Journal of
Cyber Security

Tech Science Press

https://doi.org/10.32604/jcs.2026.085438

ARTICLE

Shift-Left Security for AI-Generated Code: Detecting and Preventing Vulnerabilities at Build-Time

Bala Thripura Akasam*

Tapestry, Inc., New York, NY, USA

*Corresponding Author: Bala Thripura Akasam. Email: balathripuraa@gmail.com

Received: 11 May 2026; Accepted: 23 July 2026; Published: 21 August 2026

ABSTRACT: The widespread adoption of Artificial Intelligence (AI) coding assistants across enterprise software development teams has accelerated delivery velocity while simultaneously introducing a persistent and empirically documented security quality gap in the code these tools produce. Vulnerability classes including insecure output handling, prompt injection constructs, sensitive information disclosure patterns, and cryptographic misuse appear at elevated rates in AI-generated output regardless of model advancement, while organizational governance frameworks have failed to keep pace with the speed of AI tool deployment, creating conditions in which vulnerable code reaches production through informal risk acceptance rather than accountable remediation processes. The conventional shift-left security practice of running static application security testing earlier in the Continuous Integration pipeline (CI pipeline) is insufficient to address these conditions, as AI-amplified code volumes overwhelm static analysis triage capacity and produce finding sets that lack the runtime exploitability context needed to distinguish genuine risk from theoretical noise. This article develops a structured, research-backed blueprint for build-time security controls tailored specifically to AI-generated code, organized across three interdependent layers: a pre-pull-request policy and governance layer; a build-time detection layer combining AI-augmented and traditional static analysis, Application Programming Interface (API) schema validation, automated Software Bill of Materials (SBOM) enforcement, and pre-deployment runtime simulation; and a prioritization and feedback layer applying application security posture management correlation to convert detection volume into developer-actionable risk reduction. An evaluation framework centered on precision and recall per vulnerability class, True-Exploit Rate, mean time to remediate, supply-chain integrity coverage, and developer experience indicators provides the measurement infrastructure needed to demonstrate risk reduction outcomes rather than detection counts.

KEYWORDS: Shift-left security; AI-generated code vulnerabilities; software bill of materials; application security posture management; DevSecOps build-time controls

1 Introduction

The integration of generative AI into software development has moved from novelty to operational norm. Across enterprises of every scale, AI coding assistants now contribute directly to production codebases, generating utility functions, implementing business logic, constructing API integrations, and accelerating feature delivery at rates that were not achievable through manual authorship alone. Developer productivity gains are measurable and commercially significant: AI-assisted teams iterate faster, reduce time spent on repetitive scaffolding, and lower the expertise barrier for working across unfamiliar languages and frameworks. These advantages have driven rapid, widespread adoption, and they have done so faster than the security governance structures needed to manage the accompanying risk. As AI-generated code becomes an

increasingly large share of every organization's software inventory, the security properties of that code have become a first-order engineering concern rather than a downstream quality consideration [1].

The security consequence of this adoption curve is now empirically documented and no longer speculative. A large-scale industry survey of AI-generated code across production environments suggests that code produced with AI assistance exhibits a materially higher rate of security weaknesses than code authored without it, with vulnerabilities concentrated in well-understood and long-cataloged weakness classes [1]. A controlled benchmark evaluation spanning more than 100 large language models across four programming languages found that AI-generated code introduced exploitable security flaws in 45% of coding tasks, with security pass rates remaining flat across successive model generations rather than improving as models scale [2]. These industry-reported patterns are corroborated by peer-reviewed empirical studies including Perry et al. [3] and Pearce et al. [4]. The vulnerability classes appearing at elevated rates in AI output are not obscure edge cases; they are the same categories that secure coding disciplines have worked to suppress for decades, reintroduced at scale because AI models optimize for functional plausibility rather than security correctness. Industry expectation has generally assumed that more capable models would close the security quality gap as a byproduct of improved general performance; the empirical record does not support that assumption [3,4].

The organizational response to this documented gap has been inadequate. Enterprise adoption of AI in development workflows has accelerated considerably, yet the governance frameworks needed to manage the security implications of that adoption have not kept pace. Industry survey data indicate that while the majority of development organizations have integrated AI tooling into their delivery pipelines, fewer have established formal policies governing approved tool usage, prompt logging requirements, secret handling constraints, or third-party code provenance obligations for AI-introduced dependencies [5].

The AppSec tooling response has been equally strained. The classic shift-left practice of integrating static application security testing earlier in the CI pipeline was designed and calibrated for human-authored code volumes. AI-accelerated development has disrupted that calibration in two compounding ways. First, the sheer volume of code submitted to analysis pipelines has grown substantially. Second, and more consequentially, static analysis alone cannot determine whether a flagged vulnerability is reachable and exploitable in a running system, which means that larger finding sets do not translate into clearer risk pictures; they translate into larger triage queues with the same proportion of actionable signal [6].

This article develops a practical, research-backed blueprint for build-time security controls specifically tailored to AI-generated code—controls that go beyond detection timing to address detection quality, exploitability context, provenance enforcement, and developer experience at AI-scale volumes. The blueprint is organized around four interdependent pillars. The first characterizes the threat landscape created by AI-generated code and identifies the specific failure modes that conventional shift-left controls do not adequately address. The second defines the design objectives that a credible build-time strategy must meet. The third develops the layered control architecture. The fourth establishes an evaluation framework that measures outcomes in terms of precision, exploitability, mean time to remediate, and developer experience [5,6].

The contribution of this article is a practitioner-oriented reference architecture: a design synthesis that integrates AI-specific threat analysis, layered build-time controls, and an outcome-centered evaluation framework into a coherent, deployable blueprint. This is distinguished from a new empirical study (which would require a controlled experiment) and from a systematic literature review (which would require exhaustive search methodology); it is a design science contribution in the tradition of reference architecture papers in the security engineering literature. The architecture builds on and extends established DevSecOps pipeline frameworks, adding AI-specific threat targeting, a formal threat model, and an exploitability-centered evaluation methodology as its distinguishing elements.

2 Threat Landscape and Governance Deficit in AI-Generated Code

The threat environment surrounding AI-generated code is not a projection of future risk; it is a present and documented reality shaped by the intersection of elevated vulnerability rates, overwhelmed detection tooling, and governance structures that have not kept pace with the speed of AI adoption. The primary risk drivers active in 2025 and into 2026 are empirically grounded, organizationally entrenched, and mutually reinforcing in ways that conventional AppSec practice was not designed to absorb [7].

Threat Model

The following threat model bounds the security architecture developed in Section 3 by defining the relevant assets, adversary classes, trust boundaries, key assumptions, and scope exclusions.

Primary assets in scope include: production source code and associated intellectual property; CI/CD pipeline infrastructure (build agents, secrets management, artifact registries); the software bill of materials and associated provenance records; and developer credentials and access tokens.

Adversary classes addressed by this architecture are: (1) external attackers exploiting AI-introduced vulnerabilities after deployment—the primary risk class targeted by the detection and prioritization layers; (2) supply-chain actors targeting hallucinated or typosquatted dependencies introduced by AI-generated code—addressed by SBOM enforcement and provenance gating; and (3) insider risk arising from ungoverned AI prompt behavior, including inadvertent secret disclosure and the introduction of unapproved dependencies—addressed by the policy and governance layer.

Trust boundaries are: the pre-commit boundary (developer workstation, where pre-commit hooks operate); the repository boundary (PR merge gate, where policy and detection controls enforce); the build boundary (CI artifact generation, where SBOM and attestation gates operate); and the deployment boundary (release promotion gate, where SLSA attestation verification occurs).

Key assumptions: the CI/CD platform itself is trusted and integrity-verified; the runtime simulation environment does not have access to production data; and the AI coding assistant operates under an enterprise license with prompt logging enabled. Scope exclusion: adversarial prompt injection targeting the coding assistant itself is noted as out of scope for this architecture, with this class of risk addressed in the emerging literature on LLM-specific supply-chain attacks.

The vulnerability profile of AI-generated code is both well-documented and concerning in its composition. The established taxonomy of security risks specific to large language model applications identifies insecure output handling, prompt injection, sensitive information disclosure, and the introduction of insecure code constructs as among the most consequential and consistently observed failure modes in LLM-assisted development environments [7]. Insecure output handling occurs when LLM-generated code passes model output into downstream components without adequate validation or sanitization, creating injection pathways structurally identical to classical cross-site scripting and SQL injection vulnerabilities. Cryptographic misuse appears when models reproduce legacy or weak cryptographic patterns from training data rather than applying current secure defaults [7].

The operational consequence of this vulnerability rate is a triage burden of significant proportions that current static analysis infrastructure is not equipped to manage. Peer-reviewed analysis of AI-generated code quality indicates that automated code generation introduces security weaknesses at rates that differ meaningfully from manually authored code, with the degree of risk varying across task type, prompt construction, and the deployment domain [8]. Empirical studies specifically examining AI coding assistant output corroborate these findings across multiple programming languages: Perry et al. [3] found that GitHub Copilot-assisted developers were more likely to introduce security vulnerabilities across C, Python, and Verilog task sets; Pearce et al. [4] reported that a significant proportion of Copilot-generated code suggestions

contained CWE-listed weaknesses when evaluated against security-relevant scenarios; and Siddiq and Santos [9] extended this analysis using a multi-language benchmark dataset, confirming elevated weakness rates in Java, Python, and JavaScript outputs from multiple code generation models. These empirical findings provide the archival grounding for the vulnerability rate claims advanced throughout this article.

The fundamental problem is not detection sensitivity but exploitability context. Static analysis flags potential vulnerabilities based on code structure and pattern recognition; it cannot determine whether a flagged issue is reachable in a running system, whether an exploit path exists given the deployed configuration, or whether compensating controls elsewhere in the stack reduce realized risk. As AI tools accelerate code output across development organizations, finding sets expand proportionally, but the proportion of findings that represent genuine, reachable, and exploitable risk does not increase at the same rate [8]. Developers who receive high-volume, low-precision finding lists lose confidence in AppSec tooling and absorb the habit of treating security findings as background noise, which accelerates the accumulation of security debt precisely in the codebases where AI tools are most actively used.

Organizational governance has not kept pace with AI tool deployment, and this lag is amplifying the technical risk in compounding ways. Development organizations have broadly integrated AI coding assistants into delivery workflows without establishing the formal policy structures needed to govern their security implications. In the absence of declared usage policies, prompt logging requirements, secret handling constraints, and provenance obligations for AI-introduced dependencies, AI tools operate in an accountability vacuum [8]. The practical outcome is that known vulnerabilities progress to production not because they evade detection but because no formal structure exists to halt them.

API-first architectures and supply-chain opacity represent compounding risk vectors. AI-generated code tends to introduce and extend API surfaces rapidly, creating endpoints that may not appear in documented specifications and may not be subject to the authentication and authorization controls applied to known and governed surfaces. At the supply-chain level, the software bill of materials has been established as the foundational instrument for achieving component-level transparency that modern software security requires [10]. CISA has identified the software bill of materials as a critical element of software supply-chain risk management, providing minimum element specifications that define the data fields necessary for a bill of materials to be actionable for security purposes [10].

The cumulative implication of these conditions is that a security strategy adequate for the AI era must target the specific failure modes that LLM-generated code introduces at elevated rates, as catalogued in the established AI application security taxonomy [7]. It must incorporate exploitability context at build-time, enforce provenance continuously through automated SBOM generation and attestation verification, and address the governance gap that allows AI-generated vulnerabilities to move from detection to production without formal accountability [10].

3 A Shift-Left Security Blueprint for AI-Generated Code

A credible build-time security strategy for AI-generated code requires a rearchitected control structure that addresses the specific conditions under which AI-generated vulnerabilities are introduced and propagated (Fig. 1). The foundational principle of shift-left security is that vulnerabilities identified and remediated earlier in the software development lifecycle cost substantially less to fix and carry substantially lower organizational risk than those discovered after deployment [11]. Applied to AI-generated code, this principle demands that detection, exploitability validation, and provenance enforcement all occur within the build pipeline rather than in post-deployment testing cycles.

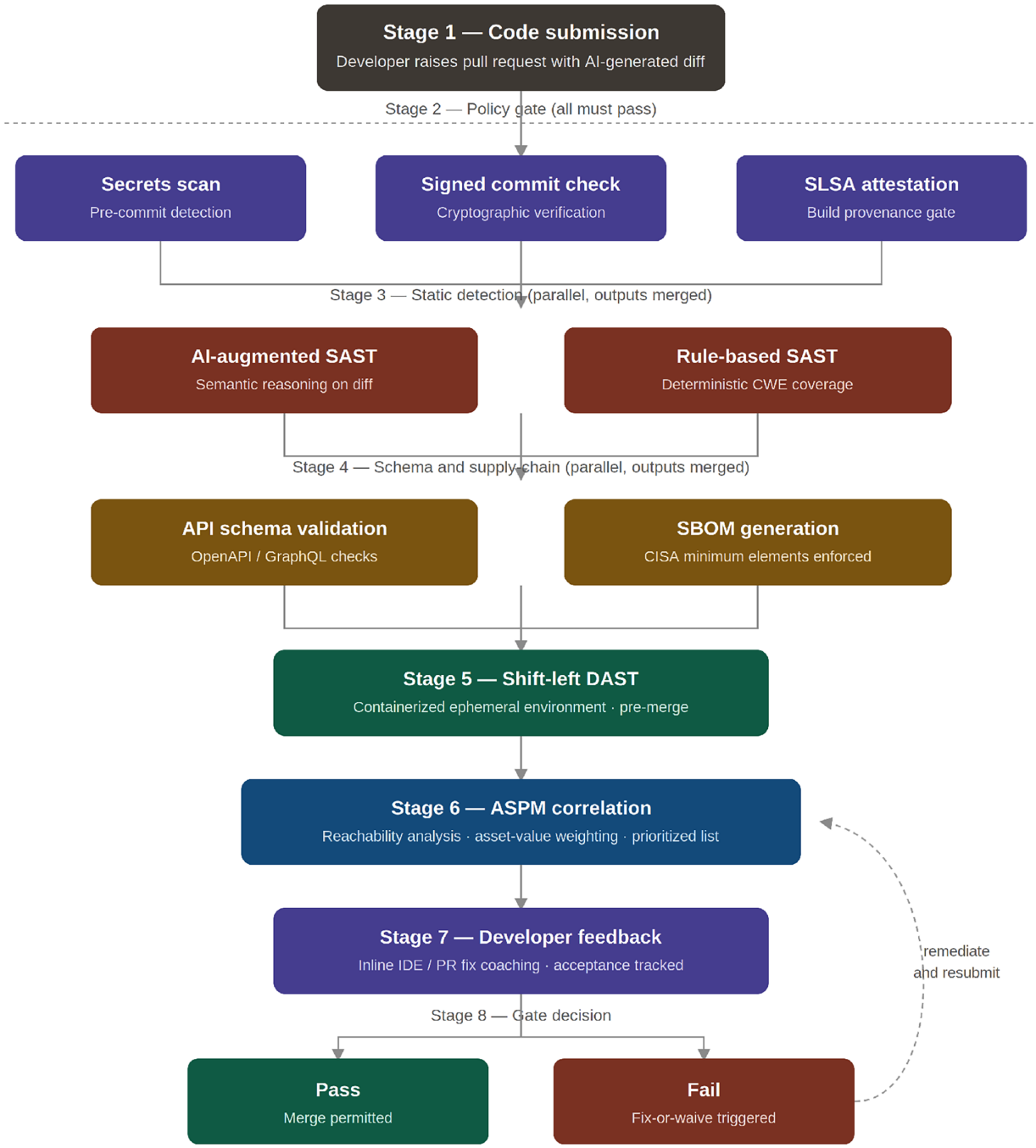

Figure 1: Shift-left security control flow for AI-generated code.

The three-layer architecture (Table 1) achieves its effectiveness through the complementarity of its constituent layers rather than through any single control. The policy layer establishes the accountability preconditions that make detection outputs attributable to identifiable AI tool usage; without it, detection findings cannot be traced to their origin, and remediation remains informal. The detection layer generates the exploitability-filtered finding sets that prioritization requires; without it, the prioritization layer has nothing to rank. The prioritization layer reduces finding volume to a level at which developer action is feasible; without it, detection alone reproduces the alert fatigue problem the blueprint is designed to solve. Residual

risks not addressed by this architecture include: runtime vulnerabilities introduced through post-merge production configuration changes outside the CI pipeline; vulnerabilities in the AI model infrastructure itself; and AI-assisted social engineering that operates outside the code generation surface. Fix-or-waive thresholds should be initially configured as block-on-critical and warn-on-high per CWE class, calibrated to the organization's observed TER baseline, to minimize developer friction during onboarding.

Table 1: Shift-left security control architecture layers, constituent controls, and primary security objectives for AI-generated code environments.

Control Layer	Constituent Controls	Primary Security Objective
Policy and Governance (Pre-PR)	AI usage policy, prompt logging, secret handling rules, SBOM entries for AI-introduced dependencies, signed commits, branch protection, pre-commit secrets scanning, SLSA attestation gates	Establish accountability and provenance obligations before AI-generated code enters shared repositories
Build-Time Detection	AI-augmented SAST, rule-based SAST, PR-level differential scans, fix-or-waive workflows, OpenAPI/GraphQL schema validation, automated SBOM generation (CycloneDX/SPDX), CISA minimum element enforcement, shift-left DAST in containerized ephemeral environments	Detect, validate, and gate AI-specific vulnerability classes and supply-chain risks within the CI pipeline before merge
Prioritization and Feedback	ASPM correlation, reachability-based ranking, asset-value weighting, IDE and PR inline fix coaching, fix acceptance tracking	Convert detection volume into developer-actionable, exploitability-prioritized finding lists that sustain adoption and prevent security debt accumulation

3.1 Policy and Governance (Pre-PR Layer)

The policy and governance layer functions as the prevention tier of the three-layer architecture in the strict sense (Fig. 2): its controls are designed to stop insecure code from being generated, committed, or promoted, rather than to detect it after the fact. Pre-commit secrets scanning, branch protection rules, SLSA attestation gates, and AI usage policy enforcement are prevention controls; SAST, DAST, and SBOM enforcement in the build layer are detection controls. Both categories are addressed in this blueprint, and the title's reference to "preventing" vulnerabilities refers specifically to this pre-PR prevention tier.

The policy and governance layer operates before any code reaches the repository and addresses the most common and consequential failure mode in AI-assisted development: the introduction of insecure code and undocumented dependencies through workflows that carry no formal accountability structure. An engineering-level AI usage policy is the foundational instrument of this layer. Such a policy must enumerate which AI coding tools are approved for use in production-facing development, establish logging and auditability requirements so that AI-assisted contributions can be traced and reviewed, prohibit the inclusion of credentials, secrets, and sensitive configuration data in AI prompts, and mandate that security-relevant

prompt patterns are applied when generating code in domains known to produce elevated vulnerability rates [12].

The provenance obligation within the policy layer is equally critical and frequently overlooked in current practice. AI coding assistants routinely introduce third-party libraries and components as part of generated code, and these introductions often bypass the deliberate dependency review that human-authored code would trigger. Requiring that all third-party components introduced through AI-generated code be documented in machine-readable software bill of materials format using CycloneDX or SPDX ensures that AI-introduced dependencies are subject to the same provenance governance as manually declared ones [10]. This requirement addresses a documented failure mode in which AI coding agents import vulnerable or non-existent open-source packages at elevated rates relative to manually authored dependency declarations, a pattern observed across production dependency graphs in 2025 [13].

Repository hygiene controls extend policy enforcement to the commit and build boundary. Signed commits establish cryptographic accountability for code contributions, ensuring that the origin of every committed change can be verified. Branch protection rules, mandatory peer review requirements, and pre-commit secrets scanning address the vectors through which insecure AI-generated content most commonly reaches shared code history. SLSA attestation gates extend this integrity enforcement to the build artifact level: only artifacts whose provenance can be cryptographically verified through a complete, auditable build chain may advance through release promotion gates [11,12].

3.2 Build-Time Detection Controls

The build-time detection layer applies a set of overlapping automated controls within the CI pipeline, each targeting a distinct dimension of the vulnerability surface that AI-generated code creates. No single control achieves adequate coverage across all dimensions; the architecture is designed for complementary coverage in which the gaps of one control are addressed by the detection scope of another [11].

AI-augmented static application security testing, deployed alongside traditional rule-based and taint-tracking SAST, forms the primary detection layer. AI-augmented analysis applies semantic reasoning to identify vulnerability patterns that rule-based engines miss because they depend on signature matching rather than contextual interpretation of code behavior. Traditional SAST contributes deterministic, auditable coverage of well-characterized weakness classes. Deploying both against pull request-level differential scans—limiting analysis scope to the lines changed in each pull request—keeps pipeline latency within bounds that developers tolerate and surfaces findings in the immediate context of the code under review [12]. Fix-or-waive workflows for critical CWE classes—specifically those observed at elevated rates in AI-generated output such as injection vulnerabilities, insecure output handling, and cryptographic misuse—enforce a formal decision at each flagged finding [12].

API schema validation integrated into the CI pipeline addresses a vulnerability surface that static code analysis does not adequately cover. Validating OpenAPI and GraphQL schemas at build-time detects authentication gaps, authorization misconfigurations, undocumented endpoints, and schema drift before APIs are deployed, targeting the shadow API problem at its origin rather than attempting to discover undocumented surfaces through post-deployment scanning [11].

Automated SBOM generation on every build, validated against the minimum element specifications established by CISA, makes dependency transparency a continuous and enforceable build-time property rather than a periodic manual audit activity [10]. Every build produces a machine-readable inventory of all components, libraries, and dependencies included in the resulting artifact. Builds that introduce components

without verifiable provenance, or dependencies pinned to versions with known unresolved high-severity advisories, are failed automatically [10].

Pre-deployment runtime simulation—referred to in this blueprint as shift-left DAST—addresses the exploitability gap that is inherent in all static analysis. Containerized ephemeral environments, configured to mirror the authentication flows, input validation surfaces, and data handling patterns of the production-bound application, run automated attack scenarios against the build under review before the pull request merges. These scenarios target the vulnerability classes most commonly introduced by AI-generated code: injection-prone input handling, authentication bypass conditions, insecure direct object reference patterns, and session management weaknesses. The output of pre-deployment runtime simulation is a pre-merge exploitability signal that confirms or refutes the exploitability of the highest-priority static findings before they enter the main branch [11].

A note on the feasibility of runtime simulation at scale: some practitioners argue that ephemeral DAST environments introduce false confidence when production topology cannot be fully mirrored, and that the overhead of environment provisioning introduces pipeline latency that high-velocity teams cannot absorb. This concern is valid and should inform scoped simulation targets rather than full-system replication. The architecture recommends scoping shift-left DAST to the highest-priority static findings per pull request—specifically those classified as critical or high by the ASPM correlation layer—rather than running comprehensive dynamic analysis on every build.

A note on model version governance: the AI-augmented SAST rule sets and threshold configurations described in this section are calibrated against specific model output distributions. When development teams upgrade their underlying AI coding assistant—whether from one major model version to another or to a fine-tuned variant—this constitutes a material change in the vulnerability generation profile that warrants a recalibration cycle. Organizations should maintain a held-out labeled dataset of AI-generated diffs representing their codebase and re-measure per-CWE precision and recall whenever the model is updated before expanding the new model's deployment. Fix-or-waive thresholds should be adjusted to reflect the recalibrated precision baseline.

3.3 Prioritization and Developer Feedback

Detection controls that generate finding outputs without translating those outputs into developer action do not reduce risk; they produce records of risk that accumulate without accountability. The prioritization and feedback layer is the mechanism by which the outputs of all upstream detection controls are converted into the minimal, high-confidence, actionable finding lists that developers can absorb and act on within normal delivery cadences. A mature shift-left posture recognizes that developer adoption is the ultimate determinant of whether any security control produces sustained risk reduction and that controls that generate excessive friction or low-precision outputs will be circumvented regardless of their technical capability [11].

Application security posture management correlation at build-time ingests the outputs of every upstream control—AI-augmented SAST, traditional SAST, API schema validation, SBOM gates, and pre-deployment runtime simulation—and applies reachability analysis and asset-value weighting to rank findings by actual risk rather than severity score alone [12]. Reachability analysis determines whether a statically identified vulnerability sits on a code path that is exercised in the deployed application, eliminating the class of findings that are technically present in the codebase but practically unreachable. Asset-value weighting adjusts priority based on the sensitivity and exposure of the component in which the finding appears [12].

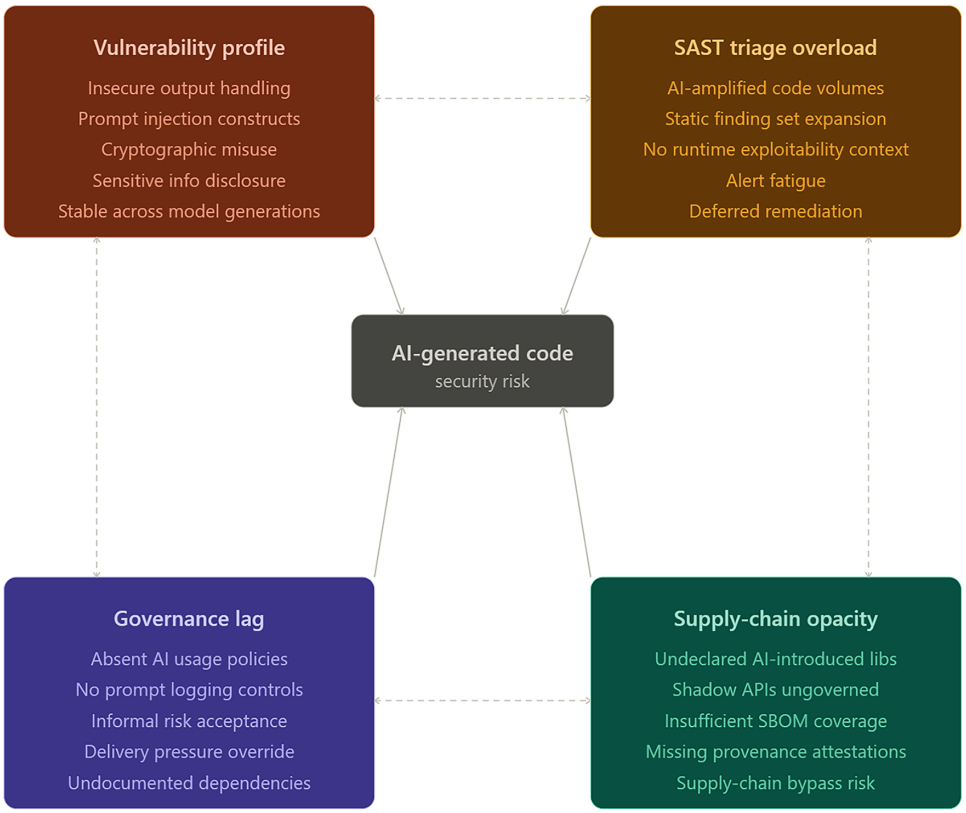

Figure 2: AI-era threat landscape and governance deficit map.

IDE and pull request fix coaching delivers AI-generated remediation suggestions at the point of code review, providing secure code alternatives with sufficient context for developers to evaluate, understand, and accept them without leaving their development environment [12]. Fix acceptance rates are tracked continuously as a leading indicator of both suggestion quality and developer trust in the tooling. A declining acceptance rate signals either that suggestions are technically inadequate, that they introduce unacceptable friction, or that developers do not understand the vulnerability the suggestion is addressing [11,12].

4 Evaluation Framework and Illustrative Outcomes

The business case for shift-left security controls in the AI code generation era cannot rest on detection volume alone. Organizations that measure the effectiveness of their AppSec programs by counting findings are measuring activity rather than outcomes, and in the context of AI-amplified code volumes, raw finding counts are more likely to reflect triage burden than risk reduction. A mature evaluation framework must shift the unit of measurement from how many vulnerabilities were detected to how much exploitable risk was eliminated, how quickly it was remediated, and at what cost to developer productivity (Table 2) [14].

Table 2: Evaluation metrics framework for shift-left security controls applied to AI-generated code, organized by measurement category and programmatic purpose.

Metric Category	Metric	Measurement Purpose
Detection Quality	Precision, Recall, F1 per CWE class (AI-SAST vs. traditional SAST)	Quantify detection accuracy and noise burden per vulnerability class and identify tooling calibration gaps
Exploitability Signal	True-Exploit Rate (TER)	Measure the percentage of static findings confirmed exploitable in pre-deployment simulation and assess signal-to-noise improvement
Remediation Efficiency	MTTR at P50 and P95 for critical findings	Capture median and tail remediation velocity; reflect combined effectiveness of detection quality, prioritization, and fix tooling
Triage Burden	Triage time per 100 findings (analyst hours); Noise Ratio (FP + non-reachable/total findings)	Quantify security team capacity consumption and isolate ASPM correlation contribution to noise reduction
Developer Experience	PR cycles added; AI fix acceptance rate; pipeline delay per PR (minutes)	Measure workflow friction and tooling adoption health; identify circumvention risk before it becomes programmatic
Supply-Chain Integrity	% dependencies meeting CISA 2025 SBOM minimum elements; % builds blocked by missing or invalid attestations	Confirm provenance coverage completeness and enforcement gate effectiveness across the build inventory

4.1 Core Metrics

Detection quality is the foundational measurement layer and must be assessed with greater granularity than aggregate finding counts allow. Precision, Recall, and F1 score computed per CWE class—evaluated separately for AI-augmented SAST and traditional rule-based SAST against AI-generated code diffs—provide the structured comparison needed to determine where each analysis approach adds marginal value and where calibration gaps remain. Precision measures the proportion of flagged findings that represent genuine vulnerabilities, directly quantifying the noise burden that developers absorb when reviewing security output [14].

The true-exploit rate (TER) is the metric that most directly addresses the triage overload problem and represents the most consequential single indicator of whether shift-left controls are delivering an exploitability signal or simply relocating noise to an earlier pipeline stage. TER is defined as the percentage of static findings confirmed exploitable through pre-deployment runtime simulation in containerized ephemeral environments. Operationally, exploitability confirmation is determined by automated attack execution via OWASP ZAP or equivalent DAST engine: a finding is classified as TER-confirmed only when the attack payload produces a response meeting a defined exploitation criterion—for example, a reflected payload in the HTTP response body for XSS findings, or an error message revealing a SQL fragment structure

for injection findings. The test oracle combines HTTP response classification and application log anomaly detection; findings that the automated oracle cannot classify are escalated to a security analyst review queue and excluded from both the numerator and denominator of the TER calculation until a manual verdict is recorded. For the illustrative case in Section 4.3, ground truth was established retrospectively by a security analyst reviewing DAST-confirmed findings against the codebase, with a 20% sample (approximately 34 of ~170 DAST-evaluated findings) audited by a second analyst to estimate inter-rater reliability; the two analysts reached agreement on exploitability classification for 87% of audited findings, corresponding to a Cohen's kappa of approximately 0.73 (substantial agreement by conventional thresholds). This reliability estimate applies to the illustrative case only and would require independent verification in a prospective study. This method is acknowledged as weaker than a controlled experiment, and readers should treat the resulting TER figures as indicative rather than definitive [14].

Mean time to remediate, computed at both the P50 and P95 percentiles for critical-severity findings, quantifies remediation efficiency across the distribution of finding complexity. P50 MTTR reflects the typical remediation experience for the median developer on a median finding; P95 MTTR captures the tail cases—complex vulnerabilities or deeply embedded weaknesses that disproportionately consume security team capacity and represent the highest sustained risk exposure. Triage time per 100 findings, measured in analyst hours, provides the operational efficiency indicator that security team managers require to assess staffing adequacy and to quantify the capacity impact of noise reduction initiatives [14].

Developer experience metrics capture the adoption dimension that ultimately determines whether shift-left controls produce sustained risk reduction or become shelfware that teams route around under delivery pressure. Pull request cycles added by security gates quantify the direct workflow friction that shift-left controls impose. AI fix acceptance rate measures the proportion of AI-generated remediation suggestions that developers accept and apply, serving as a leading indicator of both suggestion quality and developer trust in the tooling. Pipeline delay in minutes per pull request quantifies the latency that build-time security controls add to the CI cycle [15].

4.2 Supply-Chain Integrity Metrics

Supply-chain integrity measurement operationalizes the provenance and transparency obligations established in the policy and governance layer. The governance of AI-generated code introduces specific supply-chain accountability challenges that conventional dependency management practices were not designed to address: AI tools introduce components without explicit developer review, and the velocity of AI-assisted development compresses the time available for manual provenance verification [15].

The percentage of dependencies across the software inventory that satisfy CISA 2025 SBOM minimum element requirements establishes the baseline coverage level for machine-readable supply-chain transparency. This metric is computed per build and tracked over time to confirm that SBOM coverage is expanding alongside the codebase rather than degrading as new AI-introduced dependencies accumulate without documentation [14]. The percentage of builds blocked by missing or invalid provenance attestations is the enforcement-side complement to SBOM coverage measurement: a rising block rate in the early phases of program implementation is expected and desirable, reflecting the detection of previously untracked supply-chain additions. A stabilizing or declining block rate over subsequent cycles, combined with improving SBOM coverage, indicates that development teams are internalizing provenance requirements [12,15].

4.3 Illustrative Outcomes (Anonymized Fintech Case)

The following case is presented as an illustrative scenario reflecting patterns observed in enterprise implementation engagements. Results are not claimed as statistically generalizable, and causal attribution to individual controls would require controlled replication. Readers should treat the figures as indicative of the direction and magnitude of outcomes consistent with full three-layer implementation rather than as empirically established effect sizes.

The organization maintained an active codebase of approximately 1.2 million lines of Java and Python across fourteen microservices, with a development team of sixty engineers. The primary AI coding assistant deployed was GitHub Copilot (Enterprise tier). The organization had adopted AI coding assistance across its core development teams within twelve months, achieving measurable gains in feature delivery throughput. The governance and security consequences of that adoption, however, emerged within two quarters: pull request volumes surged substantially, static analysis finding sets expanded to a volume that exceeded the security team's triage capacity, and the exploitability status of the majority of flagged findings remained uncertain because runtime validation was deferred to post-deployment testing cycles running on a two-week cadence [15]. The baseline false positive rate for CWE-89 (SQL Injection) prior to intervention was 61%. A note on confounders: a new security lead joined the organization mid-evaluation, and their contributions to manual triage may have independently influenced MTTR outcomes; this is acknowledged as a limitation of the scenario.

The baseline condition before intervention was characterized by three mutually reinforcing problems that the evaluation framework was subsequently used to quantify. First, static analysis precision was low: a large proportion of findings flagged in AI-generated diffs were non-reachable or false positives. Second, the absence of pre-merge exploitability validation meant that the findings delivered to developers carried no exploitability signal. Third, pipeline gates were not enforced on SBOM coverage or provenance attestation, meaning that AI-introduced dependencies were entering production without machine-readable documentation [15].

Following a two-sprint implementation of AI-augmented SAST alongside traditional SAST with pull request-level differential scanning, automated SBOM generation enforced against CISA minimum elements, shift-left DAST in containerized ephemeral environments scoped to critical and high findings, and full ASPM correlation with reachability-based prioritization, the organization recorded the following indicative outcomes within the evaluation period (13 weeks). To provide context for the percentage changes reported: at baseline, the pipeline generated approximately 480 SAST findings per two-week sprint; post-intervention, this fell to approximately 330 findings per sprint, yielding the 31% precision improvement (from approximately 39% true-positive rate to approximately 70%). The TER baseline was approximately 18% of static findings confirmed exploitable (~87 of ~480 per sprint); post-intervention this rose to approximately 26% (~86 of ~330), representing the 42% TER improvement. Baseline triage time was approximately 14.5 analyst-hours per 100 findings, falling to approximately 6.4 analyst-hours post-intervention (56% reduction). P50 MTTR at baseline was approximately 11.2 days, falling to approximately 6.9 days post-intervention (38% reduction). No variance or confidence interval data were collected in this illustrative engagement; these figures should be treated as directional rather than statistically precise. Static analysis precision was associated with a 31% improvement; the true-exploit rate improved by 42%, a pattern consistent with the expectation that pre-deployment runtime simulation contributes to identifying exploitable findings before merge. Triage hours per 100 findings decreased by 56%. Mean time to remediate at P50 dropped by 38%, a pattern consistent with faster developer action facilitated by a smaller, higher-confidence finding list and inline fix suggestions—though other organizational factors, including the new security lead noted above, may have contributed. Per-pull-request pipeline delay remained below six minutes throughout the evaluation period (Fig. 3) [14].

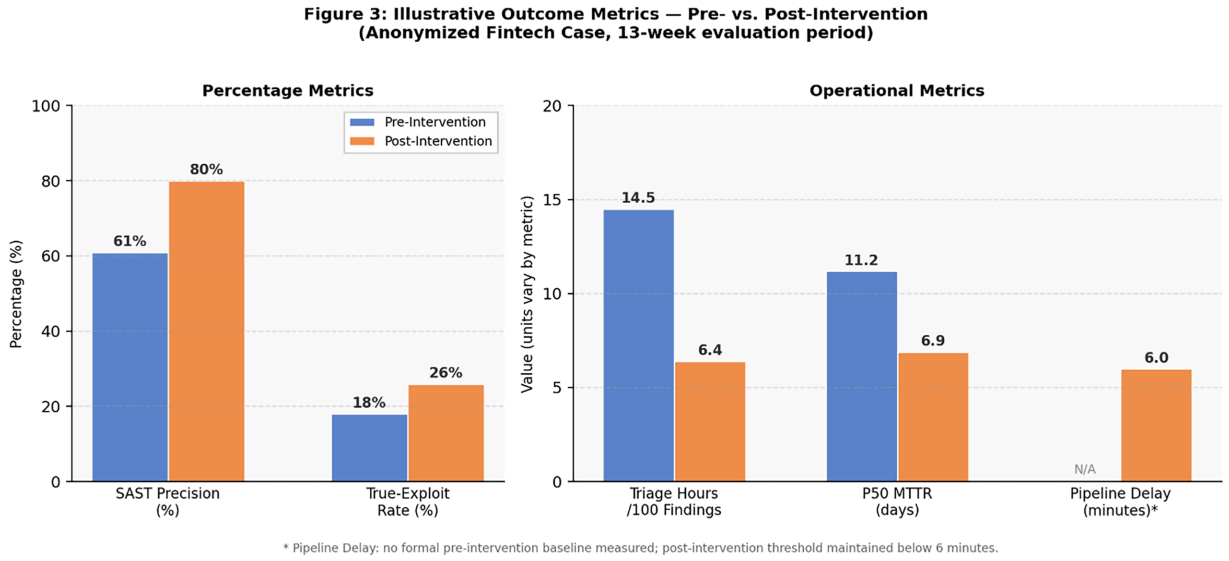

Figure 3: Illustrative Outcome Metrics: Pre- vs. Post-Intervention (Anonymized Fintech Case). Bars show approximate values at baseline and post-intervention for five primary outcome dimensions. All figures are indicative only; no variance data were collected. SAST Precision and TER are expressed as percentages; Triage Hours and P50 MTTR are absolute values; Pipeline Delay post-intervention was maintained below the six-minute threshold (no formal baseline measured). See Section 4.3 for full caveats.

These results are directionally consistent with the hypothesis that runtime-aware exploitability validation and ASPM-based correlation are variables associated with improved risk reduction outcomes in this scenario, though causal attribution cannot be established without controlled replication. Static analysis alone, regardless of the sophistication of the analysis engine, cannot close the exploitability gap that produces triage overload in AI-era development environments [14,15].

5 Conclusion

The security challenge introduced by AI-generated code is neither a temporary consequence of immature tooling nor a problem that model advancement will passively resolve; it is a structural condition of the AI-assisted development era that demands a correspondingly structural response at the build-time control layer. Conventional shift-left practice was designed for a development environment in which human-authored code volumes were manageable, static analysis triage was feasible within available security team capacity, and dependency provenance could be tracked through deliberate manual review. AI coding assistants have significantly strained each of those assumptions simultaneously: vulnerability rates in AI-generated output have remained elevated and stable across model generations in organizations with high AI adoption levels, static finding volumes have expanded beyond the triage capacity of teams operating without automated exploitability filtering, and supply-chain opacity has grown as AI tools introduce undeclared dependencies at a pace that outstrips manual governance. Organizations with lower AI adoption rates or with existing high-maturity DevSecOps programs may experience these pressures less acutely, and the blueprint is designed to be modularly adoptable rather than requiring full three-layer deployment as an initial condition. The outcomes reported in Section 4.3 are presented as an illustrative scenario; controlled evaluation would be required to establish causal attribution to any specific architectural component.

The blueprint developed across this article addresses these conditions directly by establishing a three-layer control architecture in which policy and governance obligations enforce preventive controls before

code reaches shared repositories, build-time detection controls apply overlapping AI-augmented static analysis, API schema validation, automated SBOM enforcement, and pre-merge runtime simulation to produce exploitability-informed finding sets, and application security posture management correlation reduces those finding sets to the minimal, high-confidence, developer-actionable lists that sustain adoption rather than invite circumvention. The evaluation framework accompanying the blueprint reorients program measurement away from detection volume and toward the outcome indicators—precision per vulnerability class, true-exploit rate, mean time to remediate, supply-chain coverage, and developer experience—that reflect genuine risk reduction and provide the business-relevant evidence needed to sustain organizational investment in build-time security infrastructure.

As regulatory expectations around software bill of materials minimum elements mature and supply chain transparency requirements tighten across industry and federal policy frameworks, the provenance and attestation controls described in this article will transition from competitive differentiators to baseline compliance obligations, making early adoption of the blueprint not only a security advantage but also a governance imperative. Teams that instrument build time with these controls, measure outcomes with exploitability-centered metrics, and calibrate tooling to preserve the developer experience that determines long-term adoption will be positioned to achieve sustained, measurable risk reduction as AI coding adoption continues to scale across the enterprise software development landscape.

Acknowledgement: Not applicable.

Funding Statement: The author received no specific funding for this study.

Availability of Data and Materials: Not applicable.

Ethics Approval: Not applicable.

Conflicts of Interest: The author declares no conflicts of interest.

References

1. WIZ. State of code security in 2025 [Internet]. 2025 [cited 2025 Jun 1]. Available from: https://www.wiz.io/reports/state-of-code-security-2025.
2. Veracode. 2025 GenAI code security report [Internet]. Veracode, Inc.; 2025 [cited 2025 Aug 1]. Available from: https://www.veracode.com/resources/analyst-reports/2025-genai-code-security-report/.
3. Perry N, Srivastava M, Kumar D, Boneh D. Do users write more insecure code with AI assistants? In: Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security; 2023 Nov 26–30; Copenhagen, Denmark. doi:10.1145/3576915.3623157.
4. Pearce H, Ahmad B, Tan B, Dolan-Gavitt B, Karri R. Asleep at the keyboard? Assessing the security of GitHub copilot's code contributions. In: Proceedings of the 2022 IEEE Symposium on Security and Privacy; 2022 May 22–26; San Francisco, CA, USA. doi:10.1109/sp46214.2022.9833571.
5. Vizard M. Survey surfaces widespread adoption of AI to improve DevSecOps [Internet]. DevOps.com. 2025 [cited 2025 May 1]. Available from: https://devops.com/survey-surfaces-widespread-adoption-of-ai-to-improve-devsecops/.
6. Tischler N. How AI is transforming application security testing [Internet]. Veracode Blog. 2025 [cited 2025 Apr 1]. Available from: https://www.veracode.com/blog/ai-transforming-application-security-testing/.
7. OWASP. OWASP top 10 for LLM applications 2025 [Internet]. OWASP Foundation; 2024 [cited 2025 Mar 1]. Available from: https://genai.owasp.org/resource/owasp-top-10-for-llm-applications-2025/.
8. Asare O, Nagappan M, Asokan N. Is GitHub's Copilot as bad as humans at introducing vulnerabilities in code? Empir Softw Eng. 2023;28(6):129. doi:10.1007/s10664-023-10380-1.

9. Siddiq ML, Santos JCS. SecurityEval dataset: mining vulnerability examples to evaluate machine learning-based code generation techniques. In: Proceedings of the 1st International Workshop on Mining Software Repositories Applications for Privacy and Security; 2022 Nov 18; Singapore. doi:10.1145/3549035.3561184.
10. CISA. Software bill of materials (SBOM) [Internet]. Cybersecurity and Infrastructure Security Agency; 2025 [cited 2025 Jun 1]. Available from: https://www.cisa.gov/sbom.
11. Fortinet. What is shift left security? [Internet]. Fortinet Cyber Glossary; 2025 [cited 2025 Jun 1]. Available from: https://www.fortinet.com/resources/cyberglossary/shift-left-security.
12. OWASP. OWASP DevSecOps guideline [Internet]. OWASP Foundation; 2025 [cited 2025 Jun 1]. Available from: https://owasp.org/www-project-devsecops-guideline/.
13. Endor Labs. State of dependency management 2025 [Internet]. Endor Labs; 2025 [cited 2025 Nov 1]. Available from: https://www.endorlabs.com/lp/state-of-dependency-management-2025.
14. Thevarmannil M. DevSecOps metrics & KPIs for 2026 [Internet]. Practical DevSecOps; 2025 [cited 2025 Jun 1]. Available from: https://www.practical-devsecops.com/devsecops-metrics/.
15. Chaudhuri A. Technology data security and governance in the age of AI [Internet]. Cybersecurity Tribe; 2024 [cited 2025 Jun 1]. Available from: https://www.cybersecuritytribe.com/articles/data-security-and-governance-in-the-age-of-ai.