Journal of Advances in Developmental Research (IJAIDR)

E-ISSN: 0976-4844 ● Website: www.ijaidr.com ● Email: editor@ijaidr.com

Automated Vulnerability Management in DevSecOps Pipelines for SaaS Platforms: A Practical Framework for SAST, DAST, Dependency Scanning, and Controlled Remediation

Lalith Chandra Bandaru[1], Praveen Chaitanya Jakku[2], Mohammed Shakeer Bandrevu[3]

[1,2,3]Independent Researcher

Abstract:
SaaS platforms are released, updated, and configured at a pace that traditional vulnerability management processes were not designed to support. Security findings that are discovered after deployment often move through separate dashboards, manual triage, ticket queues, and delayed sprint cycles before they are fixed. This delay creates unnecessary exposure, especially when the issue could have been detected earlier in the software delivery pipeline. DevSecOps addresses this problem by moving security checks closer to development and release workflows, but simply adding more scanners does not guarantee better security. Without aggregation, deduplication, prioritization, and clear remediation paths, scanning can increase noise rather than reduce risk.

This article presents a practical framework for automated vulnerability management in DevSecOps pipelines for SaaS platforms. The framework combines static application security testing, dynamic application security testing, dependency scanning, secrets detection, infrastructure-as-code validation, and controlled remediation workflows. It emphasizes risk-based gating, developer-friendly feedback, safe automation, and auditable remediation. The goal is not to replace security engineers or developers, but to reduce avoidable delay, improve consistency, and ensure that repeatable security fixes are handled earlier and more reliably in the delivery lifecycle.

Keywords: DevSecOps, SaaS security, vulnerability management, SAST, DAST, dependency scanning, CI/CD, automated remediation, secure software development, software supply chain security.

1. Introduction

Modern SaaS delivery depends on frequent releases, distributed development teams, reusable services, cloud platforms, third-party libraries, and automated deployment pipelines. This model gives organizations the speed they need, but it also expands the security surface. A vulnerability may appear in application code, an API endpoint, an open-source dependency, a container image, a Kubernetes manifest, a CI/CD script, or a cloud configuration. In many cases, the weakness is not caused by one large failure, but by a small insecure decision that passes through the delivery process unnoticed.

Traditional vulnerability management was usually built around periodic scans and post-release review. A security tool detected an issue, a security team reviewed it, a ticket was created, and an application team fixed it when capacity became available. That process may work for slower release cycles, but it is not well suited for SaaS platforms where code, configuration, and dependencies change continuously. The

longer a finding waits outside the engineering workflow, the greater the chance that it will be delayed, misunderstood, or deprioritized.

DevSecOps attempts to solve this problem by integrating security into development and operations practices. However, DevSecOps adoption is not only a tooling problem. Rajapakse et al. identified tool integration, automation, cultural alignment, and unclear security ownership as recurring challenges in DevSecOps adoption [1]. Akbar et al. also show that organizations need structured decision-making when adopting DevSecOps because implementation depends on technical, process, and people-related factors [2]. These findings match what many engineering teams experience in practice: adding tools is easy; making them useful, trusted, and sustainable is harder.

The purpose of this article is to present a practical framework for automated vulnerability management in DevSecOps pipelines. The framework is designed for SaaS-oriented environments where application security, dependency risk, infrastructure security, and release governance need to work together. It does not claim that every vulnerability can be fixed automatically. Instead, it separates detection from remediation and applies automation only where the fix is safe, repeatable, testable, and auditable.

2. Background

Secure software development guidance has increasingly moved toward continuous and integrated practices. The NIST Secure Software Development Framework recommends secure development activities that can be integrated into different software development lifecycle models [3]. NIST guidance on DevSecOps for microservices also highlights automated pipelines, security testing, deployment controls, and feedback mechanisms for cloud-native applications [4]. These publications support a practical direction: security should be part of the engineering system, not a separate review performed only at the end.

At the same time, SaaS platforms depend heavily on third-party components. Open-source dependencies accelerate delivery, but they also introduce supply chain risk. A dependency scanner may report many vulnerable packages, but not every reported vulnerability is equally relevant to the deployed application. Pashchenko et al. explain this problem through the idea of “actually vulnerable dependencies,” showing that vulnerability reporting can overstate risk when usage and exploitability are not considered [5]. This is important for DevSecOps pipelines because noisy dependency findings can quickly overwhelm teams.

Security testing also needs to move beyond one technique. SAST can identify many code-level weaknesses before deployment, but it may miss runtime behavior and configuration problems. DAST can test deployed behavior, but it may not explain the exact source-code location. Dependency scanning is essential for software supply chain visibility, but it must be prioritized carefully. Infrastructure-as-code scanning can catch deployment misconfigurations, but it cannot fully validate application logic. A useful DevSecOps pipeline combines these methods and gives developers a single, understandable view of risk.

Recent work on secure software development and testing also supports a more integrated approach. Casola et al. propose a model-based methodology that connects secure development and testing activities in a way that fits modern DevOps and DevSecOps pipelines [6]. Research on automated vulnerability repair is also progressing. Chen et al. studied neural transfer learning for repairing security vulnerabilities in C code [7], and Zhou et al. proposed broader input-based automatic vulnerability repair methods [8]. These approaches are promising, but production pipelines still need guardrails. An automatically suggested fix should not be treated as safe until it is reviewed, tested, and traceable.

3. Problem Statement

The main problem addressed in this article is the delay between vulnerability discovery and vulnerability remediation in SaaS delivery pipelines. Most organizations already use some security tools, but those tools often operate as separate systems. One scanner reports code issues, another reports dependency vulnerabilities, another reports secrets, and another reports infrastructure misconfigurations. Each tool may use its own severity levels, output format, dashboard, and workflow.

This creates three practical problems.
First, findings are fragmented. Developers may need to check multiple tools to understand the security status of one service. Security teams may see repeated alerts without a clear link to service ownership or release status. Operations teams may receive infrastructure findings without enough application context.
Second, findings are noisy. Security tools can produce duplicate findings, false positives, and low-context warnings. If developers repeatedly see findings that are not actionable, they lose trust in the pipeline. A DevSecOps program that creates alert fatigue can quietly become ineffective, even when the tools themselves are technically capable.
Third, remediation is inconsistent. Some findings are fixed quickly because the responsible developer is already working on the affected code. Others remain open because they are assigned late, lack ownership, or require coordination across teams. In SaaS environments, this delay can affect production systems, customer data, compliance posture, and release confidence.
The proposed framework addresses these issues by treating vulnerability management as a pipeline function rather than a separate security queue.

4. Proposed Framework

The framework has six layers: scanning, aggregation, normalization, prioritization, remediation routing, and feedback.

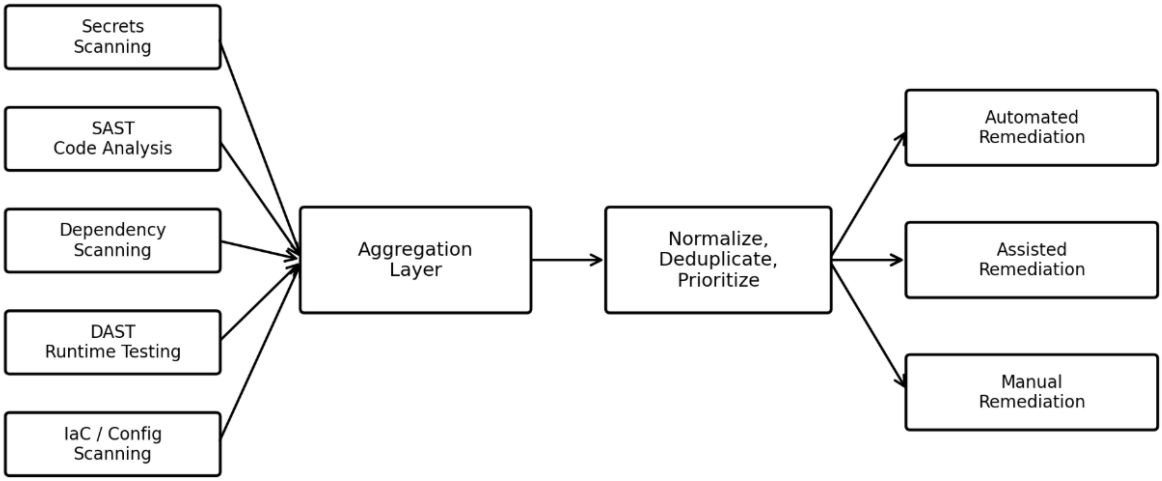

Fig. 1. Unified vulnerability management framework showing how secrets scanning, SAST, dependency scanning, DAST, and infrastructure/configuration scanning feed into aggregation, deduplication, prioritization, and remediation routing.

4.1 Scanning Layer

The scanning layer places different security checks at the pipeline stages where they provide the most useful feedback.
Secrets scanning should run as early as possible. It can be used in pre-commit hooks, pull request checks, and CI jobs to detect API keys, tokens, private keys, passwords, and service credentials before they move further into the delivery chain.
SAST should run during pull request or commit validation. It helps detect insecure coding patterns such as injection risks, unsafe input handling, insecure deserialization, weak cryptography usage, hardcoded secrets, and missing validation.
Dependency scanning should run during the build stage. It should inspect manifest files, lock files, transitive dependencies, and container layers. The output should include package name, affected version,

fixed version, severity, exploitability, and whether the dependency is used in the application path where possible.

DAST should run against a deployed test or staging environment. This allows the pipeline to inspect runtime behavior, API exposure, authentication handling, authorization behavior, response headers, and input validation.

Infrastructure-as-code scanning should validate Terraform, CloudFormation, Kubernetes manifests, Helm charts, Dockerfiles, and other deployment files. This is important because many SaaS risks come from insecure defaults, excessive permissions, public exposure, or missing runtime controls.

4.2 Aggregation Layer

The aggregation layer collects results from all scanning tools and converts them into a common finding format. A normalized finding should include scanner name, vulnerability category, severity, confidence, affected file or endpoint, affected package or resource, evidence, recommendation, owner, pipeline stage, and remediation status.

This layer prevents each tool from becoming a separate queue. Instead of asking developers and security engineers to check many dashboards, the pipeline can present one consolidated view of security risk for each service, release, or environment.

4.3 Normalization and Deduplication Layer

Different tools often describe the same issue differently. One tool may report a finding using a CWE category, another may use a CVSS score, and another may use a custom severity label. Normalization maps these results into a consistent severity and confidence model.

Deduplication reduces repeated alerts. For example, a vulnerable endpoint may be detected by SAST and DAST. A weak configuration may be detected by both a Kubernetes scanner and a cloud security scanner. The framework groups related findings using attributes such as vulnerability category, affected component, file path, endpoint, package name, resource name, and service owner.

Deduplication is not only a reporting improvement. It changes how teams respond. One well-explained finding with multiple pieces of evidence is more useful than three separate alerts that appear unrelated.

4.4 Risk-Based Prioritization Layer

A mature DevSecOps pipeline should not block every finding. Blocking everything creates frustration and slows delivery without necessarily reducing meaningful risk. Instead, the pipeline should use risk-based prioritization.

Priority should consider severity, exploitability, exposure, asset criticality, environment, dependency reachability, available fix, and business context. A critical vulnerability in an internet-facing authentication service should be handled differently from a medium-severity issue in an internal test-only component. Similarly, a vulnerable dependency that is present but not reachable should not be treated the same as one used in a production request path [5].

This does not mean lower-risk findings should be ignored. They should still be tracked, owned, and remediated. The difference is that the pipeline should reserve hard blocking for risks that justify stopping the release.

4.5 Remediation Routing Layer

Once a finding is validated and prioritized, it should move into the right remediation path. The framework uses three paths: automated remediation, assisted remediation, and manual remediation.

Automated remediation is suitable for predictable and low-risk fixes. Examples include patch-level dependency upgrades, standard security header additions, removal of exposed debug settings, correcting known insecure configuration defaults, and rotating leaked credentials through an approved workflow.

Assisted remediation is suitable when the system can suggest a fix but should not merge it without human review. Examples include infrastructure permission changes, moderate-risk dependency upgrades, container base image changes, or configuration updates that affect production behavior.
Manual remediation is required for architectural vulnerabilities, business logic flaws, authentication redesign, authorization model changes, cryptographic design changes, and major-version dependency migrations. These issues require engineering judgment and cannot be safely reduced to a template.
Infrastructure automation used in remediation should follow risk-aware practices such as modular design, secure variable handling, least-privilege execution, validation, and auditability [10]. This is especially important when automated remediation modifies infrastructure or deployment configuration rather than only application code.

4.6 Feedback Layer

The feedback layer improves the system over time. When a developer marks a finding as a false positive, the security rule should be reviewed. When an automated remediation pull request fails tests, the template should be corrected or disabled. When the same vulnerability appears across many services, the organization may need a shared secure library, coding standard, or platform-level control.
This feedback loop turns scanning into vulnerability management. Without it, the pipeline only produces alerts. With it, the organization learns from repeated patterns and reduces future risk.

5. Pipeline Implementation Model

A practical implementation can be organized across five stages: developer, commit, build, staging, and pre-production.
At the developer stage, lightweight checks should run quickly. Secrets scanning, linting, and basic static checks can prevent obvious mistakes before code reaches the shared repository. These checks should be fast and simple because developers will not accept slow local feedback during active development.
At the commit or pull request stage, SAST and dependency scanning should run with clear policy thresholds. Critical findings and high-confidence high-severity findings may block the merge. Medium and low findings may create tickets automatically, depending on the organization's policy.
At the build stage, the pipeline should scan dependencies, container images, build artifacts, and infrastructure-as-code templates. This stage is important because it validates the deployable artifact, not just the source code.
At the staging stage, DAST and API security testing should run against a realistic deployed environment. This stage helps identify runtime issues that cannot be fully understood from code alone.
At the pre-production stage, risk acceptance, release approvals, and final security validation should occur for sensitive services. This stage should not become a manual bottleneck for every change, but it should provide stronger control for high-risk releases.
Automated vulnerability remediation should also be connected with deployment-safety practices such as readiness validation, progressive rollout, observability, and rollback planning, particularly in Kubernetes-based microservice environments [9]. A generated fix is useful only if the platform can deploy it safely, observe its behavior, and roll it back if needed.

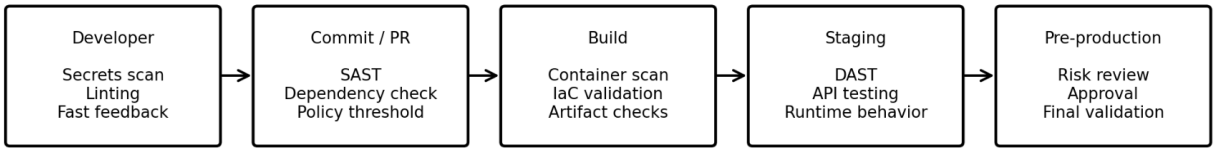

Fig. 2. Security gates across the DevSecOps pipeline, showing how different checks are placed at developer, commit, build, staging, and pre-production stages.

6. Controlled Remediation

Automated remediation is useful when it is applied carefully. It should not be treated as a goal by itself. The goal is safe and timely risk reduction.

A remediation workflow should satisfy four conditions before it is allowed to run automatically. First, the fix should be deterministic. The same input should produce the same expected change. Second, the change should be small enough to review and understand. Third, automated tests should validate the result. Fourth, rollback should be possible.

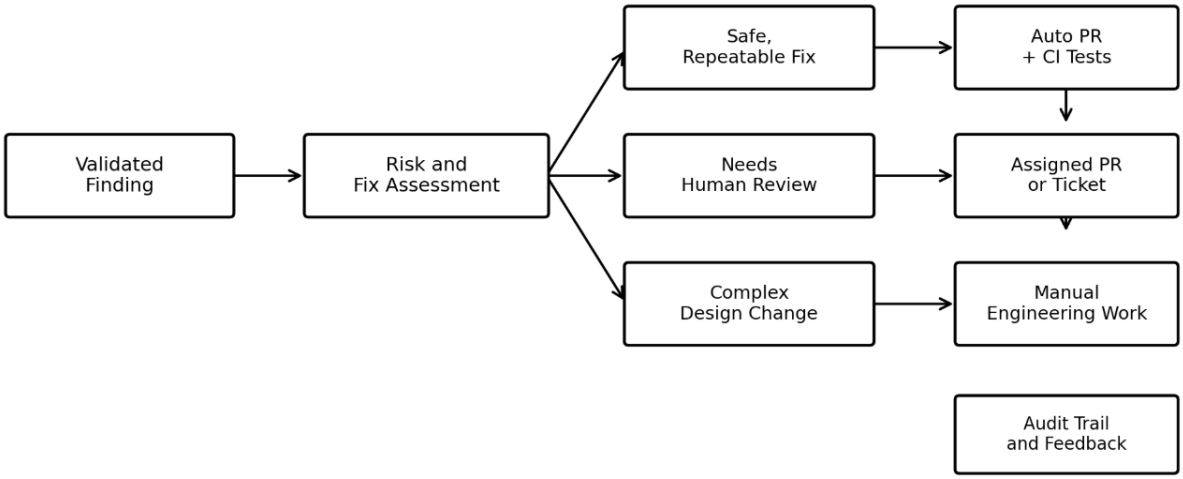

Fig. 3. Controlled remediation workflow showing how validated findings are routed into automated remediation, assisted remediation, or manual engineering work based on risk and fix complexity.

Good candidates for automated remediation include patch-level dependency updates, safe configuration corrections, missing security headers, removal of debug settings, and credential rotation after secret exposure. These changes are usually limited in scope and can be verified through automated checks.

Poor candidates include authentication redesign, authorization model changes, major-version dependency upgrades, cryptographic redesign, and business logic vulnerabilities. These changes may be security-related, but they are also design decisions. They require human review and often require coordination across teams.

Recent research shows progress in automated vulnerability repair, especially using learning-based methods [7], [8]. However, production DevSecOps pipelines should treat these techniques as assistance rather than final authority. A generated patch may compile and pass basic tests while still being incomplete or unsafe for the business context. For this reason, controlled remediation should include review, test evidence, and traceability.

7. SaaS-Specific Considerations

SaaS platforms introduce several practical security concerns that the pipeline must handle.

First, SaaS systems often depend on APIs. DAST profiles designed only for browser-based applications may miss API-specific risks or generate irrelevant noise. API testing should include authentication, authorization, input validation, rate limiting, error handling, and sensitive data exposure.

Second, SaaS platforms often have multi-tenant concerns. A vulnerability in authorization logic or tenant isolation can be more serious than a standard code defect. These findings should be escalated quickly and reviewed carefully.

Third, SaaS environments depend heavily on configuration. Identity permissions, network exposure, object storage settings, Kubernetes policies, CI/CD permissions, and runtime secrets can all create security

risk. Infrastructure-as-code scanning and cloud configuration validation should be part of the same vulnerability-management flow.

Fourth, SaaS delivery requires strong runtime visibility. Security findings should be connected with logs, metrics, traces, deployment history, and ownership metadata. Without runtime context, teams may struggle to decide whether a finding is urgent or theoretical.

For cloud-native SaaS platforms, vulnerability management is strengthened by Zero Trust controls such as workload identity, least-privilege access, network segmentation, secure secrets handling, admission control, and continuous visibility [11]. These controls reduce the chance that one weakness becomes a wider compromise.

8. Governance and Auditability

Automated vulnerability management must be auditable. Every important decision should leave evidence: what was found, which tool reported it, how severity was assigned, who owned the finding, what remediation path was selected, whether a pull request was created, which tests passed, and when the issue was closed.

This matters for compliance, but it also matters for engineering quality. When findings are traceable, teams can learn from them. They can identify repeated insecure patterns, weak ownership areas, services with recurring dependency issues, and pipeline stages that produce too much noise.

Governance should also include exception handling. Some vulnerabilities cannot be fixed immediately. In those cases, the risk should be documented, approved by the right owner, assigned an expiration date, and reviewed again. Permanent exceptions should be avoided unless there is a strong and documented reason.

A good DevSecOps program does not remove human judgment. It makes judgment easier by giving teams timely evidence, clear ownership, and consistent workflows.

9. Discussion

The practical value of automated vulnerability management is not that it finds every issue. No toolchain can promise that. Its value is that it reduces avoidable delay and improves the quality of security decisions. A scanner that creates thousands of findings without context can make security harder. Developers may ignore noisy results, security teams may spend too much time triaging duplicates, and managers may see vulnerability counts without understanding actual risk. A well-designed pipeline should produce fewer, better, and more actionable findings.

This is why aggregation and prioritization are as important as scanning. A SAST finding, a dependency vulnerability, and a DAST result should not remain isolated if they point to the same service risk. Likewise, a critical issue in a production-facing service should not wait behind low-risk findings from non-production systems.

The framework also helps reduce friction between development and security teams. Developers are more likely to act on findings when the feedback appears close to the code, explains the risk clearly, and suggests a realistic fix. Security teams are more effective when they focus on high-risk findings, policy design, and difficult remediation decisions instead of repeatedly triaging the same low-context alerts.

The strongest DevSecOps programs are not the ones with the most tools. They are the ones where tools, workflows, ownership, and remediation are connected.

10. Conclusion

SaaS platforms need vulnerability management that matches the speed and complexity of modern delivery. Periodic scanning and manual ticket queues are no longer enough when application code, dependencies, infrastructure, and configuration change continuously. Security findings must reach developers earlier, with better context and clearer remediation paths.

This article presented a practical framework for automated vulnerability management in DevSecOps pipelines. The framework combines SAST, DAST, dependency scanning, secrets detection, infrastructure-

as-code validation, aggregation, deduplication, risk-based prioritization, and controlled remediation. It treats vulnerability management as part of the delivery system rather than a separate activity after release. The main lesson is that automation should be used carefully. Detection can be widely automated, but remediation should be automated only when the fix is predictable, low-risk, testable, and auditable. Complex vulnerabilities still require human judgment. A mature DevSecOps pipeline does not remove that judgment; it supports it with timely evidence, safer workflows, and continuous feedback.

REFERENCES:

[1] R. N. Rajapakse, M. Zahedi, M. A. Babar, and H. Shen, "Challenges and solutions when adopting DevSecOps: A systematic review," Information and Software Technology, vol. 141, Art. no. 106700, 2022, doi: 10.1016/j.infsof.2021.106700.

[2] M. A. Akbar, K. Smolander, S. Mahmood, and A. Alsanad, "Toward successful DevSecOps in software development organizations: A decision-making framework," Information and Software Technology, vol. 147, Art. no. 106894, 2022, doi: 10.1016/j.infsof.2022.106894.

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

[4] R. Chandramouli, "Implementation of DevSecOps for a Microservices-Based Application with Service Mesh," NIST Special Publication 800-204C, 2022, doi: 10.6028/NIST.SP.800-204C.

[5] I. Pashchenko, H. Plate, S. E. Ponta, A. Sabetta, and F. Massacci, "Vuln4Real: A Methodology for Counting Actually Vulnerable Dependencies," IEEE Transactions on Software Engineering, vol. 48, no. 5, pp. 1592–1609, 2022, doi: 10.1109/TSE.2020.3025443.

[6] V. Casola, A. De Benedictis, C. Mazzocca, and V. Orbinato, "Secure software development and testing: A model-based methodology," Computers & Security, vol. 137, Art. no. 103639, 2024, doi: 10.1016/j.cose.2023.103639.

[7] Z. Chen, S. Kommrusch, and M. Monperrus, "Neural Transfer Learning for Repairing Security Vulnerabilities in C Code," IEEE Transactions on Software Engineering, vol. 49, no. 1, pp. 147–165, 2023, doi: 10.1109/TSE.2022.3147265.

[8] X. Zhou, K. Kim, B. Xu, D. Han, and D. Lo, "Out of Sight, Out of Mind: Better Automatic Vulnerability Repair by Broadening Input Ranges and Sources," in Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, 2024, Art. no. 88, doi: 10.1145/3597503.3639222.

[9] P. C. Jakku, "Resilient Kubernetes Deployment Strategies for Microservices: A Practical Reliability Model," International Journal for Multidisciplinary Research, vol. 3, issue. 6, Nov.–Dec. 2021, doi: https://doi.org/10.36948/ijfmr.2021.v03i06.77664

[10] P. C. Jakku, "Risk-Aware Infrastructure Automation: A Practical Framework for Secure and Maintainable Ansible Playbooks," International Journal of Leading Research Publication, vol. 2, issue. 2, Feb. 2021, doi: https://doi.org/10.70528/IJLRP.v2.i2.2165

[11] P. C. Jakku, "A Zero Trust Reference Architecture for Production-Ready Amazon EKS Environments," International Journal of Innovative Research and Creative Technology, vol. 8, issue. 2, Mar. 2022, doi: https://doi.org/10.62970/IJIRCT.v8.i2.2605004