

A Critical Review of Artificial Intelligence for Code Generation, Testing and Security Assurance Across the Software Development Lifecycle

Emmanuel E. Ajala[1*], Toussida Fatah Tanguy Minoungou[2], Harouna Wendpanga Yann Christian Sankara[3], Oluwademilade Oreoluwa Jooda[4], Chukwunonyelum Rosalyn Ezeako[5]

[1]Department of Computer Science, University of Alabama at Birmingham, Birmingham, Alabama, United States.
[2]DonorsChoose, New York, United States.
[3]Five9, San Ramon, California, United States.
[4]Goldman Sachs, Dallas, Texas, United States.
[5]University of Central Missouri, Warrensburg, Missouri, United States.
*Corresponding Author Email: emmanuelajala22@gmail.com

(Received: 29.11.2025; Accepted: 27.12.2025)

Abstract

Large language models trained on source code have moved artificial intelligence from a research curiosity at the edges of software engineering to a working component of everyday development practice. This review examines what the published evidence establishes about that shift across five phases of the software development lifecycle: requirements and design, implementation, testing, review and security assurance, and operations and maintenance. It synthesises more than 120 sources published between 2012 and 2025, from peer-reviewed empirical studies and benchmarks to practitioner frameworks and standards, and appraises the 24 primary empirical studies on which its assessments most depend against four criteria that separate replicated findings from claims resting on a single study or vendor-reported figures. The review finds that productivity gains from code assistants are real but uneven, that generated code carries a measurable security debt that users under-detect, that learned vulnerability detectors lose most of their reported accuracy when evaluated on realistic data, and that test and fuzz generation is the most consistently positive application area. It identifies benchmark contamination, label noise in vulnerability datasets, and package hallucination as three defects in the evidence base that distort current conclusions. The paper proposes an assurance loop that treats every model output as an untrusted contribution subject to static, dynamic, and human gates with recorded provenance.

Keywords: *Large language models; Code generation; Software testing; Vulnerability detection; Secure software engineering; DevSecOps; Empirical software engineering*

INTRODUCTION

Software engineering has absorbed waves of automation before, from compilers and static analysers to search-based test generation, but the arrival of large language models (LLMs) trained on source code is different in kind. Earlier tools encoded rules written by people; the current generation learns statistical regularities from billions of lines of public code and produces new code, tests, patches and explanations on demand. The observation that source code is highly repetitive and predictable, in the same sense that natural language is, was made more than a decade ago by Hindle *et al.* (2012) and gave the field its theoretical licence to apply language modelling to programs. The transformer architecture of Vaswani *et al.* (2017) and code-specific pre-training such as CodeBERT (Feng *et al.*, 2020) turned that licence into working systems, and the release of Codex (Chen *et al.*, 2021) and its deployment as GitHub Copilot put a code-generating model inside the editor of millions of developers.

The consequences are now the subject of a large and fast-growing empirical literature. Hou *et al.* (2024) catalogued 395 primary studies on LLMs for software engineering published between 2017 and 2024, and Fan *et al.* (2023) mapped the open problems. Yet the field's enthusiasm has run ahead of its evidence in several respects. Benchmarks on which models are compared are contaminated by their own training data (Sallou *et al.*, 2024). Vulnerability datasets on

which detectors are trained contain mislabelled samples at rates that invalidate reported accuracies (Croft *et al.*, 2023). Studies of developer productivity are often funded by tool vendors and measure task completion rather than delivered quality (Ziegler *et al.*, 2024). Controlled experiments on security show that AI assistance can make code worse while making its authors more confident (Perry *et al.*, 2023).

This review therefore takes a critical rather than descriptive stance. Its purpose is to establish, phase by phase across the software development lifecycle, what the peer-reviewed literature actually supports, where single studies have been treated as settled findings, and where the evidence base itself is defective. The contribution is threefold. First, the review organises the literature around lifecycle phases rather than model families, which makes it usable by practitioners who adopt tools one workflow at a time. Second, it isolates three structural defects in the evidence base, namely contamination, label noise, and package hallucination, and traces how each distorts published conclusions. Third, it proposes an assurance loop for AI-generated contributions that follows from the replicated findings. The paper follows the IMRaD structure. The relevant discussion sets out the review method and scope, and reports the results phase by phase, followed by the cross-cutting risks and an appraisal of the evidence base. The discussion section discusses the findings and presents the assurance loop, and the conclusion concludes.

METHODS

This section sets out how the review was conducted so that its conclusions can be weighed against its method. It describes the lifecycle framework used to organise the literature, the venues and date range from which sources were drawn, and the four appraisal questions applied to every primary study. The method is deliberately critical rather than enumerative. Systematic reviews of this field already exist and have counted the studies; Hou *et al.* (2024) catalogued 395 of them and Fan *et al.* (2023) mapped the open problems. What those reviews do not do is weigh the studies against each other, ask which results survive replication, and ask whether the benchmarks and datasets on which results rest are themselves sound. Those are the questions this section is designed to make answerable in the phase-by-phase analysis that follows (Sallou *et al.*, 2024).

Scope and Organising Framework

The review is organised around five phases of the software development lifecycle: requirements and design, implementation, testing, review and security assurance, and operations and maintenance. This phase-based structure follows the practice of the broader systematic reviews by Hou *et al.* (2024) and Wang *et al.* (2024), which found that the distribution of research effort across phases is highly uneven, with implementation-phase tasks such as code generation attracting the largest share of studies and requirements-phase tasks the smallest. Fig. 1 presents the organising framework, mapping the principal AI capabilities reviewed to each phase and identifying the cross-cutting concerns of human oversight, provenance, evaluation integrity and governance that apply at every phase. Within each phase, the review distinguishes three classes of evidence: benchmark evaluations of model capability, controlled experiments or field studies with human participants, and analyses of the datasets and benchmarks themselves. The third class is given particular weight because defects in evaluation infrastructure propagate into every conclusion built on it (Chakraborty *et al.*, 2022).

Source Selection

Sources were drawn from the flagship software engineering and security venues in which the primary empirical work on this topic appears: the International Conference on Software Engineering (ICSE), the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), the International Symposium on Software Testing and Analysis (ISSTA), the IEEE Symposium on Security and Privacy, the USENIX Security Symposium, the ACM Conference on Computer and Communications Security (CCS), IEEE Transactions on Software Engineering, ACM Transactions on Software Engineering and Methodology, ACM Computing Surveys and Empirical Software Engineering, together with the machine learning venues (NeurIPS, ICML, ICLR) at which code-model benchmarks are introduced. Foundational pre-

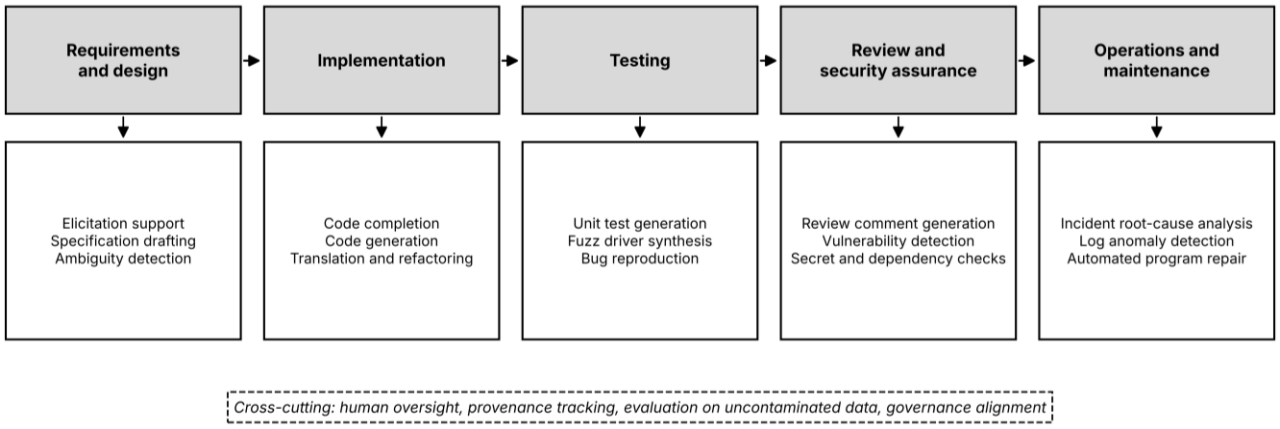


Fig. 1: Organising framework for the review: principal AI capabilities by lifecycle phase, with cross-cutting concerns that apply at every phase.

Table 1: Scope of the review by lifecycle phase, with the tasks examined and representative primary studies.

Phase	Tasks examined	Dominant technique class	Representative primary studies
Requirements and design	Elicitation support, specification drafting, ambiguity detection	Prompted general-purpose LLMs	Ronanki *et al.* (2023); Arora *et al.* (2024)
Implementation	Code completion, code generation, benchmark evaluation, usability	Code-pretrained LLMs; controlled user studies	Chen *et al.* (2021); Li, Y. *et al.* (2022); Nguyen and Nadi (2022); Barke *et al.* (2023); Liang *et al.* (2024); Ziegler *et al.* (2024)
Testing	Unit test generation, fuzzing, bug reproduction	Search-based baselines; LLM-augmented search; prompted generation	Fraser and Arcuri (2011); Lemieux *et al.* (2023); Schäfer *et al.* (2024); Xia *et al.* (2024); Deng *et al.* (2023)
Review and security assurance	Review comment generation, vulnerability detection, security of generated code	Pre-trained code models; graph neural networks; controlled experiments	Li, Z. *et al.* (2022); Zhou *et al.* (2019); Chakraborty *et al.* (2022); Pearce *et al.* (2022); Perry *et al.* (2023); Ding *et al.* (2025)
Operations and maintenance	Incident root-cause analysis, program repair	Fine-tuned and in-context LLMs; APR baselines	Ahmed *et al.* (2023); Chen *et al.* (2024); Xia *et al.* (2023); Le Goues *et al.* (2012)

LLM work is included where it establishes a baseline against which learned approaches must be judged, for example search-based test generation (Fraser and Arcuri, 2011) and genetic-programming repair (Le Goues *et al.*, 2012). Table 1 summarises the scope by phase, the tasks examined and the representative studies on which each phase's assessment rests. The review privileges peer-reviewed publications; where a widely cited result exists only as a technical report, it is identified as such and treated as provisional. Two further strands are drawn on where they describe the environments into which AI tooling is deployed rather than the tooling itself. Okoruwa *et al.* (2024) survey the evolution of enterprise threat activity and the research directions it has opened, which situates the security findings of the discussion of review and security assurance in the threat landscape organisations actually face. Fadayomi *et al.* (2019) set out risk-based cybersecurity assurance, in which controls are selected against assessed risk and the availability of data is treated as an assurance objective in its own right; that framing is adopted in the proposed assurance loop, where the gates applied to generated code are proportioned to the risk of the change rather than applied uniformly.

Appraisal Criteria

Each primary study was appraised against four questions that recur throughout the review. First, was the evaluation data plausibly outside the model's training distribution, or could the result reflect memorisation? Sallou *et al.* (2024) show that data contamination is the single most common unaddressed threat in LLM-based software engineering research. Second, were the labels or ground truth independently validated? Croft *et al.* (2023) found label inaccuracy rates between 20 and 71 percent across widely used vulnerability datasets. Third, did the study measure the outcome that matters to practice, such as defects delivered or vulnerabilities shipped, or a proxy such as task completion time or benchmark pass rate? Fourth, who funded and conducted the study, and did the authors have a commercial interest in the tool evaluated? These criteria are applied throughout and underpin the distinction drawn in the discussion of replicated findings and provisional claims between replicated findings and provisional claims.

Source Selection Procedure

Sources were identified in three stages during the preparation of this review in 2025, and the procedure is set out so that the selection can be audited. In the first stage, the proceedings of the venues listed in the source-selection discussion for the years 2012 to 2025 were searched through the ACM Digital Library, IEEE Xplore, and DBLP using the terms large language model, code generation, code completion, program synthesis, test generation, fuzzing, program repair, vulnerability detection, code review, incident management, and AIOps, each combined with the venue name and year. In the second stage, the reference lists of the two systematic reviews that bound the field, Hou *et al.* (2024) for software engineering tasks and Wang *et al.* (2024) for testing, were screened for empirical studies that the first stage had not returned, and the citing papers of the three data-quality replications (Chakraborty *et al.*, 2022; Croft *et al.*, 2023; Ding *et al.*, 2025) were screened in the same way. In the third stage, practitioner and applied sources describing the deployment environment were added from the applied literature identified in the source-selection discussion, subject to the relevance and date limits in Table 2.

A source was eligible for citation if it was published in or before 2025, if it bore directly on the use or governance of AI in one of the five lifecycle phases, and if it was either peer-reviewed or an official standard or regulation. A source was eligible for the appraisal in Table 5 only if it met three further conditions: it reported a primary empirical result (a benchmark evaluation, a study with human participants, or an audit of a dataset or benchmark) rather than a survey, framework, or position; it was the study on which a phase finding in the relevant discussion or a replicated or provisional claim in the discussion of replicated findings and provisional claims rests; and it was published at one of the flagship venues in the source-selection discussion, so that the venue's own review supplies a baseline of methodological scrutiny. Applying those conditions to the cited corpus yielded the 24 studies in Table 5. Table 2 records the eligibility criteria and the principal reasons for exclusion at each stage. Surveys were excluded from appraisal because the criteria concern evaluation design; vendor-authored

Table 2: Eligibility criteria for citation and for appraisal, with principal exclusion reasons.

Stage	Criterion applied	Principal reasons for exclusion
Citation in the review	Published in or before 2025; bears directly on AI use or governance in a lifecycle phase; peer-reviewed, or an official standard or regulation	Published after 2025; topic outside the five phases (for example agricultural, clinical or marketing applications of AI); preprint with no peer-reviewed version where a peer-reviewed alternative existed
Appraisal in Table 5 (condition 1)	Primary empirical result: benchmark evaluation, study with human participants, or audit of a dataset or benchmark	Surveys, systematic reviews, conceptual frameworks, standards and position papers
Appraisal in Table 5 (condition 2)	The study on which a phase finding in the results section, or a replicated or provisional claim in replicated findings and provisional claims sub-section, rests	Empirical studies cited for context only, whose result no finding depends on
Appraisal in Table 5 (condition 3)	Published at a flagship venue listed in source selection	Empirical studies at other venues, which are cited in the phase sections but not appraised because venue review is used as the baseline of scrutiny

studies were retained because their exclusion would have removed the productivity evidence that the review exists to examine, and their vendor involvement is recorded as an appraisal outcome instead (Sallou *et al.*, 2024).

RESULTS

This section reports the review findings across the five lifecycle phases, then examines risks that cut across phases and appraises the primary studies underpinning the phase-by-phase assessments. Within each phase, the evidence is organised into the three classes described in the lifecycle framework: benchmark evaluations, studies with human participants, and analyses of datasets and benchmarks. Where a finding rests on a single study or a vendor-conducted study, this limitation is stated alongside the finding, allowing readers to assess the evidence in context (Hou *et al.*, 2024; Sallou *et al.*, 2024).

Requirements and Design

The earliest lifecycle phases are where the leverage of any intervention is highest, because a defect in a requirement or an architectural decision propagates into every downstream artefact and is the most expensive to correct once code exists. They are also the phases where language, rather than code, is the primary medium, which ought to favour language models. The published evidence does not yet bear this expectation out. The systematic review by Hou *et al.* (2024) found requirements and design tasks to be the least studied of all lifecycle activities, and the few studies that exist are small and rely on expert ratings of generated artefacts rather than on project outcomes. This section reviews what has been established for elicitation and specification and, more briefly, for design, and sets out why the appropriate claim is modest (Arora *et al.*, 2024).

Elicitation and Specification

The requirements phase has received the least attention from the LLM research community, which is paradoxical given that natural-language processing is the technology's native strength. Ronanki *et al.* (2023) compared requirements elicited by ChatGPT with those produced by requirements engineering students and practitioners on the same prompts and found the model's outputs rated favourably on abstraction, atomicity, consistency and correctness, though the study design, which relied on expert ratings of short generated lists, cannot speak to the completeness or feasibility of the requirements in a real project. Arora *et al.* (2024) examined the broader role of LLMs across elicitation, specification, analysis, and validation and concluded that the most promising near-term use is as a drafting and questioning assistant that proposes candidate requirements and surfaces ambiguities for a human analyst to resolve, rather than as an autonomous elicitor. The business-analysis literature approaches the same problem from the demand side. Alieme *et al.* (2024) propose a framework for converting an ambiguous business problem into requirements that an automation system can act on, and their central finding is that the hard work lies in making decision boundaries, data availability, and exception handling explicit, none of which a model can infer from a vague brief.

Two limitations recur in this small literature. The first is that generated requirements inherit the biases and omissions of the prompt; a model cannot elicit what the prompt author did not know to ask about, and stakeholder conflict, which is the central difficulty of elicitation in practice, is invisible to a model reading a single description. The second is the absence of longitudinal evidence. No published study has yet followed a project from LLM-assisted elicitation through to delivery and measured whether requirements defects, which are among the most expensive to correct late, were reduced. The evidence therefore supports a modest claim: LLMs are useful for drafting and for generating counter-questions, and unproven for anything more (Arora *et al.*, 2024). Alieme *et al.* (2025) compare how Agile, Waterfall and hybrid processes handle elicitation for AI-enabled process automation and find that the burden shifts away from screens and transactions towards specifying training data, acceptable error rates and escalation paths; a requirements process that still elicits the former while ignoring the latter will produce a system whose behaviour nobody specified.

Design and Architecture

Design-phase applications are even thinner in the peer-reviewed literature. The systematic review by Hou *et al.*

(2024) records design and architecture tasks as a small fraction of the primary studies they catalogued, and most of those concern the generation of design artefacts such as class diagrams or API sketches from textual descriptions rather than architectural decision-making under constraints. Fan *et al.* (2023) identify design-level reasoning as an open problem precisely because architectural quality is judged over years against changing requirements, which no current benchmark captures. What the literature does establish is that models can produce plausible-looking design documents quickly, and that plausibility is a hazard: a fluent but shallow design rationale can anchor a team's thinking before the hard trade-offs have been examined. Until design-phase evaluations with real architects and real systems are published, the appropriate posture is to treat model-generated design artefacts as structured prompts for human deliberation, consistent with the drafting role identified for requirements (Arora *et al.*, 2024).

What a Requirements-Phase Evidence Base Would Require

Three kinds of studies are missing and would be inexpensive to run. The first is a comparison of requirements defects, counted at acceptance testing, between projects whose elicitation used a language model as a drafting assistant and matched projects that did not; Ronanki *et al.* (2023) supply the rating instruments, and Arora *et al.* (2024) the task taxonomy, so the design is ready. The second is a study of ambiguity detection, in which a model is asked to flag underspecified requirements in a real backlog, and its flags are compared with the ambiguities that later caused rework. The third is a longitudinal study of the specification burden that Alieme *et al.* (2025) identify, measuring whether teams that specify data, error tolerance, and escalation at requirements time experience fewer production incidents in the resulting automated system. Until studies of this kind are published, the requirements phase remains the part of the lifecycle where the technology's native strength in language is least supported by evidence of benefit (Hou *et al.*, 2024).

Implementation

Implementation is the phase in which AI assistance is most widely deployed and most intensively studied, and it is the phase on which the strongest commercial claims rest. This section examines the evidence in three parts. It first asks what the standard benchmarks measure and shows, through three independent lines of work, that function-level pass rates systematically overstate engineering competence. It then reviews the controlled experiments, field studies, and surveys of developer productivity, distinguishing vendor-conducted telemetry studies from independent user studies and distinguishing measures of perceived productivity from measures of delivered output (Ziegler *et al.*, 2024; Mozannar *et al.*, 2024). Finally, it considers the human-factors evidence on how assistants change the way programmers work and learn, which bears on the long-run quality of the profession rather than on the speed of any single task (Becker *et al.*, 2023).

Capability: What Benchmarks Measure and What They Miss

The implementation phase is where the evidence is deepest and where the gap between benchmark capability and delivered value is most visible. Chen *et al.* (2021) introduced the HumanEval benchmark of 164 hand-written programming problems with unit tests alongside the Codex model, and the pass@k metric they defined became the field's standard yardstick. Li, Y. *et al.* (2022) demonstrated with AlphaCode that a model could reach a median ranking in the top 54.3 percent of human participants in Codeforces competitions, a result that was widely reported as evidence of human-level programming. Both results measure the ability to produce a short, self-contained function from a precise specification, which is a small and atypical fraction of professional programming.

Three subsequent lines of work show how much the standard benchmarks overstate. Liu *et al.* (2023) augmented HumanEval with far more thorough test cases in the EvalPlus framework and found that pass rates for leading models fell by between 19.3 and 28.9 percent, meaning that a substantial share of code previously counted as correct contained bugs

Table 3: Principal empirical studies of AI code assistants in the implementation phase, with study design and the finding each supports.

Study	Design	Finding supported
Nguyen and Nadi (2022)	Benchmark evaluation of Copilot on LeetCode problems in four languages	Correctness ranged from 27 percent (JavaScript) to 57 percent (Java); correct solutions had lower complexity than human solutions
Vaithilingam *et al.* (2022)	Within-subjects user study, 24 participants	No significant completion-time advantage; participants preferred Copilot as a starting point
Barke *et al*. (2023)	Grounded-theory observation of 20 programmers	Two interaction modes (acceleration and exploration); gains concentrated in acceleration
Dakhel *et al.* (2023)	Comparison against human programmers on algorithmic problems	Lower correctness than humans; Copilot's buggy solutions easier to repair
Liang *et al.* (2024)	Survey of 410 developers	Rejections driven by unmet functional and non-functional requirements and missing project context
Mozannar *et al.* (2024)	Instrumented sessions with behavioural coding	Substantial session time spent verifying suggestions, bounding achievable speed-up
Ziegler *et al.* (2024)	Telemetry and survey of Copilot users (vendor study)	Acceptance rate predicts self-reported productivity; outcome quality not measured

the original tests did not reach. Jimenez *et al.* (2024) constructed SWE-bench from 2,294 real GitHub issues in twelve Python repositories and reported that the best model at the time of publication resolved only 1.96 percent of them when required to edit a real codebase rather than write a function from scratch. Sallou *et al.* (2024) demonstrated that HumanEval-style problems and their solutions are present in the public corpora on which models are trained, so that benchmark scores partly measure recall. Taken together, these studies establish that function-level pass rates are not evidence of competence at repository-level engineering, and that any benchmark older than the model under test should be presumed contaminated unless shown otherwise.

Productivity: Controlled and Field Evidence

Claims about developer productivity carry the largest commercial weight and warrant the closest scrutiny. Table 3 summarises the principal empirical studies. Ziegler *et al.* (2024), writing from GitHub, analysed telemetry from Copilot users and found that the rate at which developers accept suggestions is the strongest predictor of self-reported productivity, which is a finding about perception rather than output. Nguyen and Nadi (2022) evaluated Copilot's suggestions on LeetCode problems across four languages and found correctness varying from 27 percent in JavaScript to 57 percent in Java, with the correct solutions typically of lower complexity than human-written ones. Dakhel *et al.* (2023) compared Copilot against human programmers on fundamental algorithmic problems and found that Copilot produced correct solutions less often but that its buggy solutions were often easier to repair than students' errors, a nuance that matters for how the tool is positioned in a workflow.

The user-study literature adds mechanisms to these outcome measures. Barke *et al.* (2023) observed programmers using Copilot and identified two distinct modes of interaction: an acceleration mode in which the programmer already knows what to write and uses the model to type it faster, and an exploration mode in which the programmer uses suggestions to discover an approach. Gains were concentrated in acceleration mode. Mozannar *et al.* (2024) instrumented programming sessions and found that a large share of session time was spent verifying suggestions rather than writing, which caps the achievable speed-up and explains why acceptance rate correlates with perceived productivity. Liang *et al.* (2024) surveyed 410 developers and found that the dominant reason for rejecting assistants was that generated code did not meet functional or non-functional requirements, and that developers found the tools hardest to use for tasks requiring project-specific context. Vaithilingam *et al.* (2022) found in an early study that participants did not complete tasks faster with Copilot but preferred it because it offered a useful starting point. The consistent picture is of a tool that reduces typing and lookup effort for well-understood tasks while shifting effort towards verification, with no peer-reviewed field study yet demonstrating a reduction in delivered defects. Adoption itself is uneven across organisations and regions, and the diffusion literature explains why: Oteri and Chibwaye (2022) model early technology adoption in emerging and transitional economies econometrically and find that adoption timing is governed by infrastructure readiness, skills availability and institutional trust rather than by the capability of the technology, which implies that assistant adoption will follow the organisations already equipped to verify what the tools produce. Two distinctions should be kept in view when reading these results. Perceived productivity, which is what Ziegler *et al.* (2024) and Liang *et al.* (2024) measure through self-report and acceptance rate, is not the same quantity as measured output, which Vaithilingam *et al.* (2022) and Dakhel *et al.* (2023) measure through task completion and correctness, and the two diverge in the studies reviewed. Every productivity result above was obtained with GitHub Copilot or the Codex model that underlies it, between 2021 and 2024, on Python, JavaScript or Java tasks of function or script scale; the findings are observations about those models, languages and task settings, and the generalisation to other assistants, languages and repository-scale work is an interpretation rather than a result.

Human Factors and Skill Formation

A further strand of evidence concerns the effect of assistants on how programmers think and learn, which bears on the long-run quality of the profession rather than the speed of any one task. Becker *et al.* (2023) reviewed the implications of code generation for programming education and documented the concern that students who receive working code before they understand the problem may fail to develop the debugging and decomposition skills on which later competence depends. Barke *et al.* (2023) observed that exploration-mode use, in which the programmer does not yet know the solution, was the mode most prone to accepting plausible but wrong suggestions, which is the same population at greatest educational risk. Mozannar *et al.* (2024) showed that the verification burden falls disproportionately on the programmer's ability to read and judge unfamiliar code, which is itself a skill that assistants may erode if they reduce the amount of code programmers write by hand. None of these studies measures long-run skill outcomes directly; they establish the mechanism and the plausibility of the concern, and the review treats the long-run effect as an open question rather than a finding. The education literature frames the mechanism explicitly: Adams (2023) argues that critical thinking atrophies when a generative tool supplies the answer before the learner has formed the question, and proposes that assessment be redesigned around the reasoning rather than the artefact. The workforce literature adds the economic framing; Essandoh *et al.* (2025) examine how automation of routine knowledge work redistributes rather than removes job roles, and routine coding is one of the roles being redistributed towards specification and verification. The constructive counterpart is the human-in-the-loop literature, which treats human judgement as a designed component of the workflow rather than a residual. Ladapo *et al.* (2022a) survey the state of the art and identify the placement of the human – at labelling, at model selection, or at output review – as the design decision that determines whether the loop improves the model or merely slows it; Ladapo *et al.* (2024) apply the pattern to generative annotation and show that review-and-correct

loops improve both the immediate output and the model's subsequent behaviour, which is the mechanism the assurance loop of the proposed assurance loop relies on.

Delivery Pipelines and Platform Engineering

A code assistant's output reaches production through a delivery pipeline, and the governance of that pipeline, not the capability of the model, decides whether generated code improves or degrades what ships. The sources in this subsection describe pipeline and platform governance rather than AI directly; they are included because they define the control points at which the gates of the proposed assurance loop operate, and each is cited for that purpose only. Badmus *et al.* (2018) systematically reviewed continuous integration and delivery pipeline strategies on the Salesforce platform and found that deployment outcomes depended on branching discipline, environment management, and release approval rather than on the pipeline tool chosen. Badmus *et al.* (2020) then compared the Copado, Flosum, and Salesforce DX toolchains across enterprise deployment contexts and reached the same conclusion from the tooling side: the toolchains differ in convenience, but the control points at which a change is reviewed and promoted are what determine risk. Those control points are exactly where a model's suggestions must be gated, because an assistant that writes directly into a branch bypasses them. Olagunju *et al.* (2024) reach the same conclusion at programme level, proposing an applied performance optimisation framework for high-demand technology programmes and system rollouts in which stabilisation after release, not speed to release, is the measure of success.

Infrastructure code deserves the same treatment as application code, and increasingly comes from the same assistants. Mbonu *et al.* (2020b) examine governance for Terraform-based enterprise cloud deployments and show that a misconfiguration committed as code propagates to every environment the pipeline touches, which makes review of generated infrastructure code a security control rather than a style matter. Oshoba *et al.* (2020) propose an audit-trail model for cloud configuration management in which every configuration change is recorded immutably with its author and approval, a design that maps directly onto the provenance record of the proposed assurance loop once the author may be a model. Ojukwu *et al.* (2025) examine the WebAssembly component model and WASI as a portable cloud-to-edge runtime and document its sandboxing and interoperability trade-offs, which matter because generated code is increasingly deployed to runtimes the developer did not choose and may not understand.

Platform change also shapes how assistants are adopted. Fadero *et al.* (2025) describe the secure separation of IT services after a merger, acquisition, or ownership transition, a setting in which an assistant trained on one organisation's conventions may generate code that violates the other's, and in which provenance of generated changes is needed to unwind them. Badmus *et al.* (2025) propose a governance framework for AI-assisted CRM workflows that treats the deployment of an assistant as a change subject to risk assessment, readiness review and defined rollback rather than as a feature toggle. Across this literature, the consistent lesson is that the organisations best placed to benefit from assistants are those that already govern their pipelines well (Ziegler *et al.*, 2024). Two further contributions treat the organisational side of such transitions as a data problem. Oteri and Chibwaye (2025) review change management analytics for large-scale system transitions and argue that adoption risk and resistance can be predicted from operational data rather than assessed retrospectively; Chibwaye and Oteri (2023) examine data-integrated ERP migration for large government workforces and identify data governance, not system functionality, as the binding constraint. Both apply directly to the introduction of assistants, which is a system transition whose adoption risk is rarely measured.

Testing

Testing differs from the other phases reviewed in one decisive respect: the output of a test generator can be checked by running it. A generated test either compiles, executes, and passes or it does not, and a fuzzing campaign either finds a confirmed crash or it does not. This gives the testing literature a quality of evidence that is unavailable for code generation, where correctness is judged by benchmarks, or for requirements, where it is judged by raters. The long tradition of search-based test generation also supplies a strong non-learned baseline against which learned approaches can be measured (Fraser and Arcuri, 2011). This section reviews unit test generation, fuzzing and input generation, and the limits of generated tests, and concludes that testing is the application area with the most consistently positive and best-validated record, subject to one unsolved problem, the test oracle (Wang *et al.*, 2024).

Unit Test Generation

Testing is the phase in which learned approaches have the clearest positive record, in part because the output can be checked by execution. The relevant baseline is search-based test generation, exemplified by EvoSuite for Java (Fraser and Arcuri, 2011) and Pynguin for Python (Lukasczyk and Fraser, 2022), which evolve test suites to maximise coverage. These tools achieve high coverage but produce tests that are hard to read and that encode the behaviour of the code under test, including its bugs, as oracles. Lemieux *et al.* (2023) addressed a specific weakness of search-based generation, the coverage plateau that occurs when the search cannot construct the inputs needed to reach new code, by invoking a code model to propose example tests when the search stalls. Their CodaMosa system improved coverage on a majority of the modules studied, and the design is instructive because it uses the model to escape local optima rather than to replace the search.

Schäfer *et al.* (2024) evaluated prompted generation alone with their TestPilot system on 25 npm packages and reported median statement coverage of 70.2 percent and median branch coverage of 52.8 percent, with a substantial fraction of generated tests passing and a smaller fraction failing because of incorrect assertions rather than syntax errors. The systematic survey by Wang *et al.* (2024) covering 102 studies confirms unit test generation as the most studied testing application and identifies oracle correctness as the central

unsolved problem: a model that writes an assertion matching the current behaviour of the code produces a regression test, not a test of intent. Kang *et al.* (2023) demonstrated a related capability, the reproduction of bugs from natural-language reports, with their LIBRO approach generating reproducing tests for roughly a third of the bugs in the Defects4J benchmark, which is useful for triage even where the generated test is not retained. Borah *et al.* (2022) catalogue the applications of AI across test case generation, test prioritisation and defect prediction and find that prioritisation, where the model orders existing tests rather than writing new ones, is the application with the fewest failure modes because the oracle is already present. Mbonu *et al.* (2021) examine AI techniques for secure software testing and automated regression control and make a point that the generation literature overlooks: a regression suite that a model has generated or re-ordered is itself a change to the system under configuration management, and modifications to it should be reviewed with the same rigour as modifications to the code it protects.

Fuzzing and Input Generation

Fuzzing has benefited from LLMs in a different way, by lowering the cost of constructing the harnesses and seed inputs that fuzzing requires. Deng *et al.* (2023) showed that a code model could generate valid, diverse programs exercising deep-learning library APIs with no task-specific training, discovering previously unknown bugs in PyTorch and TensorFlow. Xia *et al.* (2024) generalised the approach in Fuzz4All, which uses a model to produce and mutate inputs for systems that consume programs in several languages and found bugs in widely used compilers and interpreters. These results are robust in the sense that the bugs found were confirmed by maintainers and fixed, which is a stronger form of validation than a benchmark pass rate. The mechanism is also well understood: fuzzing's bottleneck is the generation of structurally valid inputs that reach deep program states, and a model trained on a language's corpus is a good generator of valid inputs in that language (Wang *et al.*, 2024).

Limits of Generated Tests

The positive record in testing needs two qualifications. First, coverage is a weak proxy for fault detection, and the studies that report coverage rarely report mutation scores or real faults found by the generated suites; Schäfer *et al.* (2024) are unusual in examining assertion quality directly. Second, generated tests inherit any defect in the model's understanding of the code's intent, so a model that misreads a specification will generate tests that enshrine the misreading. The oracle problem identified by Wang *et al.* (2024) is not solved by scale. The sound uses of generated tests are therefore those in which a human or an independent artefact supplies the oracle: regression-test capture of behaviour already reviewed, bug reproduction from reports in which the expected behaviour is stated (Kang *et al.*, 2023), and fuzzing, in which the oracle is a crash or sanitizer violation rather than a generated assertion (Xia *et al.*, 2024). At the acceptance level, Eyetsemitan *et al.* (2023) describe a structured user acceptance testing framework for small organisations deploying new technology and show that acceptance failures are dominated by mismatches between what was built and what the user intended; generated unit tests, however numerous, cannot detect that class of failure because they test the code against itself.

Review and Security Assurance

This section examines the gate between a change and the main branch, and the security of what passes through it. It covers four topics that the literature usually treats separately, but that belong together in practice. Automated code review asks whether a model can direct or replace reviewer attention. Learned vulnerability detection asks whether a model can find security weaknesses in existing code, and is the topic on which the evidence base is most severely compromised by defective datasets (Croft *et al.*, 2023). The security of generated code asks whether assistants introduce weaknesses, and whether their users notice, and has been examined with unusual rigour, including a randomised controlled experiment (Perry *et al.*, 2023). Supply-chain risk and package hallucination describe a new attack vector that assistants create and that conventional review was not designed to catch (Spracklen *et al.*, 2025).

Automated Code Review

Code review is the primary human gate between a change and the main branch, and it is also one of the costliest activities in mature teams. Tufano *et al.* (2022) showed that pre-training a transformer on code and then fine-tuning on review data improved the model's ability to propose the revised code a reviewer would request, and Li, Z. *et al.* (2022) released CodeReviewer, pre-trained on a large corpus of review activity across nine languages, which improved on prior work for three tasks: predicting whether a change needs a comment, generating the comment, and generating the revised code. Both studies report modest absolute performance, with exact-match rates for generated revisions in the low tens of percent, and both note that the most useful outputs are the shortest, such as flagging a changed file as needing attention. The practical implication is that review models are better deployed as triage aids that prioritise reviewer attention than as substitutes for review (Hou *et al.*, 2024). The governance of the review process matters as much as the quality of the model's comments. Badmus *et al.* (2022) set out a framework for Git-based version control and code review governance in Salesforce development teams, specifying protected branches, mandatory reviewer assignment, and traceability from change to requirement; those are the rules through which a model's suggestions should be routed, and a team that lacks them will not be rescued by a review model.

Learned Vulnerability Detection

Vulnerability detection has attracted a large body of deep-learning research and is the area in which the evidence base is most compromised. Early systems such as VulDeePecker (Li *et al.*, 2018) and the Draper representation-learning approach (Russell *et al.*, 2018) reported high accuracy on datasets of labelled functions. Zhou *et al.* (2019) introduced Devign, a graph neural network over program

representations, with a dataset drawn from real C projects, and Fu and Tantithamthavorn (2022) extended line-level localisation with LineVul. Reported F1 scores above 60 percent on these datasets created an impression that learned detectors were close to practical.

Three replication studies dismantled that impression. Chakraborty *et al.* (2022) evaluated the leading models on a more realistic dataset with a natural ratio of vulnerable to non-vulnerable functions and found their performance fell by more than half, with the models relying on spurious features such as variable names rather than vulnerability semantics. Steenhoek *et al.* (2023) examined the variability of deep-learning vulnerability detectors across training runs and found that the models disagreed on a large fraction of individual predictions even when their aggregate scores were similar, which is incompatible with dependable use. Croft *et al.* (2023) audited the datasets themselves and found that between 20 and 71 percent of labels were inaccurate across widely used collections, with duplication rates of up to 20 percent and uncertain labels in a further large fraction. Ding *et al.* (2025) then constructed PrimeVul with rigorous labelling and de-duplication and found that a code model which scored an F1 of 68 percent on the BigVul dataset scored 3 percent on PrimeVul, and that no model evaluated approached a usable detection rate at an acceptable false-positive rate. Table 4 sets these results side by side. The conclusion is not that learned detection is impossible but that the field's reported progress to 2023 was largely an artefact of its data.

The same data-quality caution applies to adjacent learned detectors, which share the training-data dependency even where their reported accuracy is high. Idika *et al.* (2021) report deep-learning malware classification for cloud-native microservices in edge architectures, with accuracy bounded by the representativeness of the malware corpus used for training. Ajagbe *et al.* (2025) evaluate feature selection for machine-learning intrusion detection and show that the choice of features changes detection rates by more than the choice of learner, which is the same lesson Chakraborty *et al.* (2022) drew about spurious features in vulnerability detection. Adebayo *et al.* (2023) systematically review machine-learning anomaly detection for industrial control systems and find that most studies evaluate on simulated or testbed traffic rather than on production systems. Amadi *et al.* (2025b) catalogue the cyber-physical attack surface of legacy SCADA networks and document the gap between vulnerabilities that are known and vulnerabilities that any deployed detector can observe. Detection is in any case only the first step: Adegbite *et al.* (2024a) describe a vulnerability governance architecture for power and utilities corporations that runs from exposure mapping through prioritisation to remediation verification, and it is at the remediation-verification stage that a detector with uncertain precision imposes its real cost, because every false positive consumes verification effort.

Security of Generated Code

The security of code written by assistants has been examined with more rigour than most topics in this review, and the findings are consistent. Pearce *et al.* (2022) prompted Copilot in 89 scenarios drawn from the MITRE top 25 weakness categories, generated 1,689 programs and found that approximately 40 percent were vulnerable. Asare *et al.* (2023) presented Copilot with the code preceding real historical vulnerabilities and found that it reproduced the original vulnerability in roughly a third of cases and the fix in roughly a quarter, which indicates that the model is neither systematically better nor systematically worse than the human authors whose code it learned from. Khoury *et al.* (2023) found that ChatGPT produced insecure code in most of the 21 programs they requested and corrected the weaknesses only when explicitly asked, confirming that security must be requested rather than assumed.

The decisive study is the controlled experiment by Perry *et al.* (2023), who randomised 47 participants to complete security-relevant programming tasks with or without an AI assistant. Participants with the assistant wrote significantly less secure code on most tasks and were significantly more likely to believe their code was secure. Sandoval *et al.* (2023) reached a more reassuring conclusion in a user study of C programming, finding that assistance did not increase severe security bugs, but their tasks were narrower, and their participants were students working in a single language. The

Table 4: Reported performance of learned vulnerability detectors before and after the data-quality replications.

Study	Evaluation setting	Headline result
Zhou *et al.* (2019)	Design a graph neural network on project-level C dataset	F1 above competing baselines on the authors' dataset; established the benchmark later shown to be noisy
Fu and Tantithamthavorn (2022)	LineVul transformer on BigVul, function and line level	Function-level F1 of 91 percent on BigVul; line-level localisation improved over prior work
Chakraborty *et al.* (2022)	Re-evaluation on realistic class balance and de-duplicated data	Performance fell by more than half; models keyed on spurious lexical features
Steenhoek *et al.* (2023)	Variability across training runs and model families	Models disagreed on a large share of individual predictions despite similar aggregate scores
Croft *et al.* (2023)	Audit of label accuracy in four widely used datasets	20 to 71 percent inaccurate labels; up to 20 percent duplicates
Ding *et al.* (2025)	PrimeVul with rigorous labelling and leakage control	A 7B code model fell from F1 of 68 percent (BigVul) to 3 percent (PrimeVul)

weight of evidence supports two claims: generated code contains security weaknesses at rates comparable to or higher than human-written code in the same context, and assistance degrades the author's ability to recognise those weaknesses. The second claim is the more important, because it defeats the assumption that human review will catch what the model introduces. The applied security-engineering literature anticipated the need for structural rather than discretionary controls. Okoruwa *et al.* (2020) propose privacy-centric security engineering for digital and cloud systems in which protection is designed into the architecture rather than reviewed into the code afterwards, an approach that does not depend on the reviewer's confidence being well calibrated. Odejobi *et al.* (2025) extend this to AI-augmented secure software engineering, positioning deep-learning threat detection as one layer in a pipeline that still terminates in a human decision, and noting that the detection layer must be evaluated on code the detector did not see during training, which returns the argument to the data-quality findings of the discussion of learned vulnerability detection. The same scoping applies here. Pearce *et al.* (2022) and Asare *et al.* (2023) studied Copilot in 2021 and 2022 on C and Python scenarios; Perry *et al.* (2023) used an OpenAI Codex model with 47 participants on five tasks in Python, JavaScript and C; Khoury *et al.* (2023) studied ChatGPT in early 2023 on 21 programs in five languages. The claim that assistance degrades security is therefore well supported for those models, languages and periods, and the review treats its extension to later models as a provisional interpretation that each new model generation should be required to re-test.

Supply Chain and Package Hallucination

A security risk specific to generative tools emerged after the early security studies. Spracklen *et al.* (2025) analysed 576,000 code samples generated by sixteen models across Python and JavaScript and found that 19.7 percent of the packages recommended did not exist, amounting to more than 205,000 unique hallucinated package names, with open-source models hallucinating at roughly four times the rate of commercial ones. A hallucinated package name is an attack surface: an adversary who registers the name on a public registry with malicious content will have it installed by every developer who follows the model's suggestion. This extends the supply-chain attack taxonomy of Ladisa *et al.* (2023), which already documented typosquatting and dependency confusion as practical attack vectors, with a new vector in which the model, rather than the developer, supplies the mistaken name. The risk is compounded by indirect prompt injection, demonstrated by Greshake *et al.* (2023), in which content retrieved by a model from a web page or a repository can redirect its behaviour, so that an assistant reading a poisoned README may recommend a poisoned dependency. The practical controls are conventional, namely allow-listed registries, lockfiles and dependency scanning, but they must be made mandatory because the human judgement that previously caught an unfamiliar package name is now the thing being bypassed (Spracklen *et al.*, 2025). Where assistants are given tools and autonomy, the attack surface widens further. Nwakamma *et al.* (2024b) catalogue prompt injection, tool poisoning, memory manipulation, and excessive agency as the principal threats to agentic enterprise workflows, each of which turns the model into a confused deputy that acts on an attacker's instructions with the developer's privileges. Adebayo *et al.* (2022) add that models deployed inside operational technology are themselves targets for adversarial manipulation of their inputs, so the threat model must cover the assistant as an attack target and not only the code it writes.

Identity and Configuration Controls for Assistants as Principals

Once an assistant can act, it is a principal in the organisation's identity system and should be governed as one; the identity and configuration literature cited here predates agentic assistants, and its relevance is that the controls it describes are the ones an assistant must be placed under. Ogbole *et al.* (2021) set out an identity-centric zero trust architecture for enterprise security governance in which every access is authenticated and authorised on its own merits regardless of network location; applied to assistants, this means that a model's ability to read a repository, open a pull request, or deploy a change is granted narrowly and logged. Ahmed *et al.* (2021) apply zero trust to operational technology in North American critical infrastructure and show that the architecture's value lies in containing the blast radius of a compromised principal, which is precisely the risk an injected or poisoned assistant presents. Mbonu *et al.* (2020a) review identity and access management integration across hybrid and multi-cloud environments, and Mbonu *et al.* (2022b) examine access governance optimisation in large AWS estates; both find that over-privileged service identities are the most common and most dangerous finding in cloud audits, and an assistant granted a human developer's credentials is the newest instance of that pattern. Aliliele *et al.* (2023) propose continuous monitoring for cloud misconfiguration as an enterprise risk control, and the same monitoring is what catches an assistant-generated configuration change that a reviewer waved through. Upreti and Oteri (2025) set out a security design model for resilient network architectures in large-scale cloud platforms in which segmentation and least privilege are designed into the network rather than applied as policy afterwards, which is the architectural counterpart to treating the assistant as a narrowly privileged principal.

Operations and Maintenance

Operations was an early and relatively mature adopter of machine learning under the AIOps label, and maintenance has been the subject of learned defect prediction for two decades, so this phase offers a longer record against which to judge the current wave. The section reviews three topics. Incident management and root-cause analysis are where large language models have entered operations, principally through work at a single large cloud provider (Ahmed *et al.*, 2023; Chen *et al.*, 2024). Automated program repair has a decade of pre-LLM history and a sharp recent discontinuity in reported capability that must be read against benchmark contamination (Xia *et al.*, 2023). Defect prediction and maintenance analytics are included because they illustrate how an earlier generation of optimistic machine-learning

results in software quality was undermined by evaluation defects, a pattern this review finds repeating (Hall *et al.*, 2012).

Incident Management and Root-Cause Analysis

Operations was an early adopter of machine learning under the label of AIOps, with Dang *et al.* (2019) describing production deployments at Microsoft for anomaly detection and incident triage, and Notaro *et al.* (2021) surveying the methods applied to failure management. LLMs entered this space through incident management. Ahmed *et al.* (2023) fine-tuned models on more than 40,000 historical incidents and evaluated the recommended root causes and mitigations with practising on-call engineers, who rated a majority as useful, though the exact-match accuracy against the recorded root cause was low. Chen *et al.* (2024) built an in-context approach that retrieves similar historical incidents to ground the model's analysis and reported improved root-cause accuracy together with a reduction in the hallucinated causes that plagued the fine-tuned baseline. Both studies are internal to a single large cloud provider and use that provider's incident data; the results have not been replicated at organisations with smaller incident histories, where the retrieval base that grounds the model would be thinner (Notaro *et al.*, 2021). The security-operations literature describes the surrounding architecture. Dosunmu and Ogundele (2022) set out frameworks for integrating threat intelligence into enterprise decision-making and show that intelligence is only useful when it reaches the decision at the moment the decision is made, which is the role an LLM-based incident assistant is meant to play. Dosunmu and Ogundele (2025) then examine security orchestration and automation models for accelerating incident detection and response, and find that automation shortens response time most where the playbook is well defined and least where judgement is required, which bounds where a model should be allowed to act. Adegbite *et al.* (2024b) propose an AI-driven security operations centre architecture for the utility sector that combines threat intelligence, behavioural analytics and automated response, and Amadi *et al.* (2025a) frame AI-based security posture management as a layer for continuous adherence to the NIST Cybersecurity Framework; in both designs the model proposes and a human disposes, the same division of labour that Microsoft's on-call engineers found acceptable. The IT service management literature supplies the process layer. Olagunju *et al.* (2023) propose a governance model that integrates change, incident and problem management into one performance-driven framework, and its analytical dimension, in which recurring incidents are traced to the changes that caused them, is exactly the feedback loop that provenance for generated changes would make possible. Oteri *et al.* (2025c) describe cloud-based IT service management architecture and orchestration on the ServiceNow platform and show that automated incident workflows depend on a maintained configuration management database, the artefact into which generated changes must be recorded. Olagunju *et al.* (2025) propose an operational reliability and service assurance framework for enterprise systems serving large user populations that places AI-driven anomaly detection and root-cause analysis inside a policy-governed control plane rather than alongside it, which is the deployment pattern the Microsoft studies describe informally.

Automated Program Repair

Automated program repair (APR) predates LLMs by a decade. GenProg (Le Goues *et al.*, 2012) demonstrated genetic-programming repair on real bugs, and Monperrus (2018) catalogued the field's growth. The weakness of classical APR was overfitting: patches that passed the test suite while being semantically wrong. Xia *et al.* (2023) evaluated nine pre-trained models directly on the Defects4J, QuixBugs and ManyBugs benchmarks and found that the models fixed substantially more bugs than previous template-based and learning-based APR tools, with the largest model fixing the most and with an infilling formulation, in which the model completes a masked buggy region given surrounding context, outperforming full-function generation. The result is robust in that it was obtained with a fixed protocol across tools and benchmarks, but it inherits the contamination problem: Defects4J bugs and their fixes are public and plausibly present in training data (Sallou *et al.*, 2024). Repository-level repair, measured on SWE-bench, remained very weak at the time of that benchmark's publication (Jimenez *et al.*, 2024), and the gap between benchmark repair and repository repair mirrors the gap observed for code generation in the discussion of implementation.

Defect Prediction and Maintenance Analytics

The older tradition of defect prediction from static code and process metrics is relevant because it illustrates what happened the last time machine learning was applied to a software quality problem without adequate attention to evaluation. Menzies *et al.* (2007) showed that simple learners on static code attributes could predict defective modules with useful recall. Hall *et al.* (2012) reviewed 208 fault-prediction studies and found that reported performance depended heavily on the dataset, the modelling technique and the evaluation design, with many studies reporting results that could not be compared or reproduced. Ray *et al.* (2016) later showed that buggy code is measurably less natural, in the language-model sense, than correct code, which both motivates the use of language models for maintenance and explains why they succeed: the signal is real but shallow. The lesson for the current wave is that the field has already lived through a cycle of optimistic results undermined by evaluation defects, and the replications reviewed in the discussion of learned vulnerability detection show the cycle repeating (Hall *et al.*, 2012).

Cloud Operations and Platform Change

Machine learning entered cloud operations before language models did, and its record there is instructive for one reason: the dependency of every learned operational model on the quality and coverage of its telemetry is the same dependency that governs the incident assistants of the discussion of incident management and root-cause analysis and the field-telemetry stage of the assurance loop. Ahmed *et al.* (2020) built a predictive model for cloud resource scaling and found

that the gains came from anticipating load a few minutes ahead rather than from any sophistication in the learner, a reminder that the value of prediction lies in the action it enables. Mayo *et al.* (2021) designed an AI-predictive maintenance model for e-commerce systems using cloud analytics and showed that the model's usefulness depended on the quality of the telemetry it was fed, which is the same dependency that the incident-management studies of the discussion of incident management and root-cause analysis exhibit. Platform migration is the setting in which assistants are most often introduced, and it is the setting in which their lack of organisational context is most costly. Ladapo *et al.* (2022b) distil best practices for enterprise cloud migration and identify dependency mapping and security re-baselining as the steps most often skipped; Ladapo *et al.* (2025) report lessons from the migration of a South African retail bank's applications and information systems to cloud infrastructure, where undocumented dependencies caused the largest delays. A model asked to generate migration code cannot discover those dependencies, and the provenance record proposed in the proposed assurance loop is the only way to trace a migration defect back to the generated change that caused it (Sallou *et al.*, 2024). Olagunju *et al.* (2022) apply the same logic to predictive capacity planning across multi-programme IT portfolios and find that forecast accuracy matters less than the lead time the forecast gives to a decision; Upreti and Oteri (2023) develop an analytical model for cloud rightsizing from long-term usage forecasts and show that over-provisioning, not under-provisioning, is the dominant cost in practice. The telecommunications core offers the most demanding case. Oteri *et al.* (2025a) propose an operational reliability engineering framework for sustained high availability in 5G core networks serving millions of subscribers, and Oteri *et al.* (2025b) set out conceptual and architectural models for cloud-native 5G packet core orchestration on Kubernetes and OpenStack; in both, observability and automated remediation are designed around the failure modes of cloud-native software, which is the environment into which generated infrastructure and application code is now deployed. Adeniji *et al.* (2025), reviewing hybrid AI-based control for harmonic distortion in urban distribution networks, state the same requirement for learned controllers that the discussion of learned vulnerability detection found unmet for learned vulnerability detectors: behaviour outside the training data must be bounded before the model is trusted.

Cross-Cutting Risks and Governance

The risks examined in this section are not confined to a single phase. Hallucination affects generated requirements, code, tests, dependencies and incident analyses alike; memorisation of licensed code is a property of the model rather than of any workflow; and the regulatory instruments now in force in the European Union and the United States apply to the organisation's use of AI as a whole. A final cross-cutting concern is that the tooling which embeds a model is itself software with distinctive engineering and security problems, and an organisation that adopts assistants takes on the task of assuring that tooling as well as the code it helps produce (Amershi *et al.*, 2019; Greshake *et al.*, 2023). The section treats each in turn and shows that every one of them points to the same practical requirement, recorded provenance for every model contribution, which the assurance loop of the proposed assurance loop places at its centre (Ji *et al.*, 2023).

Hallucination, Confidence and Verification Cost

Hallucination, the production of fluent output that is wrong, is the generic failure mode of language models (Ji *et al.*, 2023), and its software-engineering form is code that compiles, reads well, and is incorrect, insecure, or dependent on something that does not exist. The reviewed studies show the phenomenon at every phase: assertions that encode wrong behaviour (Schäfer *et al.*, 2024), vulnerable idioms reproduced from training data (Asare *et al.*, 2023), package names that do not exist (Spracklen *et al.*, 2025) and incident root causes that were never there (Chen *et al.*, 2024). The common feature is that the output does not carry a usable signal of its own reliability. Mozannar *et al.* (2024) quantified the verification cost this imposes on programmers, and Perry *et al.* (2023) showed that the cost is often not paid, because assistance raises confidence while lowering correctness. Calibration, the property that a model's expressed confidence tracks its actual accuracy, is therefore a first-order requirement for any deployment in which a person decides whether to check the output, and it is almost never reported in the software-engineering literature (Wang *et al.*, 2024). One response to the absence of a reliability signal is to use a second model as a judge of the first. Nwakamma *et al.* (2024a) review LLM-as-a-judge methods for automated model governance and release gating and document the position, verbosity, and self-preference biases that limit them; a judge model can triage, but it cannot replace executable or human verification for the same reason the first model could not.

Memorisation and Intellectual Property

Models trained on public code can reproduce it. Carlini *et al.* (2021) demonstrated that training data could be extracted verbatim from large language models, and Yu *et al.* (2023) constructed CodeIPPrompt to measure how readily code models emit licensed code when prompted, finding that all the models they tested did so at non-trivial rates and that the risk was concentrated in prompts resembling the openings of well-known files. The practical consequences for an organisation are twofold: generated code may carry licence obligations the organisation has not reviewed, and the organisation's own proprietary code, if used to fine-tune or prompt a hosted model, may be exposed. Neither consequence is addressed by the model's own behaviour. They are addressed by provenance recording, which Fig. 2 places at the centre of the assurance loop, so that every accepted suggestion can be traced to the prompt, model version and reviewer decision that produced it (Yu *et al.*, 2023). The legal literature frames the exposure from the organisation's side. Annan (2024) examines the tension between algorithmic accountability, which demands that a model's behaviour be explicable, and trade secret protection, which resists disclosure of how it was built, and concludes that organisations deploying models will have to document use without disclosing internals. Ominyi *et al.* (2025) review

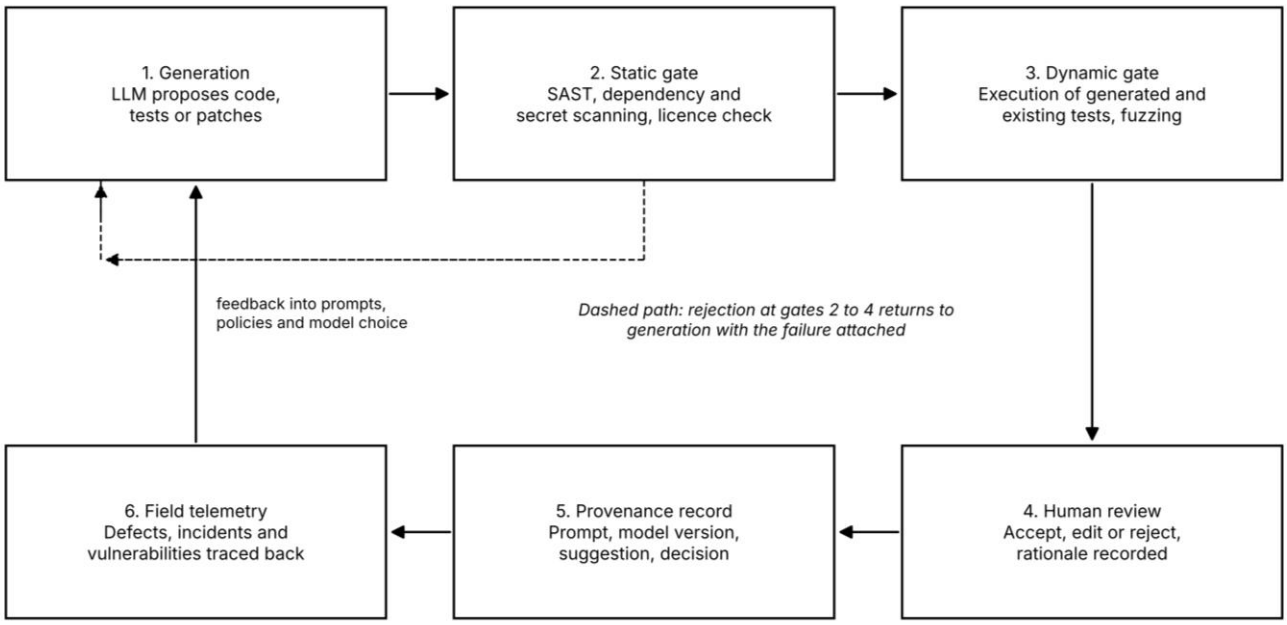


Fig. 2: The proposed assurance loop for AI-generated contributions. Each contribution passes static, dynamic and human gates; provenance is recorded for every decision; field telemetry closes the loop by attributing defects and incidents back to the generation context.

the legal privilege and confidentiality problems that arise when enterprise material is passed to hosted language models, including the risk that privileged content is waived by being placed in a prompt, a risk that applies directly to source code and incident data.

Regulatory and Standards Context

Two governance instruments shape how organisations in major markets will be expected to deploy these tools. The European Union's Artificial Intelligence Act, Regulation (EU) 2024/1689, entered into force in August 2024 and imposes transparency, documentation and risk-management obligations on providers and deployers of general-purpose AI systems, with the obligations on general-purpose models applying from August 2025. In the United States, the NIST Artificial Intelligence Risk Management Framework 1.0 (NIST, 2023) sets out a voluntary structure of govern, map, measure and manage functions, and the NIST Secure Software Development Framework, SP 800-218 (NIST, 2022), already requires organisations supplying software to the federal government to attest to secure development practices that apply equally to generated and hand-written code. ISO/IEC 42001:2023 provides a certifiable management system standard for AI. None of these instruments is specific to software engineering, and none resolves the technical questions this review has identified, but together they establish that an organisation cannot treat AI-generated code as outside its existing secure-development obligations, which makes the assurance loop of the proposed assurance loop a compliance matter as well as an engineering one (NIST, 2022). The readiness literature fills in how these obligations are met in practice. Ominyi and Anichukwueze (2024) conceptualise responsible AI governance structures for automated decision systems in regulated industries and identify an inventory of AI uses as the prerequisite for every other control. Ominyi *et al.* (2024) review organisational readiness for the EU Artificial Intelligence Act across multinational enterprises and find documentation and inventory to be the most common gaps, which for software organisations means that assistants adopted team by team without central record are already a compliance exposure.

Audit and Continuous Assurance of AI Use

Audit is the function that turns governance obligations into evidence, and the audit literature has begun to adapt to AI in two directions at once: auditing the organisation's use of AI, and using AI to audit. Elebe and Anunagba (2025) set out responsible AI in IT auditing through evidence validation, human oversight, and regulatory compliance, and their central requirement, that every AI-produced finding be traceable to validated evidence, is the same requirement the assurance loop of the proposed assurance loop places on generated code. Aliliele *et al.* (2025) review AI-assisted continuous auditing systems for technology risk and cybersecurity oversight and find that continuous auditing is only as good as the completeness of the logs it reads, which is why provenance must be captured at generation time and not reconstructed afterwards. Mbonu *et al.* (2022a) describe AI-enabled IT general controls and SOX audit automation and show that change-management controls, which govern who may alter production code, are the controls most affected when an assistant can propose changes. Uchechi *et al.* (2022) review real-time compliance pipelines for automated risk monitoring and enterprise data governance, the operational form that continuous assurance of AI use will take (NIST, 2023).

Engineering the Systems That Contain Models

A final cross-cutting concern is that software which embeds a model is itself software with distinctive engineering

problems. Sculley *et al.* (2015) identified the hidden technical debt of machine-learning systems, including entanglement, data dependencies and feedback loops, and Amershi *et al.* (2019) documented from a case study at Microsoft how the machine-learning workflow differs from conventional development in data management, model customisation and

Table 5: Appraisal of 24 primary studies against the four criteria stated in the appraisal criteria section.

Study	Phase	Evaluation data outside training distribution	Ground truth	Outcome class	Independent of tool vendor
Chen *et al.* (2021)	Implementation	Yes at publication (hand-written problems); contaminated for later models	Executable tests	Proxy (pass@k)	No (OpenAI)
Li, Y. *et al.* (2022)	Implementation	Yes (contests after training cut-off)	Executable judge	Proxy (contest rank)	No (DeepMind)
Nguyen and Nadi (2022)	Implementation	No (public LeetCode problems)	Executable tests	Proxy (correctness)	Yes
Vaithilingam *et al.* (2022)	Implementation	Not applicable (user study)	Task completion	Proxy (time, preference)	Yes
Dakhel *et al.* (2023)	Implementation	No (public algorithmic problems)	Executable tests	Proxy (correctness)	Yes
Barke *et al.* (2023)	Implementation	Not applicable (observation)	Coded behaviour	Mechanism	Yes
Liu *et al.* (2023)	Implementation	No (HumanEval retained)	Augmented executable tests	Proxy (pass@k)	Yes
Liang *et al.* (2024)	Implementation	Not applicable (survey)	Self-report	Perception	Yes
Mozannar *et al.* (2024)	Implementation	Not applicable (instrumented sessions)	Coded behaviour	Proxy (time allocation)	No (Microsoft)
Jimenez *et al.* (2024)	Implementation	Partly (public issues; contamination acknowledged)	Executable tests	Repository task	Yes
Ziegler *et al.* (2024)	Implementation	Not applicable (telemetry)	Self-report	Perception	No (GitHub)
Lemieux *et al.* (2023)	Testing	No (public Python modules)	Execution	Proxy (coverage)	No (Microsoft)
Schäfer *et al.* (2024)	Testing	No (public npm packages)	Execution	Proxy (coverage, pass rate)	No (GitHub)
Deng *et al.* (2023)	Testing	Yes (new inputs)	Confirmed bugs	Delivered (real faults)	Yes
Xia *et al.* (2024)	Testing	Yes (new inputs)	Confirmed bugs	Delivered (real faults)	Yes
Pearce *et al.* (2022)	Security	Yes (new scenarios)	CodeQL and manual review	Delivered (vulnerability rate)	Yes
Asare *et al.* (2023)	Security	No (historical CVEs)	Real CVE fixes	Delivered (vulnerability reproduction)	Yes
Perry *et al.* (2023)	Security	Not applicable (randomised experiment)	Manual security grading	Delivered (insecure code)	Yes
Chakraborty *et al.* (2022)	Security	Yes (realistic re-split)	Re-validated labels	Proxy (F1)	Yes
Croft *et al.* (2023)	Security	Not applicable (dataset audit)	Manual label validation	Label accuracy	Yes
Ding *et al.* (2025)	Security	Yes (leakage-controlled)	Validated labels	Proxy (F1, false-positive rate)	Yes
Spracklen *et al.* (2025)	Security	Yes (fresh prompts)	Registry existence check	Delivered (hallucinated packages)	Yes
Ahmed *et al.* (2023)	Operations	Unclear (internal incidents)	Engineer rating	Perception	No (Microsoft)
Xia *et al.* (2023)	Operations	No (public Defects4J)	Executable tests	Proxy (bugs fixed)	Yes

Table 6: Totals of the Table 5 classifications by appraisal criterion.

Criterion	Count	Studies
Contamination criterion not applicable	7	Vaithilingam *et al.* (2022); Barke *et al.* (2023); Liang *et al.* (2024); Mozannar *et al.* (2024); Ziegler *et al.* (2024); Perry *et al.* (2023); Croft *et al.* (2023)
Evaluation data outside training distribution: yes	8	Chen *et al.* (2021); Li, Y. *et al.* (2022); Deng *et al.* (2023); Xia *et al.* (2024); Pearce *et al.* (2022); Chakraborty *et al.* (2022); Ding *et al.* (2025); Spracklen *et al.* (2025)
Evaluation data outside training distribution: partly	1	Jimenez *et al.* (2024)
Evaluation data outside training distribution: unclear	1	Ahmed *et al.* (2023)
Evaluation data outside training distribution: no	7	Nguyen and Nadi (2022); Dakhel *et al.* (2023); Liu *et al.* (2023); Lemieux *et al.* (2023); Schäfer *et al.* (2024); Asare *et al.* (2023); Xia *et al.* (2023)
Delivered outcome measured	6	Deng *et al.* (2023); Xia *et al.* (2024); Pearce *et al.* (2022); Asare *et al.* (2023); Perry *et al.* (2023); Spracklen *et al.* (2025)
Vendor-affiliated authorship	7	Chen *et al.* (2021); Li, Y. *et al.* (2022); Mozannar *et al.* (2024); Ziegler *et al.* (2024); Lemieux *et al.* (2023); Schäfer *et al.* (2024); Ahmed *et al.* (2023)

the difficulty of modular reasoning. Greshake *et al.* (2023) added a security dimension by showing that applications which pass retrieved content to a model can be hijacked by instructions embedded in that content. An organisation that adopts AI assistants in its development process is therefore taking on two engineering problems at once: assuring the code the model helps produce, and assuring the model-integrated tooling itself. The second problem is less visible and less studied, and the review identifies it as an area in which the software-engineering community has more to contribute than it has so far (Amershi *et al.*, 2019).

Appraisal of the Primary Studies

Table 5 applies the four appraisal criteria described in the methods to the 24 primary empirical studies on which the phase-by-phase assessments most depend; the surveys, frameworks and standards cited elsewhere in the review are not audited because the criteria concern empirical evaluation design. The pattern is sharper than the narrative alone conveys. Of the seventeen studies to which the contamination criterion applies, eight used evaluation data plausibly outside the model's training distribution, one did so only partly, one could not be determined from the published report, and seven did not; two of the eight are the replications that exposed the problem in the others. The seven studies for which the criterion is not applicable are user studies, surveys, telemetry analyses, a randomised experiment, and a dataset audit, none of which evaluates a model on held-out data. Only six of the 24 measured a delivered outcome, meaning a real fault found, a vulnerability shipped, or a package that does not exist, rather than a proxy such as coverage, pass rate, or self-reported productivity, and every one of those six sits in the testing or security phase, none in implementation. Seven of the 24 were authored or co-authored by the vendor of the tool or model evaluated, and those seven are concentrated in the capability-benchmark, productivity, and test-generation literature, which is where the strongest commercial claims are made. The table does not impeach any individual study, several of which are careful about their own limitations, but it shows that the field's most-cited positive results and its most rigorous evaluations are largely different studies (Sallou *et al.*, 2024; Croft *et al.*, 2023).

Table 6 totals the classifications in Table 5 by criterion so that the counts in the preceding paragraph can be checked directly against the table.

DISCUSSION

This section draws the phase-by-phase findings together. It first separates the results that have been replicated by independent groups using different methods from the claims that rest on a single study, a vendor study or a benchmark now known to be contaminated, because the two categories warrant different levels of confidence and the field has not always kept them apart (Sallou *et al.*, 2024). It then proposes an assurance loop for AI-generated contributions that follows directly from the replicated findings, and that treats every model output as an untrusted contribution subject to static, dynamic and human gates with recorded provenance. The replicated findings are then used to specify the assurance loop in enough detail that an organisation can test it against its own telemetry (Chakraborty *et al.*, 2022; Croft *et al.*, 2023).

Replicated Findings and Provisional Claims

Reading across the five phases, the evidence divides into findings that have been replicated by independent groups with different methods and claims that rest on a single study, a vendor study, or a contaminated benchmark. Four findings meet the replication standard, each stated with the models, languages, and period to which the underlying studies apply. First, AI assistance with Copilot-class assistants between 2021 and 2024 speeds up well-understood implementation tasks in mainstream languages and shifts effort to verification; this is supported by user studies from independent groups (Barke *et al.*, 2023; Mozannar *et al.*, 2024; Liang *et al.*, 2024). Second, code generated by Codex-based assistants and ChatGPT between 2021 and 2023, in C, Python, and JavaScript, carries security weaknesses at rates that are not lower than human-written code and that users under-detect (Pearce *et al.*, 2022; Asare *et al.*, 2023; Perry *et*

al., 2023). Third, learned vulnerability detectors trained on the standard datasets do not generalise, because the datasets are defective (Chakraborty *et al.*, 2022; Croft *et al.*, 2023; Ding *et al.*, 2025). Fourth, LLM-assisted test and fuzz generation finds real bugs in real systems when the oracle is supplied by execution rather than by the model (Deng *et al.*, 2023; Xia *et al.*, 2024; Lemieux *et al.*, 2023).

Three widely repeated claims remain provisional. The claim that assistants reduce delivered defects has no peer-reviewed field evidence. The claim that models can repair real bugs at scale is supported on benchmarks that are plausibly in training data and is contradicted by repository-level evaluation (Xia *et al.*, 2023; Jimenez *et al.*, 2024). The claim that models can perform requirements elicitation is supported only by small expert-rating studies (Ronanki *et al.*, 2023). A fourth claim, that function-level benchmark scores track engineering competence, is now contradicted by three separate lines of evidence and should be retired (Liu *et al.*, 2023; Jimenez *et al.*, 2024; Sallou *et al.*, 2024).

An Assurance Loop for AI-generated Contributions

The replicated findings point to a single design principle: a model's output is an untrusted contribution and must pass the same gates, with the same evidence, as a contribution from an unknown external contributor. Fig. 2 sets out an assurance loop built on that principle. Generation (gate 1) is followed by a static gate (gate 2) comprising static application security testing, dependency resolution against an allow-listed registry, secret scanning and licence checking, which together address the vulnerability patterns documented by Pearce *et al.* (2022), the package hallucinations documented by Spracklen *et al.* (2025) and the memorisation risk documented by Yu *et al.* (2023). A dynamic gate (gate 3) executes existing tests and any generated tests, with fuzzing where harnesses exist, which supplies the execution oracle that the discussion of testing identified as the sound basis for generated tests. Human review (gate 4) is retained but is informed by the evidence from gates 2 and 3 and by the finding of Perry *et al.* (2023) that reviewers' confidence is unreliable, so review guidance should direct attention to the categories in which generated code is known to fail rather than relying on general vigilance.

Two further elements distinguish the loop from a conventional pipeline. A provenance record (gate 5) captures, for every accepted or rejected suggestion, the prompt, the model and version, the suggestion itself and the reviewer's decision and rationale. This record is what makes the licensing obligations of the discussion of memorisation and intellectual property and the documentation obligations of the discussion of regulatory and standards context dischargeable, and it is also the data without which the field's central unanswered question, whether assistance changes delivered quality, cannot be answered. Field telemetry (gate 6) links production defects, incidents and vulnerabilities back to the provenance record so that the organisation learns which kinds of generated contribution fail, and feeds that learning into prompts, policies and model selection. Rejections at any gate return to generation with the failure attached, which converts the gates from filters into a training signal for the prompting practice. The loop is deliberately conservative; its cost is justified by the replicated finding that the alternative, trusting the output and the reviewer's instinct, demonstrably ships insecure code (Perry *et al.*, 2023). Static analysis, dynamic testing, and human review are of course already present in secure development practice, and the NIST Secure Software Development Framework (NIST, 2022) requires all three. Table 7 sets the loop against a conventional DevSecOps pipeline so that the contribution can be stated precisely. Four elements are new. First, the unit of assurance is the model contribution rather than the commit: each suggestion is gated on its own, so that a single pull request assembled from several suggestions carries a provenance record for each. Second, gate selection is risk-based rather than uniform: the gates applied to a contribution are chosen from the risk class of the change, following the risk-based assurance principle of Fadayomi *et al.* (2019). Third, rejection is a training signal: a

Table 7: The proposed assurance loop compared with a conventional DevSecOps pipeline.

Element	Conventional DevSecOps pipeline	Proposed assurance loop
Unit of assurance	The commit or pull request	Each model contribution, individually gated and recorded, then the commit
Static gate	SAST and dependency scanning on the branch	SAST, allow-listed dependency resolution, secret scanning and licence check on the contribution, before merge into the branch
Dynamic gate	Test suite on the branch	Existing tests plus generated tests executed on the contribution; fuzzing where harnesses exist; generated assertions flagged for oracle review
Human review	General review by an assigned reviewer	Review directed by a checklist of the weakness categories in which generated code is known to fail, with evidence from gates 2 and 3 attached
Gate selection	Uniform for every change	Risk-based: gates chosen from the risk class of the change (Table 8)
Rejection	Pull request returned to the author	Returned to generation with the failing evidence attached; escalation to human authorship after repeated failure
Provenance	Commit author and message	Prompt, model and version, suggestion, gate results, reviewer decision and rationale, for every contribution
Field feedback	Incident tickets linked to commits, where practised	Defects, incidents and vulnerabilities attributed to generation context and fed back to prompts, policies and model choice

Table 8: Risk classes, gates applied and acceptance thresholds in the assurance loop.

Risk class	Definition	Gates applied	Acceptance thresholds
Low	Changes confined to documentation, tests, formatting or non-executing configuration	Static gate; human review	No new SAST finding of any severity; no new dependency; reviewer acceptance
Medium	Changes to application logic that do not touch authentication, authorisation, cryptography, input parsing, data persistence or infrastructure	Static gate; dynamic gate; human review	No new SAST finding at medium severity or above; all existing tests pass; generated tests achieve a mutation score at or above the project baseline; reviewer acceptance against the checklist
High	Changes to authentication, authorisation, cryptography, input parsing, data persistence, infrastructure code or build and deployment configuration	Static gate; dynamic gate including fuzzing where a harness exists; review by two reviewers, one with security responsibility	No new SAST finding of any severity; all dependencies resolved from the allow-list; all existing tests pass; fuzzing campaign of a defined duration with no new crash; both reviewers accept

rejection at any gate returns to generation with the failing evidence attached, so that the prompt, the policy, or the model is adjusted before the next attempt. Fourth, field telemetry is joined to provenance so that production defects are attributed to generation context, which no conventional pipeline records because conventional pipelines do not know which lines a model wrote.

The loop is made testable by three operational rules. Risk-based gate selection assigns each contribution to one of three risk classes from the files it touches and the privileges it exercises, and Table 8 states the gates and acceptance thresholds for each class; the thresholds are starting values to be tuned on each organisation's own telemetry rather than universal constants, and the mutation-score threshold in particular follows the finding of Schäfer *et al.* (2024) that coverage alone does not establish that a generated test detects faults. Escalation is governed by two rules: a contribution rejected twice at the same gate is withdrawn from generation and assigned to a human author, and any rejection at the human-review gate for a security weakness triggers a review of the prompt and policy that produced the contribution. Measurement is governed by four indicators that the provenance record makes computable: the rejection rate at each gate by risk class, the proportion of accepted contributions later attributed to a production defect or vulnerability, the median verification time per contribution, and the share of contributions with a complete provenance record, which should be 100 percent. These indicators are what an organisation would report to show whether assistants are improving delivered quality, and their absence from the published literature is the gap identified in the appraisal of the primary studies (Ziegler *et al.*, 2024).

CONCLUSION

Artificial intelligence has become a working part of software development, and the published evidence is now rich enough to say with some precision what it does and does not deliver. Across the lifecycle, assistants accelerate implementation tasks that the programmer already understands, generate tests and fuzz inputs that find real bugs when execution supplies the oracle, and draft requirements, designs, review comments and incident analyses that are useful as starting points for human judgement. They also produce code with security weaknesses at rates no lower than human code, while making their users more confident; they recommend dependencies that do not exist; and they are evaluated on benchmarks and datasets that are, in the case of vulnerability detection especially, too defective to support the conclusions drawn from them.

The appropriate response is neither rejection nor trust but engineering. Every model output should be treated as a contribution from an untrusted source and passed through static, dynamic and human gates with recorded provenance, so that the organisation can discharge its licensing and regulatory obligations and, over time, learn from its own field data which uses of these tools improve delivered quality and which do not. The research community's task is to repair its evidence base: uncontaminated benchmarks, validated labels, reported calibration and longitudinal field studies. Until that work is done, the honest summary of the state of the art is that these tools make certain kinds of software work faster, make certain kinds of error more likely, and have not yet been shown to make software better.

Grant Support Details

The present research did not receive any financial support to conduct the research.

Conflict of Interest

The authors declare that there is no conflict of interest regarding the publication of this manuscript. In addition, the ethical issues, including plagiarism, informed consent, misconduct, data fabrication and/ or falsification, double publication and/or submission, and redundancy, have been completely observed by the authors.

REFERENCES

1) Adams, V. (2023) 'Preserving critical thinking in the era of generative artificial intelligence: evidence-based frameworks for human-centered learning', *International Journal of Social Sciences and Management Research*, 9(11), pp. 321-332. https://doi.org/10.56201/ijssmr.v9.no11.2023.pg321.332
2) Adebayo, A., Adegbite, M.P. and Ahmed, M.O. (2022) 'Adversarial machine learning in critical infrastructure: a conceptual framework for threat modeling AI-enabled OT systems', *World Journal of Innovation and Modern Technology*, 6(1), pp. 184-234. https://doi.org/10.56201/wjimt.v6.no1.2022.pg184.234
3) Adebayo, A., Adegbite, M.P. and Ahmed, M.O. (2023) 'AI-augmented threat detection in industrial control systems: a systematic

review of machine learning approaches for ICS anomaly detection', *International Journal of Engineering and Modern Technology*, 9(3), pp. 287-340. https://doi.org/10.56201/ijcsmt.v9.no3.2023.pg287.340

4) Adegbite, M.P., Adebayo, A. and Ahmed, M.O. (2024a) 'A vulnerability governance architecture for power and utilities corporations: from exposure mapping to remediation verification', *World Journal of Innovation and Modern Technology*, 8(6), pp. 185-246. https://doi.org/10.56201/wjimt.v8.no6.2024.pg185.246
5) Adegbite, M.P., Adebayo, A. and Ahmed, M.O. (2024b) 'An AI-driven security operations architecture for utility sector SOCs: integrating threat intelligence, behavioral analytics, and automated response', *International Journal of Engineering and Modern Technology*, 10(11), pp. 197-256. https://doi.org/10.56201/ijemt.v10.no11.2024.pg197.256
6) Adeniji, I.O., Shittu, H., Shittu, M.A., *et al.* (2025) 'Mitigating harmonic distortion in urban distribution networks using hybrid AI-based control models', *International Journal of Advanced Multidisciplinary Research and Studies*, 5(6), pp. 2055-2068.
7) Ahmed, K.S., Odejobi, O.D. and Oshoba, T.O. (2020) 'Predictive model for cloud resource scaling using machine learning techniques', *Journal of Frontiers in Multidisciplinary Research*, 1(1), pp. 173-183.
8) Ahmed, M.O., Adegbite, M.P. and Adebayo, A. (2021) 'Zero trust architecture for operational technology in North American critical infrastructure: a framework for implementation and resilience optimization', *International Journal of Engineering and Modern Technology*, 7(1), pp. 66-113. https://doi.org/10.56201/ijemt.vol.7.no1.2021.pg66.113
9) Ahmed, T., Ghosh, S., Bansal, C., *et al.* (2023) 'Recommending root-cause and mitigation steps for cloud incidents using large language models'. *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE 2023)*. Melbourne, Australia, pp. 1737-1749. https://doi.org/10.1109/ICSE48619.2023.00149
10) Aileme, J.I., Sule, A. and Obukadata, V.O. (2024) 'From ambiguity to AI-ready requirements: a business analysis framework for structuring business problems for intelligent process automation', *International Journal of Multidisciplinary Research and Growth Evaluation*, 5(6), pp. 2088-2117. https://doi.org/10.54660/IJMRGE.2024.5.6.2088-2117
11) Aileme, J.I., Sule, A. and Obukadata, V.O. (2025) 'Requirements elicitation for AI-enabled process automation: a comparative review of Agile, Waterfall, and Hybrid implementation approaches', *Gulf Journal of Computer Sciences*, 1(4), pp. 116-169.
12) Ailile, C., Mbonu, I.S. and Iwuanyanwu, U. (2023) 'A conceptual framework for continuous cloud misconfiguration monitoring and enterprise risk mitigation strategies', *International Journal of Scientific Research in Computer Science, Engineering and Information Technology*, 9(10), pp. 373-394. https://doi.org/10.32628/CSEIT2361071
13) Ailile, C., Mbonu, I.S., Uzoka, E., *et al.* (2025) 'A review of AI-assisted continuous auditing systems in technology risk and cybersecurity oversight', *Gyansharyam, International Scientific Refereed Research Journal*, 8(4), pp. 210-250. https://doi.org/10.32628/GISRRJ258369
14) Ajagbe, S.A., Akindolani, A., Akande, T.O., Adeyanju, K.I., Aiyeniko, O. and Ajiboye, G.O. (2025) 'Intrusion detection system with feature selection on machine learning algorithm'. *Proceedings of the 6th International Conference and Workshop on Engineering and Technology Research (ICWETR) LAUFET 2025*. pp. 24-39.
15) Amadi, C., Adebayo, A. and Ijagbemi, A.O. (2025a) 'AI-based security posture management as an enabling layer for real-time NIST Cybersecurity Framework adherence: a theory-grounded conceptual framework', *World Journal of Innovation and Modern Technology*, 9(8), pp. 99-147. https://doi.org/10.56201/wjmt.v9.no8.2025.pg99.147
16) Amadi, C., Adebayo, A. and Ijagbemi, A.O. (2025b) 'The cyber-physical attack surface of legacy SCADA networks: a systematic review of vulnerabilities, threats, and detection gaps in critical infrastructure', *International Journal of Computer Science and Mathematical Theory*, 11(8), pp. 135-169. https://doi.org/10.56201/ijcsmt.vol.11.no8.2025.pg135.169
17) Amershi, S., Begel, A., Bird, C., *et al.* (2019) 'Software engineering for machine learning: a case study'. *Proceedings of the 41st IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP 2019)*. Montreal, Canada, pp. 291-300. https://doi.org/10.1109/ICSE-SEIP.2019.00042
18) Annan, A.O. (2024) 'Algorithmic accountability and trade secret protection in artificial intelligence', *Shodhshayaram, International Scientific Refereed Research Journal*, 7(5), pp. 315-347.
19) Arora, C., Grundy, J. and Abdelrazek, M. (2024) 'Advancing requirements engineering through generative AI: assessing the role of LLMs'. In: *Generative AI for effective software development*; Nguyen-Duc, A., Abrahamsson, P. and Khomh, F. Cham: Springer, pp. 129-148. https://doi.org/10.1007/978-3-031-55642-5_6
20) Asare, O., Nagappan, M. and Asokan, N. (2023) 'Is GitHub's Copilot as bad as humans at introducing vulnerabilities in code?', *Empirical Software Engineering*, 28, 129. https://doi.org/10.1007/s10664-023-10380-1
21) Badmus, O., Dosunmu, A.A. and Anunagba, C.O. (2025) 'A governance framework for AI-assisted CRM workflows: Agentforce deployment, risk management, and organizational readiness in enterprise Salesforce environments', *International Journal of Scientific Research in Humanities and Social Sciences*, 2(1), pp. 59-80.
22) Badmus, O., Dosunmu, A.A. and Ozowara, D.E. (2018) 'A systematic review of CI/CD pipeline strategies in Salesforce DevOps: tools, practices, and deployment outcomes', *Iconic Research and Engineering Journals*, 2(6), pp. 319-333.
23) Badmus, O., Dosunmu, A.A., Ozowara, D.E., *et al.* (2020) 'A comparative framework for Salesforce DevOps tooling: evaluating Copado, Flosum, and Salesforce DX across enterprise deployment contexts', *International Journal of Multidisciplinary Research and Growth Evaluation*, 1(5), pp. 1021-1031. https://doi.org/10.54660/IJMRGE.2020.1.5.1021-1031
24) Badmus, O., Jooda, D., Ozowara, D.E., *et al.* (2022) 'A framework for Git-based version control and code review governance in Salesforce development teams', *Shodhshauryam, International Scientific Refereed Research Journal*, 5(2), pp. 473-493.
25) Barke, S., James, M.B. and Polikarpova, N. (2023) 'Grounded Copilot: how programmers interact with code-generating models'. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA1), 78, pp. 85-111. https://doi.org/10.1145/3586030
26) Becker, B.A., Denny, P., Finnie-Ansley, J., *et al.* (2023) 'Programming is hard, or at least it used to be: educational opportunities and challenges of AI code generation'. *Proceedings of the 54th ACM Technical Symposium on Computer Science Education (SIGCSE 2023)*. Toronto, Canada, pp. 500-506. https://doi.org/10.1145/3545945.3569759
27) Borah, S., Aliliele, K.C., Rakshit, S., *et al.* (2022) 'Applications of artificial intelligence in software testing'. In: *Cognitive informatics and soft computing*; Mallick, P.K. *et al.* (ed.). Lecture Notes in Networks and Systems, 375. Singapore: Springer, pp. 727-736. https://doi.org/10.1007/978-981-16-8763-1_60
28) Carlini, N., Tramèr, F., Wallace, E., *et al.* (2021) 'Extracting training data from large language models'. *Proceedings of the 30th USENIX Security Symposium*. pp. 2633-2650.
29) Chakraborty, S., Krishna, R., Ding, Y., *et al.* (2022) 'Deep learning based vulnerability detection: are we there yet?', *IEEE Transactions on Software Engineering*, 48(9), pp. 3280-3296. https://doi.org/10.1109/TSE.2021.3087402
30) Chen, M., Tworek, J., Jun, H., *et al.* (2021) 'Evaluating large language models trained on code', *arXiv preprint*, arXiv:2107.03374 (technical report). https://doi.org/10.48550/arXiv.2107.03374
31) Chen, Y., Xie, H., Ma, M., *et al.* (2024) 'Automatic root cause analysis via large language models for cloud incidents'. *Proceedings of the Nineteenth European Conference on Computer Systems (EuroSys 2024)*. Athens, Greece, pp. 674-688. https://doi.org/10.1145/3627703.3629553
32) Chibwaye, I.P. and Oteri, O. (2023) 'Advances and conceptual models in data integrated ERP migration for large scale government workforce transformation', *International Journal of Advanced Multidisciplinary Research and Studies*, 3(6), pp. 2712-2726.
33) Croft, R., Babar, M.A. and Kholoosi, M.M. (2023) 'Data quality for software vulnerability datasets'. *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE 2023)*. Melbourne, Australia, pp. 121-133. https://doi.org/10.1109/ICSE48619.2023.00022
34) Dakhel, A.M., Majdinasab, V., Nikanjam, A., *et al.* (2023) 'GitHub Copilot AI pair programmer: asset or liability?', *Journal of Systems and Software*, 203, 111734. https://doi.org/10.1016/j.jss.2023.111734
35) Dang, Y., Lin, Q. and Huang, P. (2019) 'AIops: real-world challenges and research innovations'. *Proceedings of the 41st IEEE/ACM*

International Conference on Software Engineering: Companion Proceedings (ICSE-Companion 2019). Montreal, Canada, pp. 4-5.

36) Deng, Y., Xia, C.S., Peng, H., *et al.* (2023) 'Large language models are zero-shot fuzzers: fuzzing deep-learning libraries via large language models'. *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2023)*. Seattle, USA, pp. 423-435. https://doi.org/10.1145/3597926.3598067

37) Ding, Y., Fu, Y., Ibrahim, O., *et al.* (2025) 'Vulnerability detection with code language models: how far are we?'. *Proceedings of the 47th IEEE/ACM International Conference on Software Engineering (ICSE 2025)*. Ottawa, Canada, pp. 469-481.

38) Dosunmu, A.A. and Ogundele, P.O. (2022) 'Threat intelligence integration frameworks supporting proactive enterprise cybersecurity decision making', *Gyanshauryam, International Scientific Refereed Research Journal*, 5(3), pp. 397-416.

39) Dosunmu, A.A. and Ogundele, P.O. (2025) 'Security orchestration and automation models for accelerating incident detection and response', *Computer Science and IT Research Journal*, 6(11), pp. 878-894. https://doi.org/10.51594/csitrj.v6i11.2163

40) Elebe, O. and Anunagba, C.O. (2025) 'Responsible artificial intelligence in IT auditing through evidence validation, human oversight, and regulatory compliance', *International Journal of Technology, Health and Sustainability*, 1(2), pp. 410-428.

41) Essandoh, S., Sakyi, J.K., Ibrahim, A.K., *et al.* (2025) 'Artificial intelligence and the future of work: impacts on employment and job roles', *International Journal of Multidisciplinary Futuristic Development*, 6(1), pp. 31-41. https://doi.org/10.54660/IJMFD.2025.6.1.31-41

42) Eyetsemitan, R.A., Ambali, K.B., Oyeleye, A.O., *et al.* (2023) 'User acceptance testing in small business technology deployment: a structured validation framework for lean operational environments', *International Journal of Multidisciplinary Research and Growth Evaluation*, 4(6), pp. 1512-1531. https://doi.org/10.54660/IJMRGE.2023.4.6.1512-1531

43) Fadayomi, O., Abolaji, T.O., Edivri, J., *et al.* (2019) 'Risk-based cybersecurity assurance and data availability: limitations, advances and future research opportunities', *Iconic Research and Engineering Journals*, 2(12), pp. 602-617. https://doi.org/10.64388/IREV2I12-1713779

44) Fadero, S., Jooda, D., Oyesiji, S.O., *et al.* (2025) 'IT services decoupling after mergers, acquisitions, and ownership transitions: a framework for secure infrastructure separation', *International Journal of Advanced Multidisciplinary Research and Studies*, 5(6), pp. 2471-2485.

45) Fan, A., Gokkaya, B., Harman, M., *et al.* (2023) 'Large language models for software engineering: survey and open problems'. *Proceedings of the 2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*. Melbourne, Australia, pp. 31-53. https://doi.org/10.1109/ICSE-FoSE59343.2023.00008

46) Feng, Z., Guo, D., Tang, D., *et al.* (2020) 'CodeBERT: a pre-trained model for programming and natural languages'. *Findings of the Association for Computational Linguistics: EMNLP 2020*. pp. 1536-1547. https://doi.org/10.18653/v1/2020.findings-emnlp.139

47) Fraser, G. and Arcuri, A. (2011) 'EvoSuite: automatic test suite generation for object-oriented software'. *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE 2011)*. Szeged, Hungary, pp. 416-419. https://doi.org/10.1145/2025113.2025179

48) Fu, M. and Tantithamthavorn, C. (2022) 'LineVul: a transformer-based line-level vulnerability prediction'. *Proceedings of the 19th International Conference on Mining Software Repositories (MSR 2022)*. Pittsburgh, USA, pp. 608-620. https://doi.org/10.1145/3524842.3528452

49) Greshake, K., Abdelnabi, S., Mishra, S., *et al.* (2023) 'Not what you've signed up for: compromising real-world LLM-integrated applications with indirect prompt injection'. *Proceedings of the 16th ACM Workshop on Artificial Intelligence and Security (AISec 2023)*. Copenhagen, Denmark, pp. 79-90. https://doi.org/10.1145/3605764.3623985

50) Hall, T., Beecham, S., Bowes, D., *et al.* (2012) 'A systematic literature review on fault prediction performance in software engineering', *IEEE Transactions on Software Engineering*, 38(6), pp. 1276-1304. https://doi.org/10.1109/TSE.2011.103

51) Hindle, A., Barr, E.T., Su, Z., *et al.* (2012) 'On the naturalness of software'. *Proceedings of the 34th International Conference on Software Engineering (ICSE 2012)*. Zurich, Switzerland, pp. 837-847. https://doi.org/10.1109/ICSE.2012.6227135

52) Hou, X., Zhao, Y., Liu, Y., *et al.* (2024) 'Large language models for software engineering: a systematic literature review', *ACM Transactions on Software Engineering and Methodology*, 33(8), 220. https://doi.org/10.1145/3695988

53) Idika, C.N., Salami, E.O., Ijiga, O.M., *et al.* (2021) 'Deep learning driven malware classification for cloud-native microservices in edge computing architectures', *International Journal of Scientific Research in Computer Science, Engineering and Information Technology*, 7(4), pp. 710-732.

54) Ji, Z., Lee, N., Frieske, R., *et al.* (2023) 'Survey of hallucination in natural language generation', *ACM Computing Surveys*, 55(12), 248. https://doi.org/10.1145/3571730

55) Jimenez, C.E., Yang, J., Wettig, A., *et al.* (2024) 'SWE-bench: can language models resolve real-world GitHub issues?'. *Proceedings of the Twelfth International Conference on Learning Representations (ICLR 2024)*. Vienna, Austria.

56) Kang, S., Yoon, J. and Yoo, S. (2023) 'Large language models are few-shot testers: exploring LLM-based general bug reproduction'. *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE 2023)*. Melbourne, Australia, pp. 2312-2323. https://doi.org/10.1109/ICSE48619.2023.00194

57) Khoury, R., Avila, A.R., Brunelle, J., *et al.* (2023) 'How secure is code generated by ChatGPT?'. *Proceedings of the 2023 IEEE International Conference on Systems, Man, and Cybernetics (SMC 2023)*. Honolulu, USA, pp. 2445-2451.

58) Ladapo, O.O., Dosunmu, A.A., Jooda, D., *et al.* (2022a) 'Human-in-the-loop machine learning: a state of the art', *Journal of Frontiers in Multidisciplinary Research*, 3(1), pp. 656-669. https://doi.org/10.54660/JFMR.2022.3.1.656-669

59) Ladapo, O.O., Dosunmu, A.A., Jooda, D., *et al.* (2025) 'Migration of applications and information systems to cloud computing infrastructure: lessons from a South African retail bank', *International Journal of Multidisciplinary Research and Growth Evaluation*, 6(6), pp. 1361-1375. https://doi.org/10.54660/IJMRGE.2025.6.6.1361-1375

60) Ladapo, O.O., Jooda, D., Dosunmu, A.A., *et al.* (2022b) 'Navigating digital transformation: best practices for cloud migration strategies in the enterprise', *Journal of Frontiers in Multidisciplinary Research*, 3(1), pp. 643-655. https://doi.org/10.54660/JFMR.2022.3.1.643-655

61) Ladapo, O.O., Jooda, D., Dosunmu, A.A., *et al.* (2024) 'Keeping humans in the loop: human-centered automated annotation with generative AI', *International Journal of Multidisciplinary Futuristic Development*, 5(1), pp. 81-95. https://doi.org/10.54660/IJMFD.2024.5.1.81-95

62) Ladisa, P., Plate, H., Martinez, M., *et al.* (2023) 'SoK: taxonomy of attacks on open-source software supply chains'. *Proceedings of the 2023 IEEE Symposium on Security and Privacy (SP 2023)*. San Francisco, USA, pp. 1509-1526. https://doi.org/10.1109/SP46215.2023.10179304

63) Le Goues, C., Nguyen, T., Forrest, S., *et al.* (2012) 'GenProg: a generic method for automatic software repair', *IEEE Transactions on Software Engineering*, 38(1), pp. 54-72. https://doi.org/10.1109/TSE.2011.104

64) Lemieux, C., Inala, J.P., Lahiri, S.K., *et al.* (2023) 'CodaMosa: escaping coverage plateaus in test generation with pre-trained large language models'. *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE 2023)*. Melbourne, Australia, pp. 919-931. https://doi.org/10.1109/ICSE48619.2023.00085

65) Li, Y., Choi, D., Chung, J., *et al.* (2022) 'Competition-level code generation with AlphaCode', *Science*, 378(6624), pp. 1092-1097. https://doi.org/10.1126/science.abq1158

66) Li, Z., Lu, S., Guo, D., *et al.* (2022) 'Automating code review activities by large-scale pre-training'. *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2022)*. Singapore, pp. 1035-1047. https://doi.org/10.1145/3540250.3549081

67) Li, Z., Zou, D., Xu, S., *et al.* (2018) 'VulDeePecker: a deep learning-based system for vulnerability detection'. *Proceedings of the 25th Annual Network and Distributed System Security Symposium (NDSS 2018)*. San Diego, USA. https://doi.org/10.14722/ndss.2018.23158

68) Liang, J.T., Yang, C. and Myers, B.A. (2024) 'A large-scale survey on the usability of AI programming assistants: successes and challenges'. *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering (ICSE 2024)*. Lisbon, Portugal, 52. https://doi.org/10.1145/3597503.3608128

69) Liu, J., Xia, C.S., Wang, Y., *et al.* (2023) 'Is your code generated by ChatGPT really correct? Rigorous evaluation of large language models for code generation'. *Advances in Neural Information Processing Systems 36 (NeurIPS 2023)*. New Orleans, USA.

70) Lukasczyk, S. and Fraser, G. (2022) 'Pynguin: automated unit test generation for Python'. *Proceedings of the 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion 2022)*. Pittsburgh, USA, pp. 168-172. https://doi.org/10.1145/3510454.3516829

71) Mayo, W., Ogbole, J.I., Okoruwa, P.O., *et al.* (2021) 'Designing an AI-predictive maintenance model for e-commerce systems using machine learning and cloud analytics', *International Journal of Scientific Research in Computer Science, Engineering and Information Technology*, 7(5), pp. 416-440.

72) Mbonu, I.S., Aliliele, C., Iwuanyanwu, U., *et al.* (2020a) 'A review of identity and access management integration strategies in hybrid and multi-cloud environments', *International Journal of Multidisciplinary Research and Growth Evaluation*, 1(5), pp. 795-810. https://doi.org/10.54660/IJMRGE.2020.1.5.795-810

73) Mbonu, I.S., Aliliele, C., Iwuanyanwu, U., *et al.* (2021) 'Advances in artificial intelligence techniques for secure software testing and automated regression control mechanisms', *International Journal of Scientific Research in Computer Science, Engineering and Information Technology*, 7(5), pp. 468-496. https://doi.org/10.32628/CSEIT217565

74) Mbonu, I.S., Aliliele, C., Iwuanyanwu, U., *et al.* (2022a) 'A conceptual framework for AI enabled IT general controls and SOX audit automation processes', *Gyansharyam, International Scientific Refereed Research Journal*, 5(5), pp. 384-414. https://doi.org/10.32628/GISRRJ2256239

75) Mbonu, I.S., Iwuanyanwu, U., Aliliele, C., *et al.* (2020b) 'Advances in infrastructure as code governance for secure Terraform based enterprise cloud deployments', *International Journal of Multidisciplinary Research and Growth Evaluation*, 1(5), pp. 811-828. https://doi.org/10.54660/IJMRGE.2020.1.5.811-828

76) Mbonu, I.S., Iwuanyanwu, U., Aliliele, C., *et al.* (2022b) 'Advances in cloud identity and access governance optimization in large scale AWS enterprise environments', *Shodhsharyam, International Scientific Refereed Research Journal*, 5(3), pp. 403-438. https://doi.org/10.32628/SHISRRJ225490

77) Menzies, T., Greenwald, J. and Frank, A. (2007) 'Data mining static code attributes to learn defect predictors', *IEEE Transactions on Software Engineering*, 33(1), pp. 2-13. https://doi.org/10.1109/TSE.2007.256941

78) Monperrus, M. (2018) 'Automatic software repair: a bibliography', *ACM Computing Surveys*, 51(1), 17. https://doi.org/10.1145/3105906

79) Mozannar, H., Bansal, G., Fourney, A., *et al.* (2024) 'Reading between the lines: modeling user behavior and costs in AI-assisted programming'. *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems (CHI 2024)*. Honolulu, USA, 142. https://doi.org/10.1145/3613904.3641936

80) Nguyen, N. and Nadi, S. (2022) 'An empirical evaluation of GitHub Copilot's code suggestions'. *Proceedings of the 19th International Conference on Mining Software Repositories (MSR 2022)*. Pittsburgh, USA, pp. 1-5. https://doi.org/10.1145/3524842.3528470

81) NIST (2022) *Secure Software Development Framework (SSDF) Version 1.1: Recommendations for Mitigating the Risk of Software Vulnerabilities*. NIST Special Publication 800-218. Gaithersburg, MD: National Institute of Standards and Technology. https://doi.org/10.6028/NIST.SP.800-218

82) NIST (2023) *Artificial Intelligence Risk Management Framework (AI RMF 1.0)*. NIST AI 100-1. Gaithersburg, MD: National Institute of Standards and Technology. https://doi.org/10.6028/NIST.AI.100-1

83) Notaro, P., Cardoso, J. and Gerndt, M. (2021) 'A survey of AIOps methods for failure management', *ACM Transactions on Intelligent Systems and Technology*, 12(6), 81. https://doi.org/10.1145/3483424

84) Nwakamma, S., Ojuku, J. and Oyesiji, S.O. (2024a) 'LLM-as-a-judge for automated model governance: a review of methods, biases, and release-gating practices', *World Journal of Innovation and Modern Technology*, 8(6), pp. 185-216. https://doi.org/10.56201/wjimt.v8.no6.2024.pg185.216

85) Nwakamma, S., Ojuku, J. and Oyesiji, S.O. (2024b) 'Securing agentic AI enterprise workflows against prompt injection, tool poisoning, memory manipulation, and excessive agency threats', *World Journal of Innovation and Modern Technology*, 8(6), pp. 217-248. https://doi.org/10.56201/wjimt.v8.no6.2024.pg217.248

86) Odejobi, O.D., Okonkwo, C.S., Ahiaeke Patrick, M.C., *et al.* (2025) 'AI-augmented secure software engineering: leveraging deep learning for autonomous threat detection and mitigation', *International Journal of Engineering and Modern Technology*, 11(12), pp. 101-121. https://doi.org/10.56201/ijemt.vol.11.no12.2025.pg101.121

87) Ogbole, J.I., Okoruwa, P.O., Fadayomi, O., *et al.* (2021) 'Conceptual model for identity-centric zero trust architecture in enterprise security governance', *International Journal of Scientific Research in Computer Science, Engineering and Information Technology*, 7(5), pp. 393-415.

88) Ojuku, J., Oyesiji, S.O. and Nwakamma, S. (2025) 'WebAssembly component model and WASI for portable cloud-to-edge software: language interoperability, security, performance, and container comparisons', *International Journal of Computer Science and Mathematical Theory*, 11(12), pp. 232-263. https://doi.org/10.56201/ijcsmt.vol.11.no12.2025.pg232.263

89) Okoruwa, P.O., Fadayomi, O., Abolaji, T.O., *et al.* (2024) 'Enterprise cybersecurity trends and threat evolution, advances and emerging research directions', *Global Multidisciplinary Perspectives Journal*, 1(6), pp. 194-204. https://doi.org/10.54660/GMPJ.2024.1.6.194-204

90) Okoruwa, P.O., Fadayomi, O., Akeju, B., *et al.* (2020) 'Conceptual model for privacy-centric security engineering in digital and cloud computing systems', *International Journal of Scientific Research in Computer Science, Engineering and Information Technology*, 7(5), pp. 371-389.

91) Olagunju, E.O., Edivri, J. and Oteri, O. (2022) 'Predictive capacity planning and resource utilization forecasting models for multi program and multi stakeholder IT portfolios', *Shodhshauryam, International Scientific Refereed Research Journal*, 5(3), pp. 336-356.

92) Olagunju, E.O., Edivri, J. and Oteri, O. (2023) 'A conceptual governance model for change, incident, and problem management in mission-critical enterprise IT environments', *Gyanshauryam, International Scientific Refereed Research Journal*, 6(6).

93) Olagunju, E.O., Edivri, J. and Oteri, O. (2024) 'Applied performance optimization frameworks for managing high-demand enterprise technology programs and system rollouts', *International Journal of Advanced Multidisciplinary Research and Studies*, 4(6), pp. 3059-3070. https://doi.org/10.62225/2583049X.2024.4.6.5915

94) Olagunju, E.O., Oteri, O. and Edivri, J. (2025) 'An operational reliability and service assurance framework for enterprise IT systems supporting large user populations', *International Journal of Advanced Multidisciplinary Research and Studies*, 5(6), pp. 2117-2128. https://doi.org/10.62225/2583049X.2025.5.6.5916

95) Ominyi, M. and Anichukwueze, C.C. (2024) 'Conceptualizing responsible AI governance structures for automated decision systems in regulated industries', *International Journal of Social Sciences and Management Research*, 10(11), pp. 403-430. https://doi.org/10.56201/ijssmr.v10.no11.2024.pg403.430

96) Ominyi, M., Anichukwueze, C.C. and Uzougbo, N.S. (2024) 'A systematic review of organizational readiness for the EU Artificial Intelligence Act in multinational enterprises', *World Journal of Innovation and Modern Technology*, 8(6), pp. 185-203. https://doi.org/10.56201/wjimt.v8.no6.2024.pg185.203

97) Ominyi, M., Anichukwueze, C.C. and Uzougbo, N.S. (2025) 'Reviewing legal privilege and confidentiality challenges in enterprise deployment of large language models', *Journal of Law and Global Policy*, 10(3), pp. 150-176. https://doi.org/10.56201/jlgp.vol.10.no3.2025.pg150.176

98) Oshoba, T.O., Hammed, N.I. and Odejobi, O.D. (2020) 'Blockchain-enabled compliance and audit trail model for cloud configuration management', *Journal of Frontiers in Multidisciplinary Research*, 1(1), pp. 193-201.

99) Oteri, O. and Chibwaye, I.P. (2022) 'Econometric and conceptual models for predicting early technology adoption in emerging and transitional economies', *Shodhshauryam, International Scientific Refereed Research Journal*, 5(1), pp. 451-483.

100) Oteri, O. and Chibwaye, I.P. (2025) 'Advances and conceptual models in enterprise level change management analytics for large scale organizational system transitions', *Gyanshauryam, International Scientific Refereed Research Journal*, 8(5), pp. 177-211.

101) Oteri, O., Olagunju, E.O. and Edivri, J. (2025a) 'An operational reliability engineering framework for sustained high availability in 5G core networks serving millions', *International Journal of Advanced Multidisciplinary Research and Studies*, 5(6), pp. 2129-2147.

102) Oteri, O., Olagunju, E.O. and Edivri, J. (2025b) 'Conceptual and architectural models for cloud native 5G packet core orchestration using Kubernetes and OpenStack', *Gyanshauryam, International Scientific Refereed Research Journal*, 8(5), pp. 52-92.

103) Oteri, O., Olagunju, E.O. and Edivri, J. (2025c) 'Conceptual models for cloud-based IT service management architecture and orchestration using ServiceNow platforms', *Shodhshauryam, International Scientific Refereed Research Journal*, 8(6), pp. 185-203.

104) Pearce, H., Ahmad, B., Tan, B., *et al.* (2022) 'Asleep at the keyboard? Assessing the security of GitHub Copilot's code contributions'. *Proceedings of the 2022 IEEE Symposium on Security and Privacy (SP 2022)*. San Francisco, USA, pp. 754-768. https://doi.org/10.1109/SP46214.2022.9833571

105) Perry, N., Srivastava, M., Kumar, D., *et al.* (2023) 'Do users write more insecure code with AI assistants?'. *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS 2023)*. Copenhagen, Denmark, pp. 2785-2799. https://doi.org/10.1145/3576915.3623157

106) Ray, B., Hellendoorn, V., Godhane, S., *et al.* (2016) 'On the "naturalness" of buggy code'. *Proceedings of the 38th International Conference on Software Engineering (ICSE 2016)*. Austin, USA, pp. 428-439. https://doi.org/10.1145/2884781.2884848

107) Ronanki, K., Berger, C. and Horkoff, J. (2023) 'Investigating ChatGPT's potential to assist in requirements elicitation processes'. *Proceedings of the 49th Euromicro Conference on Software Engineering and Advanced Applications (SEAA 2023)*. Durres, Albania, pp. 354-361. https://doi.org/10.1109/SEAA60479.2023.00061

108) Russell, R., Kim, L., Hamilton, L., *et al.* (2018) 'Automated vulnerability detection in source code using deep representation learning'. *Proceedings of the 17th IEEE International Conference on Machine Learning and Applications (ICMLA 2018)*. Orlando, USA, pp. 757-762.

109) Sallou, J., Durieux, T. and Panichella, A. (2024) 'Breaking the silence: the threats of using LLMs in software engineering'. *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER 2024)*. Lisbon, Portugal, pp. 102-106. https://doi.org/10.1145/3639476.3639764

110) Sandoval, G., Pearce, H., Nys, T., *et al.* (2023) 'Lost at C: a user study on the security implications of large language model code assistants'. *Proceedings of the 32nd USENIX Security Symposium*. Anaheim, USA, pp. 2205-2222.

111) Schäfer, M., Nadi, S., Eghbali, A., *et al.* (2024) 'An empirical evaluation of using large language models for automated unit test generation', *IEEE Transactions on Software Engineering*, 50(1), pp. 85-105. https://doi.org/10.1109/TSE.2023.3334955

112) Sculley, D., Holt, G., Golovin, D., *et al.* (2015) 'Hidden technical debt in machine learning systems'. *Advances in Neural Information Processing Systems 28 (NIPS 2015)*. Montreal, Canada, pp. 2503-2511.

113) Spracklen, J., Wijewickrama, R., Sakib, A.H.M.N., *et al.* (2025) 'We have a package for you! A comprehensive analysis of package hallucinations by code generating LLMs'. *Proceedings of the 34th USENIX Security Symposium*. Seattle, USA.

114) Steenhoek, B., Rahman, M.M., Jiles, R., *et al.* (2023) 'An empirical study of deep learning models for vulnerability detection'. *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE 2023)*. Melbourne, Australia, pp. 2237-2248. https://doi.org/10.1109/ICSE48619.2023.00188

115) Tufano, R., Masiero, S., Mastropaolo, A., *et al.* (2022) 'Using pre-trained models to boost code review automation'. *Proceedings of the 44th International Conference on Software Engineering (ICSE 2022)*. Pittsburgh, USA, pp. 2291-2302. https://doi.org/10.1145/3510003.3510621

116) Uchechi, N., Ominyi, M., Anichukwueze, C.C., *et al.* (2022) 'Real-time compliance pipelines: a review of automated risk monitoring and enterprise data governance', *International Journal of Multidisciplinary Futuristic Development*, 3(2), pp. 66-94. https://doi.org/10.54660/IJMFD.2022.3.2.66-94

117) Upreti, M. and Oteri, O. (2023) 'Analytical model for optimizing cloud resource allocation through rightsizing and long-term usage forecasting', *Gyansharyam, International Scientific Refereed Research Journal*, 6(1), pp. 400-424.

118) Upreti, M. and Oteri, O. (2025) 'Security design model for resilient network architectures in large scale cloud computing platforms', *Shodhsharyam, International Scientific Refereed Research Journal*, 8(5), pp. 84-104.

119) Vaithilingam, P., Zhang, T. and Glassman, E.L. (2022) 'Expectation vs. experience: evaluating the usability of code generation tools powered by large language models'. *Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems (CHI EA 2022)*. New Orleans, USA, Article 332. https://doi.org/10.1145/3491101.3519665

120) Vaswani, A., Shazeer, N., Parmar, N., *et al.* (2017) 'Attention is all you need'. *Advances in Neural Information Processing Systems 30 (NIPS 2017)*. Long Beach, USA, pp. 5998-6008.

121) Wang, J., Huang, Y., Chen, C., *et al.* (2024) 'Software testing with large language models: survey, landscape, and vision', *IEEE Transactions on Software Engineering*, 50(4), pp. 911-936. https://doi.org/10.1109/TSE.2024.3368208

122) Xia, C.S., Paltenghi, M., Tian, J.L., *et al.* (2024) 'Fuzz4All: universal fuzzing with large language models'. *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering (ICSE 2024)*. Lisbon, Portugal, Article 126. https://doi.org/10.1145/3597503.3639121

123) Xia, C.S., Wei, Y. and Zhang, L. (2023) 'Automated program repair in the era of large pre-trained language models'. *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE 2023)*. Melbourne, Australia, pp. 1482-1494. https://doi.org/10.1109/ICSE48619.2023.00129

124) Yu, Z., Wu, Y., Zhang, N., *et al.* (2023) 'CodeIPPrompt: intellectual property infringement assessment of code language models'. *Proceedings of the 40th International Conference on Machine Learning (ICML 2023)*. Honolulu, USA, PMLR 202, pp. 40373-40389.

125) Zhou, Y., Liu, S., Siow, J., *et al.* (2019) 'Devign: effective vulnerability identification by learning comprehensive program semantics via graph neural networks'. *Advances in Neural Information Processing Systems 32 (NeurIPS 2019)*. Vancouver, Canada, pp. 10197-10207.

126) Ziegler, A., Kalliamvakou, E., Li, X.A., *et al.* (2024) 'Measuring GitHub Copilot's impact on productivity', *Communications of the ACM*, 67(3), pp. 54-63. https://doi.org/10.1145/3633453