**International Journal of Academic and Applied Research (IJAAR)**
**ISSN: 2643-9603**
**Vol. 10 Issue 10, October - 2026, Pages: 13-30**

# Generative Artificial Intelligence in Software Testing

**Serif Oyindamola Oyesiji[1*], Aisha I. Abdulwahab[2], Harouna Wendpanga Yann Christian Sankara[3], Rosalyn Ezeako[4]**

[1] Harrisburg University of Science and Technology, Harrisburg, Pennsylvania, USA
[2] Cummins Inc., Indiana, USA
[3] Five9, San Ramon, California, USA
[4] University of Central Missouri, Warrensburg, Missouri, USA
[*] *Corresponding author: Serif Oyindamola Oyesiji, oyesijiserif@gmail.com*

***Abstract****: Background: Large language models now write unit tests, propose assertions, and repair failing test code inside continuous integration pipelines, yet a generated test that compiles and passes is not thereby evidence that software is correct. Objective: This review asks how generated test artefacts should be validated before they count as assurance evidence, and which organisational controls keep that evidence trustworthy. Method: An integrative review of software testing, traceability, empirical software engineering, and AI security literature published to 2026 was conducted under a predefined protocol, with sources appraised for oracle independence, fault measurement, rerun screening, and training-data contamination, and synthesised through mechanism, context, and outcome coding. Results: The evidence converges on four hazards: hallucinated or regression-captured assertions that certify existing faults, coverage gains unmatched by fault sensitivity, nondeterministic tests that erode trust in the build signal, and leakage or injection through prompts and agent tools. Marginal mutation analysis, rerun screening, requirement tracing, sandboxed execution, and evidence-informed review address these hazards unevenly. Contribution: The article proposes GATES, a framework of five constructs (grounding in intent, adversarial fault sensitivity, test determinism, endorsement, and security containment) with measures, gating rules, six falsifiable propositions, and a staged evaluation design. Implications: Generated tests should be admitted on fault-revealing, requirement-linked evidence, never on pass rates alone.*

**Keywords**: large language models; unit test generation; test oracle; mutation testing; flaky tests; prompt injection; human oversight

## 1 Introduction

Software testing has always rested on an asymmetry: a test can show that a fault is present but cannot show that faults are absent, and its value depends entirely on whether its verdict can be trusted (Myers et al., 2011). Generative artificial intelligence, and large language models (LLMs) in particular, changes the economics of producing tests without changing that asymmetry. Code-trained models can draft unit tests from a method signature, propose input values for boundary conditions, translate natural-language acceptance criteria into executable scenarios, and suggest repairs when a test breaks after refactoring (Borah et al., 2022; Mbonu et al., 2021b). Compared with earlier search-based generators, what generative models add is readability, idiomatic naming, natural-language statements of intent, and the ability to work from incomplete or non-compiling context. Those additions are exactly what make generated tests persuasive to human reviewers, and therefore exactly what make their validation a governance problem rather than a tooling detail (Ladapo et al., 2022).

### 1.1 Problem Setting: Why a Passing Generated Test Is Not Yet Evidence

A generated test has three parts, and each can fail independently. The fixture and input construction can be unrealistic, the action can call the wrong method or a method that does not exist, and the assertion can encode an expectation that is wrong. The third failure is the most consequential, because the assertion is the test oracle, the mechanism that decides whether observed behaviour is acceptable (Ammann et al., 2016; Zhu et al., 1997). When a model is asked to write assertions for existing code, the most readily available evidence about expected behaviour is the code itself. A model that infers the expected value from the implementation produces a regression oracle: the test passes on the current program, including any fault the program contains, and fails only if behaviour later changes. Where intended behaviour is documented in a requirement, the gap between a regression oracle and a specification oracle is precisely the gap that requirements traceability research has tried to close for three decades (Gotel et al., 1994; Ramesh et al., 2001).

Industrial adoption is running ahead of evidence. Surveys of unit testing practice conducted well before generative models found that developers struggle most with deciding what to assert and with maintaining tests after change (Daka et al., 2014), and those are the tasks that vendors now promise to automate. Organisational pressure then converts adoption counts, such as the number of generated tests merged or the line coverage added per sprint, into performance indicators. Coverage is a weak adequacy criterion on its own, because a statement can be executed without its result ever being checked (Zhu et al., 1997). A pipeline that rewards coverage added by generated tests therefore rewards the failure mode, execution without checking, to which a generator is most prone.

### 1.2 Research Questions and Contributions

The review addresses three research questions. RQ1 asks which failure modes distinguish LLM-generated test artefacts from human-written and search-generated tests, and how strongly each is supported by the available evidence. RQ2 asks which validity checks, including mutation analysis, rerun-based flakiness screening, requirement tracing, and sandboxed execution, discriminate between generated tests that add fault-revealing power and those that add only execution. RQ3 asks which organisational controls over data exposure, review authority, and lifecycle ownership are needed for generated tests to function as assurance evidence in regulated or security-sensitive settings (ISO/IEC/IEEE, 2018a; Ominyi et al., 2024b).

The article makes four contributions. (C1) It reframes generated assertions through signal detection theory, treating each assertion as a detector whose hit rate and false-alarm rate can be estimated from mutation and rerun evidence, which gives the oracle problem a measurable form for generative tools. (C2) It adjudicates conflicting claims about coverage, readability, and fault detection, separating findings about test production from findings about test validity. (C3) It proposes GATES, a framework of five constructs (grounding in intent, adversarial fault sensitivity, test determinism, endorsement by accountable reviewers, and security containment of data and execution), each with a measure, a failure signal, and a gating action. (C4) It states six falsifiable propositions and a staged evaluation design capable of refuting them. The framework is conceptual: it is derived from the reviewed literature and has not yet been tested in the field (Wieringa, 2014; Hevner et al., 2004).

## 2 Theoretical Grounding

Three bodies of theory frame the review. Testing theory defines what a test must do to count as evidence: it must reach a faulty location, infect program state, propagate the effect to an observable output, and reveal it through an oracle (Ammann et al., 2016). Signal detection theory supplies a vocabulary for the quality of verdicts made under uncertainty. Sociotechnical research on automation explains how people calibrate trust in machine outputs and why that calibration fails in predictable ways (Ladapo et al., 2022). Each strand is used below to derive a specific expectation about generated tests rather than as general background, and each expectation reappears as a proposition in Section 5.

### 2.1 The Oracle Problem and Four Classes of Generated Assertion

The oracle problem is the difficulty of determining, for a given input, whether the observed output is correct (Myers et al., 2011). Automated generators have historically resolved it by recording observed output and asserting that it stays the same. EvoSuite, for example, minimises its assertions to those that distinguish the current program from its mutants, producing regression oracles that are compact but that take current behaviour as ground truth (Fraser et al., 2011). LLM generators inherit the regression stance whenever the implementation under test appears in the prompt, but they add a second route to an expected value: the model's learned prior about what code of a given shape usually returns. That route can produce a specification-like oracle when the prior matches the intent, and it can also produce hallucinated assertions, expected values or exception types that neither the code nor any requirement supports.

Four classes of generated assertion are therefore worth distinguishing. A specification assertion checks intended behaviour and would fail on a faulty implementation. A regression-captured assertion checks current behaviour and is correct only if the implementation is. A hallucinated assertion checks behaviour that neither exists nor was intended, and typically fails on first execution. A vacuous assertion, for example a check that a returned object is not null when the method can never return null, passes on every plausible implementation and detects nothing. Standard generate-and-filter pipelines discard failing tests, which removes most hallucinated assertions but keeps regression-captured and vacuous ones, because both pass on the current code (Daka et al., 2014; Borah et al., 2022). The filter thus selects against the class that is visibly wrong and in favour of the classes that are invisibly weak. This selection effect is the central reason that pass rate misleads as a quality indicator for generated suites, and it motivates mutation analysis and requirement tracing as external checks (Ammann et al., 2016; Cleland-Huang et al., 2012).

### 2.2 Signal Detection as a Lens on Test Verdicts

Signal detection theory models a detector deciding whether a signal is present in noise. Applied to testing, the signal is a fault in the program under test and the detector is the test with its assertions. A hit is a failure on a faulty version, a miss is a pass on a faulty version, a false alarm is a failure on a correct version, and a correct rejection is a pass on a correct version. Two parameters summarise the detector. Sensitivity, $d'$, captures how well the test separates faulty from correct behaviour, and the criterion captures how strict its assertions are. Under the equal-variance Gaussian model, $d' = z(H) - z(F)$, where H is the hit rate, F is the false-alarm rate, and z is the inverse of the standard normal distribution function.

The lens yields three expectations for generated tests. First, the hit rate of a suite can be estimated by mutation analysis, because each non-equivalent mutant is a surrogate fault, so the mutation score is an empirical hit rate over a defined fault model (Ammann et al., 2016). Second, the false-alarm rate has two components, deterministic false alarms caused by wrong assertions and stochastic false alarms caused by flakiness, and only the second is invisible to single-run filtering. Third, loosening assertions to suppress flakiness, for instance by replacing exact equality with a range check or deleting a timing assertion, moves the criterion

rather than raising sensitivity: it lowers false alarms and hits together. A generator tuned to avoid failing tests will drift toward lax criteria unless something in the pipeline rewards hits. This explains why one tool can look excellent on one metric and poor on another.

### 2.3 Automation Bias and the Sociotechnical Review Loop

Generated tests enter a sociotechnical system in which a reviewer, usually a developer under delivery pressure, decides whether to merge them. Research on human-in-the-loop machine learning shows that human review improves outcomes only when reviewers have time, relevant information, authority to reject, and feedback on their past decisions (Ladapo et al., 2022). Automation bias, the tendency to accept machine output with less scrutiny than equivalent human output, is amplified when output is fluent and well formatted, which describes LLM-generated tests exactly. Complacency grows with the historical reliability of a tool, so a generator that is usually right is reviewed least carefully precisely when its rare errors are most costly. Deskilling occurs when oracle design is delegated for long enough that reviewers lose the practice needed to evaluate oracles.

Technology acceptance research adds that perceived usefulness and ease of use drive adoption independently of accuracy. In testing, perceived usefulness is dominated by visible outputs such as coverage deltas and test counts, whereas oracle accuracy is invisible without further analysis. Sociotechnical theory therefore predicts that, absent explicit instrumentation, adoption will be driven by visible indicators and review will be calibrated to a tool's apparent reliability rather than to its oracle validity. The design implication, developed in Section 5, is that the pipeline should place validity evidence in front of the reviewer at the moment of decision, and that reviewer decisions should themselves be audited against later fault data.

## 3 Review Protocol

The study is an integrative review. Integrative reviews combine empirical studies, conceptual work, standards, and methodological sources in order to build a framework, and they suit fields that are young, methodologically heterogeneous, and in need of theory rather than pooled effect estimates (Page et al., 2021). The protocol followed the reporting logic of PRISMA 2020 where it applies to non-meta-analytic synthesis: the question, sources, search strings, eligibility criteria, screening stages, appraisal instrument, and synthesis method were defined before screening, and deviations were recorded. Table 1 summarises the protocol and Figure 1 shows the flow from retrieval to framework construction. The review does not report pooled effect sizes, and it makes no claim of exhaustive coverage (Runeson et al., 2009).

### 3.1 Sources, Search Strings, and Eligibility

Searches covered the ACM Digital Library, IEEE Xplore, Scopus, the Web of Science Core Collection, SpringerLink, and the arXiv cs.SE listing, over a window running from January 1994, the year of an influential analysis of the requirements traceability problem (Gotel et al., 1994), to June 2026. The long window is deliberate. Generative test tools reuse concepts such as coverage adequacy, mutation analysis, and regression prioritisation whose foundations predate LLMs, and new claims should be compared against those baselines (Zhu et al., 1997; Li et al., 2007). Standards were retrieved from ISO and IEEE catalogues, with emphasis on test process and requirements engineering standards (ISO/IEC/IEEE, 2018a, 2018b).

The core search string joined three concept blocks with Boolean AND. The generation block was ("large language model" OR LLM OR "generative AI" OR "code model" OR GPT OR Codex OR "foundation model" OR "search-based test generation"). The testing block was ("unit test" OR "test generation" OR "test oracle" OR assertion OR "mutation testing" OR "mutation score" OR "flaky test" OR "regression test"). The validity and risk block was (validity OR hallucination OR correctness OR "fault detection" OR "data leakage" OR contamination OR "prompt injection" OR privacy OR "human review" OR "automation bias"). A secondary string replaced the generation block with terms for security operations and AI governance, in order to retrieve work on agentic tool risks, continuous control, and data protection bearing on the security containment construct (Nwakamma et al., 2024b; Aliliele et al., 2025b). Backward and forward citation chasing from included software engineering sources supplemented the database searches.

Sources were eligible when they (a) reported an empirical evaluation, formal definition, or design guidance concerning automated test generation, test oracles, mutation analysis, flakiness, traceability, or the security and governance of AI-assisted software work; (b) specified the artefact, setting, or measure precisely enough for a construct, mechanism, or boundary condition to be coded; and (c) appeared in peer-reviewed venues, as standards, or as scholarly books. Studies from adjacent domains, such as inspection engineering, alarm management, clinical decision support, and diagnostic laboratory quality, were admitted only to support cross-domain analogies about detection, oversight, and quality systems, and they are cited only for such general claims.

### 3.2 Screening, Appraisal, and Synthesis Logic

Screening proceeded in three stages: title and abstract screening against the eligibility criteria, full-text screening with reasons for exclusion recorded, and relevance mapping in which each included source was assigned to one or more GATES constructs. The appraisal instrument adapted guidance for empirical software engineering research (Runeson et al., 2009) to the validity concerns

specific to generated tests. Seven questions were asked of each empirical source: whether the oracle used to judge generated tests was defined independently of the generator; whether fault detection was measured with real faults, seeded mutants, or not at all; whether flakiness was screened by reruns and how many; whether benchmark projects could have appeared in the model's training data; whether model version, decoding parameters, and prompt were reported; whether human evaluation used blinded raters; and whether cost, latency, or reviewer time was measured. Standards and conceptual sources were appraised for clarity of definitions and for whether they specified controls whose effect could be tested.

Two appraisal rules were applied throughout. Rule A classed results obtained on benchmarks that were public before a model's training cut-off as potentially contaminated, and treated their effects as upper bounds rather than estimates, because memorised tests for well-known projects inflate both compilation and pass rates. Rule B refused to accept coverage results as evidence of fault detection unless mutation or real-fault analysis accompanied them, following the adequacy literature (Zhu et al., 1997; Ammann et al., 2016).

Synthesis used mechanism, context, and outcome coding. For each finding the extraction recorded the mechanism claimed (for example, that supplying method documentation improves assertion correctness), the context (language, project type, model family, prompt strategy), and the outcome measure (compilation, pass rate, coverage, mutation score, real-fault detection, flakiness, reviewer time). Findings were then grouped by construct and compared within groups. Contradictory findings were retained and examined for moderators rather than averaged away, in keeping with design science, in which artefacts are evaluated against explicit requirements in explicit contexts (Wieringa, 2014; Hevner et al., 2004). Table 1 records each protocol element with the safeguard adopted against the biases most likely in this literature.

**Table 1. Review protocol for generative test artefacts and the bias safeguard attached to each element**

| Protocol element | Rule applied in this review | Safeguard against bias |
|---|---|---|
| Questions | RQ1 failure modes; RQ2 validity checks; RQ3 organisational controls for LLM-generated test artefacts | Questions and construct map fixed before screening |
| Sources | ACM Digital Library, IEEE Xplore, Scopus, Web of Science, SpringerLink, arXiv cs.SE; ISO and IEEE standards catalogues | Backward and forward citation chasing to close database gaps |
| Window | January 1994 to June 2026 | Pre-LLM foundations (coverage, mutation, traceability) kept as baselines for new claims |
| Search strings | Generation block AND testing block AND validity or risk block; secondary string for agentic security and AI governance | Terms for flakiness, contamination, and injection included so that negative evidence is retrieved |
| Inclusion | Empirical evaluation, formal definition, or design guidance on test generation, oracles, mutation, flakiness, traceability, or AI security | Adjacent-domain sources admitted for analogy and general claims only |
| Exclusion | Benchmarks without disclosed method; opinion without argument; duplicate reports | Reasons recorded at full-text stage |
| Appraisal | Seven questions: independent oracle, fault measure, reruns, contamination, model and prompt reporting, blinded raters, cost | Rule A: pre-cut-off public benchmarks are upper bounds. Rule B: coverage alone is not fault evidence |
| Synthesis | Mechanism, context, and outcome coding by GATES construct | Contradictory findings retained and examined for moderators |

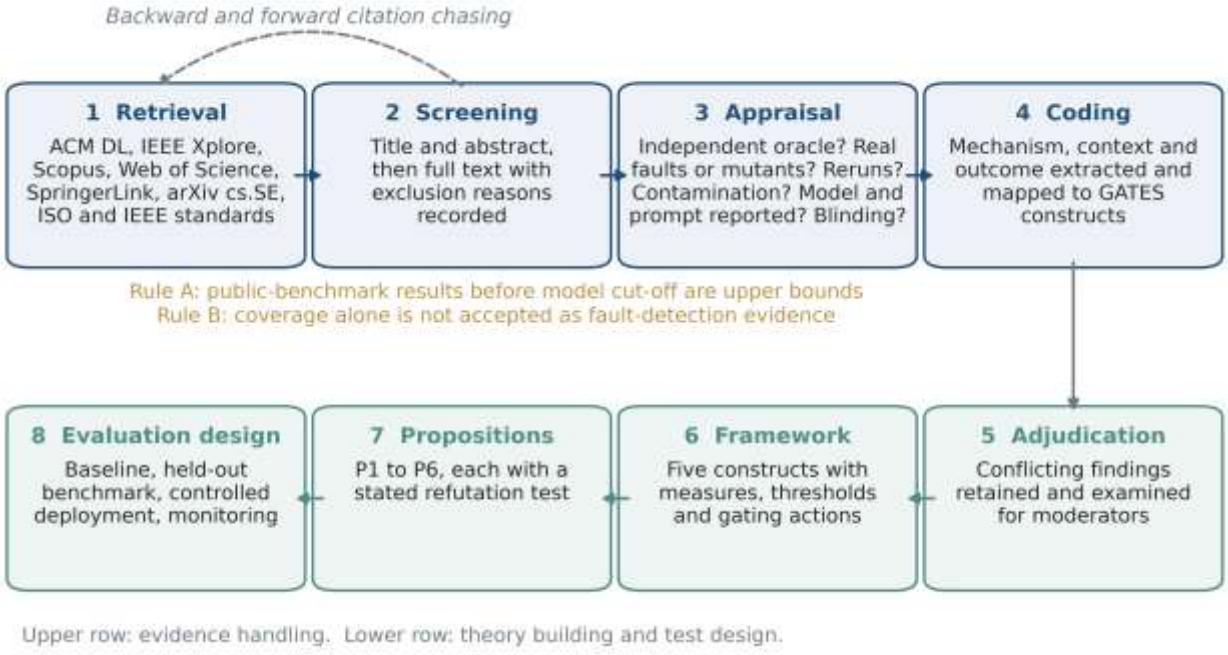


**Figure 1. Review protocol: evidence handling from retrieval to coding (upper row) and theory building from adjudication to evaluation design (lower row)**

Figure 1 separates evidence handling from theory building. A source shapes the framework only when it supports a construct, mechanism, boundary condition, or design choice; adjacent-domain sources contribute analogies only; and Rules A and B are applied before coding, so that contaminated or coverage-only results enter the synthesis already marked as upper bounds.

### 4 Synthesis of the Evidence

This section organises the evidence by failure mode rather than by tool, because tools change faster than the mechanisms that make their outputs valid or invalid. Each subsection states what the evidence supports, where it conflicts, and how the conflict is adjudicated, and Table 2 at the end of the section summarises the adjudications. Throughout, a distinction is kept between production outcomes (whether tests are generated, compile, and run) and validity outcomes (whether tests detect faults, reflect intent, and behave deterministically). Much of the enthusiasm around generative testing rests on production outcomes, whereas most of the risk lies in validity outcomes (Borah et al., 2022; Mbonu et al., 2021b).

### 4.1 Unit Test Generation: Coverage Gains and Their Limits

The most consistent finding across evaluations of LLM unit test generators is that they produce readable tests that raise line and branch coverage on code the model can see, and that raw outputs carry substantial rates of compilation failure and runtime failure that must be filtered or repaired (Borah et al., 2022). Iterative strategies, in which compiler errors and failing assertions are fed back to the model for repair, raise the share of usable tests but also raise the risk that assertions are edited to match observed behaviour, converting specification attempts into regression oracles. Compared with search-based generation, LLM tests tend to be shorter, use more meaningful identifiers, and are judged more understandable by developers, while search-based tools tend to reach higher coverage on heavily branching code when given enough search budget (Fraser et al., 2011; Daka et al., 2014).

The apparent conflict between studies reporting large coverage gains and studies reporting modest fault detection dissolves once the adequacy hierarchy is considered. Statement coverage requires only that code runs, whereas mutation adequacy requires that a changed behaviour be observed and checked (Zhu et al., 1997; Ammann et al., 2016). A generated test that calls a method and asserts only that no exception occurred adds coverage without adding checking. When evaluations report coverage alone, they measure the reach component of fault detection and leave infection, propagation, and revelation unmeasured. The adjudication adopted here is that coverage results establish an upper bound on the fault-revealing contribution of generated tests and say nothing about its size.

Context availability is the principal moderator. Models produce better tests for self-contained utility classes than for methods whose behaviour depends on project-specific types, configuration, or external services, because the relevant context is either absent from the prompt or truncated by the context window. Retrieval-augmented prompting, which inserts related classes,

existing tests, and documentation, improves compilation, and information retrieval over software artefacts is a mature technique for selecting that context (Borg et al., 2014). It also sends more proprietary code to the model provider, a trade-off examined in Section 4.5. The practical boundary is that success on open-source benchmarks does not transfer automatically to enterprise codebases with deep dependency graphs, internal frameworks, sparse documentation, and poorly stated non-functional requirements, which is where much regulated software is written.

The unit of reported improvement also matters. Studies that report the percentage of methods for which at least one passing test was produced, analogous to pass@k in code generation, answer a different question from studies that report the fault-detection power of the resulting suite. Governance decisions about trusting generated tests depend on the second, while procurement cases are commonly argued on the first.

### 4.2 Hallucinated Assertions and Regression-Oracle Capture

Hallucination in test generation takes forms that differ in how easily they are detected. Structural hallucinations, such as calls to nonexistent methods, wrong constructor signatures, or imports of unavailable packages, surface as compilation errors and are removed by the build. Semantic hallucinations, such as an expected value computed with an arithmetic slip, an exception type that the method never throws, or an assumed ordering of an unordered collection, surface as failures on the correct program and are usually removed as well, or, worse, repaired by editing the assertion to match the output. Silent errors are the dangerous class: assertions that are wrong about intent but right about current behaviour, which arise when the model's prior and a faulty implementation agree, and assertions that are vacuous (Myers et al., 2011).

Regression-oracle capture deserves separate treatment because it is systematic rather than random. If a method contains a fault and the model is shown the method, the most probable completion for an expected value is the value the method computes. The generated test then certifies the fault. A later correct fix will fail that test, and a developer who trusts the generated suite may revert the fix or edit the test. Evaluations using real-fault benchmarks reveal this pattern when tests are generated against the faulty version and then run against the fixed version: assertions that pass on the faulty version and fail on the fix are the signature of captured faults (Mbonu et al., 2021b; Borah et al., 2022). The effect is concealed in evaluations that generate tests against fixed versions and then check whether they fail on seeded faults, because in that design the implementation shown to the model is already correct.

Three mitigations appear in the literature, with different evidential support. The first withholds the implementation and prompts from the signature, documentation, and requirement only; this forces a specification stance but lowers compilation and pass rates and increases structural hallucination. The second prompts with the requirement and then uses traceability links to check that each assertion corresponds to a stated behaviour; information retrieval methods for recovering links between requirements, code, and tests are mature enough to support this check at scale (Borg et al., 2014; Cleland-Huang et al., 2012). The third uses metamorphic or differential relations, such as the invariance of a sorted result under permutation of its input, which check properties rather than values and are less susceptible to capture. The adjudication is that no single mitigation removes capture, but tracing assertions to requirements is the only one that makes capture detectable, because it supplies an independent statement of intent against which the assertion can be compared (Ramesh et al., 2001; ISO/IEC/IEEE, 2018b).

### 4.3 Mutation Score as a Validity Check

Mutation analysis seeds small syntactic changes, called mutants, into the program, for example replacing a relational operator or negating a condition, and measures the proportion of mutants that the test suite detects, or kills (Ammann et al., 2016). The mutation score is $MS = K / (M - E)$, where K is the number of killed mutants, M the number generated, and E the number of equivalent mutants, which differ syntactically from the original but not semantically. The technique rests on two hypotheses: the competent programmer hypothesis, that real programs are close to correct, and the coupling effect, that tests which detect simple faults tend also to detect the complex faults composed of them. Where these hypotheses hold, the mutation score estimates a suite's hit rate over a realistic fault distribution, the quantity that neither pass rate nor coverage can provide.

For generated tests the most informative quantity is the marginal mutation score, the additional mutants killed by generated tests beyond those already killed by the existing human-written suite: $\Delta MS = (K(H \cup G) - K(H)) / (M - E)$, where H and G denote the human and generated suites. A generated suite that raises coverage while leaving $\Delta MS$ near zero adds execution without evidence. A per-test variant, the number of mutants killed uniquely by a given test, identifies generated tests that carry distinct evidence and those that duplicate existing checks.

Equivalent mutants inflate the denominator and cannot be identified in general, so reported scores are lower bounds of true adequacy. Mutation operators model small faults, and generated tests may be weakest precisely against semantic misunderstandings of requirements that no operator reproduces. Mutation analysis is expensive, scaling roughly with the product of mutants and tests, which pushes teams to sample mutants or restrict analysis to changed code. Finally, and specific to generative pipelines, a generator given mutation feedback can learn to kill mutants with assertions tightly coupled to implementation detail,

raising the score while making tests brittle under legitimate refactoring. The mutation score is therefore a necessary but gameable check, and it should be paired with a maintenance measure such as the rate at which generated tests fail after behaviour-preserving refactoring (Li et al., 2007).

The adjudication adopted is that mutation analysis is the strongest single check currently available on the strength of generated oracles, that it should run on changed code in every merge request introducing generated tests, and that thresholds should be set on the marginal score rather than the absolute score, because the absolute score is dominated by the existing suite. Inspection engineering provides the right mental model: non-destructive testing methods are qualified by their probability of detecting seeded defects of known size, not by the area they scan. Coverage is the scanned area; the mutation score is the probability of detection.

### 4.4 Flakiness and Nondeterminism in Generated Tests

A flaky test is one that can both pass and fail on the same code in the same environment. Generated tests are susceptible for identifiable reasons: models reproduce common but fragile idioms such as fixed sleeps, assert on string renderings of hash-based collections, create shared static state across tests, and call real clocks and networks when no mock appears in the context (Daka et al., 2014). A second and distinct source of nondeterminism is the generator itself. With nonzero sampling temperature the same prompt yields different tests on different runs, and provider-side model updates change outputs even at zero temperature.

The cost of flakiness is best understood as a false-alarm problem in the sense introduced in Section 2.2. Each spurious failure consumes investigation time and, more damagingly, teaches developers that red builds are often noise. Process industries know the same dynamic as alarm fatigue: standing and nuisance alarms train operators to acknowledge rather than diagnose, degrading their response to genuine alarms. The implication for generated tests is that a small number of flaky additions can reduce the effective sensitivity of an entire suite by lowering the attention paid to every failure.

Rerun-based screening detects flakiness probabilistically. If a flaky test fails independently with probability p on each run, the probability that it fails at least once in n reruns is $1 - (1 - p)^n$. A test that fails on 5 percent of runs is caught with probability of about 0.40 after 10 reruns and about 0.92 after 50, whereas a test that fails on 1 percent of runs needs roughly 300 reruns for 95 percent detection. Panel (a) of Figure 2 plots this relation. Two consequences follow. Screening generated tests with a single execution, the default in generate-and-filter pipelines, admits most low-rate flaky tests. And the rerun budget should be derived from the flakiness rate the team is prepared to tolerate rather than chosen for convenience. Because failures often correlate with machine load and test order, the independence assumption is optimistic, and reruns under shuffled order and varied parallelism give a more realistic estimate (Li et al., 2007).

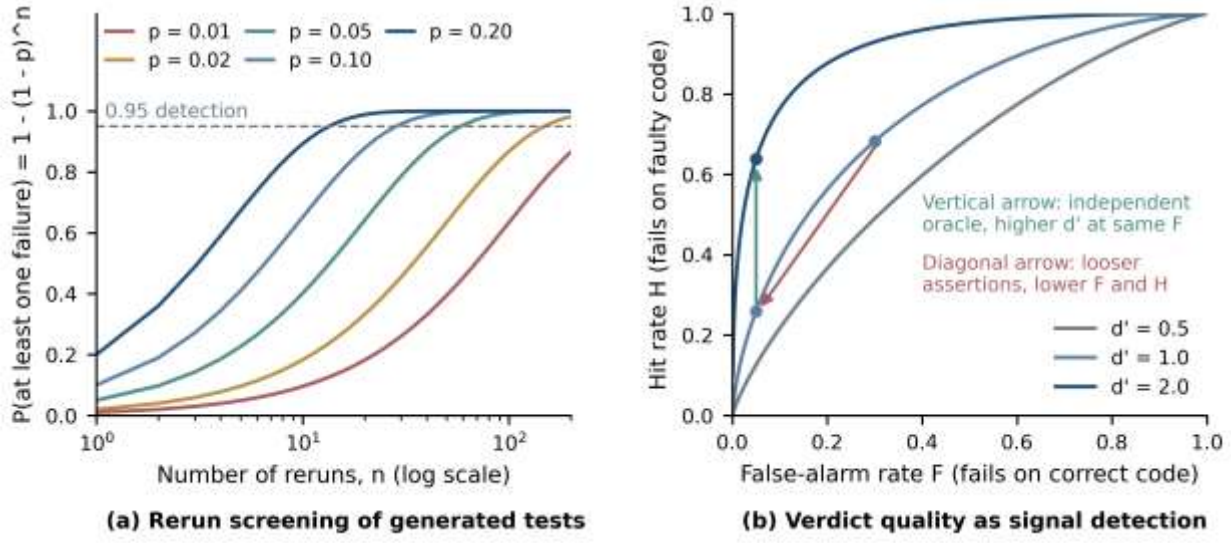


**Figure 2. Analytic illustrations of verdict quality: (a) probability that rerun screening detects a flaky generated test as a function of reruns and per-run failure rate; (b) receiver operating curves showing the difference between loosening assertions and adding an independent oracle**

Panel (b) of Figure 2 illustrates the trade-off identified in Section 2.2: loosening an assertion moves a test along its curve, whereas an independent oracle moves it to a higher curve. Anomaly detection over test history offers a complementary, continuous control. Pass and fail sequences per test form a time series in which flaky tests show intermittent transitions that are uncorrelated with code change, and methods developed for operational time series and failure prediction can flag them (Atakpa et al., 2023). The

adjudication is that rerun screening at admission and history-based monitoring in operation are complements: the first stops obvious flakiness from entering, and the second catches flakiness that emerges later from environmental change. Quarantine of detected flaky tests should be time-limited and owned, because quarantined tests that are never repaired become permanent blind spots.

### 4.5 Prompt Leakage, Data Contamination, and the Agentic Attack Surface

Security exposure in generative testing has three components that are often conflated: disclosure of proprietary information through prompts, contamination of evaluation data through training exposure, and manipulation of the generator or its tools by adversarial content. Each has its own threat model and its own control. Prompt leakage occurs when source code, configuration, fixtures, or logs sent to a hosted model contain secrets, personal data, or trade secrets. Data sensitivity classification with regulatory traceability, which labels data by category and tracks its movement, provides the basis for deciding what may leave the organisation. Data loss prevention at the boundary, applied to prompts as it is to email and storage, can redact secrets and identifiers before transmission (Aliliele et al., 2025b), and privacy-by-design role models restrict which developers may send which repositories to which endpoints. Where exposure cannot be reduced enough, locally hosted small language models, compressed through quantisation, pruning, and distillation, keep context inside the organisation at some cost in capability (Nwakamma et al., 2025).

Contamination concerns the validity of evidence rather than the confidentiality of data. When the projects in a public benchmark, often including their existing tests, were available on the web before a model's training cut-off, the model may reproduce memorised tests, and its apparent performance overstates what it would achieve on unseen proprietary code. Rule A in Section 3.2 treats such results as upper bounds, and evaluations meant to support adoption decisions should include held-out internal code or projects created after the model's cut-off (Runeson et al., 2009; Page et al., 2021). Embedding-based near-duplicate detection can be used to flag generated tests that are near copies of public tests. Where organisations fine-tune a generator on their own repositories, context-aware curation of the fine-tuning data decides which tests, fixtures and historical defects the model learns from, and therefore whether it is steered toward specification oracles or toward reproducing implemented behaviour (Oyesiji et al., 2024).

The agentic attack surface is newer and growing. Test generation agents read repository content, call tools such as build systems and package managers, and execute the code they produce. Each capability opens a channel for indirect prompt injection, in which instructions embedded in a code comment, a README, a dependency, or a fixture redirect the agent; for tool poisoning; and for excessive agency, in which the agent acts beyond its task, for example by modifying pipeline configuration or disabling failing tests (Nwakamma et al., 2024b). Generated tests are executable code, and running unreviewed code that can install packages or open network connections inside a build environment holding deployment credentials is a supply-chain risk. Zero trust controls, including least privilege for the agent's identity, restricted network egress, and ephemeral sandboxes, apply directly, and capability-based runtimes such as WebAssembly with explicit system interfaces offer fine-grained isolation for untrusted generated code (Ojukwu et al., 2025). Deep learning approaches to autonomous threat detection and mitigation in secure software engineering add a complementary control, in which the pipeline screens generated code for unsafe behaviour before it is executed (Odejobi et al., 2025). Monitoring and automated response for anomalous agent behaviour add a further layer, and adversarial simulation of injection attempts belongs in the evaluation of any agentic testing tool (Dosunmu et al., 2026b). Risk-based assurance and continuous control suit such fast-changing configurations better than point-in-time audits.

### 4.6 Human Oversight and the Economics of Review

Human review is the final gate in most generative testing workflows, and the evidence on its effectiveness is mixed in ways that theory predicts. Reviewers catch structural and obvious semantic errors reliably, but regression-captured and vacuous assertions pass review at high rates, because they look correct, they pass, and the reviewer rarely has the requirement at hand when reading the test (Ladapo et al., 2022; Daka et al., 2014). Review quality also varies with volume. A generator that proposes dozens of tests per changed method shifts the bottleneck from writing to reading. When review time per test falls below the time needed to trace an assertion to intent, review becomes approval.

The economics can be made explicit. Let a generated test save w minutes of authoring time, cost r minutes of review, and carry probability q of being a silently invalid test that, if merged, costs c minutes of later investigation. Adoption is beneficial in expectation when $w > r + q \cdot c$, before any cost of a masked fault is counted. The inequality shows why cutting review time r by trusting the tool can be self-defeating: q is itself a decreasing function of r, because less careful review admits more invalid tests. Organisations that measure only w, the authoring time saved, will overstate net benefit.

Three design moves strengthen oversight. The first surfaces validity evidence at the point of review, namely the marginal mutation contribution of each generated test, its rerun history, and its trace link, so that the reviewer judges evidence rather than appearance. The second allocates review authority by risk: tests for safety-, security-, or finance-critical modules require a named owner with domain knowledge, while tests for low-risk utilities may be admitted on automated evidence alone. The third audits reviewers by

injecting known-invalid assertions at a low rate and measuring how often they are caught. Multi-agent quality control, in which a second model critiques the first model's output, can pre-screen tests, but it inherits correlated errors when both models share training data, so it complements rather than replaces accountable review. Reviews of LLM-as-a-judge methods for automated model governance catalogue the biases of model-based evaluators alongside the release-gating practices built around them, which supports treating a judging model as one input to the admission gate rather than as the gate itself (Nwakamma et al., 2024a).

Workforce effects extend beyond individual reviews. Structured mentoring and on-the-job learning build the tacit knowledge that oracle design requires, and organisations that delegate oracle design wholesale may erode the skills needed to review it. Table 2 brings the adjudications of Sections 4.1 to 4.6 together.

**Table 2. Adjudication of conflicting evidence on LLM-generated tests**

| Theme | Claim favouring generated tests | Contrary or qualifying evidence | Adjudication |
|---|---|---|---|
| Coverage | Generated tests raise line and branch coverage quickly | Coverage gains often exceed gains in mutation or real-fault detection | Coverage is an upper bound on fault-revealing value; require marginal mutation evidence |
| Readability | LLM tests are judged more understandable than search-generated tests | Fluency raises reviewer trust independently of correctness | A maintenance benefit and an automation-bias risk at the same time |
| Assertion correctness | Iterative repair raises the share of passing tests | Repair edits assertions toward observed behaviour and can capture faults | Pass rate is not an oracle measure; trace assertions to requirements |
| Fault detection | Some generated suites find faults existing suites miss | Public-benchmark results may be inflated by training contamination | Treat as upper bound unless held-out or post-cut-off code is used |
| Determinism | Filtered generated tests pass on first execution | Single-run filtering admits low-rate flaky tests; generator output varies between runs | Rerun budget set from tolerated flakiness; pin and log model versions |
| Security | Hosted models lower setup cost and raise capability | Prompts expose code and secrets; agents widen the injection surface | Classify and redact context; sandbox execution; least-privilege agents |
| Human review | Review reliably catches hallucinated structure | Captured and vacuous assertions pass review under volume pressure | Show validity evidence at review; tier authority by risk; audit reviewers |

## 5 The GATES Framework

The synthesis supports a conditional mechanism. Generated test artefacts become assurance evidence when their assertions are traceable to independent statements of intent, when their verdicts are deterministic, when they demonstrably detect faults beyond those the existing suite detects, when their generation and execution stay within an acceptable exposure envelope, and when an accountable person endorses them with the evidence in view. Remove any one condition and the artefact reverts to plausible code. GATES names these conditions as five constructs: grounding in intent, adversarial fault sensitivity, test determinism, endorsement, and security containment. The framework draws on design science in treating the governed generation pipeline as an artefact with requirements and evaluation criteria (Hevner et al., 2004; Wieringa, 2014), and on organisational learning, which requires operational evidence to feed back into upstream design.

### 5.1 Constructs and Measures

Grounding in intent (G) is the degree to which each generated assertion corresponds to a documented requirement, acceptance criterion, interface contract, or explicitly recorded developer intention. It is measured as the proportion of generated assertions with a verified trace link, where verification means that a reviewer or an automated check has confirmed that the asserted behaviour is stated in the linked source. Its failure signal is a high share of assertions whose only justification is observed behaviour. Requirements engineering standards and traceability reference models supply the link types and the lifecycle for maintaining them (ISO/IEC/IEEE, 2018b; Ramesh et al., 2001; Cleland-Huang et al., 2012).

Test determinism (T) is the degree to which a generated test gives the same verdict on the same code and environment, and the degree to which the generation process itself can be reproduced. It is measured by the observed failure rate over n reruns with shuffled order and varied parallelism, and by recording the model identifier, version, decoding parameters, and prompt hash for

each generated test. Provenance matters because the reason a test exists is easily lost as ownership passes between people. Its failure signal is intermittent failure without code change (Atakpa et al., 2023).

Adversarial fault sensitivity (A) is the ability of generated tests to detect faults that the existing suite misses. It is measured by the marginal mutation score on changed code, supplemented where possible by detection of historical real faults replayed from version history. Its failure signal is a coverage gain with near-zero marginal kill. A maintenance counter-measure, the failure rate of generated tests after behaviour-preserving refactoring, guards against the brittle tests that mutation feedback can induce (Ammann et al., 2016; Zhu et al., 1997).

Security containment (S) is the degree to which code, data, and execution privileges exposed during generation and test execution are limited to what the task requires. It is measured by the proportion of prompts passing classification and redaction controls, the number of secrets or personal data items detected in outbound context, the privilege scope of the generation agent's identity, and the outcome of periodic injection simulations. Its failure signal is sensitive content in prompts or agent actions outside the declared scope (Nwakamma et al., 2024b).

Endorsement (E) is the degree to which a named, qualified person has reviewed a generated test together with its validity evidence and accepted accountability for its admission. It is measured by the share of generated tests in critical modules with a recorded endorser, review time per test, and each endorser's catch rate on audited injected defects. Its failure signal is approval faster than the time needed to check a trace link, or a catch rate below an agreed floor (Ladapo et al., 2022).

Table 3 operationalises the five constructs with their measures, illustrative thresholds, failure signals, and gating actions. The thresholds are starting points to be calibrated against each organisation's baseline rather than universal values, and each is attached to an owner and a response. Figure 3 shows the structure of the framework: four evidence gates between the generator and the protected test suite, all enclosed by the containment boundary, together with the moderators, the outcomes, and the feedback path.

**Table 3. GATES constructs, measures, illustrative thresholds, failure signals, and gating actions**

| Construct | Measure | Illustrative threshold | Failure signal | Gating action |
|---|---|---|---|---|
| G Grounding in intent | Share of generated assertions with a verified link to a requirement, contract, or recorded intent | At least 0.9 in critical modules; at least 0.6 elsewhere | Assertions justified only by observed output | Regenerate with requirement context or rewrite the expected value manually |
| T Test determinism | Failures over n shuffled reruns; completeness of provenance record | Zero failures in n reruns, with n derived from tolerated flaky rate; full provenance | Intermittent failure without code change; missing model version | Reject at admission; time-limited, owned quarantine in operation |
| A Adversarial fault sensitivity | Marginal mutation score on changed code; failure rate after behaviour-preserving refactoring | At least one unique kill per admitted test; refactoring failures below agreed ceiling | Coverage gain with no unique kills; brittleness after refactoring | Prune redundant tests; strengthen or relax assertions deliberately |
| S Security containment | Redaction pass rate; sensitive items in outbound prompts; agent privilege scope; injection simulation results | Zero unredacted secrets; least-privilege identity; all simulations contained | Sensitive context transmitted; agent action outside declared scope | Block request; rotate exposed secrets; restrict agent tools |
| E Endorsement | Share with named endorser; review time per test; catch rate on injected invalid assertions | All tests in critical modules endorsed; catch rate above agreed floor | Approval faster than a trace check; low catch rate | Reassign review; reduce batch size; targeted reviewer training |

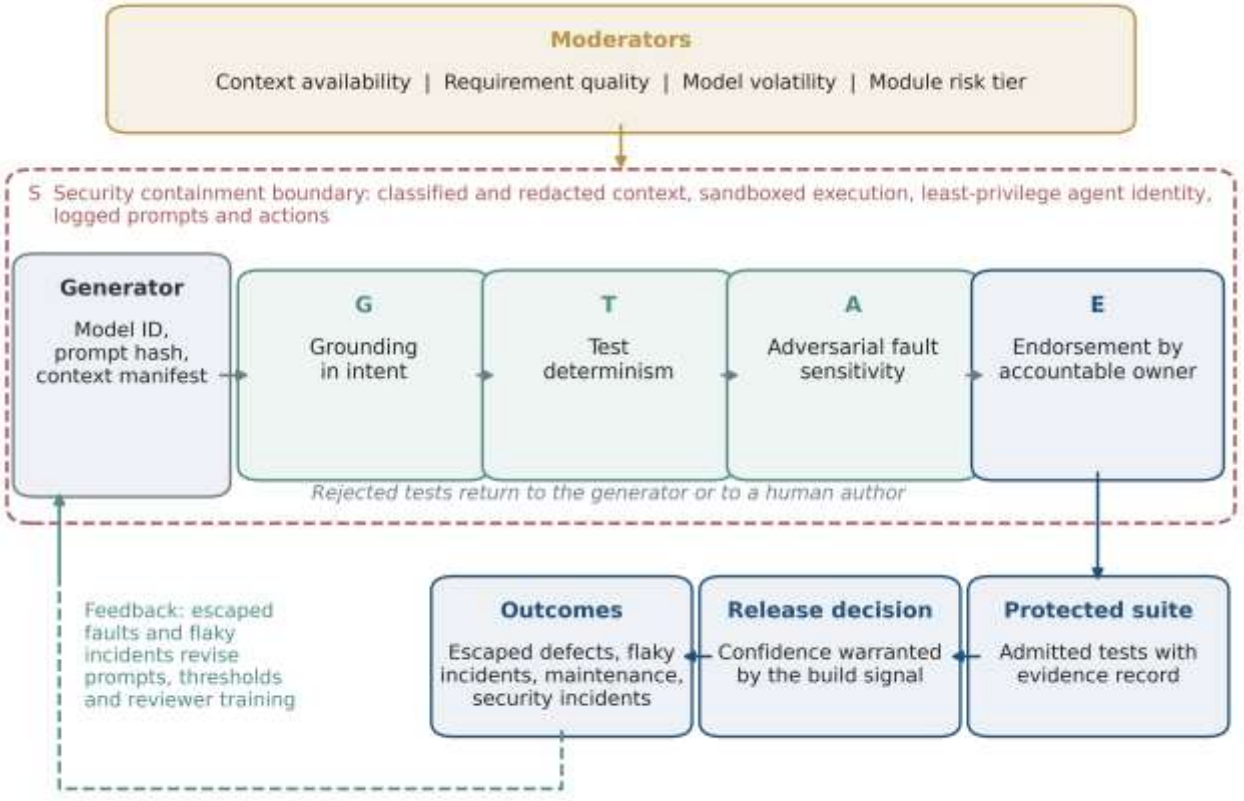


**Figure 3. The GATES framework: evidence gates between the generator and the protected suite, the containment boundary, moderators, and the feedback path**

### 5.2 Causal Pathway, Moderators, and Trade-offs

The framework makes a causal claim: the five constructs mediate the relation between adopting a generative testing tool and the outcomes that matter, namely escaped defects, release confidence that the evidence actually warrants, maintenance effort, and security incidents. Adoption affects those outcomes only through the constructs.

The constructs interact and trade off. Traceability and fault sensitivity are complementary: tracing guards against assertions that check the wrong thing, and mutation guards against assertions that check too little. Determinism and fault sensitivity are in tension, because the simplest way to remove a flaky assertion is to weaken it, which lowers hits along with false alarms, as panel (b) of Figure 2 illustrates. Containment and generation quality are in tension, because less context sent to the model means fewer leaks but more structural hallucination and lower compilation rates. Endorsement and throughput are in tension, because careful review admits fewer tests per hour. These are properties of the problem rather than defects of the framework, and multi-objective reporting that discloses each trade-off is preferable to a single composite score that hides it.

Four moderators condition the pathway, as shown at the top of Figure 3. Context availability, the degree to which relevant types, documentation, and requirements can be supplied to the generator, raises traceability and compilation. Requirement quality, including whether non-functional requirements are stated operationally, bounds the traceability that can be achieved. Model volatility, the frequency of provider-side updates, lowers reproducibility. Module risk tier, the consequence of a missed fault, determines how much evidence endorsement should demand. The last moderator reflects the general principle that governance intensity should scale with potential harm, a principle also embedded in risk-tiered AI regulation (Ominyi et al., 2024b).

A worked scenario makes the pathway concrete; its figures are illustrative rather than empirical. A payments team uses an LLM assistant to add unit tests to a currency rounding module. The assistant generates 40 tests, of which 34 compile and pass on first execution, raising branch coverage of the module considerably. Under GATES, reruns reveal two tests that fail intermittently because they assert on the string form of a hash map, and they are rejected. Mutation analysis shows that 20 of the remaining 32 tests kill no mutant that the human suite did not already kill, and they are pruned as redundant. Of the remaining 12, tracing finds three that assert a rounding result for negative amounts that the specification defines differently: the implementation contains a fault, and the tests had captured it. The endorser, a developer who knows the rounding rules, corrects the three assertions, which

now fail and expose the fault. The team admits 12 tests, fixes one defect, and records the attrition from 40 to 12 as the expected cost of evidence. A pipeline that counted passing tests would have admitted 34 and certified the fault. Structured improvement methods recommend recording such attrition so that it informs the next cycle.

### 5.3 Propositions

Six propositions follow from the synthesis. Each is stated so that a specified result would count against it.

P1 (oracle capture; G). When generated tests are produced with the implementation in the prompt, the proportion of assertions that pass on a faulty version and fail on its fix will be higher than when tests are produced from the signature and requirement alone. Test: generate suites under both conditions for real fault and fix pairs drawn from projects created after the model's training cut-off, and compare capture rates. The proposition is refuted if capture rates do not differ or are lower with the implementation present (Fraser et al., 2011; Borah et al., 2022).

P2 (decoupling of coverage and fault sensitivity; A). Across modules, the correlation between the coverage gain from generated tests and their marginal mutation score will be weaker than the corresponding correlation for human-written tests. Test: compute both correlations across modules matched on coverage gain. The proposition is refuted if the correlations do not differ or if the correlation for generated tests is stronger (Zhu et al., 1997; Ammann et al., 2016).

P3 (single-run admission; T). Generated tests admitted after a single execution will show a higher flaky-failure incidence over the following 90 days of continuous integration than generated tests admitted after at least 30 reruns under shuffled order. Test: randomise the admission policy across repositories or time blocks and compare flaky failures per thousand executions. The proposition is refuted if no difference appears (Atakpa et al., 2023).

P4 (evidence at review; E). Presenting marginal mutation evidence and trace status alongside a generated test will raise reviewers' detection of injected invalid assertions relative to presenting the test alone, without a proportional increase in review time. Test: a within-subject experiment in which professional developers review matched test sets with and without evidence panels. The proposition is refuted if detection does not rise or if review time rises in proportion (Ladapo et al., 2022).

P5 (cost of containment; S). Redacting sensitive context and restricting retrieval to code classified as safe to share will reduce the compilation and pass rates of generated tests, but will not reduce the marginal mutation score among the tests that compile. Test: compare generation under full and redacted context on the same targets. The proposition is refuted if the marginal score among compiled tests falls materially under redaction, which would mean that containment costs evidential value and not only throughput (Aliliele et al., 2025b; Nwakamma et al., 2025).

P6 (feedback learning; all constructs). Teams that route escaped defects and flakiness incidents back into prompt templates, thresholds, and reviewer training will show a declining rate of invalid generated tests over successive quarters, whereas teams without that feedback path will show a flat or rising rate as model versions change. Test: a longitudinal comparison using interrupted time series around the introduction of the feedback path. The proposition is refuted if the trends do not diverge.

The propositions are untested. They are derived from the evidence and theory set out above, and their value lies in specifying which data would show the framework to be wrong. Staged capability claims gain credibility only when each stage is tied to observable criteria of this kind (Wieringa, 2014).

## 6 Evaluation Design and Threats to Validity

A framework that cannot fail is not a scientific claim. This section specifies how GATES could be evaluated in practice and which threats would compromise that evaluation. The design follows guidance for case study and experimental research in software engineering (Runeson et al., 2009) and uses a staged structure in which each stage produces the evidence the next stage needs. It is intended for organisations that already operate continuous integration with version history, so that real-fault replay and longitudinal comparison are feasible, and it assumes ethical approval for observing developer behaviour wherever reviewer audits are involved.

### 6.1 A Staged Evaluation Design

Stage 1 establishes a baseline. For a sample of repositories, the team records the mutation score of the current human-written suite on changed code, its flakiness rate over the preceding quarter, the escaped-defect rate per release, and review time per test. Concurrent changes such as framework migrations should be logged so that they are not later attributed to the intervention.

Stage 2 is an offline benchmark on held-out code. Generation runs on code written after the model's training cut-off, or on internal code never exposed publicly, under the prompt conditions required by P1 and P5. Each generated test passes through the gating pipeline shown in Figure 4: compilation, execution on the reference version, rerun screening, mutation gating, trace verification, and endorsement, with containment checks applied to prompts and execution at every stage. Attrition at each gate is recorded,

producing an evidence profile of the generator on the organisation's own code rather than on public benchmarks. Historical faults are replayed by checking out the faulty version, generating tests, and running them against the fix, so that capture can be measured directly.

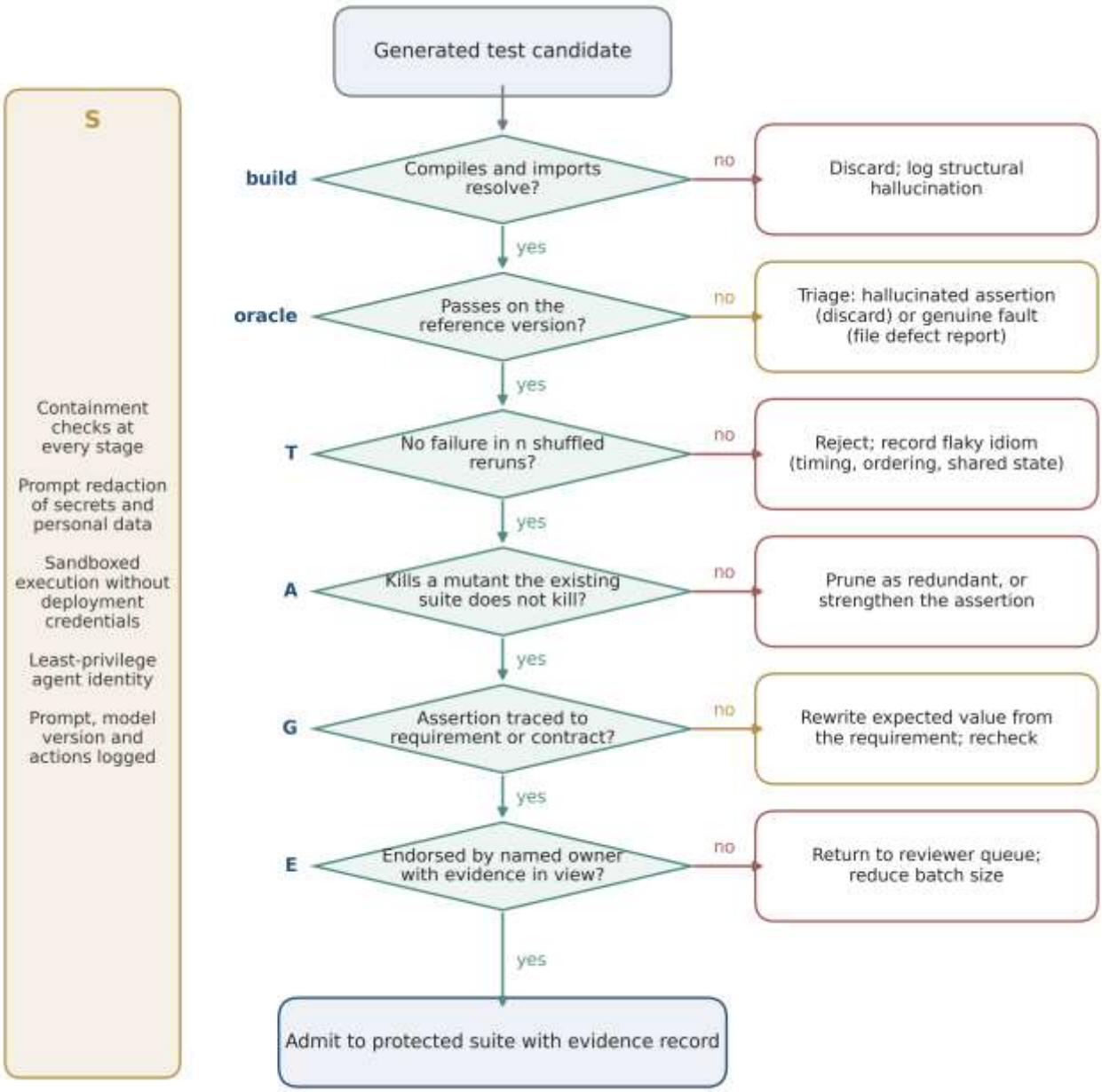


**Figure 4. Admission pipeline for generated tests, showing the decision at each gate, the action on failure, and the containment checks applied throughout**

Stage 3 is a controlled deployment. Repositories or teams are assigned to admission policies, at random where possible; where randomisation is infeasible, a stepped-wedge design introduces the policy to groups in sequence so that each group serves as its own control before introduction. Difference-in-differences and interrupted time series analyses estimate effects on flaky-failure incidence, escaped defects, and maintenance effort. Process tracing through review logs confirms whether the mechanism operated: whether reviewers opened the evidence panel, whether trace links were verified, and whether rejected tests were regenerated or rewritten. Apparent effects often shrink once implementation fidelity is measured, so fidelity should be recorded alongside outcomes.

Stage 4 is sustained monitoring. After deployment the constructs become operational indicators with owners, thresholds, and review cycles, and every model version change triggers re-baselining of the Stage 2 profile. Process mining of pipeline event logs can verify that each admitted generated test passed each gate in the prescribed order, providing an auditable record of the admission lifecycle. Real-time indicator tracking is useful only when each threshold breach triggers a defined response; otherwise it produces dashboards without decisions. Benefits depend on sustained adherence rather than initial uptake, and review discipline in particular tends to decay once the novelty of a new tool has faded.

Analysis should report effect sizes with uncertainty intervals, null and adverse findings, and the sensitivity of conclusions to threshold choices, against preregistered primary outcomes. Cross-sectional comparisons between adopting and non-adopting teams cannot separate cause from selection.

### 6.2 Threats to Validity

Table 4 lists the principal threats to the validity of an evaluation of GATES with a mitigation for each. Four threats are specific to generative testing and deserve discussion. The first concerns construct validity of the mutation score as a proxy for real fault detection. Mutation operators may not represent the faults that matter in a given domain, and equivalent mutants bias scores downward. Real-fault replay from version history provides a partial check, and the correspondence between mutation and real-fault results should be reported rather than assumed (Ammann et al., 2016).

The second is contamination as a threat to external validity. If evaluation code or its tests appeared in training data, Stage 2 results overstate performance on genuinely new code. The mitigation, using post-cut-off or private code, is itself imperfect, because providers rarely disclose cut-offs precisely and new code may still resemble memorised patterns (Runeson et al., 2009). The third is model drift as a threat to internal validity. Hosted models change without notice, so a difference between conditions measured weeks apart may reflect a model update rather than the manipulation; pinning versions, logging the model identifier for each generated test, and running conditions concurrently reduce this threat (Nwakamma et al., 2025). The fourth is reactivity in reviewer studies. Developers who know that they are audited with injected defects may review more carefully than usual, inflating catch rates. Low injection rates, delayed disclosure approved by ethics review, and comparison with unaudited periods mitigate the effect.

Conclusion validity is threatened by the small number of repositories most organisations can study and by the clustering of tests within modules, so multilevel models and cluster-robust uncertainty are required. Ethical validity concerns the monitoring of individual reviewers, which can slide into surveillance; aggregated reporting, and use of audit results for training rather than appraisal, address this concern.

**Table 4. Threats to the validity of a GATES evaluation and their mitigation**

| Validity type | Specific threat | Consequence | Mitigation |
|---|---|---|---|
| Construct | Mutation operators unrepresentative of domain faults; equivalent mutants | Mutation score misstates real fault detection | Real-fault replay; report mutation to real-fault correspondence |
| Construct | Trace links verified superficially | Traceability overstated | Second-reviewer check on a sample; defined link types |
| Internal | Provider-side model update between conditions | Model drift mistaken for treatment effect | Pin versions; log model identifier per test; run conditions concurrently |
| Internal | Concurrent framework migration or staffing change | Confounded effects on flakiness and defects | Log concurrent changes; stepped-wedge design; difference-in-differences |
| External | Benchmark contamination through training data | Performance overstated on unseen code | Post-cut-off or private code; near-duplicate detection against public tests |
| External | Open-source projects unlike enterprise codebases | Results fail to transfer | Stratify by dependency depth, documentation quality, and language |
| Conclusion | Few repositories; tests clustered within modules | Overconfident estimates | Multilevel models; cluster-robust intervals; report null results |
| Ethical | Reviewer audits used for individual appraisal | Surveillance; gaming of audits | Aggregate reporting; audit results used for training only |

## 7 Discussion

The review set out to explain when generated test artefacts count as assurance evidence. The answer is conditional and the conditions are specific: independent intent, verified determinism, marginal fault sensitivity, contained exposure, and accountable endorsement. This section draws out what that answer means for theory and for practice and engages with the strongest objections to it. The discussion is confined to generative models producing test artefacts; questions about prioritising or minimising existing suites raise different evidential issues and are addressed only where they intersect with generation (Li et al., 2007). The section also asks whether the framework would remain necessary if models became substantially more capable, a question taken up in Section 7.3.

### 7.1 Theoretical Implications

Three theoretical implications follow. First, the signal detection framing turns the oracle problem from a qualitative concern into an estimable quantity for generative tools. Mutation analysis estimates hits, reruns estimate stochastic false alarms, and trace verification estimates the share of assertions whose criterion is anchored to intent. Together they allow a generated suite to be described by sensitivity and criterion rather than by pass rate, and they predict that pass-rate optimisation will shift the criterion toward laxity without raising sensitivity (Ammann et al., 2016).

Second, the review extends traceability theory to machine-generated artefacts. Traceability research developed to connect human-authored requirements, designs, code, and tests (Gotel et al., 1994; Ramesh et al., 2001). Generative tools create artefacts whose provenance includes a model, a prompt, and a context set, none of which exist in classical trace models. GATES treats the model identifier, prompt hash, and context manifest as trace objects, which makes regeneration auditable and connects the reproducibility of generated evidence to established traceability practice (Cleland-Huang et al., 2012; Borg et al., 2014).

Third, the review qualifies sociotechnical accounts of automation bias for a setting in which the automated output is itself a check. In most automation bias research the person reviews a decision; here the person reviews a detector whose failures are silent. That difference predicts that automation bias in testing will be less visible and more persistent than in decision support, because an error in a test is revealed only when a fault escapes, often long after review. A remedy borrowed from safety practice is to treat each escaped fault as a near miss: an event to be traced back to the review that admitted the test that should have caught it.

### 7.2 Practical Implications

For engineering leaders, the first implication is to change what is counted. The number of generated tests and the coverage they add should be reported only alongside their marginal mutation score, flakiness record, and trace status. A dashboard showing coverage deltas without these companions invites the substitution of production for validity that this review has documented.

The second implication concerns where generation is used. The evidence favours generative tools for tasks with strong external oracles: generating inputs for property-based tests whose properties are written by people, drafting tests from explicit acceptance criteria, and producing scaffolding such as fixtures, mocks, and builders against which people then write assertions. It is weakest for inferring expected values for existing code without independent intent, which is the default mode of most assistants. Teams can capture much of the productivity benefit while avoiding most of the capture risk by assigning the oracle to people or to requirements and the scaffolding to the model (Myers et al., 2011).

The third implication concerns security architecture. Test generation should be treated as a data flow crossing a trust boundary, with classification, redaction, and logging at that boundary, and generated code should run in ephemeral sandboxes without deployment credentials. Agentic tools should hold least-privilege identities and should be unable to disable or delete tests without human approval, because removing a failing test is the simplest way for an agent to make a pipeline pass (Nwakamma et al., 2024b).

The fourth implication concerns people. Review authority should be tiered by module risk, reviewers should receive feedback on their audited catch rate, and junior developers should continue to write some oracles by hand so that the skill needed to review generated oracles is maintained. Readiness assessment before rollout should cover skills, workflow, and incentives, because incentives shape which outcomes a programme optimises. Diverse review teams are more likely to notice assertions that encode unstated assumptions about users and inputs.

Finally, regulated organisations should map GATES evidence to their obligations. Test process standards require documented test design and traceable results (ISO/IEC/IEEE, 2018a), and AI regulation increasingly requires documentation of automated system use and of human oversight in high-risk contexts (Ominyi et al., 2024b). The provenance and endorsement records specified by the framework serve both purposes and support the shift from periodic audit to continuous control.

### 7.3 Counter-arguments and Boundary Conditions

Three objections deserve a direct answer. The first is that the framework is too costly: mutation analysis, reruns, and trace verification consume compute and reviewer time that erode the productivity gains that motivated adoption. Mutation analysis restricted to changed code with sampled operators fits most continuous integration budgets, and rerun screening applies only at admission. More fundamentally, the relevant comparison is between validated generated tests and the realistic alternative, which is either human-written tests or unvalidated generated tests whose hidden costs arrive later as captured faults and flaky noise. Lifecycle evaluation, which counts operating and failure costs rather than acquisition cost alone, supports this reasoning.

The second objection is that models are improving quickly and the failure modes documented here will disappear. Some will diminish; structural hallucination, for instance, falls with better context handling. Regression-oracle capture, however, is not a capability limitation. It follows from the information available to the model: if the implementation is the only evidence of intent, a better model will infer the implemented behaviour more reliably, not less. Capture is therefore expected to persist, or even strengthen, as models improve unless independent intent is supplied. Containment risks likewise grow with capability, because more capable agents are given more tools and more autonomy (Nwakamma et al., 2024b; Dosunmu et al., 2026b).

The third objection is that mutation scores and trace links are themselves imperfect, so gating on them replaces one weak proxy with another. The objection is partly correct, and the framework answers it by triangulation rather than reliance on any single measure: an assertion that traces to a requirement, kills a mutant not otherwise killed, and survives reruns is more credible than one satisfying any single check. Multi-indicator approaches are standard wherever single proxies are known to be gameable.

Boundary conditions limit where GATES applies. It assumes that some statement of intent exists or can be written; in exploratory development without specifications, the traceability construct degrades to recorded developer intent, which is weaker. It assumes that a mutation tool exists for the language and that builds are fast enough to support reruns; embedded and hardware-in-the-loop testing, where each run is expensive, may not meet this assumption. Continuous integration for embedded IoT development handles the same constraint by moving tests up a fidelity ladder from simulation toward physical hardware, reserving expensive runs for checks already validated at cheaper levels (Oyesiji et al., 2026). Organisations lacking these preconditions should build them before relying on generated tests as evidence.

## 8 Limitations and Future Research Agenda

This review has limitations that bound its conclusions. The literature on LLM-based testing is young and fast-moving; many evaluations appear first as preprints, use closed models whose versions are later retired, and report on public benchmarks where contamination cannot be excluded. The synthesis therefore gives more weight to mechanisms, which are stable across model versions, than to reported effect magnitudes, which are not. The review also drew on adjacent domains for analogies about detection, oversight, and quality management; those analogies generate expectations rather than evidence about software testing, and they may fail where the domains differ. Integrative reviews involve judgement in inclusion and coding, and although the protocol and appraisal questions were fixed in advance, a second independent coder would strengthen reliability (Page et al., 2021).

Five lines of research follow. First, capture studies should generate tests against faulty versions drawn from projects created after model cut-offs and measure how often generated assertions certify the fault under different prompt conditions (P1). Such studies need shared, versioned datasets of fault and fix pairs refreshed as new models are released.

Second, oracle-strength metrics beyond mutation need development. Candidates include checked coverage, the proportion of executed statements whose effects reach an assertion; assertion-level mutation contribution; and coverage of metamorphic relations. Their correspondence with real-fault detection should be established on the same datasets, with uncertainty reported rather than point estimates alone (Zhu et al., 1997).

Third, reviewer behaviour under different evidence presentations should be studied experimentally with professional developers, including the effect of evidence panels on detection of injected invalid assertions (P4) and the long-term effect of generative assistance on oracle design skill. Longitudinal designs are needed because deskilling, if it occurs, develops over months rather than within a single experiment.

Fourth, security research should characterise indirect prompt injection against test generation agents specifically, including injection through fixtures, dependencies, and failing-test output, and should evaluate sandboxing and privilege controls under red-team conditions. The containment cost hypothesis (P5) is especially important for organisations weighing hosted against local models (Nwakamma et al., 2025; Ojukwu et al., 2025).

Fifth, organisational studies should examine how incentives shape adoption, for example whether teams evaluated on coverage adopt generated tests differently from teams evaluated on escaped defects, and whether procedural standardisation of the admission workflow improves consistency without slowing delivery.

## 9 Conclusion

Generative models have made test artefacts cheap to produce, and in doing so have exposed a distinction that testing practice could previously leave implicit: the difference between a test that runs and a test that constitutes evidence. This review has shown that the difference is measurable. Assertions can be traced to independent statements of intent; their fault sensitivity can be estimated by marginal mutation analysis; their determinism can be screened by reruns sized to a tolerated flakiness rate; the exposure created by generation can be contained through classification, redaction, and sandboxing; and their admission can be endorsed by accountable reviewers who see the evidence. GATES organises these checks into a framework with explicit measures, gating actions, six falsifiable propositions, and a staged evaluation design. Its central claim is that regression-oracle capture is a structural consequence of generating assertions from implementations, and that it persists as models improve unless intent is supplied from outside the code. Organisations that adopt generative testing on these terms can expect fewer but stronger tests, and a build signal that deserves the confidence placed in it.

**Funding:** This research received no specific grant from any funding agency in the public, commercial, or not-for-profit sectors.

**Competing interests:** None declared.

**Data availability:** No new data were generated or analysed in this study; all sources are cited in the reference list.

## References

Aliliele, C., Mbonu, I. S., Uzoka, E., & Iwuanyanwu, U. (2025b). Advances in data lakehouse governance architectures for enterprise data loss prevention and compliance assurance. Shodhshauryam, International Scientific Refereed Research Journal, 8(4), 171-213. https://doi.org/10.32628/SHISRRJ258474

Ammann, P., & Offutt, J. (2016). Introduction to Software Testing (2nd ed.). Cambridge University Press.

Atakpa, M. I., & Fobellah, A. N. (2023). Anomaly detection in financial time-series data: A conceptual model for healthcare and banking applications. International Journal of Scientific Research in Computer Science, Engineering and Information Technology, 9(2), 967-997. https://doi.org/10.32628/CSEIT2342441

Borah, S., Aliliele, K. C., Rakshit, S., & Vajjhala, N. R. (2022). Applications of artificial intelligence in software testing. In P. K. Mallick et al. (Eds.), Cognitive informatics and soft computing (Lecture Notes in Networks and Systems, Vol. 375, pp. 727-736). Springer. https://doi.org/10.1007/978-981-16-8763-1_60

Borg, M., Runeson, P., & Ardö, A. (2014). Recovering from a decade: A systematic mapping of information retrieval approaches to software traceability. Empirical Software Engineering, 19, 1565-1616. https://doi.org/10.1007/s10664-013-9255-y

Cleland-Huang, J., Gotel, O., & Zisman, A. (Eds.). (2012). Software and Systems Traceability. Springer. https://doi.org/10.1007/978-1-4471-2239-5

Daka, E., & Fraser, G. (2014). A survey on unit testing practices and problems. IEEE International Symposium on Software Reliability Engineering Workshops, 201-211. https://doi.org/10.1109/ISSREW.2014.11

Dosunmu, A. A., & Ogundele, P. O. (2026b). Integrated threat intelligence automation and simulation models for next generation cyber defense. Engineering and Technology Journal, 11(1), 8451-8462. https://doi.org/10.47191/etj/v11i01.05

Fraser, G., & Arcuri, A. (2011). EvoSuite: Automatic test suite generation for object-oriented software. ESEC/FSE, 416-419. https://doi.org/10.1145/2025113.2025179

Gotel, O. C. Z., & Finkelstein, A. C. W. (1994). An analysis of the requirements traceability problem. IEEE International Conference on Requirements Engineering, 94-101. https://doi.org/10.1109/ICRE.1994.292398

Hevner, A. R., March, S. T., Park, J., & Ram, S. (2004). Design science in information systems research. MIS Quarterly, 28(1), 75-105. https://doi.org/10.2307/25148625

ISO/IEC/IEEE. (2018a). ISO/IEC/IEEE 29119-2 Software and systems engineering: Software testing processes. International Organization for Standardization.

ISO/IEC/IEEE. (2018b). ISO/IEC/IEEE 29148 Systems and software engineering: Life cycle processes and requirements engineering. International Organization for Standardization.

Ladapo, O. O., Dosunmu, A. A., Jooda, D., & Abolaji, T. O. (2022). Human-in-the-loop machine learning: A state of the art. Journal of Frontiers in Multidisciplinary Research, 3(1), 656-669. https://doi.org/10.54660/.JFMR.2022.3.1.656-669

Li, Z., Harman, M., & Hierons, R. M. (2007). Search algorithms for regression test case prioritization. IEEE Transactions on Software Engineering, 33(4), 225-237. https://doi.org/10.1109/TSE.2007.38

Mbonu, I. S., Aliliele, C., Iwuanyanwu, U., & Uzoka, E. (2021b). 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), 468-496. https://doi.org/10.32628/CSEIT217565

Myers, G. J., Sandler, C., & Badgett, T. (2011). The Art of Software Testing (3rd ed.). Wiley.

Nwakamma, S., Ojukwu, J., & 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), 185-216.

Nwakamma, S., Ojukwu, J., & 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), 217-248. https://doi.org/10.56201/wjimt.v8.no6.2024.pg217.248

Nwakamma, S., Ojukwu, J., & Oyesiji, S. O. (2025). On-device multimodal small language models for privacy-preserving edge intelligence through quantization, pruning, distillation, and energy-efficient inference. World Journal of Innovation and Modern Technology, 9(12), 271-302. https://doi.org/10.56201/wjimt.v9.no12.2025.pg271.302

Odejobi, O. D., Okonkwo, C. S., Ahiaeke Patrick, M. C., Okeke, O. T., & Mayo, W. (2025). AI-augmented secure software engineering: Leveraging deep learning for autonomous threat detection and mitigation. International Journal of Engineering and Modern Technology, 11(12), 101-121. https://doi.org/10.56201/ijemt.vol.11.no12.2025.pg101.121

Ojukwu, J., Oyesiji, S. O., & 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), 232-263. https://doi.org/10.56201/ijcsmt.vol.11.no12.2025.pg232.263

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

Oyesiji, S. O., Anunagba, C. O., Nwakamma, S., & Sankara, H. W. Y. C. (2026). Automated validation in embedded IoT development: A review of continuous integration and delivery across the fidelity ladder. Journal of Engineering Research and Reports, 28(9), 145-157. https://doi.org/10.9734/jerr/2026/v28i92005

Oyesiji, S. O., Nwakamma, S., & Ojukwu, J. (2024). A conceptual framework for context-aware data curation in fine-tuning pipelines. International Journal of Engineering and Modern Technology, 10(11), 197-228. https://doi.org/10.56201/ijemt.v10.no11.2024.pg197.228

Page, M. J., et al. (2021). The PRISMA 2020 statement: An updated guideline for reporting systematic reviews. BMJ, 372, n71. https://doi.org/10.1136/bmj.n71

Ramesh, B., & Jarke, M. (2001). Toward reference models for requirements traceability. IEEE Transactions on Software Engineering, 27(1), 58-93. https://doi.org/10.1109/32.895989

Runeson, P., & Höst, M. (2009). Guidelines for conducting and reporting case study research in software engineering. Empirical Software Engineering, 14, 131-164. https://doi.org/10.1007/s10664-008-9102-8

Wieringa, R. J. (2014). Design Science Methodology for Information Systems and Software Engineering. Springer. https://doi.org/10.1007/978-3-662-43839-8

Zhu, H., Hall, P. A. V., & May, J. H. R. (1997). Software unit test coverage and adequacy. ACM Computing Surveys, 29(4), 366-427. https://doi.org/10.1145/267580.267590