WWW.IJRPR.COM

International Journal of Research Publication and Reviews

Journal Homepage: www.ijrpr.com ISSN 2582-7421

An Intelligent DevSecOps Framework for Securing Cloud Based CI/CD Pipelines

Valluru Damodara Rao, Mr. Suraj Soren

Department of Computer Science and Engineering, GMRIT DEEMED TO BE UNIVERSITY, Rajam, Andhra Pradesh, India
Contact: 24341A05O6@gmrit.edu.in

ABSTRACT –

Cloud software delivery relies on continuous integration and continuous delivery or deployment (CI/CD), but the same automation can distribute insecure code, dependencies, credentials, or cloud configurations at speed. This paper presents a conceptual intelligent DevSecOps framework that places deterministic security checks throughout a cloud-based pipeline and adds risk prioritization for developer attention. It does not claim an implementation or experimental result. The design combines source and dependency analysis, secret detection, infrastructure-as-code validation, container checks, policy gates, deployment controls, and runtime feedback. The framework is intended to help college-level readers understand how security can be built into, rather than appended to, automated delivery.

Keywords: *DevSecOps; cloud computing; CI/CD pipeline; cloud security; DevOps; software supply chain*

I. INTRODUCTION

The security objective is not to eliminate all delivery risk, which is impossible, but to make risk visible and manageable before it reaches a high-impact environment. A cloud pipeline should therefore protect both the software being delivered and the automation that delivers it. This includes repository permissions, runner isolation, artifact storage, deployment credentials, and the cloud accounts used by the workflow. When these components are treated as ordinary infrastructure, security becomes part of normal engineering design instead of a last-minute approval activity.

For college-level projects, the term shift left means moving appropriate security work closer to development. It does not mean moving every production-security activity into a developer laptop. Design review, secure coding, dependency checking, and secret detection can occur early; runtime monitoring and incident handling necessarily continue after deployment. A complete DevSecOps model connects these stages so that a lesson from production can improve an earlier requirement, test, or policy.

Cloud computing provides on-demand computing resources over a network. In practice, cloud applications are often changed through a CI/CD pipeline: developers commit code, an automated process builds and tests it, and an approved artifact is deployed. DevOps brings development and operations together so that this process can be fast and repeatable. DevSecOps extends that collaboration by treating security as a shared responsibility across planning, coding, building, deployment, and operations. OWASP describes the aim as putting security into DevOps early enough to keep pace with CI/CD [3].

A pipeline is both a delivery path and a security-sensitive system. A hard-coded token can expose an account; an outdated library can introduce a known vulnerability; and an unsafe Terraform or Kubernetes configuration can create a public service or excessive permission. Build systems, deployment credentials, and third-party packages also form a software supply chain. The Secure Software Development Framework (SSDF) recommends secure practices that can be integrated into each software development life cycle [2]. Therefore, security checks must be automated, visible, and tied to delivery decisions.

II. BACKGROUND AND RELATED WORK

Automated testing should be selected according to the question it can answer. SAST can identify risky patterns in source code but may not understand the final cloud environment. SCA can identify known library issues but cannot determine whether every issue is reachable in the application. IaC checks can identify an unsafe resource definition before deployment but need current policy rules. DAST can reveal behavior in a running test environment but depends on test coverage and safe configuration. Combining their evidence produces a stronger basis for a decision than treating any single result as complete assurance.

A software bill of materials supports vulnerability response because it describes the components used to produce a release. It should be generated consistently, linked to an artifact, and protected from unauthorized change. Build provenance can add information about how and where an artifact was produced. These records are valuable for traceability, but their quality depends on protected build systems and trustworthy inputs. They should be paired with access control, review, and verification rather than treated as a simple compliance document.

Supply-chain assurance is important because a release is assembled from more than the application source code. The build definition, dependencies, package registry, container base image, runner, artifact repository, deployment workflow, and cloud identity can all influence the final release. Provenance and component records help an organization answer practical questions after a vulnerability is announced: which artifact contains the component, which environment received it, and which release decision permitted it. These records support investigation but do not replace patching or configuration review.

Security testing also needs a feedback path to development. A finding that is delivered only as a long report at the end of a release cycle is less useful than a finding linked to the relevant commit, package, container layer, or infrastructure resource. Clear feedback encourages remediation as part of ordinary engineering work. It also allows repeated issues to be converted into training, reusable secure defaults, or a policy rule that prevents recurrence.

Continuous integration means that developers merge small changes frequently and validate them with automated builds and tests. Continuous delivery keeps a releasable artifact ready for deployment, while continuous

deployment automatically releases an approved change. These practices reduce the size of each change and make feedback faster, but they also make pipeline reliability important. If the pipeline accepts an unsafe component or configuration, the same automation can repeat the mistake quickly across environments.

A useful security design therefore combines prevention, detection, and response. Prevention includes protected branches, least-privilege service accounts, trusted package sources, and policy constraints. Detection includes scanners, build logs, image inspection, and runtime monitoring. Response includes blocking an unsafe release, creating a traceable ticket, rotating a leaked secret, and improving a rule after review. This layered approach is more realistic than depending on one scanner or one final manual review.

The literature and guidance also show that security evidence must be interpreted in context. A dependency finding in a development-only tool may require a different response from the same finding in an internet-facing production service. Asset value, exposure, exploitability, compensating controls, and the age of the finding can influence the response. However, context should make decisions clearer, not weaken basic controls such as secret protection or approved access management.

DevOps mainly optimizes collaboration, automation, and operational reliability. DevSecOps keeps these aims but adds explicit security requirements, security testing, evidence, and response. It does not mean that every issue is silently accepted by a tool. Teams need clear ownership, thresholds, and a way to decide whether a finding should block a release. Feio et al. present a DevSecOps framework centered on continuous security testing across pipeline phases [1]. Their work supports the central idea that security activities can be automated inside a CI/CD process.

Several control families are relevant. Static application security testing (SAST) examines source code without executing it. Software composition analysis (SCA) identifies open-source components and their known vulnerabilities. Secret detection searches commits and configuration files for passwords, tokens, and private keys. Infrastructure-as-code (IaC) checks inspect declarative cloud resources before creation. Container scanning checks image packages and configuration. Dynamic application security testing (DAST) examines a running application. These checks are complementary; one tool cannot establish complete security.

The reviewed literature emphasizes fragmented tools, alert overload, weak runtime monitoring, and configuration-policy gaps. The proposed use of GitHub Actions or Jenkins, SonarQube, Trivy, Docker, Kubernetes, Terraform, OWASP ZAP, Open Policy Agent, STRIDE, and NIST SSDF is reflected in the framework. The framework is grounded in verified standards and publications.

NIST container guidance identifies security concerns in container technologies and recommends controls for them [4]. SLSA provides a staged approach to build integrity and provenance for software artifacts [5]. SBOM guidance defines a software bill of materials as a record of component details and supply-chain relationships [6]. Together, these sources motivate checking both application code and the path by which an artifact is produced.

III. PROBLEM STATEMENT OBJECTIVES AND QUESTIONS

The problem also includes decision delay. If tool results are spread across dashboards, a developer may need to compare several formats and decide whether a finding belongs to a source file, dependency, image, or cloud setting. This increases the chance that important evidence is ignored or that teams apply inconsistent decisions. The proposed framework addresses this operational problem by collecting evidence in one record, preserving the original source, and associating each decision with a policy rule or named reviewer.

A further objective is to support explainability. A developer should be able to see why a pipeline stopped, which artifact or configuration caused the decision, whether an exception exists, and how to proceed. This is important when intelligent ranking is used because an unexplained model result can be difficult to validate. Explainability is therefore a functional requirement of the framework, not an optional presentation feature.

Cloud-based pipelines can produce many findings from different scanners, while developers still need to understand which finding matters first. Without coordinated policy, a serious secret exposure may be handled like a low-impact code-quality issue. In addition, cloud configuration, container images, pipeline permissions, and runtime behavior can introduce risks outside the source-code review process. A practical framework must connect evidence to an understandable release decision without assuming that machine learning can replace deterministic checks.

This paper has four objectives: (1) identify security controls suitable for major CI/CD stages; (2) consolidate their evidence into one risk record; (3) define policy-based allow, review, and block decisions; and (4) propose an evaluation plan for security coverage, usefulness, and delivery impact. The research questions are: RQ1: Which automated controls should be placed at each stage of a cloud CI/CD pipeline? RQ2: How can combined evidence be prioritized without removing human accountability? RQ3: How can policy gates and runtime feedback support safer deployment?

IV. PROPOSED INTELLIGENT DEVSECOPS FRAMEWORK

The framework can be implemented with modular interfaces so that a team is not tied to one vendor. For example, a source scanner, dependency scanner, or container scanner can publish a common finding format even when their internal reports differ. The orchestrator invokes the tools, stores evidence, and waits for the policy decision. This modular design allows a team to replace a tool, update a rule, or add a new cloud control without rebuilding the whole delivery process.

The framework also distinguishes between a security finding and a security incident. A finding is evidence that requires a decision or remediation; an incident is an event that may require containment, investigation, and communication under an incident-response process. Runtime monitoring can create either type depending on the evidence. Treating them differently helps prevent low-confidence pipeline alerts from being handled as emergencies while ensuring that a confirmed compromise receives appropriate escalation.

Figure 3 places the major controls at practical pipeline stages. The figure is a design map rather than a claim that every project must use every control. A small application may begin with source protection, secret detection, and dependency checking; a higher-risk cloud service may add image verification, policy-as-code, and detailed runtime monitoring.

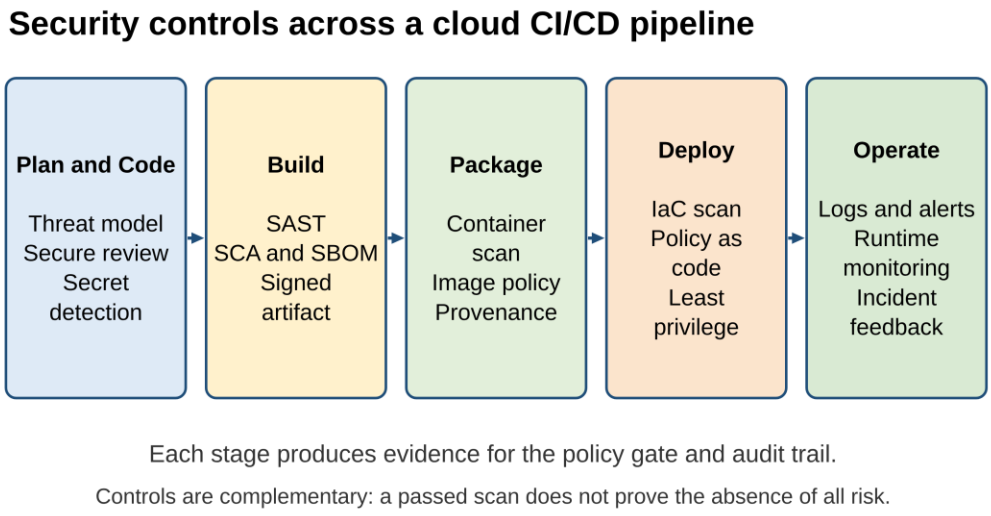


Fig. 3. Security controls mapped across cloud CI/CD stages.

The framework separates confidence from severity. Severity describes the potential impact reported by a source such as a vulnerability database or scanner. Confidence describes how strongly the evidence identifies a real issue in the current artifact or environment. A high-severity, low-confidence result may require review, whereas a confirmed exposed secret may be blocked immediately even if no numerical risk model is available. This distinction makes policy decisions more defensible and reduces the temptation to treat every alert as equally urgent.

Feedback information is retained as an audit record with the build identifier, artifact digest, evidence summary, decision, exception if any, and deployment target. An artifact digest is a cryptographic identifier for the produced file or image. Recording it links a deployment to the exact item that passed a gate. The record can support rollback, incident response, and later evaluation of whether policies were working as intended.

The framework uses a normalized finding record so that tools can contribute evidence in a consistent form. At a minimum, the record contains the tool name, pipeline run, repository and commit, affected component, finding category, severity, confidence, exposure context, policy rule, owner, and status. A normalized record avoids separate dashboards becoming separate decision processes. It also makes it possible to audit why a release was allowed, reviewed, or blocked.

Risk prioritization is applied after evidence collection, not before it. The framework first removes obvious duplicates and links related findings to the same artifact or service. It then assigns a transparent priority using deterministic factors such as a critical severity, public exposure, production environment, and a known exploit signal. The result is a queue for review, not an automatic declaration that a vulnerability is exploitable. The pipeline can display both the original scanner severity and the contextual priority so that developers can challenge or correct an inaccurate classification.

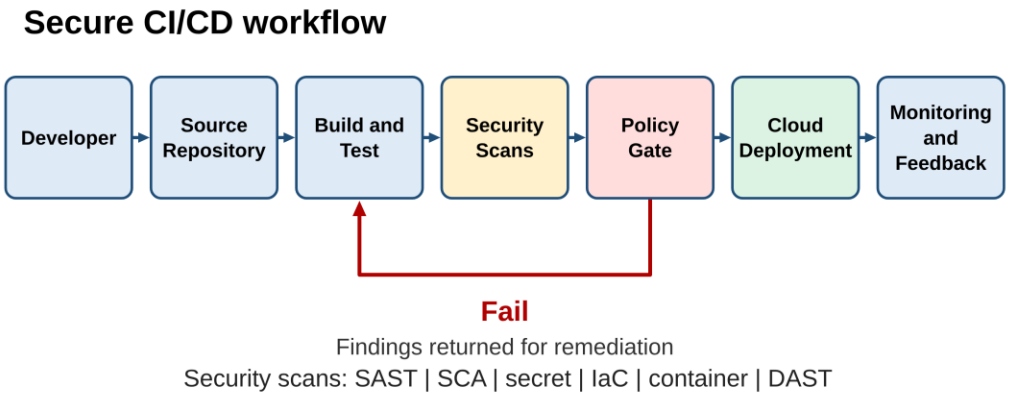


Fig. 1. Secure CI/CD workflow with feedback after a failed security decision.

Exception handling is part of secure automation. If a rule must be temporarily waived, the request should identify the affected artifact, business reason, compensating control, responsible approver, expiry date, and planned remediation. An exception must be visible to the policy gate and audit log. This prevents a short-term delivery decision from becoming an undocumented permanent bypass.

Figure 1 shows the basic workflow. Each code change begins in a controlled repository and proceeds through build and test. Security scans produce structured findings. A policy gate then applies non-negotiable rules, such as blocking confirmed secrets, forbidden public exposure, or unapproved critical risks. When a build fails, the finding is returned to the developer for remediation. A successful, traceable artifact can be deployed, then monitored for runtime events that inform later decisions.

Figure 2 expands this workflow into modules. The orchestration layer receives commits and controls repeatable jobs. Evidence collectors run SAST, SCA, secret, IaC, container, and optional DAST checks. Their normalized output includes the asset, finding type, severity, location, confidence, and remediation status. The risk-prioritization module enriches findings with deterministic factors such as exposure, asset criticality, and policy context. It may also use an explainable learned or statistical score as a secondary signal. For example, EPSS estimates the likelihood that a published CVE will be exploited in the next 30 days [7]. Such a score must not overrule a mandatory rule or conceal its reason.

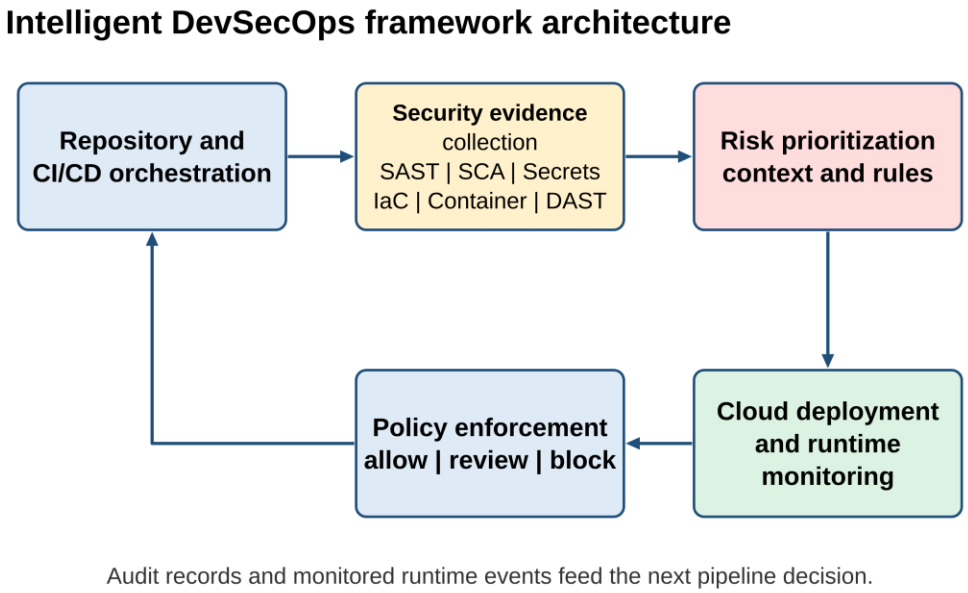


Fig. 2. Conceptual architecture of the proposed intelligent DevSecOps framework.

Policy enforcement converts security evidence into action. The policy should be version-controlled and reviewed like code. A release may be allowed when only accepted, low-risk findings remain; sent to review when the context is uncertain; or blocked for critical rule violations. The deployment module uses least privilege, protected secrets, signed or traceable artifacts, and environment separation. Monitoring collects logs, alerts, configuration changes, and runtime anomalies. It creates an audit trail and sends useful feedback to the pipeline, but it does not claim to prevent every attack.

V. SYSTEM ARCHITECTURE AND SECURITY CONTROLS

Artifact integrity deserves special attention. After a build is approved, the pipeline should deploy the identified artifact rather than rebuilding a different copy later. Immutable tags or content digests can help the deployment process select the exact evaluated image. If an artifact changes after scanning, the security evidence may no longer apply. This principle is especially useful for containerized applications where a mutable image tag can otherwise point to a different image over time.

Cloud configuration checks should consider identity, network exposure, encryption, logging, and resource configuration. Examples of policy questions include whether a storage location is public, whether a service has an overly broad role, whether a database encryption setting is absent, and whether a security group allows unnecessary inbound access. The rule should state the expected condition and the reason for it so that a developer can correct the configuration without guessing.

The policy gate in Figure 4 turns these checks into a controlled decision. A block should state the required remediation, while review should state the uncertainty and responsible owner. Allow does not mean risk-free; it means that the defined evidence and policy requirements for that release were satisfied. This wording keeps the decision honest and avoids overstating what automated security can guarantee.

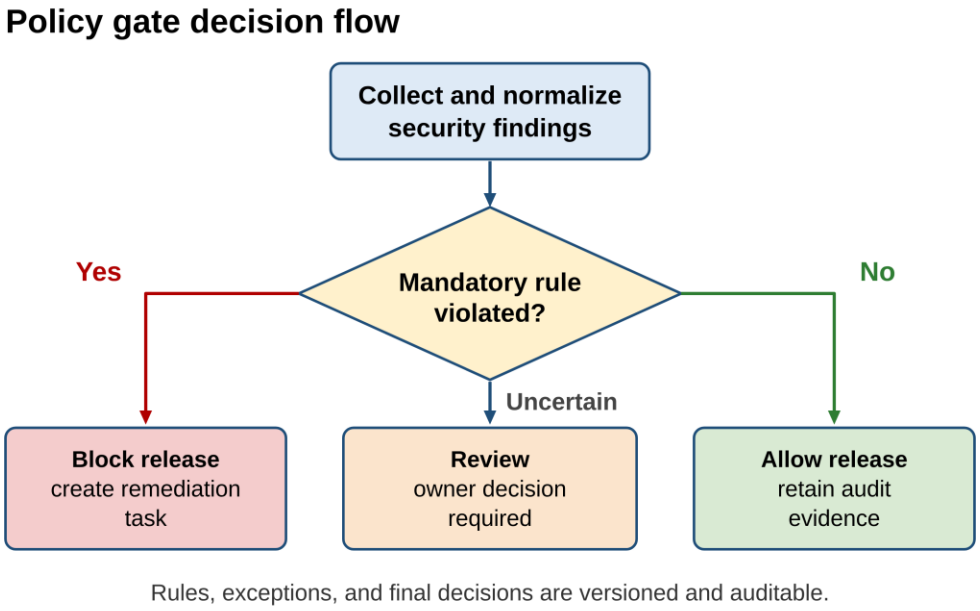


Fig. 4. Policy-gate decision flow for allow, review, and block outcomes.

Security gates should be progressive. Early commit checks should be quick and provide immediate feedback, for example secret detection, basic linting, and dependency policy checks. More expensive checks, such as full container scanning and authenticated DAST, can run after a build or in a pre-production environment. This ordering helps keep developer feedback useful while still applying deeper validation before deployment. The exact sequence depends on the application, threat model, and available infrastructure.

Access control is also a pipeline control. Build runners, deployment accounts, repositories, container registries, and cloud projects should use separate identities with only the permissions required for their task. Sensitive values should be supplied by a managed secret service rather than stored in repository files or pipeline logs. Audit records should identify the workflow, artifact, policy version, and actor or service identity associated with a deployment decision.

Policy-as-code gives the team a repeatable way to express requirements such as disallowing public storage, privileged containers, overly broad network rules, or unapproved base images. Policy files need reviews, testing, version history, and a change-management process. A policy that is not understood or is too strict will be bypassed; a policy that is never measured may leave important coverage gaps.

Area	DevOps	DevSecOps
Primary focus	Speed, collaboration, reliability	Speed with explicit security accountability
Security timing	Often separate or late	Built into planning, code, build, release, and operations
Release decision	Tests and operational checks	Tests plus security evidence and policy gate
Evidence	Build and deployment logs	Security findings, policy decisions, provenance, and audit records
Response	Fix after detection	Prioritize, block when required, remediate, and learn from runtime feedback

Table I. Comparison of DevOps and DevSecOps.

At the repository stage, branch protection, code review, signed commits where feasible, and secret detection reduce basic risks. During build, SAST and SCA examine application logic and dependencies; an SBOM can record component information. IaC validation and policy-as-code inspect cloud resources before they exist. Container scanning checks the image that will run, and NIST recommends container-aware controls that can operate at container scale [4]. Before release, the policy gate evaluates explicit thresholds, exception approvals, and evidence freshness. In operation, logging, alerts, access controls, and configuration monitoring provide a detection and learning loop.

The proposed intelligent element is intentionally bounded. It groups duplicates, ranks related findings, and adds context such as exploit likelihood and production exposure. It must display the factors behind its score, record the model or rule version, and allow a reviewer to override it. False positives, biased historical labels, adversarial inputs, and model drift are risks. Deterministic checks for secrets, known forbidden configurations, and approved policies remain authoritative.

VI. RESEARCH METHODOLOGY

The proposed evaluation can include a scenario-based demonstration before a larger experiment. Each scenario begins with a controlled commit or configuration change and records the pipeline stages, findings, priority, policy result, remediation action, and final status. Example scenarios could include a hard-coded test token, an outdated vulnerable package, a public cloud-storage configuration, and an image containing an unapproved package. The study should document whether the finding was detected at the intended stage and whether the explanation was sufficient for remediation.

A comparison baseline should be defined carefully. The baseline might use the same build environment and the same selected security tools but omit centralized evidence handling and policy orchestration. This makes it easier to evaluate the contribution of the framework rather than simply the presence of scanners. The evaluation should record costs such as additional pipeline time, configuration effort, and reviewer time alongside security-related outcomes.

To reduce bias, the test cases should represent several classes of issue rather than only cases that the chosen tools are known to detect. For example, the test suite can include a vulnerable dependency, an accidental test secret, an unsafe network rule, a container running with unnecessary privilege, and a benign configuration that should not be blocked. Each case should have a documented expected decision. A reviewer can then assess whether the pipeline provides a correct and understandable outcome.

The study should use a review protocol for policy exceptions and false positives. Two reviewers can independently label a small sample of findings as actionable, non-actionable, or unclear, then reconcile disagreements using a documented rule. This will not eliminate judgment, but it makes evaluation more transparent. The paper should report the size and limitations of the labeled sample rather than generalize beyond the evidence.

The evaluation should include both technical and human factors. Technical assessment can measure whether planned controls execute at the intended stage and whether policy decisions are reproducible from saved evidence. Human assessment can ask reviewers whether the finding summary identifies the affected asset, explains the decision, and supports remediation. Short interviews or a structured questionnaire can complement pipeline metrics, provided participants, questions, and analysis methods are documented.

Threat modeling should be performed before selecting controls. A simple STRIDE-based review can consider spoofing of identities, tampering with source or artifacts, repudiation where evidence is incomplete, information disclosure through secrets or logs, denial of service against pipeline infrastructure, and elevation of privilege through cloud roles. The output should map threats to controls and tests. This makes the framework adaptable rather than a fixed list of tools.

Reproducibility requires that the future study records its sample application version, commit history, scanner versions, rule sets, policy thresholds, cloud region or test environment, and procedure for creating test cases. Results should distinguish a tool finding from a manually confirmed issue and should state any incomplete coverage. These details allow later researchers to repeat the work or explain why two environments produced different results.

This work is a conceptual design study, not an implemented system or an experimentally evaluated framework. The recommended methodology is design science: define the problem from the reviewed literature and verified guidance; map controls to pipeline stages; design the architecture and decision rules; and evaluate the design using a reproducible prototype in later work. The design rationale is aligned with SSDF secure-development practices [2], OWASP DevSecOps guidance [3], and supply-chain integrity concepts in SLSA [5].

A future implementation can use a small cloud-native sample application with a Git repository, a CI/CD service, Docker, a Kubernetes test cluster, Terraform, and selected scanners. Test cases should contain intentionally vulnerable dependencies, synthetic exposed secrets that are revoked and nonfunctional, insecure IaC configurations, and a controlled policy violation. No genuine credentials or harmful payloads should be used. A baseline pipeline without integrated gates can be compared with the proposed pipeline under identical commits.

Evaluation measures should include control coverage by pipeline stage, true and false positive findings after manual labeling, time from commit to decision, time to remediate, number of blocked unsafe test releases, policy decision consistency, and reviewer usefulness. A study should report dataset details, tool versions, rule configurations, thresholds, hardware or cloud environment, and known limitations. Accuracy alone is inadequate for rare security events; reviewers should inspect precision, recall, workload, and the practical consequences of a false allow or false block.

VII. PROPOSED EVALUATION AND DISCUSSION

The three-outcome decision model is illustrated in Figure 4. It allows the policy gate to protect mandatory conditions without pretending that every security question has a fully automatic answer. A review path is appropriate for a medium-confidence finding in a sensitive component, an expiring exception, or a newly introduced cloud service. The reviewer decision should be recorded with a reason and expiry date where relevant, allowing later audit and policy improvement.

Figure 5 shows why runtime monitoring is connected to the pipeline. After deployment, monitoring can reveal unexpected process execution, unusual network behavior, repeated authentication failures, or configuration drift. A confirmed lesson should be analyzed by the responsible team and then translated into a safer default, scanner rule, test, or policy improvement. The diagram does not imply that every runtime alert should automatically modify a build; review protects the pipeline from noisy or misleading signals.

Meaningful adoption metrics can include the percentage of releases with complete evidence, the number of policy exceptions that expire on time, the time needed to explain a blocked release, and the proportion of runtime lessons converted into preventive controls. These measures describe process maturity, not proof that all attacks are prevented. They are useful because they connect the framework to operational work that teams can observe and improve.

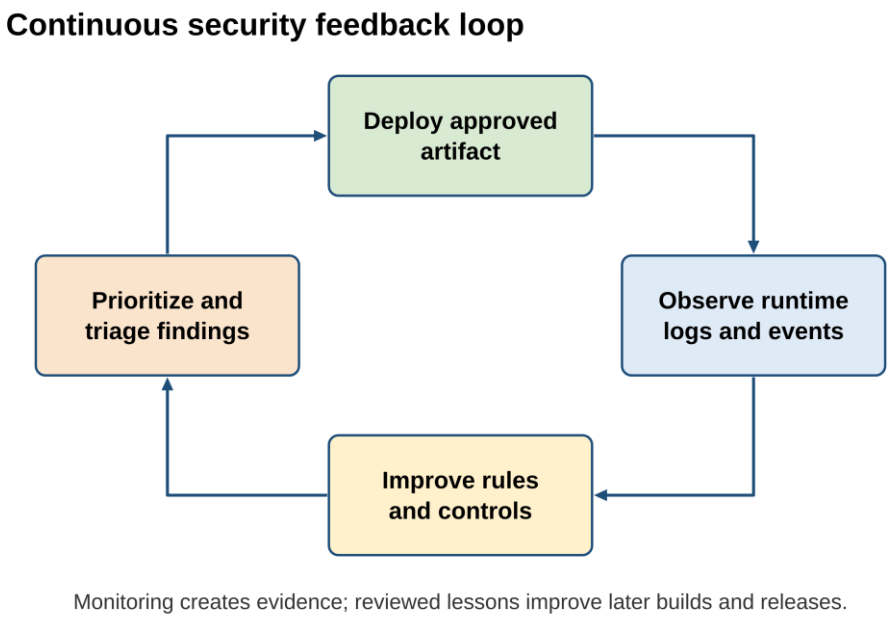


Fig. 5. Runtime feedback loop for continuous security improvement.

A practical release policy can use several outcomes instead of a simple pass or fail. Allow means that required evidence is present and no blocking rule applies. Review means that the evidence is incomplete, conflicting, or context dependent and needs an accountable decision. Block means that a mandatory control failed. This three-way decision avoids forcing uncertain findings into an automatic allow or an automatic block. It also gives teams a way to measure how often policies need refinement.

The framework should be adopted incrementally. A first phase can protect repositories and detect secrets. A second phase can add dependency, container, and IaC checks. A later phase can add provenance, deployment policy, and runtime feedback. Each phase should have a named owner, measured objectives, and an agreed rollback plan for tool failure. Incremental adoption recognizes that DevSecOps is a socio-technical change involving people, process, and technology.

A future study can compare three decision strategies: a baseline pipeline with conventional build checks, a pipeline with independent security scanners, and the proposed coordinated framework with evidence normalization and policy gates. The comparison should avoid treating the number of alerts as a security outcome. More alerts may mean more coverage, more noise, or both. The useful question is whether the process identifies meaningful unsafe conditions in time for a justified release decision.

The proposed framework is intended to support developers rather than transfer all responsibility to a scoring engine. A short remediation explanation, the affected file or artifact, policy reason, and suggested next action can reduce the time needed to understand a finding. Conversely, an opaque score without supporting evidence can increase mistrust. The design therefore keeps scanner evidence, deterministic policy reasons, and human override records visible together.

Runtime monitoring closes the loop but should be operated carefully. Logs and security events can contain personal, confidential, or operationally sensitive information. Collection and retention should follow organizational policy, access should be restricted, and alerts should be tested so that teams are not overwhelmed by low-value notifications. Runtime findings can create a backlog item or policy improvement; they should not automatically rewrite pipeline policy without review.

No performance measurements, percentages, or charts are reported because no experiment was provided. The expected outcome is not a claim of effectiveness; it is a testable proposition: integrating coordinated controls and clear policy gates should make security evidence available earlier and make release decisions easier to audit. The evaluation should test whether this happens without unacceptable pipeline delay. Human review is especially important for high-impact findings, new cloud services, and model-generated prioritization.

The framework has practical trade-offs. More scans can increase build time and create noisy alerts. Strict gates can interrupt delivery if policies are immature. Tool outputs may overlap or disagree. Cloud service configurations vary by provider, so policies need local adaptation. An organization should start with a small set of high-confidence, high-impact rules, measure the effects, and expand coverage gradually. OWASP SAMM provides a risk-driven, measurable way to improve secure development practices over iterations [8].

VIII. LIMITATIONS AND FUTURE WORK

The visual flows in this paper are conceptual. They are intended to communicate component relationships and decision points, not prescribe a production topology. A real deployment would need architecture-specific decisions for network boundaries, data residency, secret storage, build-runner isolation, identity federation, logging retention, and incident response. Those details should be developed with the organization that owns the application and its risk decisions.

Future research can evaluate whether contextual prioritization reduces triage effort without hiding important findings. It can also test policy portability across cloud providers and compare human-review workload before and after evidence normalization. Any study involving operational logs or real vulnerabilities should follow ethical and organizational approval requirements, avoid exposing credentials, and protect participant or system information.

This document is not a substitute for a threat model, regulatory assessment, or organization-specific security policy. Different applications process different data, use different cloud services, and face different operational risks. The proposed controls should therefore be tailored after identifying assets, trust boundaries, legal obligations, and acceptable risk. Future work can study how the same framework changes for public web applications, internal services, and data-processing workloads.

The framework assumes that organizations can operate or access basic security tools and maintain policy rules. Small teams may have limited expertise, budget, or time to tune scanners. A future implementation should therefore compare an open-source reference toolchain with a managed-service alternative and report maintenance effort as well as security findings. It should also consider accessibility of findings for developers who are not security specialists.

Risk scoring depends on the quality and timeliness of its inputs. A vulnerability database can be incomplete, a severity score can lack environment context, and a machine-learning signal can change as attacker behavior changes. The framework addresses these issues by keeping rules and source evidence visible, requiring review for uncertain cases, and keeping permanent controls independent of a learned score. Future work should test calibration and error cases rather than assuming that intelligence is always accurate.

The framework is conceptual and does not establish causal improvement in security, speed, cost, or compliance. It does not compare named commercial products, train a machine-learning model, or prove that any scanner finds every vulnerability. The references below were selected because their publication details or official sources could be verified. Future work should implement the reference pipeline, publish safe test artifacts and policy files, conduct a controlled evaluation, assess false-positive burden, and examine explainability, privacy, and governance for risk-ranking models.

IX. CONCLUSION

Cloud CI/CD pipelines need security controls that match the speed of automated delivery. This paper proposes an intelligent DevSecOps framework in which scanning, evidence normalization, explainable risk prioritization, policy gates, controlled deployment, monitoring, and audit records form one continuous loop. Its core contribution is a clear conceptual design rather than a claim of measured effectiveness. Deterministic controls remain the foundation; intelligent analysis is a transparent aid for prioritization and triage. A future reproducible prototype and evaluation can test whether the design improves security decision-making while preserving useful delivery flow.

REFERENCES

[1] C. Feio, N. Santos, N. Escravana, and B. Pacheco, "An Empirical Study of DevSecOps Focused on Continuous Security Testing," in 2024 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW), 2024, pp. 610-617, doi: 10.1109/EUROSPW61312.2024.00074.

[2] M. Souppaya, K. Scarfone, and D. Dodson, Secure Software Development Framework (SSDF) Version 1.1: Recommendations for Mitigating the Risk of Software Vulnerabilities, NIST SP 800-218, Feb. 2022, doi: 10.6028/NIST.SP.800-218.

[3] OWASP Foundation, "OWASP DevSecOps Guideline," accessed Oct. 16, 2026. [Online]. Available: https://owasp.org/projects/devsecops-guideline

[4] M. Souppaya, J. Morello, and K. Scarfone, Application Container Security Guide, NIST SP 800-190, Sep. 2017, doi: 10.6028/NIST.SP.800-190.

[5] SLSA, "SLSA specification," version 1.2, accessed Oct. 16, 2026. [Online]. Available: https://slsa.dev/spec/v1.2/

[6] Cybersecurity and Infrastructure Security Agency, "SBOM Resources Library: NTIA Minimum Elements," accessed Oct. 16, 2026. [Online]. Available: https://www.cisa.gov/topics/cyber-threats-and-advisories/sbom/sbomresourceslibrary

[7] FIRST, "Exploit Prediction Scoring System (EPSS)," accessed Oct. 16, 2026. [Online]. Available: https://www.first.org/epss/

[8] OWASP Foundation, "OWASP Software Assurance Maturity Model (SAMM)," accessed Oct. 16, 2026. [Online]. Available: https://owasp.org/projects/samm

APPENDIX A. QUALITY CONTROL SUMMARY

Document format: single-column layout. References: 8, each cited in the text. Figures: 5 conceptual diagrams. Tables: 1. Status: conceptual proposal; no implementation or experimental evaluation is claimed.