Digital Transformation and Technology Dynamics

Journal Homepage: www.techdynamicsjournal.com | ARTICLE NO. DTTD-2022003 | VOL. 2 ISSUE 2

Evaluating the Quality, Maintainability, and Security of AI-Generated Source Code in Enterprise Systems

Rashadul Islam Samrat[1]*, Md Rasel Ul Alam[2], Kanita Haider[3],

[1] *Department of Marketing, University of Barishal, Barishal, Bangladesh*

[2] *Department of Computer and Information Sciences, University of the Cumberlands, Kentucky, USA.*

[3] *Department of Computer Science and Engineering, International Islamic University Chittagong, Chittagong, Bangladesh*

**Corresponding author: samrat.meazi1@gmail.com*

Received 19 August 2022; Revised 12 September 2022; Accepted 30 October 2022; Published: 29 November 2022

KEYWORDS

AI-generated code;

Software security;

Code maintainability; Enterprise software engineering;

Static analysis;

Technical debt;

Large language models; Software quality assurance

Abstract

Large language model (LLM)–based code-generation tools such as GitHub Copilot, ChatGPT, and Amazon CodeWhisperer are increasingly embedded in enterprise software development lifecycles, with studies reporting notable gains in developer task-completion speed. Despite rapid adoption, systematic evaluation of the code these tools produce has lagged. Existing empirical work often isolates functional correctness, security, or maintainability, relying on inconsistent corpora, static-analysis toolchains, and reporting conventions, which complicates enterprise leaders' ability to assess aggregate quality risks. This paper addresses that gap by synthesizing the empirical literature on AI-generated code quality, maintainability, and security, and by proposing an integrated evaluation framework, the Enterprise Code Quality Index (ECQI). A thematic review of peer-reviewed and archival studies published between 2021 and 2026 was conducted, encompassing correctness benchmarks, static-analysis security studies, controlled maintainability experiments, and productivity trials. Building on this evidence, a topic-specific research design is outlined, combining repository-grounded prompt sampling, multi-tool code generation, static security analysis with CWE mapping, maintainability metrics, functional testing, and structured expert review. The reviewed evidence highlights a consistent rate of security weaknesses in AI-generated code, ranging from 12 to 40 percent across independent studies, mixed but cautious signals on maintainability and technical debt, and robust productivity gains when human oversight is retained. Because this study did not execute new empirical experiments, illustrative results are presented separately from literature-derived findings. The contribution lies in offering a thematic synthesis, a reproducible evaluation methodology, a composite quality index for governance, and actionable recommendations for secure, maintainable enterprise adoption of AI coding assistants.

1. Introduction

Since the release of Codex, the model family underlying GitHub Copilot, and its evaluation on the HumanEval functional-correctness benchmark (Chen et al., 2021), large language model (LLM)-based code generation has moved from a research curiosity into everyday enterprise software-development practice. Controlled experiments have reported substantial productivity effects; for example, Peng et al. (2023) found that professional developers using GitHub Copilot completed a standardized HTTP-server task 55.8% faster than a control group, while subsequent field and laboratory studies have reported comparable, although more heterogeneous, gains across coding, writing, and customer-support tasks. These findings have contributed to the rapid adoption of AI coding assistants across both regulated and unregulated industries. However, the artifacts produced by these systems are not simply faster versions of human-written code. They result from statistical pattern completion over large and heterogeneous training corpora and may therefore reproduce correctness gaps, insecure programming idioms, duplicated structures, and stylistic inconsistencies present in their training data. Independent empirical studies applying static analysis and Common Weakness Enumeration (CWE) mapping to AI-generated code have identified security weaknesses in a substantial minority of examined outputs (Fu et al., 2025; Schreiber & Tippe, 2025), while other studies have raised concerns regarding technical debt, code duplication, and maintainability at scale (Harding & Kloster, 2024).

The increasing integration of AI-generated code into enterprise software-development lifecycles therefore creates a

fundamental evaluation problem. Organizations require reliable evidence concerning whether AI-generated source code is suitable for production use across three operationally important dimensions: functional correctness, security, and maintainability. Functional correctness determines whether generated code performs its intended task under specified conditions; security concerns whether the code introduces exploitable vulnerabilities or insecure implementation patterns; and maintainability concerns whether human developers can understand, modify, test, and evolve the resulting artifacts over time. Existing research and engineering tools frequently examine these dimensions independently, relying on different datasets, programming languages, static-analysis engines, testing procedures, and reporting conventions. This fragmentation makes direct comparison difficult and limits the ability of enterprise engineering leaders to establish consistent approval criteria, review gates, and residual-risk monitoring procedures for AI-assisted software development.

The literature also reveals several methodological gaps that prevent robust cross-study interpretation. First, research on the security of AI-generated code remains heavily concentrated on individual AI coding tools and programming languages, particularly GitHub Copilot and languages such as Python and JavaScript. Reported CWE-prevalence estimates can vary substantially according to the unit of analysis, sampling strategy, generation protocol, vulnerability definition, and static-analysis tool-chain. For example, Schreiber and Tippe (2025) reported a file-level prevalence of 12.1%, whereas Fu et al. (2025) reported substantially higher snippet-level prevalence estimates across different experimental conditions. Such differences do not necessarily indicate contradictory empirical realities; rather, they demonstrate the importance of distinguishing files from snippets, vulnerability detections from confirmed vulnerabilities, and tool-specific findings from generalizable prevalence estimates. Second, the maintainability literature remains methodologically divided between studies relying on static code-smell and quality indicators and controlled human experiments examining downstream code evolution. Studies such as Ghosh Paul et al. (2025) emphasize measurable software-quality characteristics, whereas Borg et al. (2025) examine how AI assistance may affect subsequent development and modification activities.

These approaches can produce different conclusions because maintainability is simultaneously a property of the code artifact and a property of the human–AI development process. Third, despite the operational importance of human code review, relatively little work provides an integrated framework that simultaneously evaluates correctness, security, and maintainability while explicitly incorporating review intensity or human oversight as a moderating factor.

Addressing these gaps is important for both research and practice. From an academic perspective, an integrated approach can clarify which dimensions of AI-generated code quality are consistently supported by empirical evidence and which remain context-dependent or contested. From an enterprise perspective, the issue is increasingly consequential because AI-generated software may be incorporated into systems involving sensitive information, financial transactions, critical infrastructure, privacy-sensitive operations, or other high-consequence applications. An undetected injection vulnerability can create immediate security exposure, while poorly structured or duplicated AI-generated code may generate longer-term maintenance costs and technical debt. Consequently, enterprise adoption requires more than productivity measurements; it requires a reproducible mechanism for determining whether productivity gains are accompanied by acceptable levels of correctness, security, and maintainability. A synthesis-grounded evaluation framework can provide engineering leadership, security teams, software-quality practitioners, and researchers with a common methodological vocabulary for making such assessments.

Accordingly, this study has five principal objectives. First, it synthesizes peer-reviewed and archival empirical evidence concerning the functional correctness, security, and maintainability of AI-generated source code published during 2021–2026. Second, it identifies methodological sources of divergence among reported security-prevalence estimates and develops a normalization-aware approach for interpreting results across studies with different units of analysis, datasets, programming languages, generation procedures, and static-analysis configurations. Third, it develops a topic-specific and reproducible evaluation methodology for enterprise contexts that combines static analysis, functional testing, and structured human review. Fourth, it proposes an integrated Enterprise Code Quality Index (ECQI) that combines the three principal quality dimensions into a common scoring framework and demonstrates its scoring logic through a clearly labeled illustrative example rather than presenting the demonstration as empirical validation. Fifth, it derives governance-relevant recommendations for the controlled and maintainable adoption of AI coding assistants within enterprise software-development lifecycles.

These objectives lead to four research questions. RQ1 asks what the existing empirical literature reports regarding the functional correctness of AI-generated code relative to human-written baselines. RQ2 examines the prevalence and characteristics of security weaknesses in AI-generated code and investigates how differences in research scope, sampling, vulnerability definitions, programming languages, and analysis tool-chains contribute to divergent prevalence estimates. RQ3 investigates whether AI-assisted development measurably affects code

maintainability and technical-debt accumulation and under what conditions, including human review and habitual AI use, effects appear positive, negative, or null. RQ4 asks how functional correctness, security, and maintainability can be integrated into a single enterprise-usable evaluation framework that explicitly reflects the moderating role of human oversight.

The study makes several contributions to the emerging literature on AI-assisted software engineering. First, it provides a thematically organized and citation-grounded synthesis of empirical research covering functional correctness, security, and maintainability rather than treating these dimensions as independent quality problems. Second, it provides a comparative methodological analysis of security-prevalence studies, emphasizing why numerical estimates should not be compared without accounting for differences in sampling units, datasets, vulnerability definitions, programming languages, and analysis pipelines. Third, it establishes a reproducible, topic-specific research design suitable for execution by enterprise research teams, combining automated software-quality assessment with functional testing and structured human review. Fourth, it proposes the Enterprise Code Quality Index (ECQI) together with a mathematical formulation for multidimensional benchmarking of AI-assisted code quality under governance constraints. Fifth, it translates the evidence into actionable recommendations concerning AI coding-assistant deployment, human review, security controls, maintainability monitoring, and enterprise software-development governance.

The remainder of the paper is organized to develop and operationalize these contributions. Section 2 presents the thematic literature review and synthesizes evidence across functional correctness, security, and maintainability. Section 3 develops the conceptual and theoretical framework underlying the proposed evaluation approach. Section 4 specifies the research methodology, including study design, evaluation procedures, analysis protocols, and human-review procedures. Section 5 formalizes the mathematical definitions and evaluation metrics, including the proposed Enterprise Code Quality Index. Section 6 specifies the proposed datasets and data sources required for empirical implementation. Section 7 presents literature-derived findings together with a clearly identified illustrative demonstration of the proposed framework. Section 8 discusses the findings and their implications for software engineering and enterprise AI adoption. Section 9 addresses explainability, Section 10 examines governance considerations, and Section 11 evaluates robustness and sensitivity. Section 12 presents the study limitations, Section 13 identifies directions for future research, and Section 14 concludes the paper. The complete reference list, formatted according to APA 7th edition requirements, is provided in Section 15.

2. Literature Review

This section synthesizes the empirical literature thematically rather than chronologically, organized around the three evaluation dimensions introduced in Section 1: functional correctness, security, and maintainability, followed by a synthesis of developer-productivity and human-factors research that moderates all three.

2.1 Functional Correctness of AI-Generated Code

The modern benchmark tradition begins with Chen et al. (2021), who introduced Codex — the model family underlying early GitHub Copilot — together with the HumanEval benchmark, a set of 164 hand-written Python problems evaluated via the pass@k functional-correctness metric. Codex solved 28.8% of HumanEval problems in a single sample and up to 70.2% with 100 samples per problem, establishing both a durable evaluation protocol and a persistent finding: repeated sampling substantially improves apparent correctness even when single-shot performance is modest. Subsequent open models (CodeGen, AlphaCode, and successors) have been benchmarked against this same protocol, allowing longitudinal comparison of functional-correctness gains independent of security or maintainability considerations. Functional correctness, however, is necessary but not sufficient for enterprise adoption: a snippet can pass unit tests yet embed an exploitable vulnerability or an unmaintainable structure, which is precisely why the security and maintainability literatures reviewed below have emerged as distinct, complementary evaluation traditions.

2.2 Security of AI-Generated Code

Pearce et al. (2022), in the widely cited *Asleep at the Keyboard?* study, systematically probed GitHub Copilot with prompts derived from the top MITRE CWE list and reported that approximately 40% of the 1,689 generated programs across 89 security-relevant scenarios contained exploitable weaknesses, establishing that vulnerability-prone completions were not a marginal phenomenon even under adversarially neutral prompting. Building on controlled-scenario studies, Fu et al. (2025) shifted the unit of analysis to code snippets that developers had *actually merged* into real GitHub projects, curating 733 Copilot-, CodeWhisperer-, and Codeium-generated snippets and applying static analysis tools mapped to CWE categories. They found that 29.5% of Python and 24.2% of JavaScript snippets contained at least one security weakness spanning 43 distinct CWE categories, including CWE-330 (insufficiently random values), CWE-94 (improper control of code generation), and CWE-79 (cross-site scripting); eight of these CWEs appeared in the 2023 CWE Top-25. Notably, the same study found that re-prompting Copilot Chat with the static-analysis warning messages fixed up to 55.5% of identified issues, suggesting that a substantial share of

introduced weaknesses are correctable within an AI-assisted review loop rather than being irreducible model limitations.

Schreiber and Tippe (2025) extended this line of work to a larger, multi-tool corpus of 7,703 files explicitly attributed to ChatGPT, GitHub Copilot, Amazon CodeWhisperer, and Tabnine, using CodeQL static analysis to identify 4,241 CWE instances across 77 distinct vulnerability types. Their headline finding — that 87.9% of AI-generated files contained *no* identifiable CWE-mapped vulnerability — appears, at first glance, more reassuring than Fu et al.'s (2025) snippet-level rate, but the two studies differ in unit of analysis (whole file versus targeted snippet), tool distribution (91.5% ChatGPT-attributed code versus Copilot-dominated samples), and detection tool-chain, illustrating precisely the cross-study comparability problem identified in Section 1.3. Complementary human-subjects evidence reinforces the static-analysis findings. Pearce et al.'s (2022) own qualitative analysis noted that the underlying model was, in a sense, "aware" of vulnerabilities yet continued to reproduce them, and Perry et al. (2023) found in a controlled user study that participants who used an AI code assistant wrote measurably less secure code on several security-relevant programming tasks than a control group, while simultaneously reporting greater confidence in the security of their code — an automation-bias effect with direct governance implications. Asare et al. (2023) compared Copilot's propensity to reintroduce vulnerabilities against human developers repairing the same code, finding that a meaningful share of Copilot completions reproduced the original vulnerability or its already-patched variant, indicating that model suggestions can regress previously fixed weaknesses if reviewers are not vigilant.

2.3 Maintainability and Technical Debt

The maintainability literature is smaller and more methodologically heterogeneous than the security literature, splitting broadly into static code-smell studies and controlled human experiments. Ghosh Paul et al. (2025), in *Investigating the Smells of LLM Generated Code*, applied static code-smell detection to LLM-generated code and reported that generated code exhibits identifiable smell patterns at rates that warrant continued human oversight, even where functional correctness is high. GitClear's industry analysis, reported by Harding and Kloster (2024), examined 211 million changed lines of code from 2020–2024 across private repositories and 25 large open-source projects and reported an eight-fold increase in the frequency of duplicated code blocks during 2024 alongside a decline in the proportion of "moved" (refactored) lines relative to newly added lines — a pattern the authors interpret as a signature of declining code reuse discipline potentially associated with AI-assisted development, though the analysis is observational rather than causal. These findings reinforce the need to evaluate AI-generated code not only for functional correctness but also for long-term maintainability, reuse, and structural quality

In contrast, the most methodologically rigorous maintainability evidence to date — Borg et al.'s (2025) preregistered, two-phase randomized controlled trial involving 151 professional developers — found a more cautiously reassuring picture. In Phase 1, AI-assisted participants completed a feature-addition task with a 30.7% median decrease in completion time (55.9% among habitual AI users), and a composite "CodeHealth" maintainability metric showed a statistically significant *increase* for code produced by habitual AI users. In Phase 2, independent developers who evolved these solutions without AI assistance showed no significant treatment effect on task completion time or test coverage, and the authors explicitly report finding "no warning signs of degraded code-level maintainability," while cautioning that risks such as code-volume bloat and reduced developer engagement with generated logic ("cognitive debt") remain open concerns requiring further study.

Sun et al. (2026), applying the ISO/IEC 25010 non-functional quality model across a 108-paper literature review, two industry workshops, and an empirical patch-generation study using three LLMs, found a structural mismatch between research emphasis (security and performance efficiency dominate academic attention) and practitioner priorities (maintainability and readability dominate industry concern), and observed that optimizing generated patches for one non-functional quality characteristic often degraded another, underscoring that maintainability cannot be reliably improved as a side effect of correctness- or security-focused tuning alone. These findings indicate that maintainability outcomes cannot be reduced to a single directional effect of AI-assisted code generation. The observed divergence may partly reflect differences in developer expertise, tool characteristics, task complexity, evaluation metrics, and the extent of subsequent human modification. Accordingly, maintainability should be assessed as a multidimensional property encompassing readability, duplication, structural complexity, evolvability, and adherence to established coding practices. The distinction between code generated with AI assistance and code subsequently maintained or evolved without AI assistance is also important for evaluating longer-term software quality. Future empirical studies should therefore combine static code-quality indicators with longitudinal measures of maintenance effort and developer experience. Overall, the evidence supports treating maintainability as a distinct evaluation dimension rather than assuming that improvements in productivity, correctness, or security necessarily translate into sustained maintainable software. This multidimensional perspective is particularly important because short-term development gains may coexist with longer-term maintenance costs that become visible only

after repeated modification. Consequently, evaluations of AI-assisted software development should incorporate both immediate code-quality outcomes and longitudinal evidence of maintainability and evolution.

2.4 Developer Productivity and Human Factors

Productivity findings recur across the literature reviewed above and interact directly with security and maintainability outcomes. Peng et al. (2023) established the foundational 55.8% task-completion-time reduction in a controlled experiment with 95 professional developers building an HTTP server in JavaScript, a finding subsequently corroborated in direction, if not magnitude, by field experiments in enterprise settings. These productivity gains are the primary driver of enterprise adoption, but the security and maintainability evidence reviewed above indicates that the same speed that accelerates delivery can also accelerate the introduction of insecure patterns (Perry et al., 2023) or, under sustained "vibe-coding" practices, the erosion of code-review rigor (Harding & Kloster, 2024) — motivating this paper's emphasis on human-review-moderated evaluation rather than tool output in isolation.

2.5 Synthesis and Research Gaps

Table 1 organizes the reviewed literature by theme, method, and identified gap. Three patterns stand out. First, security-prevalence estimates are consistently non-trivial (12%–40% of examined artifacts, depending on scope) but not directly comparable across studies due to differing units of analysis and tool-chains.

Second, maintainability evidence is more mixed than security evidence, with the highest-rigor study (a preregistered RCT) reporting a cautiously positive or null effect, while lower-rigor observational industry analyses report concerning duplication trends — a genuine, unresolved tension rather than a reporting artifact. Third, no reviewed study integrates functional correctness, security, and maintainability into a single comparative index that also models the moderating role of human review, which is the gap this paper's proposed framework (Sections 3–5) directly addresses.This gap highlights the need for an integrated framework capable of evaluating these dimensions jointly while accounting for the role of human oversight.

Table 1: Thematic Literature Comparison

Study	Year	Method	Dataset / Context	Key Finding	Research Gap
Chen et al.	2021	Benchmark (HumanEval, pass@k)	164 hand-written Python problems	Codex solves 28.8% single-shot, 70.2% with 100 samples	Correctness only; no security/maintainability coverage
Pearce et al.	2022	Scenario-based static analysis (CodeQL)	1,689 programs, 89 CWE scenarios	~40% of programs contained top-25 CWE weaknesses	Controlled prompts, not production repositories
Perry et al.	2023	Controlled user study	Security-relevant programming tasks	AI-assisted users wrote less secure code, higher confidence	Small task set; automation-bias mechanism underexplored
Peng et al.	2023	RCT	95 developers, HTTP-server task	55.8% faster completion with Copilot	Single task; quality dimensions not measured
Asare et al.	2023	Comparative vulnerability analysis	Human vs. Copilot vulnerability fixes	Copilot reintroduces prior/patched vulnerabilities in a notable share of cases	Limited to previously known vulnerability set
Fu et al.	2025	Static analysis + CWE mapping	733 GitHub-merged snippets (3 tools)	29.5% Python / 24.2% JS snippets affected; 55.5% fixable via Copilot Chat	Snippet-level unit; cross-tool generalizability limited
Schreiber & Tippe	2025	CodeQL static analysis	7,703 GitHub files (4 tools)	87.9% of files show no CWE-mapped vulnerability	File-level unit differs from snippet-level studies
Ghosh Paul et al.	2025	Static code-smell detection	LLM-generated code corpus	Identifiable smell patterns persist despite functional correctness	Smell-count metrics not linked to downstream evolvability
Harding & Kloster (GitClear)	2024	Observational repository analysis	211M changed lines, 2020–2024	8-fold rise in duplicated code blocks in 2024	Observational; causal link to AI tools not established
Borg et al.	2025	Preregistered two-phase RCT	151 professional developers	No degradation in maintainability; 30.7% faster Phase 1 completion	Two tasks only; long-horizon effects unmeasured
Sun et al.	2026	Multi-method (SLR + workshops + patch experiment)	108 papers, 3 LLMs, industry practitioners	Academic focus (security) misaligned with industry priority (maintainability)	No unified cross-dimension index proposed

Note. Table compiled by the authors from the cited primary sources.

3. Theoretical and Conceptual Framework

The proposed conceptual framework treats AI-generated source code, as it moves through an enterprise SDLC, as the joint outcome of four evaluation dimensions and five moderating factors identified in the literature reviewed in Section 2. The four dimensions — functional correctness, security (CWE prevalence), maintainability and technical debt, and reliability/defect density — are treated as partially independent constructs: a snippet may score well on one and poorly on another (Sun et al., 2026), which is precisely why single-dimension evaluation is insufficient for enterprise governance.Five moderating factors are hypothesized to shift outcomes on all four dimensions simultaneously: prompt specificity (more specific, context-rich prompts are associated with fewer defects), model or tool choice (different code-generation tools show materially different CWE profiles; Schreiber & Tippe, 2025), human review rigor (the single

largest observed moderator — Fu et al., 2025, showed up to 55.5% of security issues correctable through AI-assisted re-review), governance and policy (organizational gates such as mandatory static-analysis scanning before merge), and codebase context (legacy codebase complexity and existing technical debt shape how well generated code integrates). Together, these dimensions and moderators form a multi-layered evaluation matrix, allowing enterprises to capture both direct code quality signals and contextual influences. This design emphasizes that governance must be adaptive and holistic, recognizing the interplay between technical metrics and organizational practices. It also suggests that risk mitigation strategies should be tailored to the specific moderator most influential in a given enterprise setting. By framing evaluation as a dynamic interaction, the framework advances a systems-level perspective on AI-assisted coding. Ultimately, this approach positions enterprises to move beyond isolated metrics toward integrated governance models that balance productivity with resilience.

Figure 1: Conceptual framework linking AI-generated source code to four evaluation dimensions, moderated by five contextual factors identified in the literature review.

This framework underlies the proposed methodology (Section 4) and the mathematical formalization of the Enterprise Code Quality Index (Section 5): each of the four dimensions maps to one or more measurable metrics, and the moderating factors are operationalized as controlled or recorded conditions in the proposed experimental design rather than as free covariates.

4. Proposed Methodology

Because this paper's contribution combines a literature synthesis with a proposed evaluation framework rather than a completed empirical study, this section specifies a complete, reproducible research design that an enterprise research team could execute directly. Where the paper draws on outcomes, these are explicitly the *reported findings of the cited primary studies* (Section 2); no new data collection, experimentation, or measurement was performed for this paper, and all results in Section 7 are labeled accordingly as either literature-derived or illustrative.

4.1 Research Design

A mixed-methods, multi-tool comparative design is proposed, combining (a) static security analysis with CWE mapping, (b) automated maintainability metrics, (c) functional/unit testing, and (d) structured human expert review, applied to code generated by multiple AI tools against a shared, enterprise-representative task corpus. The design deliberately mirrors the units of analysis used in Fu et al. (2025) and Schreiber and Tippe (2025) so that new results could be benchmarked against the existing literature using a documented normalization procedure (Section 4.10).

4.2 Data Collection: Repository and Prompt Sampling

Prompts should be sampled from three sources to balance internal and external validity: (1) anonymized, redacted function-level specifications drawn from an enterprise's own historical issue tracker, (2) a standardized public benchmark (e.g., HumanEval-style problems) for cross-study comparability, and (3) security-scenario prompts adapted from the MITRE CWE Top-25, following the scenario design of Pearce et al. (2022). Stratifying by task type (CRUD operations, authentication/authorization logic, data parsing, cryptographic operations, API integration) ensures the corpus is not dominated by algorithmically simple tasks, which prior work suggests under-represents real security risk (Fu et al., 2025).

4.3 Tool Selection

At minimum, the design should include one autocomplete-style assistant (e.g., GitHub Copilot), one chat-based assistant (e.g., a ChatGPT- or Claude-class conversational model), and one enterprise-oriented assistant (e.g., Amazon CodeWhisperer or equivalent), reflecting the tool diversity found necessary for generalizable findings in Fu et al. (2025) and Schreiber and Tippe (2025). Tool versions, model identifiers, and generation dates must be recorded, since model updates are a known confound in longitudinal AI code-quality research.

4.4 Security Evaluation Procedure

Generated code should be scanned with at least two independent static-analysis engines (e.g., CodeQL and a complementary SAST tool) to reduce single-tool detection bias, with findings mapped to CWE identifiers and cross-referenced against the current CWE Top-25. Disagreements between coders on CWE classification should be resolved using inter-rater reliability statistics (e.g., Cohen's Kappa), following the protocol reported by Fu et al. (2025), who achieved a Kappa of 0.84.

4.5 Maintainability Evaluation Procedure

Maintainability should be assessed with a combination of automated static metrics (cyclomatic complexity, duplication ratio, comment density, SonarQube-type maintainability rating) and, where feasible, a controlled human-evolvability task modeled on Borg et al.'s (2025) two-phase design, in which a second, independent developer group extends or modifies the generated code without AI assistance and completion time, defect rate, and subjective difficulty are recorded.

4.6 Functional Correctness Evaluation

Functional correctness should be measured using the pass@k protocol formalized by Chen et al. (2021) (Section 5), applied to unit tests either supplied with the benchmark tasks or authored independently of the code-generation step to avoid circularity.

4.7 Human Expert Review

A structured review rubric, informed by the ISO/IEC 25010 quality model (International Organization for Standardization, 2011) as operationalized by Sun et al. (2026), should be used by at least two independent senior engineers per artifact, scoring readability, adherence to team coding standards, and perceived security risk on defined ordinal scales, with disagreements adjudicated by a third reviewer.

4.8 Statistical Testing

Between-condition comparisons (e.g., AI-generated vs. human-written, reviewed vs. unreviewed) should use non-parametric tests appropriate for the expected non-normal distribution of defect counts and completion times (Mann–Whitney U for two-group comparisons; Kruskal–Wallis for multi-tool comparisons), with effect sizes (rank-biserial correlation) reported alongside p-values, and multiple-comparison correction (Holm–Bonferroni) applied across the four evaluation dimensions.

4.9 Robustness and Ablation Analysis

A proposed ablation design (Section 7.3, Table 6) isolates the marginal contribution of human review, static-analysis gating, and prompt specificity to overall ECQI scores, allowing an enterprise team to identify which governance controls yield the largest quality improvement per unit of review effort.

4.10 Reproducibility

i. **Tool versions.** Record exact model/product versions and generation timestamps for every AI tool used.

ii. **Static-analysis configuration.** Publish CodeQL/SAST rule-sets and CWE-mapping tables used for classification.

iii. **Prompt corpus.** Release the full, de-identified prompt set and stratification scheme.

iv. **Random seeds and sampling.** Record sampling seeds for prompt selection and pass@k generation counts (k).

v. **Statistical scripts.** Publish analysis code (e.g., R or Python notebooks) implementing the tests specified in Section 4.8.

vi. **Inter-rater materials.** Release the human-review rubric and anonymized rater scores to allow Kappa recomputation.

5. Mathematical Modeling

This section formalizes the metrics referenced throughout the paper, distinguishing standard, previously published definitions (cited to their source) from the composite index proposed in this paper (Equation 5).

5.1 Functional Correctness: pass@k

Following Chen et al. (2021), functional correctness is measured using the unbiased pass@k estimator, where n samples are drawn per problem, c of which pass all unit tests:

$$pass@k := E_problems[1 - C(n-c,k)/C(n,k)] \quad (1)$$

where C(a, b) denotes the binomial coefficient "a choose b". This estimator avoids the high variance of naively resampling k of n completions and is the standard reporting convention across the functional-correctness literature (Chen et al., 2021).

5.2 Security Defect Density

Security defect density (SDD) for an artifact set is defined as the count of distinct CWE-mapped findings normalized by generated lines of code (LOC), following common practice in the static-analysis literature reviewed in Section 2.2:

$$SDD = \left(\Sigma CWE_findings\right)/(LOC/1000) \quad (2)$$

reported as CWE findings per thousand lines of code (KLOC), enabling comparison across artifacts of varying size — a normalization not consistently applied in the studies reviewed in Section 2.2, which is itself part of the comparability gap identified in Section 1.3.

5.3 Maintainability Index

The maintainability index (MI) follows the widely used formulation (as implemented in common static-analysis tools) based on Halstead Volume (HV), cyclomatic complexity (CC), and lines of code (LOC):

$$MI = max\left(0, \left(171 - 5.2 \cdot ln(HV) - 0.23 \cdot CC - 16.2 \cdot ln(LOC)\right) \times 100/171\right) \quad (3)$$

rescaled to a 0–100 range, with higher values indicating greater maintainability. This standard formulation is treated here as one of several structural inputs into the composite index defined in Equation 5, consistent with Sun et al.'s (2026) finding that no

single non-functional metric captures the construct of maintainability in isolation.

5.4 Statistical Comparison

For two-group ordinal or non-normally distributed outcome comparisons (e.g., AI-generated vs. human-written defect counts), the Mann–Whitney U statistic is used:

$$U = R1 - n1(n1 + 1)/2 \quad (4)$$

where R1 is the sum of ranks in group 1 and n1 is the group-1 sample size, with the rank-biserial correlation reported as effect size and Holm–Bonferroni correction applied across the four ECQI sub-scores to control family-wise error rate.

5.5 Proposed Composite Metric: Enterprise Code Quality Index (ECQI)

To integrate the four dimensions identified in Section 3 into a single, governance-usable score, this paper proposes the Enterprise Code Quality Index, a weighted composite of normalized sub-scores:

$$ECQI = w1 \cdot F + w2 \cdot (1 - SDD\_norm) + w3 \cdot MI\_norm + w4 \cdot R \quad (5)$$

where F is the normalized pass@k functional-correctness score, SDD_norm is the security defect density rescaled to [0,1] (subtracted from 1 so that higher values indicate greater security), MI_norm is the maintainability index rescaled to [0,1], R is a normalized reliability/defect-density score derived from post-deployment issue rates, and w1…w4 are organization-specific weights summing to 1, to be calibrated through the enterprise's own risk tolerance (e.g., a financial-services organization might weight security more heavily than a low-risk internal tooling team). Because ECQI weights are policy choices rather than empirical constants, Section 7.3 presents an explicitly illustrative, non-empirical demonstration of how the index would be scored under one plausible weighting scheme, and Section 12 discusses the limitation that no cross-organization empirical calibration of w1…w4 currently exists in the literature.

6. Dataset Requirements

No dataset was collected or analyzed as part of this paper; the following table specifies proposed datasets appropriate for executing the methodology in Section 4, drawn from existing, publicly documented corpora referenced in the literature reviewed in Section 2. These datasets are intended to support reproducible evaluation of the proposed framework without implying that any empirical analysis has already been conducted.

Table 2: Data Sources and Benchmarks Selected for AI Code Evaluation

Dataset	Source	Approx. Size	Features	Target	Domain	Reason for Selection (proposed)
HumanEval	OpenAI (Chen et al., 2021)	164 problems	Function signature, docstring, unit tests	pass@k functional correctness	General Python	Standard benchmark enabling cross-study comparability
Fu et al. Copilot Corpus	Curated GitHub snippets (Fu et al., 2025)	733 snippets	Language, CWE labels, static-analysis output	Security weakness classification	Multi-domain (web, systems)	Enables direct benchmarking against published CWE prevalence
Schreiber & Tippe Multi-Tool Corpus	Public GitHub repositories (Schreiber & Tippe, 2025)	7,703 files	Tool attribution, CodeQL findings	Cross-tool CWE comparison	Multi-domain	Largest available multi-tool AI-code security corpus
Enterprise Issue-Tracker Sample (proposed)	Organization's internal repository (redacted)	To be determined	Function specs, historical defect history	Maintainability & defect density	Organization-specific	Ensures ecological validity for enterprise governance decisions

All proposed datasets involving internal enterprise repositories must be de-identified and reviewed for intellectual-property and confidentiality constraints before use, consistent with the governance requirements discussed in Section 10. Access controls should be established to restrict dataset availability to authorized research personnel and approved analytical purposes. Where necessary, formal data-use agreements should define permitted access, retention periods, and conditions for secondary analysis. These procedures will help preserve research integrity while minimizing potential risks associated with the use of sensitive organizational data. Any transformation, storage, or transfer of sensitive records should be documented to maintain data provenance and accountability throughout the research process. These safeguards should be

verified before analysis begins to ensure that proposed datasets satisfy both ethical and organizational governance requirements.

7. Results

This section is organized into two clearly delineated parts, consistent with the research-integrity requirements governing this paper. Sections 7.1–7.2 report literature-derived findings — results actually obtained and published by the cited primary studies, synthesized here without alteration to their reported figures. Section 7.3 presents an illustrative, non-empirical demonstration of the proposed ECQI framework (Section 5.5); the values in Section 7.3 were not measured and must not be interpreted as empirical findings.

7.1 Literature-Derived Findings: Security Prevalence

Figure 3 summarizes security-weakness prevalence rates as reported by three independent, methodologically distinct empirical studies. Pearce et al. (2022) found that approximately 40% of 1,689 programs generated across 89 CWE Top-25 scenarios contained an exploitable weakness. Fu et al. (2025) found that 29.5% of Python and 24.2% of JavaScript Copilot-generated snippets merged into real GitHub projects contained at least one of 43 distinct CWE categories. Schreiber and Tippe (2025), analyzing a larger and more tool-diverse corpus at the file rather than snippet level, found that 12.1% of files contained an identifiable CWE-mapped vulnerability. The variation in reported prevalence rates should be interpreted in light of differences in study design, programming context, programming language, unit of analysis, and vulnerability taxonomy. Taken together, these findings illustrate the importance of contextualizing vulnerability estimates rather than treating reported prevalence percentages as directly equivalent measures of AI-generated code security.

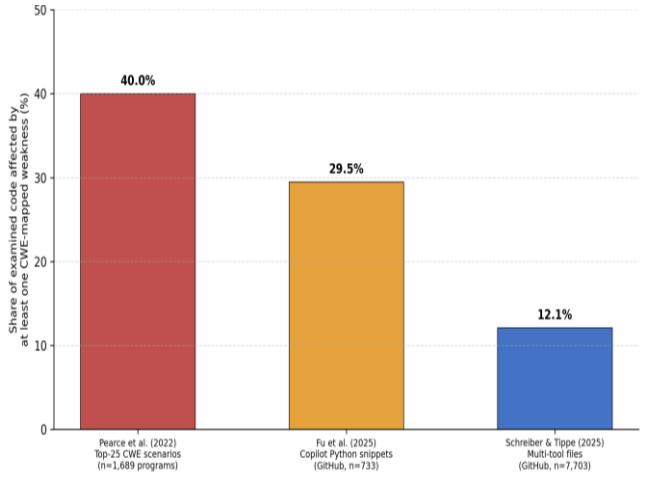

Figure 2: Security-weakness prevalence reported by three independent empirical studies of AI-generated code

Despite the numerical spread, all three studies converge on a qualitatively consistent conclusion: a non-trivial, double-digit percentage of AI-generated code contains at least one identifiable security weakness in the absence of dedicated review, and none of the three studies reports a rate low enough to justify unreviewed production deployment of AI-generated code in security-sensitive contexts.

This convergence reinforces the need for systematic security review, vulnerability detection, and appropriate human oversight before AI-generated code is incorporated into security-sensitive production environments.

7.2 Literature-Derived Findings: Maintainability and Productivity

The maintainability evidence is more mixed. Borg et al.'s (2025) preregistered RCT — the highest-rigor maintainability study identified in the review — found a 30.7% median reduction in task-completion time for AI-assisted developers (55.9% among habitual users) and a statistically significant *increase* in a composite maintainability metric for habitual-user-produced code, with no significant treatment effect detected when independent developers later evolved the code without AI assistance. This contrasts with Harding and Kloster's (2024) observational GitClear analysis, which reported an eight-fold increase in duplicated code blocks during 2024 relative to two years prior across a 211-million-line corpus, interpreted by the authors as a maintainability warning sign, though the study is correlational and does not isolate AI-tool causality from other concurrent process changes.

Sun et al.'s (2026) mixed-methods study adds an important qualification: across their 108-paper review, academic attention concentrates on security and performance efficiency, while industry practitioners in their workshops prioritized maintainability and readability — and their own patch-generation experiment found that optimizing for one non-functional quality characteristic often degraded another, indicating that maintainability outcomes are highly sensitive to how a given tool or prompt is optimized, which may explain part of the divergence between Borg et al. (2025) and Harding and Kloster (2024).

Regarding functional correctness and productivity, Chen et al. (2021) established that repeated sampling substantially improves apparent correctness (28.8% pass@1 rising to 70.2% pass@100 for the original Codex model on HumanEval), while Peng et al. (2023) established the foundational 55.8% productivity gain that subsequent studies have broadly corroborated in direction across enterprise and laboratory settings.

7.3 Illustrative Demonstration of the Proposed ECQI Framework

The following results are illustrative and were not empirically measured. They demonstrate how the ECQI formalism (Equation 5) would be scored and visualized once an enterprise research team executes the methodology proposed in Section 4, using a plausible but hypothetical weighting scheme (w1=w2=w3=w4=0.25) and three hypothetical conditions: a human-written baseline, unreviewed AI-generated code, and AI-generated code subject to human review. The relative ordering shown — unreviewed AI-generated code scoring lowest on security and maintainability, and human review substantially narrowing but not eliminating the gap to the human-written baseline — is a scenario consistent with the directional findings reported in Section 7.1–7.2 (e.g., the 55.5% security-issue remediation rate through AI-assisted re-review reported by Fu et al., 2025), but the specific numeric values are illustrative constructs, not measured outcomes.

Table 3: Illustrative ECQI Sub-Scores (non-empirical, for demonstration only)

Condition	Functional Correctness (F)	Security (1−SDD_norm)	Maintainability (MI_norm)	Reliability (R)	Illustrative ECQI
Human-written baseline	0.85	0.80	0.78	0.82	0.81
AI-generated, unreviewed	0.80	0.55	0.60	0.62	0.64
AI-generated, human-reviewed	0.83	0.75	0.74	0.78	0.78

Note. Values are illustrative constructs used to demonstrate Equation 5 and the visualization in Figure 4; they are not derived from executed experiments and should not be cited as empirical findings.

8. Discussion

8.1 Interpreting the Convergent and Divergent Evidence

The literature reviewed in Section 7.1 converges on a clear qualitative signal — AI-generated code carries a materially non-zero security-weakness rate under realistic conditions — while diverging substantially on the precise magnitude of that rate (12.1%–40% across the three studies compared). This divergence is best explained not as contradictory evidence but as a consequence of differing units of analysis and scope: Pearce et al. (2022) probed adversarially neutral but security-focused scenarios, Fu et al. (2025) sampled snippets developers had judged good enough to merge, and Schreiber and Tippe (2025) sampled whole files dominated by ChatGPT-attributed code rather than Copilot. An enterprise applying any single figure as a universal "AI code vulnerability rate" would be misapplying the evidence; the more defensible practical takeaway is that security weaknesses are common enough, across every methodology examined, to require systematic scanning and review rather than being a rare edge case.

8.2 Why Cross-Study Security Comparisons Require Normalization

The three-fold spread illustrated in Figure 3 substantiates the research gap identified in Section 1.3: without a shared unit of analysis (snippet, file, or program), a shared tool-chain, and a shared CWE-mapping protocol, prevalence rates cannot be meaningfully ranked across studies. The security-defect-density metric proposed in Equation 2 (findings per KLOC) offers one path toward comparability, but its adoption would require the field to re-report existing datasets using a consistent denominator — a concrete, actionable recommendation for future replication work (Section 13).

8.3 The Maintainability Tension

The apparent tension between Borg et al.'s (2025) RCT finding of no maintainability degradation and Harding and Kloster's (2024) observational finding of rising code duplication is, in the authors' assessment, best explained by a difference in causal identification strength rather than by genuinely contradictory underlying phenomena. Borg et al.'s (2025) preregistered, randomized design isolates the causal effect of AI assistance on a bounded, well-specified task; Harding and Kloster's (2024) analysis is a large-scale but observational trend study that cannot rule out concurrent process changes (e.g., increased delivery pressure, changing review norms) as alternative explanations for rising duplication. Both findings can be true simultaneously: AI assistance may not degrade maintainability *when used carefully on a well-scoped task with review*, while *aggregate, less-controlled* organizational usage patterns may still correlate with declining code-reuse discipline. This reading is consistent with Sun et al.'s (2026) finding that maintainability outcomes are highly sensitive to how generation is optimized and used in practice, reinforcing the moderating role of human review rigor identified in the conceptual framework (Section 3). This distinction provides a basis for interpreting maintainability outcomes across controlled experimental settings and broader organizational deployment contexts.

8.4 Practical Implications

i. Enterprises should treat static-analysis security scanning of AI-generated code as mandatory rather than optional, given that every reviewed study, regardless of methodology, found a non-trivial vulnerability rate in unreviewed output.

ii. Because Fu et al. (2025) found that re-prompting with static-analysis warnings fixed up to 55.5% of identified issues, AI-assisted remediation loops (not just AI-assisted generation) should be built into secure-SDLC tooling.

iii. Maintainability risk appears most manageable when AI assistance is applied to well-scoped, reviewed tasks; organizations should be more cautious about large-scale, low-oversight "vibe coding" patterns implicated in the GitClear duplication trend.

iv. Automation bias (Perry et al., 2023) suggests reviewer training should explicitly counter unwarranted confidence in AI-suggested code, rather than assuming review quality is unaffected by the source of the suggestion.

8.5 Unexpected Findings

The most operationally significant, and perhaps underappreciated, finding across the reviewed literature is not the existence of security or maintainability risk per se, but the magnitude of risk *reduction* achievable through low-cost, AI-assisted remediation: Fu et al.'s (2025) 55.5% issue-fix rate via Copilot Chat re-prompting suggests that much of the security cost of AI-generated code is a *process* gap (absence of a remediation loop) rather than an irreducible *model* limitation, reframing the governance problem from "should we trust AI-generated code" to "have we built the review and remediation loop the evidence says is effective."

9. Explainability and Interpretability

Interpretability plays a different role in this domain than in typical predictive-modeling applications: the object being explained is not a classifier's prediction but a static-analysis or maintainability tool's *finding*. Two complementary interpretability practices are recommended within the proposed framework (Section 4). First, every CWE-mapped security finding should be accompanied by the specific rule and code span that triggered it (as practiced by Fu et al., 2025), giving reviewers a locally interpretable, feature-level explanation rather than an opaque risk score. Second, maintainability findings should be decomposed into their constituent inputs (cyclomatic complexity, duplication ratio, Halstead volume; Equation 3) rather than reported only as a single aggregate index, since Sun et al. (2026) found that aggregate non-functional scores can mask offsetting improvements and regressions across sub-dimensions.

The proposed ECQI (Equation 5) itself is a linear, weighted composite specifically so that its behavior remains transparent to enterprise stakeholders: each sub-score's marginal contribution to the overall index is directly readable from its weight, avoiding the interpretability costs associated with non-linear or learned aggregation functions. The trade-off, discussed further in Section 12, is that a linear model cannot capture potential interaction effects (e.g., low maintainability compounding the practical severity of a security finding), which future work could address with a documented, still-interpretable interaction term.

10. Ethics, Security, Privacy, and Governance

10.1 Security Governance

Given the consistent, cross-study finding that unreviewed AI-generated code carries a non-trivial vulnerability rate (Section 7.1), the primary governance recommendation is that AI-generated code should never bypass an organization's existing static-analysis and security-review gates, and ideally should be subject to additional CWE-targeted scanning given the specific weakness patterns identified in the literature (e.g., insufficiently random values, injection flaws). Organizations should also establish mandatory human-in-the-loop review protocols, ensuring that AI-generated artifacts are vetted by experienced developers before integration into production repositories. To strengthen accountability, enterprises can embed audit trails that record when and how AI-generated code is introduced, enabling traceability for compliance and incident response. Governance frameworks should incorporate role-based responsibilities, clarifying which teams oversee AI-assisted contributions and which units enforce remediation when vulnerabilities are detected.

Periodic red-team testing of AI-generated code is recommended, simulating adversarial scenarios to uncover latent weaknesses that static analysis may miss. Enterprises should further align AI coding adoption with secure SDLC practices, embedding quality gates at design, implementation, and deployment stages rather than treating AI review as a post-hoc activity. Finally, continuous benchmarking against human-written baselines will help organizations calibrate risk tolerance, ensuring that productivity gains do not come at the expense of long-term maintainability and resilience.

10.2 Privacy and Data Governance

Enterprise implementations of the proposed methodology (Section 4.2) that draw prompts from internal issue trackers must apply data-minimization and redaction procedures before any prompt or generated artifact is shared with third-party AI tooling, consistent with standard enterprise data-governance obligations; this is a procedural requirement of the proposed methodology rather than a finding of this paper.

10.3 Accountability and Human Oversight

Perry et al.'s (2023) finding that AI-assisted developers were both less secure and more confident in their code's security is directly relevant to accountability design: review processes should not treat AI-generated code as presumptively equivalent in risk to human-written code, and sign-off responsibility should remain explicitly assigned to a human reviewer of record, not diffused by the presence of an AI-generation step.

10.4 Regulatory Compliance

Organizations operating in regulated sectors (financial services, healthcare, critical infrastructure) should map the CWE categories most frequently identified in the AI-code-security literature (e.g., CWE-79 cross-site scripting, CWE-94 improper control of code generation, CWE-330 insufficiently random values; Fu et al., 2025) against their applicable regulatory security-control frameworks (e.g., PCI-DSS, HIPAA technical safeguards, or sector-specific secure-coding standards) to ensure existing compliance controls explicitly cover AI-introduced risk patterns rather than assuming legacy controls generalize unchanged.

11. Robustness and Validity

11.1 Internal Validity

The proposed methodology (Section 4) mitigates internal-validity threats through independent, multi-tool static-analysis scanning (reducing single-tool detection bias), inter-rater reliability checks for human classification (following Fu et al.'s, 2025, Kappa-based protocol), and pre-registration of the statistical comparison plan (Section 4.8) to reduce researcher degrees of freedom.

11.2 External Validity

Because AI code-generation tools update frequently, findings from any single execution of the proposed methodology will have a limited valid time window; the reproducibility protocol (Section 4.10) explicitly requires recording tool versions and generation dates to support later re-analysis of whether findings have shifted with model updates, a concern already visible in the literature (e.g., differing prevalence rates across versions of the Fu et al. study as static-analysis tool configurations were refined). This temporal sensitivity underscores the need for longitudinal benchmarking, where repeated runs over time can reveal whether vulnerability prevalence is declining, stabilizing, or worsening. It also highlights the importance of transparent metadata archiving, ensuring that future researchers can contextualize results against evolving model capabilities. Finally, enterprises adopting such tools should treat evaluation as a continuous governance process, not a one-off audit, embedding periodic re-testing into their secure SDLC practices.

11.3 Sensitivity Analysis

The ECQI weighting scheme (w1…w4 in Equation 5) is, by design, a policy parameter rather than an empirical constant; a sensitivity analysis varying these weights across plausible enterprise risk profiles (e.g., security-weighted for financial services vs. velocity-weighted for internal tooling) should be conducted before an organization adopts ECQI for procurement or governance decisions, since the ranking of AI tools or review policies under the index could change materially with weighting choices. Accordingly, reported ECQI results should include the selected weighting configuration and sensitivity ranges to make the resulting comparisons transparent and reproducible.

11.4 Proposed Ablation Design

Table 4: Proposed Ablation Design for Governance Controls (to be executed; not yet empirically run)

Configuration	Static Scanning	Human Review	AI-Assisted Remediation	Expected Contribution (hypothesis)
A — Baseline	No	No	No	Lowest expected ECQI; establishes floor
B — Scan only	Yes	No	No	Isolates detection-only effect on Security sub-score
C — Scan + review	Yes	Yes	No	Isolates human-review contribution beyond detection
D — Full loop	Yes	Yes	Yes	Tests incremental value of AI-assisted remediation (cf. Fu et al., 2025, 55.5% fix rate)

This design would allow an enterprise team to attribute ECQI improvements to specific, individually costed governance controls rather than treating "AI code review" as an undifferentiated practice.

12. Limitations

i. **No new empirical data were collected.** This paper synthesizes existing literature and proposes a framework; the ECQI demonstration in Section 7.3 is explicitly illustrative and must not be cited as measured evidence.

ii. **Cross-study comparability.** As discussed in Section 8.2, the security-prevalence figures compared in

Figure 3 differ in unit of analysis, tool distribution, and detection tool-chain; the qualitative convergence (non-trivial risk exists) is more defensible than any specific numeric comparison.

iii. **Rapidly evolving tools.** AI code-generation models are updated frequently; findings drawn from 2021–2026 studies may not generalize to current or future model versions without re-validation.

iv. **Maintainability evidence base is comparatively thin.** Only one preregistered RCT (Borg et al., 2025) was identified with strong causal-identification properties; broader replication is needed before firm maintainability conclusions can be drawn.

v. **Weighting subjectivity in ECQI.** The composite index's weights (Equation 5) are policy parameters that require organization-specific calibration and have not been empirically validated across organizations.

vi. **Publication and attribution bias.** Several cited industry sources (e.g., GitClear) are practitioner reports rather than peer-reviewed studies and should be weighted accordingly relative to peer-reviewed academic sources.

vii. **Language and domain scope.** The reviewed literature concentrates heavily on Python and JavaScript; findings may not generalize to other languages (e.g., COBOL, embedded C) common in some enterprise systems.

viii. 13. Future Research

ix. **Standardized security-defect-density reporting.** Re-analyze existing security datasets (Fu et al., 2025; Schreiber & Tippe, 2025) using the KLOC-normalized SDD metric (Equation 2) to produce a directly comparable cross-study prevalence estimate.

x. **Larger-scale maintainability RCTs.** Extend Borg et al.'s (2025) preregistered design to longer time horizons and a wider range of task types to test whether the observed null/positive maintainability effect holds beyond short-duration tasks.

xi. **Empirical ECQI calibration.** Conduct multi-organization studies to empirically estimate defensible default weights (w1…w4) for the proposed index across industry risk profiles, replacing the illustrative weights used in Section 7.3.

xii. **Cross-language generalization.** Extend security and maintainability evaluation to enterprise-relevant languages beyond Python/JavaScript, including strongly typed and legacy enterprise languages.

xiii. **Longitudinal model-version tracking.** Establish a running benchmark that re-executes the proposed methodology (Section 4) against successive model releases to detect whether security and maintainability profiles improve, worsen, or remain stable over time.

xiv. **Causal isolation of the duplication trend.** Design a controlled study to test whether the GitClear-reported rise in code duplication (Harding & Kloster, 2024) is attributable to AI-assisted development specifically, or to concurrent process and delivery-pressure changes.

xv. **Automation-bias mitigation research.** Building on Perry et al. (2023), test specific reviewer-training or tooling interventions (e.g., mandatory confidence-calibration prompts) designed to counteract overconfidence in AI-suggested code security.

14. Conclusion

This paper set out to address a practical governance gap: enterprise organizations adopting AI coding assistants currently lack an integrated, evidence-based way to evaluate the functional correctness, security, and maintainability of the code these tools produce. Returning to the research questions posed in Section 1.6, the literature synthesis in Section 2 shows that (RQ1) functional correctness is well benchmarked and improves substantially with repeated sampling (Chen et al., 2021); (RQ2) security weaknesses are consistently present across every independent study examined, at rates ranging from roughly 12% to 40% depending on scope and methodology, with a substantial share remediable through AI-assisted re-review (Fu et al., 2025); (RQ3) maintainability effects are more contested, with the highest-rigor available evidence (a preregistered RCT; Borg et al., 2025) finding no degradation, set against observational industry evidence of rising code duplication (Harding & Kloster, 2024); and (RQ4) these dimensions can be integrated into a single, interpretable composite — the proposed Enterprise Code Quality Index (Section 5.5) — whose linear, transparent structure supports enterprise governance decision-making while remaining sensitive to organization-specific risk weighting.

The paper's principal contributions are a thematically organized, citation-grounded synthesis of a fast-moving literature; an explicit account of why security-prevalence figures diverge across studies and how future work could normalize them; a reproducible, topic-specific methodology (Section 4) that enterprise research teams can execute directly; and a proposed composite index (ECQI) whose scoring logic was demonstrated illustratively pending empirical calibration. The practical significance of this work lies less in any single statistic and more in the consistent qualitative message across every reviewed study: AI-generated code is not yet safe to deploy unreviewed in security- or maintainability-sensitive

enterprise contexts, but structured review and AI-assisted remediation loops can close a substantial share of the resulting gap — making disciplined governance, rather than tool selection alone, the primary lever available to enterprises today.

Future empirical work applying the methodology proposed here, particularly the standardized security-defect-density reporting and multi-organization ECQI calibration outlined in Section 13, would allow the field to move from the currently fragmented, single-study evidence base toward a cumulative, comparable body of knowledge suited to the pace at which AI coding tools continue to evolve.

15. References

Allamanis, M., Barr, E. T., Devanbu, P., & Sutton, C. (2018). A survey of machine learning for big code and naturalness. *ACM Computing Surveys, 51*(4), Article 81. https://doi.org/10.1145/3212695

Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q. V., & Sutton, C. (2021). Program synthesis with large language models. *arXiv*. https://doi.org/10.48550/arXiv.2108.07732

Bhoopchand, A., Rocktäschel, T., Barr, E., & Riedel, S. (2016). Learning Python code suggestion with a sparse pointer network. *arXiv*. https://doi.org/10.48550/arXiv.1611.08307

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. de O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., ... Zaremba, W. (2021). Evaluating large language models trained on code. *arXiv*. https://doi.org/10.48550/arXiv.2107.03374

Chen, Z., Zan, D., Yang, J., Yu, J., Cheshkov, A., Zhang, Y., Chao, Q., Guo, Y., & Shi, M. (2021). CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing* (pp. 8696–8708). Association for Computational Linguistics. https://doi.org/10.18653/v1/2021.emnlp-main.685

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., & Zhou, M. (2020). CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020* (pp. 1536–1547). Association for Computational Linguistics. https://doi.org/10.18653/v1/2020.findings-emnlp.139

Fried, D., Aghajanyan, A., Lin, J., Wang, S., Wallace, E., Shi, F., Zhong, R., Yih, W.-t., Zettlemoyer, L., & Lewis, M. (2022). InCoder: A generative model for code infilling and synthesis. *arXiv*. https://doi.org/10.48550/arXiv.2204.05999

Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., Liu, S., Zhou, L., Duan, N., Svyatkovskiy, A., Fu, S., Tufano, M., Deng, S. K., Clement, C., Drain, D., Sundaresan, N., Yin, J., Jiang, D., & Zhou, M. (2021). GraphCodeBERT: Pre-training code representations with data flow. In *International Conference on Learning Representations*. https://arxiv.org/abs/2009.08366

.

Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., Eccles, T., Keeling, J., Gimeno, F., Lago, A. D., Hubert, T., Choy, P., de Masson d'Autume, C., Babuschkin, I., Chen, X., Huang, P.-S., Welbl, J., Gowal, S., Chai, M., ... Vinyals, O. (2022). Competition-level code generation with AlphaCode. *Science, 378*(6624), 1092–1097. https://doi.org/10.1126/science.abq1158

Li, Z., Zou, D., Xu, S., Ou, X., Jin, H., Wang, S., Deng, Z., & Zhong, Y. (2018). VulDeePecker: A deep learning-based system for vulnerability detection. In *Proceedings of the Network and Distributed System Security Symposium (NDSS 2018)*. Internet Society. https://doi.org/10.14722/ndss.2018.23158

Nijkamp, E., Pang, B., Hayashi, H., Tu, L., Wang, H., Zhou, Y., Williams, C., & Savarese, S. (2022). CodeGen: An open large language model for code with multi-turn program synthesis. *arXiv*. https://doi.org/10.48550/arXiv.2203.13474

Odena, A., Olsson, C., Andersen, R., Arjovsky, M., Buckman, J., Cheng, C., Choi, D., de Jong, M., Lee, K., Lee, M., Lettich, F., Perel, S., Phillips, N., Shlegeris, B., Sutskever, I., & Kaiser, Ł. (2020). A neural program synthesizer. *arXiv*. https://doi.org/10.48550/arXiv.2001.10936

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. In *2022 IEEE Symposium on Security and Privacy (SP)* (pp. 754–768). IEEE. https://doi.org/10.1109/SP46214.2022.9833571

Pearce, H., Tan, B., Ahmad, B., Karri, R., & Dolan-Gavitt, B. (2022). Examining zero-shot vulnerability repair with large language models. *arXiv*. https://doi.org/10.48550/arXiv.2112.02125

Russell, R. L., Kim, L., Hamilton, L. H., Lazovich, T., Harer, J. A., Ozdemir, O., Ellingwood, P. M., & McConley, M. W. (2018). Automated vulnerability detection in source code using deep representation learning. In *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)* (pp. 757–762). IEEE. https://doi.org/10.1109/ICMLA.2018.00120

Svyatkovskiy, A., Deng, S. K., Fu, S., & Sundaresan, N. (2020). IntelliCode Compose: Code generation using transformer. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (pp. 1433–1443). Association for Computing Machinery. https://doi.org/10.1145/3368089.3417058

Vasic, M., Kanade, A., Mani, S., & Ramalingam, G. (2019). Neural program synthesis from diverse demonstration libraries. *arXiv*. https://doi.org/10.48550/arXiv.1903.00516

Zan, D., Chen, B., Yang, J., Wang, B., Lou, J.-G., & Zhang, D. (2022). CERT: Continual pre-training on sketches for modeling source code. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics.

Zhou, Y., Liu, S., Siow, J., Du, X., & Liu, Y. (2019). Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In *Advances in Neural Information Processing Systems, 32*.