code. In a controlled user study exploring how developers use AI assistants for security-related programming tasks in Python, JavaScript, and C, AI-assisted users wrote less secure code than those without AI access while simultaneously holding stronger confidence in the security of their AI-generated output [14]. This confidence-accuracy gap is particularly dangerous in audit contexts: developers who trust AI-generated scripts may deploy insufficiently validated code as control logic or compliance evidence without adequate scrutiny.

Experiments in prompt engineering further demonstrated that programmers produced more secure code when they carefully crafted their prompts, wrote helper functions, or tuned model parameters at the outset of a session, confirming that blind acceptance of AI output constitutes a significant security risk. A concrete example of bias replication involves database credentials: when prompted to generate a Python snippet connecting to a PostgreSQL database to extract trial balance data, an AI assistant may replicate patterns observed in training data by hard-coding the database username and password directly into the code. Secure practice instead calls for environment variables, a secrets manager such as HashiCorp Vault or AWS Secrets Manager, or configuration files excluded from version control via .gitignore—patterns that AI assistants may not default to unless explicitly specified in the prompt [15].

3.3. Verification Debt

The gap between code acceptance and code validation is further compounded by misalignment between developers' mental models and the actual behavior of AI-driven code completion tools. An elicitation study conducted through co-design workshops with developers found that participants expressed highly diverse and often conflicting preferences regarding when AI suggestions should appear, how they should be displayed, and the level of granularity they should provide, confirming that no single interaction model fits all users or tasks [16]. This misalignment leads developers to accept AI-generated suggestions under conditions where they lack sufficient understanding of what the tool has produced or why, deepening verification debt over time.

In audit environments where code serves as control logic for ICFR and SOX compliance testing, unexamined acceptance of AI suggestions can lead to incorrect analytics results, missed control deficiencies, and materially elevated non-compliance risk. For instance, an audit developer may receive an inline AI suggestion for a date-filtering function that appears correct at a glance—filtering transactions to the fiscal year ending December 31—but actually uses a less-than operator instead of a less-than-or-equal-to operator, silently excluding all transactions on the final day of the reporting period. This class of off-by-one error, trivial in appearance but material in consequence, exemplifies the verification debt that accumulates when AI suggestions are accepted without line-by-line scrutiny, and could directly affect the completeness of the population tested in a SOX controls engagement.

Aspect	Key Insight
Scenarios Evaluated	89 scenarios from MITRE's Top 25 CWE list
Programs Generated	1,689 programs across all scenarios
Vulnerability Rate	Approximately 40% of generated programs
Vulnerability Types	SQL injection, path traversal, hard-coded credentials
SQL Injection Example	String concatenation in Flask/Django vs. parameterized queries
Credential Bias	Hard-coded passwords instead of env variables/secrets managers
Mental Model Gap	Diverse preferences on timing/display deepen verification debt

Table 3. Security risks and verification debt in AI-generated audit code [7–9]

4. Best Practices For Validating Ai-Assisted Code In Sox And Icfr Testing

4.1. Security Validation and Static Analysis

Reducing the risks of AI-generated code in the audit setting is a multi-layered undertaking involving code

review, security validation, governance, and ongoing monitoring. Any validation framework should begin with the recognition that AI-generated code inherits the security posture of the patterns represented in its training data, and that web applications and data-facing scripts—central to audit analytics—constitute particularly high-risk surfaces. Research on the use of large language models for secure web application development has confirmed that while models can accelerate the development of functional code, they do not inherently enforce secure coding practices or follow best security standards unless explicitly directed through well-crafted prompts and supplementary validation layers [17].

For audit teams developing scripts that interact with financial databases, process sensitive control populations, or generate compliance evidence, relying on AI-generated code without layering static analysis, dynamic testing, and manual security review creates unacceptable exposure. In practice, this validation layer can be implemented using tools such as Bandit for Python security analysis, SonarQube for multi-language code quality and vulnerability scanning, and CodeQL for semantic code analysis—each configurable to detect the vulnerability categories most relevant to audit scripts, including SQL injection in database queries, insecure deserialization in data import routines, and hard-coded credentials in connection strings [18]. Mandatory peer review should treat every AI-generated contribution with the same rigor applied to human-written code, with reviewers specifically examining logic correctness, input validation practices, query parameterization, and adherence to the security requirements of the specific audit engagement.

4.2. Reproducibility and Audit Trail Management

A concern beyond security is whether the quality and productivity results of AI-assisted code generation can be replicated across different contexts and environments. Researchers have found that many published results on LLM-assisted software engineering suffer from reproducibility challenges arising from sensitivity to prompt wording, model version, temperature settings, and evaluation approach, generating doubts about the consistency and generalizability of reported outcomes [19]. This reproducibility problem has direct consequences for audit work when code is used in SOX and ICFR testing. A script may work correctly in the development environment but produce different results in production, where a different ERP schema, dataset profile, or model version may be present.

For example, an AI-generated Python script using pandas to calculate an aging analysis of accounts receivable may produce correct results during development testing against a sample dataset but fail when deployed against production data containing NULL values in date fields, non-standard date formats from international subsidiaries, or extremely large datasets that exceed memory thresholds. Version-controlled documentation of AI interactions—including prompts, model identifiers, parameter settings, and rationale for accepting or modifying outputs—should be maintained using tools such as Git with structured commit messages, dedicated prompt registry repositories, or documentation platforms integrated into the team's CI/CD pipeline to strengthen the audit trail and ensure traceability [20].

4.3. Prompt Governance and CI/CD Integration

The design and construction of prompts is central to AI-assisted program development yet is frequently overlooked as a governance concern. Research on the programmatic nature of prompting has demonstrated that prompt design and composition are the primary determinants of the quality, correctness, and reliability of AI outputs, and that effective prompting requires the same level of expertise and rigor as conventional software engineering practice [21]. Audit analytics teams should manage prompt writing with the same discipline applied to production code. This includes developing a library of prompt templates for commonly encountered audit scenarios, versioning these templates, reviewing them for compliance with regulatory requirements and organizational coding standards, and updating them as regulatory guidance evolves. A representative prompt template might specify: "Extract the complete population of [transaction type] from [ERP system] for fiscal period [date range] using parameterized queries, returning results as a pandas DataFrame filtered by [specific criteria], with no hard-coded credentials."

The final verification layer consists of CI/CD pipelines—implemented in environments such as Jenkins, GitLab CI, or GitHub Actions—that enforce quality, style, and security checks on each commit and detect regressions or logic drift before changes reach production audit environments [22]. These pipelines can be configured to automatically execute Bandit or SonarQube scans, run unit test suites using pytest, and enforce code coverage thresholds, creating a systematic quality gate that applies equally to AI-generated and human-written code.

Aspect	Key Insight
Security Tools	Bandit (Python), SonarQube (multi-language), CodeQL (semantic)
LLM Security Gap	No inherent secure coding without guided prompts + validation
Reproducibility Risk	Dev-passing scripts fail on production NULLs, dates, memory
Audit Trail	Git commits, prompt registries, CI/CD documentation
Prompt Discipline	Prompt construction requires same rigor as code development
Template Example	Standardized prompts: transaction type, ERP, date range, format
CI/CD Enforcement	Jenkins/GitLab CI/GitHub Actions enforce checks per commit

Table 4. Validation framework for AI-assisted code in SOX/ICFR testing [10–12]

5. Discussion And Framework Integration

5.1. Probabilistic Nature of AI Output and the Hybrid Development Model

Auditing functions increasingly develop custom analytics and controls testing scripts, making the reliability, correctness, and security of these scripts critical for ICFR and SOX compliance. AI code assistants can amplify productivity and support standardization of common analytical patterns, but they should not replace the technical judgment, domain expertise, or structured verification processes that underpin audit assurance. The fundamental challenge in integrating AI-generated code into compliance-sensitive workflows lies in the nature of the underlying models. Research evaluating large language models trained on code has demonstrated that while these models exhibit impressive capabilities in generating functionally plausible completions across multiple programming languages, their outputs are fundamentally probabilistic rather than deterministic—they optimize for statistical likelihood of correctness based on training patterns rather than for guaranteed adherence to functional specifications, security requirements, or domain-specific compliance logic [23].

For audit teams, this characteristic is consequential. Scripts generated by AI assistants may appear syntactically correct and functionally reasonable while silently deviating from the precise business rules, threshold calculations, or regulatory requirements governing the specific control being tested. As an example, an AI assistant asked to write a script testing the operating effectiveness of an automated control that flags journal entries above a materiality threshold might correctly implement the comparison logic but use a greater-than operator instead of a greater-than-or-equal-to operator, or apply the threshold at the line-item level instead of the entry level—a minor syntactic deviation that produces systematically incorrect test results. Figure 1 illustrates the hybrid development model recommended for audit teams, in which AI performs first-generation code generation and scaffolding while human specialists conduct extensive review, testing, and architectural integration to ensure both development speed and compliance integrity [24].

5.2. Multi-Dimensional Quality Evaluation for Audit Evidence

The evaluation of AI-assisted audit code should be conducted within the context of established software quality models. Research on software quality models shows that software quality is not determined solely by functional correctness but encompasses multiple dimensions including reliability, maintainability, efficiency, portability, and usability, and that no single metric or test can capture all quality characteristics affecting a software product's fitness for purpose [25]. This multi-dimensional quality perspective is especially relevant when audit analytics scripts serve as evidence in SOX or ICFR testing.

Audit teams can operationalize multi-dimensional quality assessment by adopting a structured code review checklist evaluating AI-generated scripts across the following dimensions: (1) Functional correctness—does the script produce accurate results against a known test dataset representative of the full population? (2) Maintainability—can another auditor understand and modify the script without the original developer's guidance? (3) Robustness—does the script handle edge cases such as NULL values, negative amounts, foreign currency transactions, and fiscal year boundary conditions? (4) Security—does the script use parameterized queries, avoid hard-coded credentials, and properly sanitize any user inputs? Even if expected results can be replicated on standard transactions, a script that fails in edge cases or cannot be independently reviewed by multiple auditors should not be considered satisfactory audit evidence. These dimensions align with ISO/IEC 25010, which provides a structured vocabulary for evaluating software product quality across functional and non-functional characteristics [26].

5.3. Engineering Competencies and Cultural Alignment

Achieving quality and conformance standards in AI-assisted audit development requires that team members possess and continually cultivate fundamental software engineering skills needed to critically review AI-generated outputs. Research examining the body of skills required for effective software engineering practice has emphasized that development competency extends well beyond the ability to write syntactically correct code, encompassing requirements analysis, systematic testing, quality assurance, and evaluating whether a given implementation genuinely satisfies its intended specification [27].

Audit development teams adopting AI coding assistants should invest in developer education covering not only the capabilities and limitations of AI tools but also core engineering disciplines such as unit testing with pytest or unittest, integration testing against representative datasets, and regression testing to ensure script modifications do not introduce unintended changes to previously validated outputs [28]. Establishing the right cultural norms is equally critical: teams should adopt a professional posture in which AI recommendations are treated as drafts requiring scrutiny rather than finished implementations ready for deployment [29]. Test-driven development (TDD) approaches—in which test cases are written prior to production code—may prove especially effective in AI-assisted audit settings, as they provide objective acceptance criteria against which AI-generated code can be evaluated immediately, ensuring that compliance scripts carry the same evidentiary weight required by audit standards [30].

Aspect	Key Insight
Probabilistic Risk	AI optimizes likelihood, not specification adherence
Quality Checklist	Correctness, maintainability, robustness (NULLs, FX), security
Evidentiary Standard	Independent reviewability by uninvolved auditors required
Testing Frameworks	pytest/unittest for unit, integration, and regression testing
TDD Strategy	Pre-written tests set objective acceptance criteria for AI code
Cultural Shift	AI output treated as drafts needing professional scrutiny

Table 5. Hybrid framework and competency model for compliant audit development [13–15]

6. Conclusion

AI coding assistants deliver tangible benefits in accelerating audit tool development by reducing repetitive scripting effort, generating functional code for tasks such as population extraction and distribution testing, and supporting standardization across audit engagements. However, these efficiency gains carry significant risks within regulated compliance environments. Cross-category security vulnerabilities including SQL injection, path traversal, and credential exposure continue to persist in AI-generated output at rates that should concern any compliance-sensitive organization. Factors including uncritical developer trust in AI recommendations, lack of reproducibility across model versions and prompt configurations, and ongoing mental model misalignment have created conditions where unverified code can compromise the integrity of control evidence and undermine the defensibility of audit conclusions.

Addressing these risks requires a structured, multi-layered response: mandatory peer review; integration of tools such as Bandit, SonarQube, and CodeQL into CI/CD pipelines for automated validation; formalized prompt libraries with version governance; comprehensive audit trail documentation; and continuous investment in test-driven development practices and software engineering competencies. The hybrid framework proposed in this paper—embedding AI-assisted development within a structure of human judgment, professional scrutiny over AI suggestions, and multi-dimensional quality assessment—enables audit teams to capture the productivity improvements offered by AI tools without sacrificing the assurance integrity that SOX and ICFR testing demand. Future research should examine the long-term reliability of AI-generated audit scripts across diverse ERP environments and investigate how prompt governance frameworks can be standardized at the industry level to support consistency in AI-assisted compliance testing.

Acknowledgments

Not applicable.

Funding Information

Authors state no funding involved.

Author Contributions Statement

This journal uses the Contributor Roles Taxonomy (CRediT). The contributions of the author are indicated below.

Name of Author	C	M	So	Va	Fo	I	R	D	O	E	Vi	Su	P	Fu
Karishma Velisetty	✓				✓	✓			✓	✓			✓	

C: Conceptualization | M: Methodology | So: Software | Va: Validation | Fo: Formal Analysis | I: Investigation | R: Resources | D: Data Curation | O: Writing–Original Draft | E: Writing–Review & Editing | Vi: Visualization | Su: Supervision | P: Project Administration | Fu: Funding Acquisition

Conflict Of Interest Statement

Authors state no conflict of interest.

Data Availability

The authors confirm that the data supporting the findings of this study are available within the article. As this is a conceptual and review-based paper, no primary datasets were generated.

References

[1] J. Becker et al., "Measuring the Impact of Early-2025 AI on Experienced Open-Source Developer Productivity," arXiv, Jul. 2025. [Online]. Available: https://arxiv.org/pdf/2507.09089

[2] S. Srikanth et al., "AI-Driven Developer Ecosystem," IJRSI, Jul. 2025. [Online]. Available: https://rsisinternational.org/journals/ijrsi/articles/ai-driven-developer-ecosystem/

[3] A. Arivoli, "The Impact of AI-Generated Code on Software Quality and Developer Productivity," IOSR-JCE, 2025. [Online]. Available: https://www.iosrjournals.org/iosr-jce/papers/Vol27-issue1/Ser-1/L2701017682.pdf

[4] V. Stray et al., "Developer Productivity With and Without GitHub Copilot: A Longitudinal Mixed-Methods Case," arXiv, Jan. 2026. [Online]. Available: https://arxiv.org/pdf/2509.20353

[5] G. Giray et al., "An Empirical Study of Generative AI Adoption in Software Engineering," arXiv, Dec. 2025. [Online]. Available: https://arxiv.org/pdf/2512.23327

[6] L. N. Hyseni and A. Deraku, "Comparative Analysis of GitHub Copilot and ChatGPT in Web Application Development: An Experimental Study," IJCESEN, Apr. 2025. [Online]. Available: https://www.ijcesen.com/index.php/ijcesen/article/view/1846/791

[7] H. Pearce et al., "Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions," IEEE S&P, 2022. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9833571

[8] N. Perry et al., "Do Users Write More Insecure Code with AI Assistants?" arXiv, 2023. [Online]. Available: https://arxiv.org/html/2211.03622v3

[9] G. Desolda et al., "Understanding User Mental Models in AI-Driven Code Completion Tools: Insights from an Elicitation Study," ScienceDirect, Nov. 2025. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1071581925002058

[10] K. Kiashemshaki et al., "Secure Coding for Web Applications: Frameworks, Challenges, and the Role of LLMs," arXiv, Aug. 2025. [Online]. Available: https://arxiv.org/html/2507.22223v2

[11] M. L. Siddiq et al., "Large Language Models for Software Engineering: A Reproducibility Crisis," arXiv, Nov. 2025. [Online]. Available: https://arxiv.org/pdf/2512.00651

[12] L. Beurer-Kellner et al., "Prompting Is Programming: A Query Language for Large Language Models," arXiv, 2023. [Online]. Available: https://arxiv.org/pdf/2212.06094

[13] M. Chen et al., "Evaluating Large Language Models Trained on Code," arXiv, 2021. [Online]. Available: https://arxiv.org/pdf/2107.03374

[14] H. Chauhan et al., "A Review of Software Quality Models for the Evaluation of Software Products," IJCSE, 2016. [Online]. Available: https://ijcseonline.org/index.php/j/article/view/1125/1118

[15] Y. Sedelmaier and D. Landes, "SWEBOS—The Software Engineering Body of Skills," I-JEP. [Online]. Available: https://online-journals.org/index.php/i-jep/article/view/4047/3386

[16] GitHub, "GitHub Copilot: Your AI Pair Programmer," GitHub, Inc., San Francisco, CA, USA, 2023. [Online]. Available: https://github.com/features/copilot

[17] E. Özgül et al., "A Systematic Review of Studies on the Use of Generative Artificial Intelligence Tools in Programming Education," IEEE Access, vol. 12, pp. 58701–58725, 2024. https://doi.org/10.1109/ACCESS.2024.3390829

[18] PCAOB, "Auditing Standard No. 5: An Audit of Internal Control Over Financial Reporting That Is Integrated with An Audit of Financial Statements," PCAOB, Washington, DC, USA, 2007. [Online].

Available: https://pcaobus.org/Standards/Auditing/Pages/Auditing_Standard_5.aspx

[19] S. Cantor, "A Guide to Data Analytics in Internal Audit," Internal Auditor, Feb. 5, 2026. [Online]. Available: https://www.becker.com/blog/cia/a-guide-to-data-analytics-in-internal-audit

[20] OWASP, "OWASP Top Ten Web Application Security Risks," Open Web Application Security Project, 2021. [Online]. Available: https://owasp.org/www-project-top-ten/

[21] HashiCorp, "Vault: Secrets Management and Data Protection," HashiCorp, Inc., San Francisco, CA, USA, 2023. [Online]. Available: https://www.vaultproject.io/

[22] SonarSource, "SonarQube: Continuous Code Quality and Security," SonarSource SA, Geneva, Switzerland, 2023. [Online]. Available: https://www.sonarqube.org/

[23] T. Winters, T. Manshreck, and H. Wright, Software Engineering at Google: Lessons Learned from Programming Over Time. Sebastopol, CA, USA: O'Reilly Media, 2020.

[24] J. Humble and D. Farley, Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. Upper Saddle River, NJ, USA: Addison-Wesley, 2010.

[25] T. Zhang et al., "A Survey of Controllable Text Generation Using Transformer-Based Pre-Trained Language Models," ACM Computing Surveys, vol. 56, no. 4, pp. 1–38, 2024. https://doi.org/10.1145/3617680

[26] ISO/IEC, "ISO/IEC 25010:2011 Systems and Software Engineering—Systems and Software Quality Requirements and Evaluation (SQuaRE)—System and Software Quality Models," ISO, Geneva, Switzerland, 2011. https://www.iso.org/standard/35733.html

[27] K. Beck, Test Driven Development: By Example. Boston, MA, USA: Addison-Wesley, 2002.

[28] D. Poccia, "AI-Assisted Software Development Lifecycle," Sep. 25, 2024. [Online]. Available: https://dev.to/aws/ai-assisted-software-development-lifecycle-289k

[29] AICPA, "Statement on Auditing Standards No. 145: Understanding the Entity and Its Environment and Assessing the Risks of Material Misstatement," AICPA, New York, NY, USA, 2021. [Online]. Available: https://us.aicpa.org/research/standards/auditattest/sas

[30] PCAOB, "AS 2301: The Auditor's Responses to the Risks of Material Misstatement," PCAOB, Washington, DC, USA, 2010. [Online]. Available: https://pcaobus.org/Standards/Auditing/Pages/AS2301.aspx