

Securing the Software Development Lifecycle: Designing and Implementing a DevSecOps Framework for CI/CD

¹Bipin Shivaji Ghorpade, ²Dr. Ayaz Ahmed Faridi, ³Dr. Kalyan Devappa Bamane

¹Research Scholar, Department of Computer Science and Engineering, D Y Patil College of Engg
Akurdi, Pune, India

¹Associate professor, Department of CSE, Kalinga University Raipur, Atal Nagar-Nava Raipur,
Chhattisgarh, India

³Department of Computer Engg, Kalinga University Raipur, Atal Nagar-
Nava Raipur, Chhattisgarh, India

Corresponding Author Email - ghorpadebipin23@gmail.com

Abstract - Continuous integration and continuous delivery/deployment (CI/CD) enable frequent software releases, but security weaknesses may remain undetected when security testing is separated from development and deployment activities. This paper presents a DevSecOps framework that incorporates dependency analysis, static code analysis, container image scanning, and dynamic application security testing into an AWS-based CI/CD workflow. The framework combines Snyk, SonarQube, Aqua Security, and OWASP ZAP with security gates, centralized reporting, and remediation feedback. Its evaluation considers reported detection percentages, detection times, gate-triggered blocking, and resource overhead. The preliminary observations indicate that the tools address different security concerns and impose different execution costs. However, the available measurements lack the denominators, repeated-run statistics, and ground-truth annotations required for a comparative effectiveness assessment. The contribution is therefore a structured integration framework and a qualified analysis of preliminary observations, rather than a validated ranking of security tools. The framework provides a basis for repeatable evaluation of security controls across the software delivery lifecycle.

Keywords—DevSecOps; CI/CD; secure software development; security automation; vulnerability management; security gates.

1. Introduction

CI/CD practices automate software integration, testing, packaging, and delivery. Continuous integration validates frequent code changes, while continuous delivery maintains deployable software and continuous deployment automates release to production. These practices shorten feedback cycles, but functional tests alone cannot establish that source code, third-party dependencies, container images, and running applications satisfy security requirements.

DevSecOps integrates security responsibilities and verification activities into development and operational workflows. Its objective is to make security assessment continuous and actionable. Automated checks can provide feedback before release, while development, security, and operations teams jointly manage remediation and exceptions. This approach requires both technical integration and clear ownership of security decisions.

A practical challenge is that individual scanners examine different artifacts and produce findings with different meanings. Dependency analysis concerns vulnerable software packages; static analysis examines source code; container scanning evaluates packaged components; and dynamic testing examines a running application. A coherent workflow must determine where each check runs, how findings are interpreted, and which conditions prevent an artifact from progressing.

This paper proposes a staged DevSecOps framework for an AWS-based CI/CD environment. Its contributions are: (i) a mapping of complementary security checks to pipeline stages; (ii) an explicit security-gate and remediation workflow; and (iii) a structured presentation of preliminary measurements with their interpretation limits. The framework emphasizes traceable decisions rather than assuming that successful completion of automated scans guarantees security.

2. Literature Review and Research Gap

The literature survey considers tool selection, continuous security testing, pipeline security, and organizational adoption. Table 1 summarizes the studies considered in the survey. Figures 1 and 2 organize security tools by practice, distinguishing tool categories from individual tool examples.

Table 1. Summary of the studies listed in the literature survey

Study / authors	Year	Approach	Reported focus
DevSecOps Toolchain for CI/CD: A Systematic Mapping Study — R. Oladipo, S. Misra, and S. Khan	2021	Systematic mapping	Identification and functional evaluation of 43 tools.
A Framework for Evaluating and Selecting Security Tools for DevSecOps — M. Shabtai and M. Kessem	2021	Framework development	Tool selection based on effectiveness, usability, and integration.
DevSecOps: A Survey on Recent Advances and Practices — V. Sharma, S. Misra, and S. Khan	2021	Survey	Overview of practices and supporting tools.
Continuous Security Testing in DevOps: A Systematic Mapping Study — E. Masoumi and M. G. Jaatun	2020	Systematic mapping	Review of 45 studies on continuous security testing.
A Model-Driven Framework for Security Testing of DevOps Pipelines — S. Falcão, A. Sampaio, and J. C. C. Maldonado	2020	Framework development	Model-driven integration of security testing.
DevSecOps Tools: A Systematic Mapping Study and Research Agenda — J. Martins, G. Rodrigues, and F. Silva	2020	Systematic mapping	Classification of 46 tools by function and usage.
Securing DevOps: A Systematic Mapping Study — M. Balasubramanian and P. Mehta	2020	Systematic mapping	Review of 80 studies addressing DevOps security.
A Survey on Security Challenges and Solutions in DevOps — A. Saberi and R. A. Koushkestani	2019	Survey	Security challenges and potential technical solutions.

Study / authors	Year	Approach	Reported focus
DevOps Security: An Exploration Study with Insights and Opportunities for Researchers — A. Kumar, A. Bhattacharya, and M. Medela	2019	Exploratory study	Security practices and research opportunities.
A Comparative Study of Security Testing Tools for Web Applications in DevOps — P. Karunakaran, M. Bhuyan, and R. Islam	2019	Comparative study	Comparison of four tools for effectiveness, usability, and integration.

These themes motivate an integration problem: a list of available tools does not by itself define a reproducible security workflow. Pipeline design must connect scan outputs to release decisions, distinguish code-quality findings from confirmed vulnerabilities, and record the overhead introduced by each check. The present framework addresses these operational requirements.

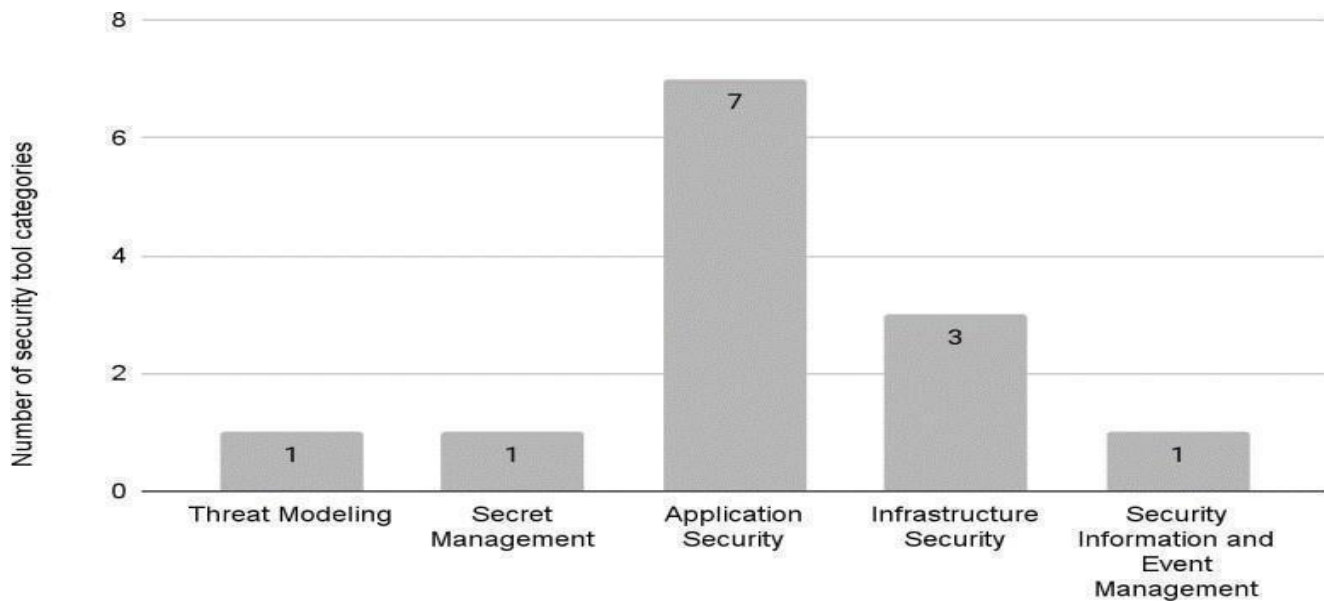


Fig. 1. Distribution of security-tool categories across security practices. Each value represents the number of distinct tool categories assigned to the corresponding practice.

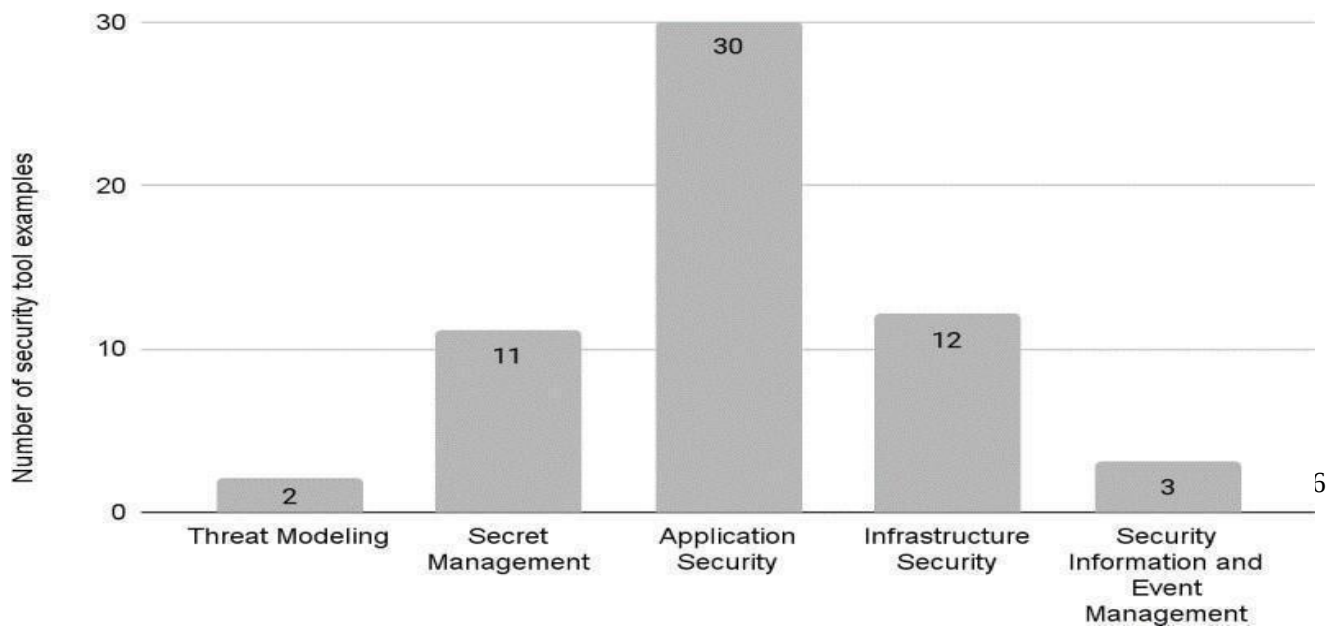


Fig. 2. Distribution of security-tool examples across security practices. Each value represents the number of individual tool examples assigned to the corresponding practice.

3. Problem Statement and Objectives

The problem addressed is how to incorporate complementary security checks into a CI/CD pipeline while maintaining explicit release criteria and measurable execution overhead. Security checks applied only near release can delay remediation, whereas poorly configured automated checks can generate excessive alerts or block releases without sufficient justification.

The framework aims to detect issues at appropriate lifecycle stages, prevent promotion when defined security policies are violated, provide actionable feedback to developers, and collect evidence for evaluating detection coverage and pipeline cost. It also separates security-policy rejection from scanner malfunction and infrastructure failure, because these events require different corrective actions.

4. Proposed DevSecOps Framework

4.1. Architecture and Workflow

The framework organizes delivery into source acquisition, build and static checks, image analysis, staging deployment, dynamic testing, release approval, and operational monitoring. AWS CodePipeline coordinates the workflow and AWS CodeBuild supports build and testing tasks. The deployment target may use the AWS environment described in the implementation plan, including Elastic Beanstalk where appropriate. The precise repository, deployment configuration, and resource specifications must be recorded for reproducibility.

Figure 3 illustrates the proposed workflow. A code change initiates the pipeline. Dependency analysis and source-code checks run before artifact promotion. Container scanning evaluates the packaged image before deployment to the test environment. Dynamic testing then evaluates the application in staging. Production promotion occurs only when all required checks complete and the applicable security gates pass. Failed gates return findings to developers, who remediate the issues and initiate a new run.

4.2. Tool Integration

Table 2 defines the intended responsibility of each tool. These roles are complementary and do not imply that the tools share an equivalent vulnerability population. Findings should retain the originating tool, artifact identifier, severity, affected component, and remediation status. Where multiple tools report the same issue, deduplication is required before calculating aggregate counts.

Table 2. Mapping of security tools to pipeline stages and assessment targets.

Tool	Pipeline stage	Assessment target	Intended output
Snyk	Build / dependency validation	Third-party dependencies	Dependency findings and remediation guidance
SonarQube	Source analysis / build validation	Source-code quality and security-related issues	Analysis findings and quality-gate status
Aqua Security	Image validation before staging	Container images and packaged components	Image findings and policy status
OWASP ZAP	Dynamic testing in staging	Running application within an authorized test scope	Dynamic findings and scan report

4.3. Environment Configuration and Security Gates

The AWS environment is configured with network segmentation, restricted connectivity, and appropriate access permissions for build and deployment components. Security groups and virtual private cloud settings should reflect the access required by each stage. Pipeline credentials and access to scan reports require protection because the pipeline itself is part of the application's security boundary.

Security gates translate scan findings into explicit promotion decisions. The design requires documented severity thresholds, quality criteria, exception rules, and ownership. The numerical thresholds are not specified in the available experimental description and must therefore be supplied before reproduction. A failed mandatory scan must not be interpreted as a clean result. Approved exceptions should identify an owner, justification, and expiry date.

A passing gate indicates compliance with the configured policy for the checks performed; it does not establish the absence of vulnerabilities. Scanner coverage, configuration, and application reachability remain relevant limitations. Dynamic scanning identifies potential weaknesses but does not automatically remediate them; remediation remains a separate development activity.

4.4. Reporting, Monitoring, and Iteration

Scan reports and gate outcomes are associated with the corresponding commit, build, and artifact. Notifications communicate blocked stages and actionable findings to the responsible team. Monitoring also records execution failures and resource use so that operational problems can be distinguished from security-policy violations.

The improvement cycle consists of reviewing findings, confirming their relevance, applying fixes, and rerunning affected checks. Changes to rules, scanner configuration, and exceptions should be versioned. This makes it possible to explain why the same application may produce different results across runs.

5. Evaluation Methodology

5.1. Evaluation Design

The evaluation design compares a baseline CI/CD workflow with a security-enabled workflow using the same application revisions and execution resources. Secure and intentionally vulnerable application variants are proposed to test detection and gate behavior. However, application identifiers, vulnerability inventories, tool versions, run counts, and raw logs are not available in the present experimental description. The observations in Section 6 are consequently treated as preliminary descriptive values.

The original research plan also includes surveys, interviews, and organizational case studies. No participant counts, instruments, interview records, or qualitative analyses are reported. These activities therefore remain planned extensions and are not used as evidence for the present findings. Any future participant-based study should document recruitment, informed consent, confidentiality measures, and its analysis procedure.

5.2. Execution Procedure

The proposed evaluation first establishes the baseline execution time and resource consumption. Security checks are then integrated at the stages specified in Table 2. The same code revisions are processed under consistent conditions, and each run records scanner results, elapsed times, gate decisions, and resource measurements. Confirmed vulnerabilities and non-vulnerable cases are needed to distinguish true detections from incorrect alerts. Repeated runs are required to quantify variability.

Dynamic testing must run against an authorized, isolated staging application with a defined scope. Authentication, reachable routes, test data, and scan duration should be recorded. A staging result should not be described as a production vulnerability finding unless production was actually assessed under an appropriate test procedure.

5.3. Evaluation Metrics

Table 3 defines the metrics needed for a reproducible evaluation. These definitions specify the intended analysis; they cannot retrospectively establish how the reported percentages were calculated. In particular, an alert judged low priority is not automatically a false positive. A false positive is a finding determined not to represent the issue asserted by the scanner.

Table 3. Evaluation metrics and required measurement definitions.

Metric	Definition	Required evidence
Detection rate (%)	$100 \times TP / (TP + FN)$	Ground-truth vulnerabilities within each tool's defined scope
Precision (%)	$100 \times TP / (TP + FP)$	Manual or independently validated finding labels
Gate-blocking rate (%)	$100 \times \text{blocked eligible runs} / \text{total eligible runs}$	Stage-specific run counts and gate outcomes
Detection time	Elapsed time from a stated start event to a relevant finding	Timestamped events and a consistent timing origin
Pipeline overhead (%)	$100 \times (\text{security-enabled time} - \text{baseline time}) / \text{baseline time}$	Matched baseline and security-enabled runs
Resource overhead (%)	$100 \times (\text{security-enabled usage} - \text{baseline usage}) / \text{baseline usage}$	Specified resource, units, sampling interval, and matched baseline

TP, FP, and FN denote true positives, false positives, and false negatives, respectively. Resource overhead should be reported separately for CPU, memory, storage, or another measured quantity. End-to-end pipeline overhead cannot be inferred by adding scan times when tasks overlap or execute in parallel.

6. Preliminary Results and Discussion

6.1. Detection and Gate Outcomes

Table 4 consolidates the reported detection percentages, time ranges, and blocking rates. The percentages describe different assessment targets, and their denominators are not documented. They therefore cannot establish a ranking of tool effectiveness or an overall vulnerability-detection rate. The 64% reported for SonarQube includes code-quality issues and should not be interpreted as a confirmed vulnerability recall value.

Table 4. Preliminary tool-specific observations reported for the integrated pipeline.

Tool	Reported detection	Reported time	Reported blocking
Snyk	72% of dependency-related vulnerabilities	1–2 min	18% of builds
SonarQube	64% of code-quality issues, including security-related issues	1–2 min	12% of builds
Aqua Security	53% of container-image security risks	3–5 min	10% of deployments
OWASP ZAP	47% of dynamic application vulnerabilities	8–10 min	6% of releases

The original report associates Snyk and SonarQube with early feedback, Aqua Security with image analysis, and OWASP ZAP with longer dynamic testing. This pattern is consistent with the different stages being assessed,

but the timing origin is not defined sufficiently to determine whether the values are scan duration or time to first detection. The blocking percentages likewise use different run populations and must not be summed to obtain an overall failure rate.

6.2. Resource Consumption

The reported resource observations are summarized in Table 5. The resource type, measurement interval, baseline, and aggregation procedure are unspecified. These figures therefore indicate a reported overhead trend rather than a reproducible performance measurement or a demonstrated cost-benefit result.

Table 5. Preliminary resource-overhead observations reported for security scanning.

Tools	Reported additional load	Interpretation constraint
Snyk and SonarQube	Approximately 10%	Unclear whether measured separately or jointly; resource and baseline unspecified.
Aqua Security and OWASP ZAP	Approximately 20–25%	Unclear whether measured separately or jointly; resource and baseline unspecified.

6.3. Interpretation of Findings

The principal architectural benefit is layered assessment: an issue missed by one type of analysis may still be identified at another stage. Early dependency and source checks provide an opportunity to remediate before artifact promotion. Container and dynamic checks extend assessment to packaged components and application behavior. Nevertheless, complementary coverage does not prove that a particular proportion of all vulnerabilities has been eliminated.

The longer reported dynamic-testing time highlights a potential trade-off between scan depth and delivery speed. The appropriate response depends on application risk, reachable functionality, and execution constraints. Parallel independent checks or different scan schedules may reduce delay, but their benefits must be measured rather than assumed.

Qualitative observations in the draft suggest that dynamic testing required more configuration than the other integrations. No standardized usability instrument, task-completion measurements, or participant data are available, so the observations cannot support a general claim of ease of use. Similarly, false-positive and false-negative behavior is mentioned without labeled counts, preventing calculation of precision, recall, or error rates.

The evaluation does not demonstrate that this toolchain would have prevented a specific historical breach. Such claims require evidence about the incident’s attack path and the controls actually assessed. Application scanning also does not, by itself, establish protection against compromise of build infrastructure or artifact distribution.

6.4. Limitations and Threats to Validity

The primary limitation is incomplete experimental reporting. Missing sample sizes and ground truth prevent verification of detection percentages. Unspecified tool versions, rules, deployment details, and vulnerability-database snapshots limit reproducibility. The absence of repeated trials and variability estimates prevents statistical comparisons. Missing baseline measurements also prevents a defensible conclusion about total pipeline latency or cost.

Different scanners evaluate different targets, creating a construct-validity risk if their percentages are treated as equivalent accuracy measures. Generalization is also limited because application size, language, dependency structure, and test coverage are unspecified. A broader evaluation should include multiple applications and workloads, consistent configurations, and independently adjudicated findings.

7. Conclusion and Future Work

This paper presents a DevSecOps framework that integrates dependency analysis, source-code assessment, container scanning, and dynamic testing into an AWS-based CI/CD workflow. Security gates and remediation feedback connect these checks to artifact-promotion decisions. The reported preliminary observations illustrate differences in detection scope, timing, and blocking behavior, but do not establish comparative accuracy, aggregate detection effectiveness, or statistically validated performance improvement.

Future work should provide a reproducible application benchmark, ground-truth vulnerability labels, versioned configurations, and matched baseline measurements. Repeated executions should quantify variability in scan duration and resource use. Planned surveys and case studies may complement this technical evaluation once their methods and results are documented. These steps would permit a stronger assessment of the framework's effectiveness, overhead, and practical adoption.

References

- [1] T. Rangnau, R. v. Buijtenen, F. Fransen and F. Turkmen, "Continuous Security Testing: A Case Study on Integrating Dynamic Security Testing Tools in CI/CD Pipelines," 2020.
- [2] K. Zunnurhain and S. R. Duclervil, "A New Project Management Tool Based on DevSecOps," 2020.
- [3] E. C. Burkard, "Usability Testing within a DevSecOps Environment," 2020.
- [4] J. S. S. Vadlamudi, "A Novel Approach to Onboarding Secure Cloud-Native Acquisitions into Enterprise Solutions," 2021.
- [5] R. N. Rajapakse, M. Zahedi, M. A. Babar and H. Shen, "Challenges and Solutions When Adopting DevSecOps: A Systematic Review," 2021.
- [6] J. P. Costa, M. Cancar and N. Petrovic, "Security in DevSecOps: Applying Tools and Machine Learning to Verification and Monitoring Steps," 2023.
- [7] A. Ibrahim, A. H. Yousef and W. Medhat, "DevSecOps: A Security Model for Infrastructure as Code Over the Cloud," 2022.
- [8] T. Chen and H. Suo, "Usability Testing within a DevSecOps Environment," 2022.
- [9] A. M. Putra and H. Kabetta, "Implementation of DevSecOps by Integrating Static and Dynamic Security Testing in CI/CD Pipelines," 2022.
- [10] X. Sun, Y. Cheng, X. Qu and H. Li, "Design and Implementation of Security Test Pipeline Based on DevSecOps," 2021.