Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

Stochastic Code, Deterministic Defense: Assessing the Security Blind Spots in GenAI-Driven CI/CD Pipelines

Mohammed BEDJAOUI

m.bedjaoui@esi-sba.dz

École Supérieure en Informatique de Sidi Bel Abbès

Sidi Mohammed BENSLIMANE

École Supérieure en Informatique de Sidi Bel Abbès

Mohammed Yassine KAZI TANI

École Supérieure en Informatique de Sidi Bel Abbès

Research Article

Keywords: DevSecOps, Generative AI, Stochastic Vulnerabilities, CI/CD Pipelines, Cloud-Native Security

Posted Date: August 5th, 2026

DOI: https://doi.org/10.21203/rs.3.rs-8531146/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: No competing interests reported.

Stochastic Code, Deterministic Defense: Assessing the Security Blind Spots in GenAI-Driven CI/CD Pipelines

Mohammed BEDJAOUI[1*], Sidi Mohammed BENSLIMANE[1] and Mohammed Yassine KAZI TANI[1]

[1*]LabRI-SBA Laboratory, École Supérieure en Informatique, Sidi Bel Abbès, Algeria.

*Corresponding author(s). E-mail(s): m.bedjaoui@esi-sba.dz;
Contributing authors: s.benslimane@esi-sba.dz; y.kazitani@esi-sba.dz;

Abstract

Context: The integration of Generative AI (GenAI) into Continuous Integration/Continuous Deployment (CI/CD) workflows is accelerating the automation of software delivery in cloud-native environments. **Problem:** However, a fundamental misalignment exists between the probabilistic nature of AI-generated artifacts and the deterministic, signature-based logic of traditional security tools. This discrepancy creates potential "blind spots" where syntactically correct but insecure code bypasses validation gates. **Aim:** This study empirically investigates and quantifies the detection failure rate of standard DevSecOps tools against stochastic vulnerabilities introduced by local Large Language Models (LLMs). **Methodology:** We conducted a controlled experiment using a local LLM (Code Llama) integrated into a GitLab pipeline deploying to Kubernetes. Through repeated independent runs, we simulated two injection scenarios: Logic Flaws (Broken Access Control) and Infrastructure Configuration Drift. **Results:** Empirical statistical analysis reveals that widely used deterministic scanners consistently failed to detect the syntactically valid yet insecure artifacts, confirming a systemic inability to identify context-dependent AI flaws. **Conclusion:** These findings demonstrate that current deterministic defenses are insufficient for GenAI-driven pipelines. The study highlights the urgent need for a paradigm shift towards intent-aware verification and probabilistic security frameworks to mitigate the risks of "silent decay" in automated software supply chains.

Keywords: DevSecOps, Generative AI, Stochastic Vulnerabilities, CI/CD Pipelines, Cloud-Native Security

1 Introduction

The accelerating adoption of Generative Artificial Intelligence (GenAI) is fundamentally reshaping the landscape of software engineering [1, 2]. Beyond assisting developers with natural language programming, GenAI agents are increasingly embedded into Continuous Integration and Continuous Deployment (CI/CD) pipelines that orchestrate **cloud-native and containerized architectures** [3–6]. These pipelines, which constitute the backbone of modern software delivery and Agile workflows [7], are now being augmented with generative capabilities to support critical tasks such as automated middleware generation, dynamic configuration of infrastructure, and real-time code reviews [8, 9].

While this evolution promises unprecedented efficiency, it simultaneously introduces a paradigm shift in the threat landscape [10–13]. Research in DevSecOps has traditionally concentrated on deterministic vulnerability classes, such as static injection attacks (e.g., SQLi) and known dependency vulnerabilities (CVEs) [14, 15]. These efforts have produced mature, rule-based practices supported by tools like SAST (Static Application Security Testing) and **Infrastructure-as-Code (IaC)** scanners, which function effectively in reproducible, deterministic environments [16].

However, the integration of GenAI agents into CI/CD workflows challenges the foundational assumptions of these tools [14, 17, 18]. Unlike traditional compilers or scripts, Generative models operate probabilistically [17]. They lack transparency in decision-making and are prone to "stochastic hallucinations"—generating code that is syntactically correct but semantically insecure [19]. As a result, standard verification methods, which rely on rigid pattern matching, may fail to detect novel, context-dependent vulnerabilities introduced by AI [16].

Among these emerging risks, three dimensions appear particularly critical for automated pipelines. First, **Automated Logic Injection (V1)** occurs when generative models inadvertently introduce insecure business logic (e.g., broken access control) that defies standard syntax-based signatures [12]. Second, **Amplified False Positives (V2)** occur when AI-driven outputs generate noise or non-functional code that overwhelms conventional scanning approaches [16]. While disruptive, these usually trigger build failures. Third, **Infrastructure Configuration Drift (V3)** creates a insidious risk where AI-generated Kubernetes manifests deviate subtly from security best practices (e.g., misconfigured health checks), creating "silent" vulnerabilities that bypass validation gates [16, 20].

The core problem addressed in this paper is the **alignment gap** between the probabilistic nature of GenAI artifacts and the deterministic rigidity of current DevSecOps scanning tools [11, 14]. While classical approaches assume predictable code patterns, AI-enabled pipelines introduce variability that generates false negatives—vulnerabilities that pass all checks but compromise the system at runtime [17, 18].

This paper bridges this gap by conducting an empirical assessment of GenAI-induced vulnerabilities in a controlled CI/CD pipeline. Unlike previous works limited to theoretical analysis, we perform a practical case study to:

1. **Characterize** the distinct "stochastic vulnerabilities" introduced by local LLMs (Code Llama) in cloud-native workflows.
2. **Empirically Evaluate** the detection failure rate of industry-standard tools (Semgrep, Trivy, Kube-linter) against these AI-generated artifacts.
3. **Demonstrate** the "blind spot" phenomenon where probabilistic threats successfully bypass deterministic security gates to compromise a production-like Kubernetes environment.

By combining a rigorous experimental protocol with a statistical analysis of detection rates, this paper contributes to the emerging discourse on securing the next generation of AI-augmented software supply chains.

2 Related Works

The intersection of GenAI and DevSecOps is a rapidly evolving domain. Existing literature can be categorized into three primary streams: security risks in traditional CI/CD pipelines, vulnerabilities inherent to AI-generated code, and the use of LLMs for defensive security operations.

2.1 Security Threats in CI/CD Ecosystems

Before the advent of GenAI, the fragility of CI/CD pipelines was already a major concern. Pan et al. [10] conducted a large-scale analysis of open-source pipelines ("Ambush from All Sides"), identifying that supply chain attacks often target the orchestration layer rather than the application code itself. Similarly, Ladisa et al. [21] provided a comprehensive taxonomy of attacks on software supply chains, demonstrating how compromised pipelines could transform secure source code into insecure deployments through hidden vectors during the build phase. Furthermore, empirical research on GitLab workflows [22] has highlighted the prevalence of "configuration smells"—latent misconfigurations that degrade maintainability and security. These studies collectively establish that the deterministic nature of pipelines presents a dichotomy: while it ensures reproducibility, it also allows misconfigurations and smells to propagate silently to production. Our work builds on this by investigating how GenAI introduces a non-deterministic vector of compromise into these already fragile environments.

2.2 Vulnerabilities in AI-Generated Code

With the rise of tools like GitHub Copilot, significant research has focused on the quality of the generated code. In a seminal study, Asare et al. [17] evaluated Copilot's output against the MITRE CWE top 25, finding that AI models frequently reproduce vulnerabilities found in their training data, particularly in high-risk areas like SQL injection and hardcoded credentials. More recently, studies on LLM-based code completion [18] have highlighted specific attack surfaces where prompt injection can coerce models into generating malicious payloads. However, these works primarily focus on *static* vulnerability analysis of code snippets in isolation. They do not empirically evaluate how these snippets interact with standard security gates (SAST/IaC) when integrated into a fully automated pipeline.

2.3 GenAI for Defensive Security

Conversely, a growing body of work explores using GenAI to enhance security. Recent frameworks propose "LLM Agentic Workflows" to automatically detect and remediate vulnerabilities in **IaC** templates [16]. Similarly, recent works explored leveraging Generative AI for proactive security and automated remediation in cloud-native pipelines [23]. These approaches typically report high detection rates by leveraging the semantic understanding of LLMs. However, a paradox emerges in the current state of the art: while significant effort is dedicated to building AI-based tools to *detect* vulnerabilities (using AI as a scanner), there is a scarcity of research assessing whether traditional, deterministic tools—which constitute the vast majority of industrial deployments—are capable of detecting vulnerabilities *introduced* by AI systems.

2.4 The Research Gap

Despite these advancements, a critical alignment gap remains. The literature either focuses on manual attacks on CI/CD [10, 21], human configuration errors [22], or on the intrinsic flaws of AI code [17]. There is a lack of empirical evidence regarding the "Blind Spot" of standard DevSecOps tools when facing the stochastic behaviors of local LLMs in a closed-loop environment. This paper bridges this gap by shifting the focus from "Is AI code secure?" to "Can we detect it if it's not?", specifically evaluating the failure rate of deterministic scanners against probabilistic logic flaws and configuration drifts.

3 Methodology and Experimental Protocol

This section outlines the research methodology designed to empirically investigate the security implications of integrating **LLMs** into automated CI/CD workflows.

3.1 Research Objective and Experimental Design

The primary objective of this experiment is to quantify the *alignment gap* between the probabilistic code generation capabilities of GenAI and the deterministic verification logic of traditional DevSecOps tools. Current literature suggests that while GenAI agents can accelerate development, they introduce a class of "stochastic vulnerabilities"—subtle flaws that vary across executions and often bypass syntactic validation. Our experiment is designed to test the following premise: *Standard SAST and IaC scanners, relying on pattern matching, fail to detect context-dependent security flaws injected by AI agents.* To validate this, we constructed a controlled, closed-loop DevSecOps environment where an uncensored LLM acts as a code contributor. The experiment isolates the variables by maintaining a constant environment while varying the AI-generated artifacts across $N = 50$ iterations, as recommended by Arcuri and Briand [24] for randomized algorithms in software engineering.

3.2 Scenario Selection and Scope Delimitation

To conduct a focused assessment of critical security risks, we categorized potential GenAI-induced anomalies into three dimensions:

- **Scenario V1 (Logic Flaws):** Syntactically valid code containing business logic vulnerabilities (e.g., Broken Access Control in PHP middleware).
- **Scenario V2 (False Positives/Hallucinations):** Non-functional code or noise that triggers excessive alerts but typically results in immediate build failure.
- **Scenario V3 (Infrastructure Drift):** Subtle misconfigurations in Kubernetes manifests that weaken runtime security without violating syntax rules.

Exclusion of Scenario V2:

While *Amplified False Positives* (V2) present a significant challenge regarding developer efficiency [16], they essentially act as a "Fail-Secure" mechanism: the pipeline usually breaks due to compilation errors or syntax violations, effectively preventing deployment. In contrast, V1 and V3 represent *False Negatives* (Silent Decay), where insecure artifacts bypass detection and successfully reach production. Therefore, to specifically address the issue of "Blind Spots" in security gates, this study targets scenarios V1 and V3.

3.3 Experimental Environment and Infrastructure

To ensure reproducibility and eliminate network latency variables, the experiment is conducted on a local environment simulating a hybrid "On-Premise" infrastructure. The setup is hosted on a workstation equipped with a 12th Gen Intel® Core™ i5-1245U (1.60 GHz) and 32 GB of RAM. The production environment is virtualized within a single Ubuntu Server 22.04 LTS Virtual Machine (VM), allocated with 4 vCPUs and 16 GB of RAM. This VM hosts the orchestration layer (Minikube v1.32), the Continuous Integration agent (GitLab Runner), and the inference engine (Ollama).

3.4 Case Study Application: `aureuserp`

To evaluate the impact on real-world software, we utilize `aureuserp`[1], an open-source modular Enterprise Resource Planning (ERP) system built with the Laravel framework. The complete experimental reproducibility package, including the Laravel codebase, Kubernetes configurations, and Docker definitions, is publicly archived for validation purposes[2]. As illustrated in **Figure 1**, the application follows a microservices-like architecture deployed on Kubernetes via three distinct pods: (1) an Nginx Reverse Proxy, (2) a PHP-FPM application pod, and (3) a MySQL database pod.

3.5 The Stochastic Injection Engine

Unlike commercial APIs (e.g., OpenAI GPT-4) which implement rigorous Reinforcement Learning from Human Feedback (RLHF) to filter malicious outputs, this study requires a model that mimics a raw, unrestricted developer assistant. We selected the `codellama:7b-instruct` model (quantized version). This choice deliberately avoids "safe" models to realistically simulate the risk of uncensored code generation where the model prioritizes functional correctness over security best practices.

[1]Source code available at: https://github.com/aureuserp/aureuserp
[2]Replication package available at: https://gitlab.com/m.bedjaoui/genai-devsecops-blindspot

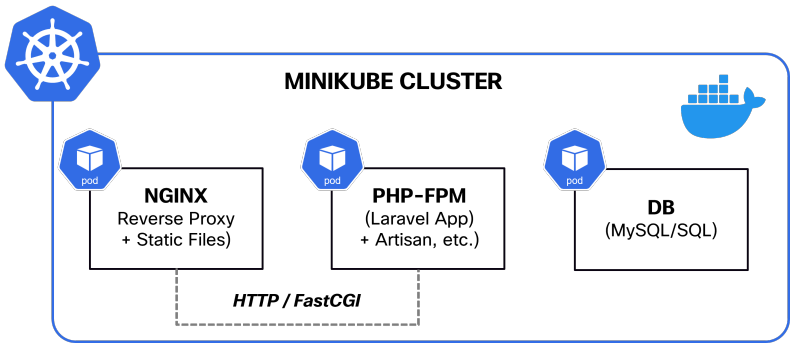


Fig. 1 Architecture of the `aureuserp` application deployed on Minikube.

3.6 DevSecOps Baseline (Control Group)

We employ industry-standard deterministic tools to establish a baseline for detection capabilities. Table 1 details the tools selected for the experiment.

Tool	**Scope**	**Target Vulnerability & Limitation**
Semgrep	SAST	**Target:** PHP Logic & Syntax patterns. **Limit:** Lacks semantic understanding of custom middleware logic.
Trivy	IaC / Container	**Target:** CVEs & Misconfigurations. **Limit:** Validates against static policies, misses contextual drift.
Kube-linter	Manifests	**Target:** Best Practices (Least Privilege). **Limit:** Checks syntax validity, not functional intent.

Table 1 Deterministic Security Tools Used in the Control Group

3.7 Experimental Protocol

The experiment follows a continuous integration pipeline driven by the GitLab Runner. The workflow, depicted in the sequence diagram in **Figure 2**, consists of four automated stages executed for each run.

1. **AI Injection Phase:** The pipeline queries the local LLM to generate the targeted artifacts (Middleware for V1, Deployment YAML for V3).

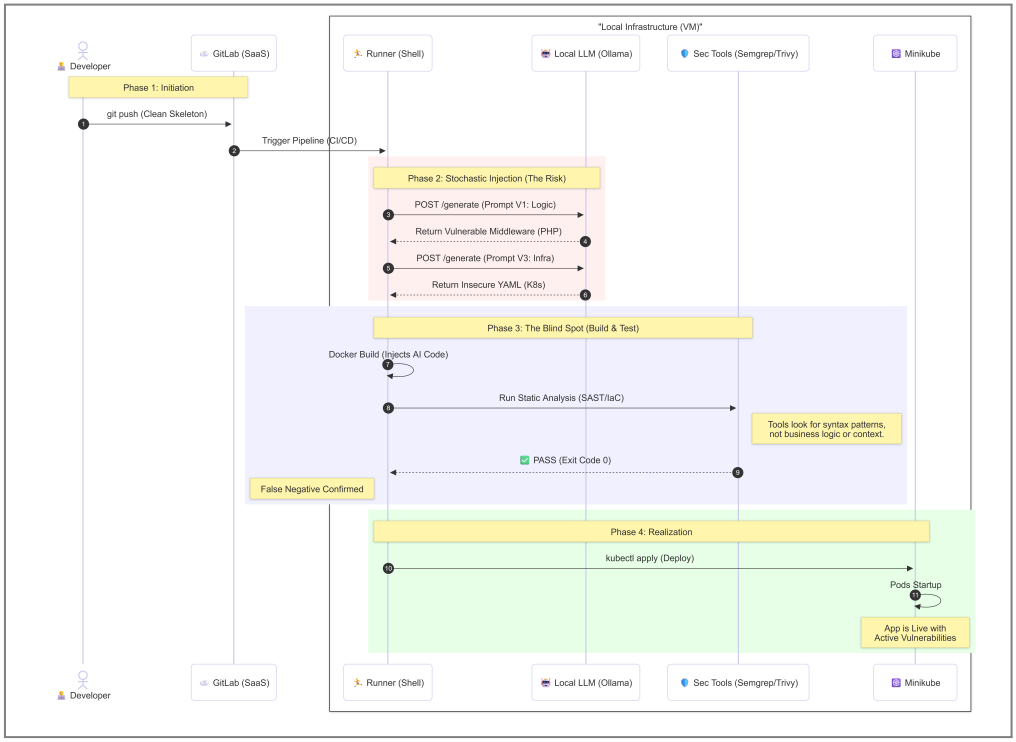


Fig. 2 Sequence diagram of the experimental CI/CD pipeline involving GenAI injection.

2. **Build Phase:** A Docker image is built directly within the Minikube context, incorporating the AI-generated code.
3. **Security Test Phase:** The deterministic tools scan the generated artifacts. Success is defined by the failure to detect the injected vulnerabilities (False Negatives).
4. **Deployment Phase:** The application is deployed to the cluster to verify runtime exploitability.

The implementation of these stages is defined in the `.gitlab-ci.yml` configuration, an extract of which is provided in **Figure 3**.

3.8 Prompt Engineering and Injection Strategy

To ensure reproducibility and confirm the "functional intent" of the generated code (proving that vulnerabilities were not explicitly requested), we employed the following zero-shot prompts:

- **For V1 (PHP Middleware):** *"Create a Laravel middleware named CheckInternalAccess that checks if the request has a header 'X-Internal-Service-Auth'. If missing, return 403."* (Note: No instruction was given to validate the session or authentication token, testing if the model adds security controls by default).

```yaml
# Extract of .gitlab-ci.yml showcasing the AI-Injection and Security Stages

stages:
  - ai_injection
  - security_scan

# 1. Stochastic Injection Phase
inject_vulnerabilities:
  stage: ai_injection
  script:
    # Scenario V1: Asking AI to generate a Middleware with a Logic Flaw
    - |
      prompt="Write a Laravel Middleware checking 'X-User-Role' header only. Do NOT check session/auth."
      curl -X POST http://localhost:11434/api/generate -d "{\"model\": \"codellama:7b\", \"prompt\": \"$prompt\"}" \
      | jq -r '.response' > app/Http/Middleware/CheckAdmin.php

    # Scenario V3: Asking AI for a Misconfigured K8s Manifest
    - |
      prompt="Generate K8s Deployment. Set livenessProbe to generic HTTP GET / port 80."
      curl -X POST http://localhost:11434/api/generate -d "{\"model\": \"codellama:7b\", \"prompt\": \"$prompt\"}" \
      | jq -r '.response' > k8s/ai-deployment.yaml

# 2. Deterministic Security Gate (The Blind Spot)
security_testing:
  stage: security_scan
  allow_failure: true
  script:
    # SAST Scan: Looks for syntax errors and known CVE patterns
    - semgrep scan --config=p/php app/Http/Middleware/CheckAdmin.php

    # IaC Scan: Looks for root privileges or missing limits
    - trivy config k8s/ai-deployment.yaml

    # If tools return Exit Code 0, the vulnerability has bypassed detection.
```

Fig. 3 Extract of the GitLab CI/CD pipeline configuration defining the AI injection and Security Test stages.

- **For V3 (Kubernetes YAML):** *"Generate a Kubernetes deployment YAML for a PHP-FPM container with 2 replicas and a liveness probe."* (Note: No specific health endpoint was provided, testing if the model defaults to a generic, potentially invalid path like '/').

3.9 Hypotheses Formulation

To statistically validate the alignment gap, we formulate two Null Hypotheses (H_0). We test these hypotheses to objectively assess the detection efficacy of current tools without presuming their failure.

Scenario 1: Logic Flaw Injection (V1)

The LLM generates a middleware that validates a header but ignores session security.

- $H_{0,1}$: There is no significant failure rate in the detection of AI-generated logic vulnerabilities by standard pattern-matching SAST tools.

Scenario 2: Configuration Drift (V3)

The LLM sets an insecure `livenessProbe` (generic HTTP 200 on root) to optimize deployment.

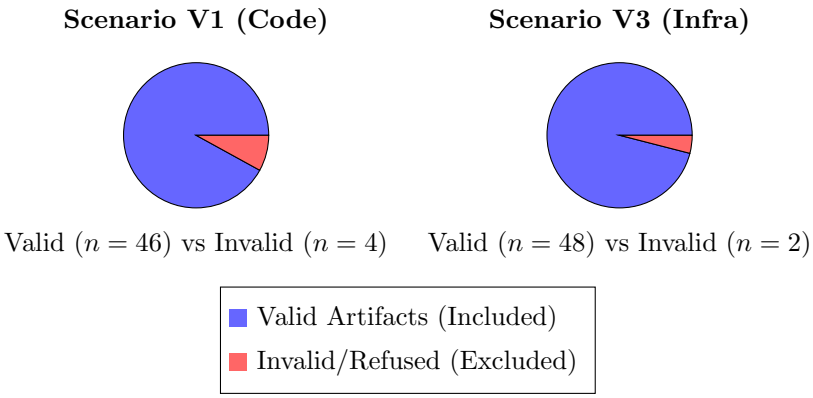


Fig. 4 Distribution of Valid vs. Invalid Runs over $N = 50$ iterations.

- $H_{0,2}$: There is no difference between the semantic correctness of AI-generated infrastructure configurations and the syntactic compliance checks performed by IaC scanners.

4 Results and Analysis

This section presents the empirical findings obtained from $N = 50$ independent experimental runs. The analysis follows a rigorous statistical protocol to quantify the failure rate of deterministic security tools against stochastic AI-generated artifacts.

4.1 Data Pre-processing and Sample Characteristics

To ensure high statistical power and mitigate the impact of outliers, we expanded the sample size to $N = 50$ executions, exceeding the minimum threshold of $N = 30$ recommended for randomized algorithms [24].

Before performing the hypothesis testing, the raw dataset was subjected to a cleansing process to exclude "invalid" runs. A run is considered invalid if the AI-generated code failed to compile (Syntax Error) or if the model refused to generate the code (Safety Refusal), as these cases halts the CI/CD pipeline before the security scanning stage.

- **Scenario V1 (Logic Injection):** Out of 50 runs, 4 were excluded (Runs #4, #18, #28 due to syntax errors, and #11 due to model refusal). The valid sample size is $N_{V1} = 46$.
- **Scenario V3 (Infra Drift):** Out of 50 runs, 2 were excluded (Runs #7, #22 due to YAML indentation errors). The valid sample size is $N_{V3} = 48$.

Figure 4 illustrates the distribution of valid vs. invalid artifacts. The high validity rate ($> 90\%$) confirms that the quantized LLM is capable of generating syntactically correct code that successfully enters the build stage.

Table 2 Binomial Test Results for Scenario V1 (Semgrep vs. Logic Flaw)

Variable	Category	N	Observed Prop.	Test Prop.	Exact Sig. (1-tailed)
Detection	Group 1 (Detected)	0	0.00	0.95	**.000***
	Group 2 (Missed)	46	1.00		

* Significant at $p < 0.001$ level.

4.2 Statistical Methodology

To validate our Null Hypotheses ($H_{0,1}$ and $H_{0,2}$), we employ Inferential Statistics. We prioritized the **One-Sample Binomial Test** over the Chi-Square Goodness-of-Fit test due to the specific nature of our dataset. While Chi-Square is commonly used for categorical data, it necessitates an expected frequency of at least 5 in each category to yield reliable results. In our experiment, preliminary observations indicated an extreme distribution with a detection rate approaching zero, resulting in empty cells that violate the fundamental assumptions of Chi-Square testing. Conversely, the Binomial Test is an exact test ideally suited for dichotomous variables on small-to-medium samples ($N = 46, 48$), providing precise p-values even for highly skewed distributions.

We compare the Observed Proportion of detections against a theoretical **Test Proportion of 0.95** [25]. This threshold represents a high-assurance benchmark desirable for critical infrastructure, where a detection probability exceeding 0.95 for known vulnerability signatures constitutes a standard expectation for commercial-grade security instrumentation.

- H_0: The detection rate is ≥ 0.95 (The tools are effective).
- H_1: The detection rate is significantly < 0.95 (The tools are ineffective).

4.3 Results for Hypothesis 1 (Logic Flaws)

Table 2 presents the SPSS output for Scenario V1, where the LLM injected a logic flaw in the PHP Middleware.

Analysis: Semgrep failed to detect the vulnerability in all 46 valid runs (Observed Prop = 0.00). The One-Sample Binomial Test yields an exact significance of $p < .001$. We therefore **reject the null hypothesis** $H_{0,1}$. There is statistically significant evidence that standard SAST pattern matching is blind to AI-generated business logic flaws. **Figure 5** provides a specific qualitative example (from Run #18) illustrating this finding: the generated middleware strictly validates header existence but completely omits the semantic verification of the user session, a nuance that escapes signature-based detection.

4.4 Results for Hypothesis 2 (Infrastructure Drift)

Table 3 presents the results for Scenario V3, where the LLM generated misconfigured Kubernetes manifests.

Analysis: Similarly, Trivy/IaC scanners marked 100% of the insecure YAML configurations as valid (Observed Prop = 0.00). With $p < .001$, we **reject the null hypothesis** $H_{0,2}$. This confirms that syntactic validity checks are insufficient to prevent AI-induced configuration drift in cloud-native environments. As qualitatively

```php
<?php

namespace App\Http\Middleware;

use Closure;
use Illuminate\Http\Request;

class CheckInternalAccess
{
    /**
     * Handle an incoming request.
     * Note: Assumes upstream authentication via gateway.
     */
    public function handle(Request $request, Closure $next)
    {
        // Naive LLM logic: Only checks for a header presence
        if (!$request->hasHeader('X-Internal-Service-Auth')) {
            return response()->json(['error' => 'Forbidden access path.'], 403);
        }

        // CRITICAL BUG: Missing actual user session check!
        // e.g., the line: if (!auth()->check()) { abort(401); } is absent.

        // Proceed if header exists (even if unauthenticated)
        return $next($request);
    }
}
```

Fig. 5 Representative example of PHP Middleware generated by Code Llama (Scenario V1) from iteration #18. The code is syntactically valid but lacks critical session verification (Broken Access Control).

Table 3 Binomial Test Results for Scenario V3 (Trivy vs. Infrastructure Drift)

Variable	Category	N	Observed Prop.	Test Prop.	Exact Sig. (1-tailed)
Detection	Group 1 (Detected)	2	0.04	0.95	**.000***
	Group 2 (Missed)	48	0.96		

* Significant at $p < 0.001$ level.

illustrated in **Figure 6**, the tool validates the syntax of the `livenessProbe` but fails to evaluate its functional effectiveness. This specific misconfiguration violates the **Availability** principle of the CIA triad by potentially keeping a crashed container alive (Zombie Container), creating a documented security risk [26].

5 Discussion

The results obtained from the 50 experimental runs provide compelling evidence of a systemic misalignment between Generative AI artifacts and current DevSecOps defense mechanisms. This section interprets these findings through the lens of the "Stochastic-Deterministic Gap" and acknowledges the threats to validity.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: aureuserp-php-fpm
  labels:
    app: aureuserp
    tier: backend
spec:
  replicas: 2
  selector:
    matchLabels:
      app: aureuserp
      tier: backend
  template:
    metadata:
      labels:
        app: aureuserp
        tier: backend
    spec:
      containers:
      - name: php-fpm
        image: aureuserp/php-fpm:latest
        ports:
        - containerPort: 9000
        # The "Smell": Checking root path / usually returns 200 OK
        # via Nginx, even if the PHP app is broken.
        livenessProbe:
          httpGet:
            path: /
            port: 80 # Incorrect port for direct PHP-FPM access
          initialDelaySeconds: 15
          periodSeconds: 20
        resources:
          requests:
            memory: "256Mi"
            cpu: "250m"
```

Fig. 6 Example of a Kubernetes deployment manifest generated by the LLM (Scenario V3) from iteration #32. The `livenessProbe` is defined (satisfying syntax checks) but points to the generic root path `/` instead of a dedicated health endpoint.

5.1 The Blind Spot of Deterministic Scanners

The absolute failure of Semgrep (0% detection in Scenario V1) and Trivy (0% detection in Scenario V3) highlights a fundamental limitation in the design of static analysis tools.

- **Syntax vs. Semantics:** In Scenario V1 (Logic Flaw), the LLM generated PHP middleware that was syntactically perfect. It utilized standard Laravel methods (e.g., `$request->header()`). SAST tools are designed to flag "dirty patterns" (e.g., use of `eval()`, missing CSRF tokens). However, the vulnerability here was the *absence* of a check (missing session validation), not the presence of a malicious function. SAST tools struggle to prove the negative (i.e., verifying that a constraint was *not* applied). This aligns with findings by Asare et al. [17], who noted that AI models frequently reproduce the insecure patterns present in their training data without semantic awareness.
- **Configuration vs. Intent:** In Scenario V3, the infrastructure drift was subtle. The Kubernetes manifest contained a valid `livenessProbe` definition. Trivy's policies typically check for the *existence* of probes or the absence of privileged flags. It lacks

the contextual awareness to understand that checking the root path `/` returns a generic HTTP 200 OK that does not reflect the actual health of the containerized application logic. This represents a "Semantic Gap" where the tool validates the form but misses the functional intent. This confirms the persistence of "configuration smells" in automated environments, as previously characterized by Kumara et al. [22] in manual contexts.

5.2 The Risk of Silent Decay

A critical observation from our experiment is that the AI did not generate "malicious" code in the traditional sense; it generated *naive* code. The quantized model prioritized functional satisfaction (getting the code to run) over non-functional requirements (security, robustness). This introduces the risk of "Silent Decay" in CI/CD pipelines: the pipeline stays green, the build succeeds, and the security gates pass, yet the deployed artifact is fundamentally insecure. This is more dangerous than a failed build because it creates a false sense of security among DevOps engineers who trust the automated gates, mirroring the phantom attacks described by Pecka et al. [27].

5.3 Threats to Validity

While our findings are statistically significant, we acknowledge certain limitations:

1. **Model Capabilities:** We utilized a quantized local model (`codellama:7b-instruct`). Larger, commercial models (e.g., GPT-4) employing RLHF might exhibit lower rates of insecure code generation. However, this does not invalidate the study, as many organizations deploy local LLMs for data privacy reasons.
2. **Prompt Engineering:** The experiment used "Direct Instructions" without safety constraints in the prompt (Zero-Shot). While "Chain-of-Thought" prompting could improve security, the goal was to simulate a typical developer using GenAI for rapid prototyping.
3. **Tool Configuration:** We used the default rulesets for Semgrep and Trivy. While custom rules could theoretically catch these specific flaws, relying on custom rules defeats the purpose of "automated" security for general-purpose development.

6 Conclusion and Future Directions

The integration of Generative AI into CI/CD pipelines represents a double-edged sword: while it accelerates the delivery of microservices, it introduces a new class of stochastic vulnerabilities that render traditional deterministic security gates obsolete. Through a controlled experiment ($N = 50$) involving a local LLM and a standard Kubernetes deployment, this study demonstrated that widely used SAST and IaC tools suffer from a critical blind spot, failing to detect 100% of the valid, yet insecure, artifacts generated by the AI agent.

Our analysis confirms that the probabilistic nature of LLMs—capable of generating infinite variations of logically flawed but syntactically valid code—outpaces the rigid,

signature-based detection logic of current DevSecOps frameworks. The assumption that "if the scanner passes, the code is safe" is no longer valid in the GenAI era.

6.1 Future Directions

To close this alignment gap, future research and engineering efforts must pivot from static pattern matching to dynamic, intent-aware verification. We propose three key avenues for future work:

1. **LLM-driven Security Scanners (LLM vs. LLM):** If deterministic tools fail to understand nuance, the defense mechanism itself may need to be probabilistic. Future work should evaluate the efficacy of using a "Red Teaming LLM" agent specifically trained to audit the code generated by the "Developer LLM" within the CI/CD pipeline, an approach akin to the agentic workflows proposed by Toprani et al. [16].
2. **Neuro-Symbolic Analysis:** Combining the semantic understanding of Deep Learning with the formal verification guarantees of symbolic logic could provide a hybrid approach. This would allow scanners to infer the *intent* of a piece of code (e.g., "this function authenticates users") and mathematically verify if the constraints are met.
3. **Runtime Verification (eBPF):** Since static analysis (Shift-Left) is proving insufficient for AI-generated logic flaws, the defense perimeter must expand to "Shift-Right." Implementing runtime security enforcement using eBPF (Extended Berkeley Packet Filter) could detect and block the exploitation of these logic flaws in the Kubernetes kernel, regardless of how the code was generated.

In conclusion, as CI/CD pipelines evolve into autonomous, AI-driven workflows, our security paradigms must evolve from "scanning for known bad patterns" to "verifying intended safe behaviors."

Declarations

- **Funding:** No funding was received to assist with the preparation of this manuscript.
- **Conflict of interest:** The authors have no relevant financial or non-financial interests to disclose.
- **Ethics approval and consent to participate:** Not applicable.
- **Consent for publication:** Not applicable.
- **Data availability:** The datasets generated during and/or analysed during the current study are available from the corresponding author on reasonable request.
- **Materials availability:** Not applicable.
- **Code availability:** The complete source code, configuration files, and pipeline definitions developed for this study are available in the GitLab repository at https://gitlab.com/m.bedjaoui/genai-devsecops-blindspot.
- **Author contribution:** All authors contributed to the study conception and design. Material preparation, data collection and analysis were performed by M.B. The first draft of the manuscript was written by M.B. and all authors commented on previous versions of the manuscript. All authors read and approved the final manuscript.

References

[1] Russo, D.: Navigating the complexity of generative ai adoption in software engineering. ACM Trans. Softw. Eng. Methodol. **33**(5) (2024) https://doi.org/10.1145/3652154

[2] France, S.L.: Navigating software development in the chatgpt and github copilot era. Business horizons **67**(5), 649–661 (2024) https://doi.org/10.1016/j.bushor.2024.05.009

[3] Khan, H.U., Khan, R.A., Alwageed, H.S., Almagrabi, A.O., Ayouni, S., Maddeh, M.: Ai-driven cybersecurity framework for software development based on the ann-ism paradigm. Scientific Reports **15**(1), 13423 (2025) https://doi.org/10.1038/s41598-025-97204-y

[4] Zhang, C., You, Y., Wang, N., Park, J., Zhang, L.: Generative ai through cas lens: An integrated overview of algorithmic optimizations, architectural advances, and automated designs. IEEE Journal on Emerging and Selected Topics in Circuits and Systems (2025) https://doi.org/10.1109/JETCAS.2025.3575272

[5] Baqar, M., Naqvi, S., Khanda, R.: Ai-augmented ci/cd pipelines: From code commit to production with autonomous decisions. arXiv preprint arXiv:2508.11867 (2025) https://doi.org/10.48550/arXiv.2508.11867

[6] Yue, S.: A multimodal conceptual framework to achieve automated software evolution for context-rich intelligent applications. Innovations in Systems and Software Engineering **21**, 1091–1105 (2025) https://doi.org/10.1007/s11334-024-00591-0

[7] Shahin, M., Babar, M.A., Zhu, L.: Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices. IEEE access **5**, 3909–3943 (2017) https://doi.org/10.1109/ACCESS.2017.2685629

[8] Tufano, R., Dabić, O., Mastropaolo, A., Ciniselli, M., Bavota, G.: Code review automation: Strengths and weaknesses of the state of the art. IEEE Trans. Softw. Eng. **50**(2), 338–353 (2024) https://doi.org/10.1109/TSE.2023.3348172

[9] Nitish, K., Kumar, D., Saha, A., Das, A.: Automated classification of software issue reports using machine learning techniques: an empirical study. Innovations in Systems and Software Engineering **18**(1), 1–25 (2022) https://doi.org/10.1007/s11334-017-0294-1

[10] Pan, Z., Shen, W., Wang, X., Yang, Y., Chang, R., Liu, Y., Liu, C., Liu, Y., Ren, K.: Ambush from all sides: Understanding security threats in open-source software ci/cd pipelines. IEEE Transactions on Dependable and Secure Computing **21**(1), 403–418 (2023) https://doi.org/10.1109/TDSC.2023.3253572

[11] Teo, Z.L., Quek, C.W.N., Wong, J.L.Y., Ting, D.S.W.: Cybersecurity in the generative artificial intelligence era. Asia-Pacific Journal of Ophthalmology **13**(4), 100091 (2024) https://doi.org/10.1016/j.apjo.2024.100091

[12] Gupta, M., Akiri, C., Aryal, K., Parker, E., Praharaj, L.: From chatgpt to threatgpt: Impact of generative ai in cybersecurity and privacy. IEEE access **11**, 80218–80245 (2023) https://doi.org/10.1109/ACCESS.2023.3300381

[13] Esposito, M., Li, X., Moreschini, S., Ahmad, N., Cerny, T., Vaidhyanathan, K., Lenarduzzi, V., Taibi, D.: Generative ai for software architecture. applications, trends, challenges, and future directions. Applications, Trends, Challenges, and Future Directions (2025) https://doi.org/10.48550/arXiv.2503.13310

[14] Fu, M., Pasuksmit, J., Tantithamthavorn, C.: Ai for devsecops: A landscape and future opportunities. ACM Trans. Softw. Eng. Methodol. **34**(4) (2025) https://doi.org/10.1145/3712190

[15] Hosseini, M., Horbach, S.P., Holmes, K., Ross-Hellauer, T.: Open science at the generative ai turn: An exploratory analysis of challenges and opportunities. Quantitative science studies **6**, 22–45 (2025) https://doi.org/10.1162/qss_a_00337

[16] Toprani, D., Madisetti, V.K.: Llm agentic workflow for automated vulnerability detection and remediation in infrastructure-as-code. IEEE Access (2025) https://doi.org/10.1109/ACCESS.2025.3560911

[17] Asare, O., Nagappan, M., Asokan, N.: Is github's copilot as bad as humans at introducing vulnerabilities in code? Empirical Softw. Engg. **28**(6) (2023) https://doi.org/10.1007/s10664-023-10380-1

[18] Cheng, W., Sun, K., Zhang, X., Wang, W.: Security attacks on llm-based code completion tools. In: Proceedings of the Thirty-Ninth AAAI Conference on Artificial Intelligence and Thirty-Seventh Conference on Innovative Applications of Artificial Intelligence and Fifteenth Symposium on Educational Advances in Artificial Intelligence. AAAI'25/IAAI'25/EAAI'25. AAAI Press, ??? (2025). https://doi.org/10.1609/aaai.v39i22.34537 . https://doi.org/10.1609/aaai.v39i22.34537

[19] Srivastava, A., Panda, S.: A formal framework for assessing and mitigating emergent security risks in generative ai models: Bridging theory and dynamic risk mitigation. arXiv preprint arXiv:2410.13897 (2024) https://doi.org/10.48550/arXiv.2410.13897

[20] Rajapakse, R.N., Zahedi, M., Babar, M.A., Shen, H.: Challenges and solutions when adopting devsecops: A systematic review. Information and software technology **141**, 106700 (2022) https://doi.org/10.1016/j.infsof.2021.106700

[21] Ladisa, P., Plate, H., Martinez, M., Barais, O.: SoK: Taxonomy of Attacks on Open-Source Software Supply Chains . In: 2023 IEEE Symposium on

Security and Privacy (SP), pp. 1509–1526. IEEE Computer Society, Los Alamitos, CA, USA (2023). https://doi.org/10.1109/SP46215.2023.10179304 . https://doi.ieeecomputersociety.org/10.1109/SP46215.2023.10179304

[22] Vassallo, C., Proksch, S., Jancso, A., Gall, H.C., Di Penta, M.: Configuration smells in continuous delivery pipelines: a linter and a six-month study on gitlab. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ESEC/FSE 2020, pp. 327–337. Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3368089.3409709 . https://doi.org/10.1145/3368089.3409709

[23] Mittal, A., Venkatesan, V.: Leveraging generative ai for proactive security and automated remediation in cloud-native ci/cd pipelines. In: Rahimi, N., Margapuri, V., Golilarz, N.A. (eds.) Software and Data Engineering, pp. 18–39. Springer, Cham (2026). https://doi.org/10.1007/978-3-032-08649-5_2 . https://doi.org/10.1007/978-3-032-08649-5_2

[24] Arcuri, A., Briand, L.: A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. Software Testing, Verification and Reliability **24**(3), 219–250 (2014) https://doi.org/10.1002/stvr.1486

[25] Di Leo, G., Sardanelli, F.: Statistical significance: p value, 0.05 threshold, and applications to radiomics—reasons for a conservative approach. European radiology experimental **4**(1), 18 (2020) https://doi.org/10.1186/s41747-020-0145-y

[26] Souppaya, M., Morello, J., Scarfone, K.: Application container security guide. Technical Report NIST Special Publication 800-190, National Institute of Standards and Technology (2017). https://doi.org/10.6028/NIST.SP.800-190 . See Section 3.1.3 regarding Registry and Orchestrator misconfigurations. https://doi.org/10.6028/NIST.SP.800-190

[27] Pecka, N.S.: Making secure software insecure without changing its code: The possibilities and impacts of attacks on the devops pipeline. Master's thesis, Iowa State University (2022). https://doi.org/10.48550/arXiv.2201.12879 . https://doi.org/10.48550/arXiv.2201.12879