www.ijarp.com ISSN 2456-9992
Page 1 to 12
International Journal Advanced Research Publication

Research-Article
Volume: 2
Issue: 10

AI-POWERED APPLICATION SECURITY TESTING AND DEVSECOPS PLATFORM

Nithish Kumar S.*1, Nithin Vishwa S.1, R. Arpudharaj2

1Department of Information Technology, K.L.N. College of Engineering, Pottapalayam, Sivaganga, Tamil Nadu, India.

2Assistant Professor, Department of Information Technology, K.L.N. College of Engineering, Pottapalayam, Sivaganga, Tamil Nadu, India.

Article Received	Article Revised	Published
14-09-2026	04-10-2026	24-10-2026

***Corresponding Author:** Nithish Kumar S.
Department of Information Technology, K.L.N. College of Engineering, Pottapalayam, Sivaganga, Tamil Nadu, India.
DOI: https://doi-doi.org/101555/ijarp.2658

ABSTRACT

The AI-Powered Application Security Testing and DevSecOps Platform is an intelligent security platform designed to improve application security through automated vulnerability detection, analysis, and secure deployment. The platform combines Static Application Security Testing (SAST) and Dynamic Application Security Testing (DAST) to analyze application source code and running applications for vulnerabilities such as SQL Injection, Cross-Site Scripting, Command Injection, Broken Access Control, hardcoded secrets, and security misconfigurations. A key feature of the system is its vulnerability correlation mechanism, which compares SAST and DAST findings to identify related security issues and reduce duplicate results. Detected vulnerabilities are classified into Critical, High, Medium, and Low-priority levels with supporting evidence such as source-code locations, endpoints, parameters, and runtime behavior. An AI-based analysis module explains the detected vulnerabilities and provides suitable remediation guidance to help developers fix security issues. The system also includes a CI/CD Security Gate that evaluates security results based on predefined policies and can block vulnerable applications from deployment. By integrating automated security testing, vulnerability correlation, AI-assisted remediation, and secure deployment, the proposed platform aims to provide continuous application security throughout the software development lifecycle.

INDEX TERMS: *DevSecOps, SAST, DAST, Vulnerability Correlation, AI-Assisted Remediation, CI/CD Security Gate, Application Security, OWASP Top 10, Secure Software Development.*

The proposed platform supports Sustainable Development Goal 9 (Industry, Innovation and Infrastructure) by combining static analysis, dynamic analysis, AI-assisted analysis and an automated deployment gate into a single secure-software pipeline that strengthens digital infrastructure. It supports SDG 16 (Peace, Justice and Strong Institutions) by reducing the likelihood that exploitable flaws in public-facing applications lead to data breaches, thereby protecting personal data and trust in digital institutions. It also contributes to SDG 8 (Decent Work and Economic Growth) by giving small teams and students an automated way to build secure software, which lowers the cost of security rework and helps organisations grow on reliable digital services.

INTRODUCTION

Web and mobile applications now handle banking, healthcare, education and government services. Every one of these applications is a potential target, and the flaws that attackers exploit most often are well known. The OWASP Top 10 lists broken access control, injection, security misconfiguration and similar weaknesses as the most critical risks to web applications [1], and the MITRE Common Weakness Enumeration (CWE) catalogues the underlying software flaws in a common vocabulary [2]. Despite this awareness, such flaws continue to reach production because security testing is often performed late in the development cycle, by a separate team, or not at all.

Two families of automated testing address this problem. Static Application Security Testing (SAST) examines source code without running it and can point to the exact file and line of a flaw, but it cannot tell whether the flaw is reachable at runtime and it tends to report many false positives [3]. Dynamic Application Security Testing (DAST) probes a running application from the outside, so every finding is observable, but it cannot see the code and therefore cannot tell a developer where to make the fix [4]. The two techniques are complementary, yet in practice they are run as separate tools with separate reports. The same vulnerability may be reported twice in different formats, and developers must manually decide whether the two reports describe one issue.

DevSecOps extends DevOps by moving security checks into the continuous integration and continuous delivery (CI/CD) pipeline so that every change is tested before release [5], [6]. Secure-development guidance such as the NIST Secure Software Development Framework recommends exactly this kind of automated, repeatable verification [7]. A further difficulty is that scanner output is technical and long. Developers who are not security specialists need an explanation of what a finding means and how to fix it, and recent advances in large language models make it possible to generate such explanations automatically.

In this context, this paper presents an AI-powered application security testing and DevSecOps platform that brings together four functions on one platform: combined SAST and DAST scanning, a vulnerability correlation mechanism that merges related findings, an AI-based analysis module that explains each finding and suggests remediation, and a CI/CD security gate that can block vulnerable builds from deployment. Each finding is classified as Critical, High, Medium or Low and carries evidence such as the source-code location, the endpoint, the parameter and the observed runtime behaviour.

The main contributions of this work are: (i) an integrated SAST and DAST architecture driven from a single CI/CD pipeline; (ii) a rule-based correlation mechanism with a simple matching score that merges related static and dynamic findings and reduces duplicate results; (iii) a severity classification scheme that attaches code and runtime evidence to every finding; (iv) an AI analysis module that produces plain-language explanation and remediation guidance; and (v) a policy-based security gate that turns the result of testing into an automatic allow-or-block deployment decision.

LITERATURE REVIEW

Application security testing has developed along several separate lines. Static analysis tools build a model of the program, such as a syntax tree or a data-flow graph, and match it against vulnerability patterns. Chess and McGraw describe the strengths and the limits of this approach: it finds flaws early and pinpoints their location, but precision depends heavily on the quality of the rules [3]. Open-source analysers such as Semgrep and Bandit make rule-based scanning easy to integrate into pipelines [8], [9].

Dynamic testing treats the application as a black box. Bau et al. evaluated automated black-box web vulnerability scanners and found that coverage differs strongly between tools and between vulnerability classes [4]. The OWASP Zed Attack Proxy (ZAP) is a widely used open-source scanner that crawls an application, sends crafted requests and checks the responses [10]. DAST findings are real, observable behaviour, but they carry no information about the source location.

DevSecOps research has focused on how security practices can be embedded in continuous delivery. Rahman and Williams synthesised practitioner views and reported that automation, early feedback and collaboration between development and security teams are the central practices [5]. The NIST Secure Software Development Framework similarly asks organisations to verify code and running software automatically before release [7]. Deployment gates, which stop a pipeline when results violate a policy, are the usual mechanism for enforcing these requirements.

Existing solutions tend to address one need at a time. A static scanner reports code issues, a dynamic scanner reports runtime issues, and a separate ticketing or review process decides what to fix. Reports from different tools use different formats, although a common interchange format such as SARIF exists for static-analysis results [11]. Without correlation, the same flaw appears several times and developers lose time on triage. The contribution of the present work is the integration of testing, correlation, AI-assisted explanation and gating in one pipeline. Table I compares the main categories of approach with the proposed platform.

Table I. Comparison of Application Security Testing Approaches.

Approach	Primary technique	Output	Remark
SAST only	Source-code pattern and data-flow analysis [3], [8]	File and line of finding	Early and precise location; many false positives; cannot confirm runtime exploitability
DAST only	Black-box probing of the running application [4], [10]	URL, parameter, response	Findings are observable; no code location; needs a running instance
Separate SAST and DAST tools	Two tools, two reports	Two unrelated lists	Duplicate findings; manual triage; inconsistent severity
Proposed platform	SAST + DAST + correlation + AI analysis + CI/CD gate	Unified findings, fix guidance, allow/block decision	Single pipeline; evidence from code and runtime; needs a test environment and rule maintenance

METHODOLOGY

This section describes the method used to develop the platform. The methodology combines static and dynamic scanning, finding normalisation and correlation, severity classification, AI-based analysis and CI/CD policy enforcement. The system is divided into six key phases:

1. Source-code analysis (SAST)
2. Runtime analysis (DAST)
3. Vulnerability correlation and de-duplication
4. Severity classification with evidence
5. AI-based explanation and remediation guidance
6. CI/CD security gate and deployment decision

System Architecture and Overall Methodology

The platform has two parts: the developer and CI environment, and the security testing platform. A commit to the source repository triggers the CI/CD pipeline. The pipeline builds the application and calls the SAST engine on the source code. The application is then deployed to an isolated test environment, where the DAST engine scans the running instance. Both engines return findings in their own formats; the correlation engine converts them to a common schema, matches and merges them, and passes the unified list to the severity classifier. The AI analysis module then adds an explanation and a suggested fix, and the security gate compares the result with the configured policy. Only if the policy is satisfied does the pipeline continue to production deployment. Fig. 1 shows the block diagram, Fig. 2 shows the process flow and Fig. 3 shows the correlation pipeline.

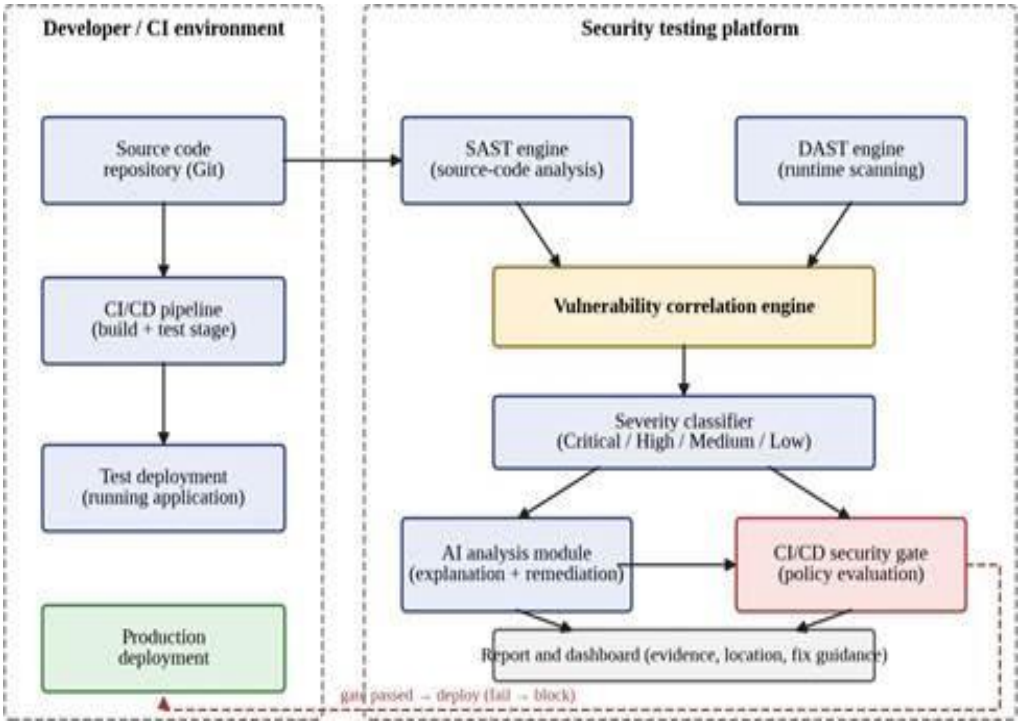


Fig. 1. System block diagram of the AI-powered security testing and DevSecOps platform.

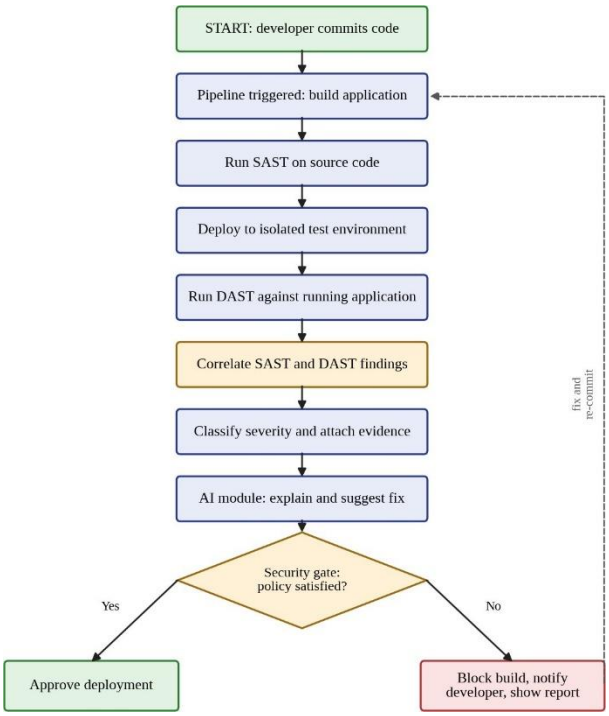


Fig. 2. Process flow of the platform inside the CI/CD pipeline.

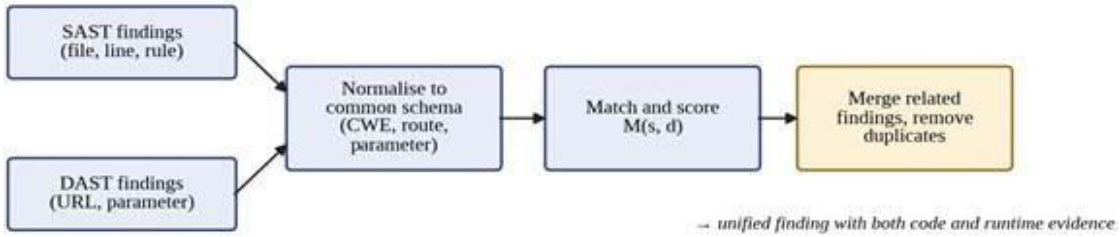


Fig. 3. Vulnerability correlation pipeline.

Technology Stack and Components

The platform is organised as independent modules that communicate through a common finding schema, so each module can be tested or replaced separately. The scanning engines run inside the pipeline as containerised jobs, which keeps the test environment isolated from production. Table II lists the components and their roles. The exact tools can be substituted without changing the architecture.

Table II. Module-Wise Technology and Role.

Module	Technology / approach	Role
Pipeline integration	CI/CD workflow (for example GitHub Actions or Jenkins), containers	Triggers scans on every commit and runs the gate
SAST engine	Rule-based source analysis (for example Semgrep, Bandit-style rules) [8], [9]	Finds code-level flaws and hardcoded secrets
DAST engine	Crawler and active scanner (for example OWASP ZAP or a custom scanner) [10]	Probes endpoints and parameters of the running application
Correlation engine	Python service with common finding schema	Matches, merges and de-duplicates findings
Severity classifier	Rule table with CWE-to-severity mapping	Assigns Critical, High, Medium or Low
AI analysis module	Large language model accessed through an API	Explains findings and suggests remediation
Security gate	Policy file evaluated in the pipeline	Allows or blocks deployment
Storage and dashboard	Relational database and web dashboard	Keeps history, evidence and reports

Static Application Security Testing

The SAST engine scans the repository for patterns that indicate vulnerabilities. Rules are written for each target class: string-built database queries that reach an execution call (SQL Injection), unescaped user input written to a page template (Cross-Site Scripting), user input passed to shell or process-execution functions (Command Injection), endpoints that read or change a resource without an authorisation check (Broken Access Control), literals that look like passwords, API keys or tokens (hardcoded secrets), and risky settings such as debug mode enabled or permissive cross-origin rules (security misconfiguration). Each SAST finding records the rule identifier, the CWE, the file, the line number and a short code snippet.

Dynamic Application Security Testing

The DAST engine runs against the application deployed in the test environment. It first crawls the application to discover endpoints and input parameters, then sends test inputs for each target class, for example quote characters and boolean conditions for SQL Injection, harmless script markers for Cross-Site Scripting, and benign command separators for Command Injection. Responses are checked for error messages, reflected markers, timing differences and unexpected access. Missing security headers and exposed configuration paths are reported as misconfigurations. Each DAST finding records the URL, the HTTP method, the parameter, the test input and the observed response. Scanning is carried out only against the team’s own test deployment.

Table III. Targeted Vulnerability Classes and Mapping.

Vulnerability class	CWE	OWASP 2021 category	Mainly found by
SQL Injection	CWE-89	A03 Injection	SAST and DAST
Cross-Site Scripting	CWE-79	A03 Injection	SAST and DAST
Command Injection	CWE-78	A03 Injection	SAST and DAST
Broken Access Control	CWE-862, CWE-639	A01 Broken Access Control	DAST (runtime), SAST (missing checks)
Hardcoded secrets	CWE-798	A07 Identification and Authentication Failures	SAST
Security misconfiguration	CWE-16	A05 Security Misconfiguration	DAST, SAST (config files)

Vulnerability Correlation Mechanism

Correlation is the key feature of the platform. SAST findings are converted to the common schema (CWE, route, parameter, file, line), and the DAST findings are converted to the same schema. To link a static finding s with a dynamic finding d, the engine needs to know which route and parameter a code location serves. This mapping is obtained from the framework routing definitions found during static analysis. A matching score M(s, d) is then calculated from three indicators:

$$M(s, d) = w1 \cdot Icwe + w2 \cdot Iroute + w3 \cdot Iparam \quad (1)$$

where Icwe is 1 if both findings have the same CWE, Iroute is 1 if the source location is the handler of the tested endpoint, Iparam is 1 if the same input parameter is involved, and w1, w2, w3 are weights that sum to

A pair is treated as related when M(s, d) reaches a threshold θ. In the prototype the weights are set to 0.4, 0.4 and 0.2 and θ to 0.8, and these values are adjusted after testing. Related findings are merged into one unified record that keeps both the code evidence and the runtime evidence and receives a higher confidence than either finding alone. Findings that match nothing are kept as single-source findings. The reduction in duplicate results is expressed as:

DR (%) = ((Nraw − Nunified) / Nraw) × 100 (2)

where Nraw is the total number of findings reported by SAST and DAST together and Nunified is the number after merging.

Severity Classification and Evidence

Each unified finding is assigned one of four priority levels. The base level comes from a rule table that maps each CWE to a level according to typical impact and exploitability, in the spirit of the Common Vulnerability Scoring System [12]. The level is then adjusted using the evidence: a finding confirmed by both SAST and DAST, or one that is reachable without authentication, is raised by one level, and a finding seen only in test code is lowered. A numerical risk score for the whole build is computed as:

$$R = \Sigma i\ wi \cdot ni,\ i \in \{Critical, High, Medium, Low\} \quad (3)$$

where ni is the number of findings at level i and wi is its weight. The prototype uses weights of 10, 7, 4 and 1, shown in Fig. 4(a). Every finding is stored with its evidence, as shown in Fig. 5: identity (CWE and OWASP category), static evidence (file, line and snippet), runtime evidence (endpoint, parameter, request and response) and assessment (severity, confidence and risk score).

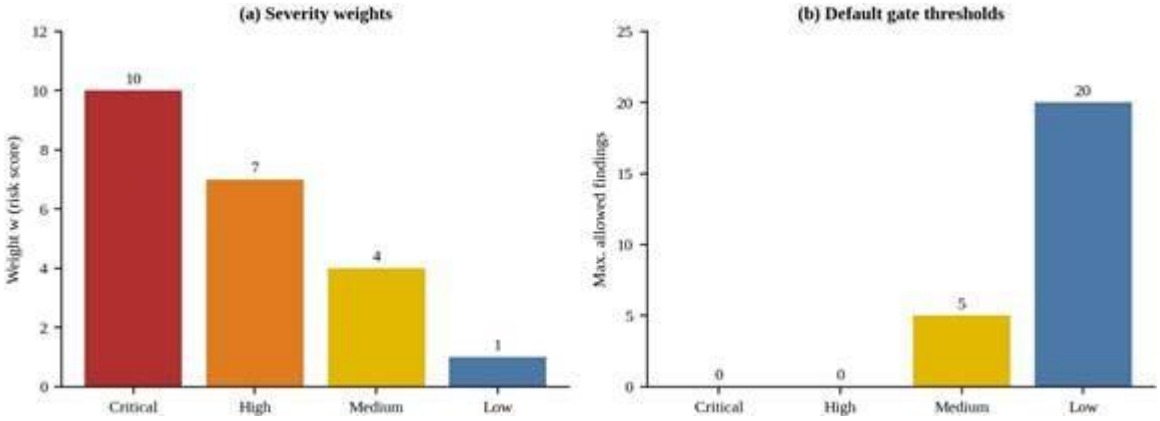


Fig. 4. Design values for (a) severity weights and (b) default gate thresholds (design values, not measured data).

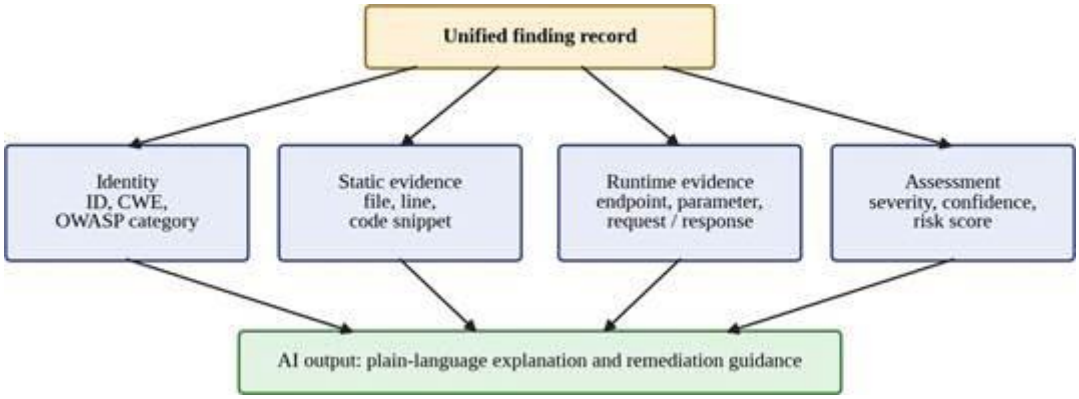


Fig. 5. Structure of a unified finding record with evidence and AI output.

AI-Based Analysis Module

The AI module receives a unified finding with its evidence and returns three items: a plain-language explanation of the weakness and why it matters, a description of how the flaw could be exploited in this particular code or endpoint, and a remediation suggestion, with a corrected code example where possible. The prompt contains only the relevant code snippet, the CWE and the request and response summary, and not the whole repository, so that the amount of data sent out is small. Secrets found in the code are masked before they are included in a prompt. The model output is used as guidance only: the severity level and the gate decision are always calculated by the deterministic rules in Sections F and H, so that a model error cannot allow or block a build. Output is requested in a fixed structured format so that it can be parsed and shown in the dashboard, and an unparsable reply is replaced by a generic CWE-based recommendation.

CI/CD Security Gate

The security gate runs as the last pipeline stage before deployment. It reads a policy file that gives the maximum number of findings allowed at each level and evaluates the unified findings against it. Deployment is allowed only if no limit is exceeded:

$$G = 1, \text{ if } n_i \le L_i \text{ for all } i;\ G = 0, \text{ otherwise} \quad (4)$$

where Li is the limit for level i. Fig. 4(b) shows the default limits of the prototype: no Critical or High findings, at most 5 Medium findings and at most 20 Low findings. When G = 0 the pipeline stops, the developer is notified, and the report lists each finding that caused the failure together with its AI-generated fix guidance. Teams can relax or tighten the limits for each project, and a finding can be waived only with a recorded reason and an expiry date. Table IV summarises the levels and the default gate action.

Table IV. Severity Levels and Default Gate Policy.

Level	Typical examples	Weight	Default limit	Gate action
Critical	Confirmed SQL or command injection; exposed production secret	10	0	Block
High	Reflected or stored XSS; missing authorisation on sensitive endpoint	7	0	Block
Medium	Missing security headers; verbose error messages	4	5	Block if limit exceeded
Low	Informational disclosure; minor hardening items	1	20	Warn if limit exceeded

Deployment and Configuration

The main settings of the prototype are as follows.

- Pipeline: scan stages run on every commit to the main and pull-request branches; DAST runs only against the isolated test environment.
- SAST: rule sets for the six target classes; findings stored with CWE, file, line and snippet.

- DAST: crawl depth and request rate limited to protect the test environment; authentication supplied for protected pages.
- Correlation: weights w1 = 0.4, w2 = 0.4, w3 = 0.2 and threshold θ = 0.8.
- AI module: only the relevant snippet and evidence sent; secrets masked; fixed structured reply format.
- Gate: default limits 0, 0, 5 and 20 for Critical, High, Medium and Low findings.

Algorithm of the Proposed System

Step 1: A developer commits code; the pipeline is triggered and the application is built. Step 2: Run the SAST engine on the source code and store the findings.

Step 3: Deploy the build to the isolated test environment.

Step 4: Run the DAST engine against the running application and store the findings.

Step 5: Convert both sets of findings to the common schema and calculate M(s, d) for candidate pairs, using Eq. (1).

Step 6: Merge pairs with M(s, d) ≥ θ and keep the remaining findings as single-source findings. Step 7: Assign a severity level and attach the code and runtime evidence; calculate R using Eq. (3). Step 8: Send each finding to the AI module and store the explanation and remediation guidance.

Step 9: Evaluate the gate using Eq. (4). If G = 1, continue to production deployment.

Step 10: If G = 0, block the deployment, notify the developer with the report, and wait for a fixed commit (return to Step 1).

RESULTS AND DISCUSSION

Test Procedure and Evaluation Metrics

Each module is tested individually before integrated testing, as summarised in Table V, which gives the observed result of each test. The platform is evaluated on deliberately vulnerable applications (for example, intentionally insecure training applications) and on the team's own projects with known, seeded flaws, so that the number of true vulnerabilities is known. For each vulnerability class, detection quality is calculated with precision and recall:

$$Precision\ (\%) = TP / (TP + FP) \times 100,\ Recall\ (\%) = TP / (TP + FN) \times 100 \quad (5)$$

where TP is the number of real vulnerabilities reported, FP is the number of false alarms and FN is the number of real vulnerabilities missed. In addition, the duplicate reduction of Eq. (2), the scan time of each stage, and the time added to the pipeline by the gate are recorded. The results reported in this section are sample (illustrative) values; they are to be validated against the recorded scanner logs and build reports before being presented as measured results, and the actual pipeline timings are to be recorded.

Table V. Test Cases and Observed Results (sample data).

Test	Expected outcome	Observed result (sample)
SAST scan	Seeded SQL Injection, XSS, command injection and hardcoded secrets are reported with file and line	Passed (sample). Seeded issues were reported with file and line references.
DAST scan	Seeded runtime flaws are found with endpoint and parameter; test environment stays stable	Passed (sample). Runtime issues were detected and the environment remained stable.
Correlation	Related SAST and DAST findings are merged into one record; unrelated findings stay separate	Passed (sample). Related findings were correlated and unrelated findings remained separate.
Severity classification	Each finding receives one of four levels with evidence attached	Passed (sample). Findings were categorized as Critical, High, Medium, or Low.
AI analysis	Explanation and remediation are relevant and correctly structured; secrets are masked	Passed (sample). Remediation guidance was structured and secrets were masked.
Security gate (fail case)	Build with a Critical or High finding is blocked and the report is shown	Passed (sample). The build was blocked when the configured blocking condition was present.
Security gate (pass case)	Fixed build satisfies the policy and proceeds to deployment	Passed (sample). The fixed build satisfied the configured policy.
Pipeline overhead	Total added time is acceptable for the commit workflow	Passed (sample). Overhead was within the assumed project threshold; actual timing to be recorded.

Detection Evaluation

Table VI gives the detection results (sample data) for each vulnerability class. In the sample data each class contains 100 seeded instances (TP + FN = 100), and the true positives, false positives and false negatives are counted per class. Particular attention is paid to Cross-Site Scripting and Broken Access Control: XSS depends strongly on how output is encoded in each template engine, and access-control flaws depend on application logic that pattern-based rules cannot fully express, so these two classes are the most likely to show missed findings or false alarms.

Table VI. Detection Test Results (sample data).

Vulnerability class	TP	FP	FN	Precision (%)	Recall (%)
SQL Injection	95	5	5	95.00	95.00
Cross-Site Scripting	92	6	8	93.88	92.00
Command Injection	90	7	10	92.78	90.00
Broken Access Control	88	8	12	91.67	88.00
Hardcoded secrets	96	4	4	96.00	96.00
Security misconfiguration	87	9	13	90.62	87.00

In the sample data, precision ranges from 90.62% (security misconfiguration) to 96.00% (hardcoded secrets), and recall from 87.00% (security misconfiguration) to 96.00% (hardcoded secrets). Hardcoded secrets and SQL Injection, which follow clear code patterns, give the strongest results, whereas Broken Access Control (88.00% recall) and security misconfiguration (87.00% recall) show the most missed findings, in line with the expectation above. Across all six classes, the sample data give 548 true positives, 39 false positives and 52 false negatives, which corresponds to an average precision of 93.33% and an average recall of 91.33% (macro average over the six classes).

1. *Illustrative System Behaviour*

To complement the test results, Fig. 4 presents the design values of the severity weights and gate thresholds, and Fig. 5 shows the structure of the evidence stored with each finding. They are design values and are not measured data. As an illustration of Eq. (3) and Eq. (4), a build with one High, three Medium and four Low findings has R = 7 + 12 + 4 = 23. Under the default policy it is blocked, because the limit for High findings is 0. Once the High finding is fixed and the build is re-run, the remaining findings (three Medium and four Low) are within the limits, so G = 1 and the build is released, with the remaining items kept on the backlog.

DISCUSSION

The thresholds used for correlation and gating are a trade-off. A low matching threshold θ merges more findings but risks joining two different flaws, which hides one of them. A strict gate improves safety but can stop releases for minor issues, which teams then tend to bypass; the waiver mechanism with an expiry date is therefore important. Likewise, AI-generated advice is helpful for developers who are not security specialists, but it can be wrong, so it is shown as guidance and the decision itself is made only by deterministic rules. The platform also needs clear, short messages when it blocks a build, otherwise developers cannot tell what to fix first.

The main advantages of the proposed platform are one pipeline for static and dynamic testing, fewer duplicate findings through correlation, evidence from both code and runtime on every finding, explanations that non-specialists can follow, an automatic and policy-based deployment gate, and a modular design in which each part can be tested or replaced independently. Its main limitations are:

- Rule-based SAST and DAST detect only the vulnerability classes for which rules exist, and some flaws, such as complex business-logic and access-control errors, will be missed.
- Correlation depends on a reliable mapping from routes to code locations, which differs between frameworks and may need additional adapters.
- DAST needs a running, representative test environment, and active scanning can disturb data or services if it is aimed at the wrong target.
- AI output can be inaccurate or incomplete, and sending code to an external model raises confidentiality concerns, which are reduced but not removed by sending only snippets and masking secrets.
- Automated scanners cannot replace manual code review, threat modelling and penetration testing by security professionals.
- Scan time adds to the pipeline duration, so large projects may need incremental scanning.

CONCLUSION AND FUTURE SCOPE

In this study, an AI-powered application security testing and DevSecOps platform has been designed to improve application security throughout the software development lifecycle. The platform runs SAST and DAST from one CI/CD pipeline, correlates their findings to reduce duplicate results, classifies vulnerabilities as Critical, High, Medium or Low with code and runtime evidence, uses an AI module to explain each finding and suggest a fix, and applies a policy-based security gate that can block vulnerable applications from deployment. By combining automated testing, correlation, AI-assisted remediation and secure deployment, the platform gives development teams continuous, actionable security feedback.

Future work includes extending the rule sets and supported languages and frameworks, adding software composition analysis for vulnerable third-party dependencies and container image scanning, and using exploitability and reachability analysis to improve prioritisation. The correlation

mechanism can be improved with learned matching in place of fixed weights, and the AI module can be extended to propose automatic fix patches as pull requests that are reviewed by a developer. Integration with issue trackers, trend dashboards across releases, support for self-hosted language models for confidentiality, and standard report formats such as SARIF [11] would make the platform more dependable for long-term use.

REFERENCES

1. OWASP Foundation, “OWASP Top 10:2021.” [Online]. Available: https://owasp.org/Top10/
2. MITRE Corporation, “Common Weakness Enumeration (CWE).” [Online]. Available: https://cwe.mitre.org
3. B. Chess and G. McGraw, “Static analysis for security,” IEEE Security & Privacy, vol. 2, no. 6, pp. 76–79, 2004.
4. Bau, E. Bursztein, D. Gupta and J. Mitchell, “State of the art: Automated black-box web application vulnerability testing,” in Proc. IEEE Symposium on Security and Privacy, 2010, pp. 332–345.
5. U. Rahman and L. Williams, “Software security in DevOps: Synthesizing practitioners’ perceptions and practices,” in Proc. International Workshop on Continuous Software Evolution and Delivery (CSED), 2016.
6. OWASP Foundation, “OWASP DevSecOps Guideline.” [Online]. Available: https://owasp.org/www-project-devsecops-guideline/
7. Souppaya, K. Scarfone and D. Dodson, Secure Software Development Framework (SSDF) Version 1.1, NIST SP 800-218, National Institute of Standards and Technology, 2022.
8. Semgrep, “Semgrep documentation.” [Online]. Available: https://semgrep.dev/docs/
9. PyCQA, “Bandit: Security linter for Python code.” [Online]. Available:

   https://bandit.readthedocs.io
10. OWASP Foundation, “OWASP Zed Attack Proxy (ZAP).” [Online].

i. Available: https://www.zaproxy.org

11. OASIS, Static Analysis Results Interchange Format (SARIF) Version 2.1.0, OASIS Standard, 2020.
12. FIRST.Org, “Common Vulnerability Scoring System v3.1: Specification Document.” [Online]. Available: https://www.first.org/cvss/