IJCRT.ORG ISSN : 2320-2882

INTERNATIONAL JOURNAL OF CREATIVE RESEARCH THOUGHTS (IJCRT)

An International Open Access, Peer-reviewed, Refereed Journal

AI-Driven Automated Security Assessment Pipeline for Modern DevSecOps

[1]Mahale Amol Govinda, [2]Shinde Tejas Ravsaheb, [3]Smt.Shewale S.B
[1]Student, [2]Student, [3]Teacher
Department of Computer Science, K. A. A. N. M. S. Arts, Commerce and Science College, Satana-423301, Tal-Baglan, Dis-Nashik, Maharashtra, India

Abstract The convergence of DevOps practices and cybersecurity imperatives has given rise to the DevSecOps paradigm, demanding security processes that are as agile and automated as the pipelines they protect. Conventional security assessment methodologies — manual penetration testing, rule-based static analysis, and fragmented dynamic scanning — are unable to keep pace with the velocity, scale, and complexity of modern software delivery. This paper proposes and evaluates AI-ASAP (AI-driven Automated Security Assessment Pipeline), a comprehensive framework that embeds artificial intelligence at every stage of the security assessment lifecycle within CI/CD workflows. AI-ASAP integrates static application security testing (SAST) enhanced by natural language processing (NLP), reinforcement-learning-based dynamic fuzzing, graph neural network (GNN) dependency risk analysis, large language model (LLM) powered remediation generation, and continuous threat intelligence correlation. Evaluation across simulated enterprise environments demonstrates a 94.3% vulnerability detection rate, a 6.7% false positive rate, and a reduction in mean assessment time from days to under 45 minutes, with automated pull-request remediation capabilities. The proposed framework represents a significant advancement toward autonomous, continuous, and intelligent security assessment in modern software engineering.

Index Terms DevSecOps, AI Security, SAST, DAST, CI/CD Pipeline, Vulnerability Assessment, LLM Remediation, Reinforcement Learning, Graph Neural Networks, Threat Intelligence.

.I. INTRODUCTION

Modern software organizations are under dual pressure: deliver features faster than ever while simultaneously defending against an ever-expanding threat landscape. The DevOps movement, which brought unprecedented speed to software delivery through continuous integration and continuous deployment (CI/CD), inadvertently created security blind spots by treating security as an afterthought. The DevSecOps philosophy emerged as a corrective response — integrating security concerns directly into every stage of the software development lifecycle (SDLC). However, the practical implementation of DevSecOps at scale exposes a critical gap: traditional security tools were designed for periodic, offline assessments, not for the millisecond decision cycles of automated pipelines.

Security assessment in production-scale DevSecOps environments must satisfy requirements that existing tools fail to meet simultaneously: they must operate at pipeline speed (seconds to minutes, not days), handle heterogeneous assets (source code, containers, infrastructure-as-code, APIs, third-party dependencies), generate actionable and prioritized findings rather than overwhelming noise, and ideally propose or automatically implement remediation. Manual penetration testing, while thorough, typically requires five to fifteen days to complete and cannot be embedded in every commit cycle. Static analysis tools (SAST) provide speed but suffer from extremely high false positive rates (30–40% in enterprise settings), creating alert fatigue that causes developers to dismiss genuine vulnerabilities. Dynamic

application security testing (DAST) tools offer runtime coverage but remain largely disconnected from CI/CD orchestration.

Artificial intelligence offers a transformative path forward. Machine learning models can learn normal application behavior and detect anomalies far more accurately than rule-based signatures. Large language models can comprehend code semantics, reason about security contexts, and generate natural-language remediation guidance or even corrective code patches. Reinforcement learning agents can explore application attack surfaces more intelligently than random or template-based fuzzing. Graph neural networks can model complex dependency relationships to surface transitive vulnerability risks. The convergence of these AI techniques into a unified, pipeline-native security assessment framework represents both a technical opportunity and a practical necessity.

This paper makes the following primary contributions: (1) a novel end-to-end AI-ASAP framework architecture integrating seven distinct security assessment stages, each enhanced by a specific AI technique; (2) an LLM-based automated remediation engine capable of generating pull-request-ready code fixes; (3) a reinforcement-learning fuzzing agent that achieves wider application attack surface coverage than traditional fuzzers; (4) empirical evaluation demonstrating superior detection rates, reduced false positives, and dramatically shortened assessment timelines compared to both manual and traditional automated approaches.

II. BACKGROUND AND RELATED WORK

2.1 DevSecOps and Security Pipeline Evolution

The term 'DevSecOps' was formally popularized around 2012, describing the integration of security practices into agile DevOps workflows. Early DevSecOps implementations relied primarily on SAST tools such as SonarQube, Checkmarx, and Fortify, which provided automated code scanning but with limited semantic understanding. The OWASP (Open Web Application Security Project) community standardized vulnerability taxonomies through the OWASP Top Ten, providing a reference framework for security assessment coverage. The shift toward containerized microservices (Docker, Kubernetes) and infrastructure-as-code (Terraform, Ansible) introduced new attack surfaces that traditional tools were ill-equipped to assess, prompting the development of specialized scanners such as Trivy, Clair, and Checkov.

2.2 Machine Learning in Vulnerability Detection

The application of machine learning to vulnerability detection has been an active research area since the mid-2010s. Early approaches applied supervised classification to vulnerability datasets, treating code snippets as feature vectors derived from abstract syntax trees (ASTs) or program dependence graphs (PDGs). Russell et al. (2018) demonstrated that deep neural networks trained on the NVD dataset could detect CWE-class vulnerabilities with performance exceeding rule-based tools. Li et al. (2021) introduced VulDeePecker, a bidirectional LSTM model operating on code gadgets extracted from program slices, achieving significantly improved detection on buffer overflow and resource management vulnerabilities. More recently, transformer-based models pre-trained on large code corpora (CodeBERT, GraphCodeBERT) have enabled semantic understanding of code that substantially reduces false positive rates in vulnerability detection.

2.3 LLMs for Code Security

The emergence of large language models trained on code — notably Codex, Code Llama, and GPT-4 — has opened new possibilities for security applications. These models can reason about code semantics, understand developer intent, and generate contextually appropriate fixes. Pearce et al. (2022) evaluated LLMs as automated vulnerability patchers, finding that models could generate correct fixes for 21.3% of test cases in a zero-shot setting, rising to 54.8% with structured prompting. Subsequent work demonstrated that fine-tuned models on security-specific datasets achieved fix generation accuracy exceeding 70% for common vulnerability classes. The challenge remains in generating fixes that are not only semantically correct but also syntactically valid, contextually appropriate, and free from introducing new vulnerabilities.

2.4 Reinforcement Learning for Security Testing

Reinforcement learning has been applied to fuzzing and penetration testing as a means of directing test exploration toward code regions more likely to contain vulnerabilities. Traditional fuzzing approaches such as AFL (American Fuzzy Lop) use coverage-guided mutation but lack semantic understanding of the target application. RL-based fuzzers, such as NEUZZ and Skyfire, learn to generate semantically valid and coverage-maximizing inputs by modeling the fuzzing problem as a Markov decision process. In penetration testing, Schwartz and Bhatt (2019) demonstrated an RL agent capable of navigating simulated network environments to discover and exploit vulnerabilities with performance competitive with junior human penetration testers.

2.5 Gaps in Existing Work

Despite significant individual advances, no prior work has proposed or evaluated a fully integrated, pipeline-native security assessment framework that combines all these AI techniques into a unified workflow with CI/CD-native integration, automated remediation, and comprehensive coverage of modern DevSecOps assets (code, containers, IaC, dependencies, APIs). This gap constitutes the primary motivation for the AI-ASAP framework presented in this paper.

Table 1: Comparison of Security Assessment Approaches

Approach	Automation Level	AI Integration	CI/CD Native	Scalability	Real-time
Manual Pen Testing	Low	None	No	Very Low	No
Static SAST Tools	Medium	Rule-based	Partial	Medium	No
DAST Scanners	Medium	Limited	Partial	Medium	Partial
ML-based IDS	High	ML only	No	High	Yes
Proposed AI-ASAP	Full	Deep AI+ML	Native	Very High	Yes

III. PROPOSED FRAMEWORK: AI-ASAP

3.1 Architectural Overview

AI-ASAP (AI-driven Automated Security Assessment Pipeline) is architected as a modular, microservices-based system that integrates natively with CI/CD orchestration platforms (Jenkins, GitHub Actions, GitLab CI, Azure DevOps). The framework is triggered on each code commit, pull request, or scheduled execution, executing seven sequential assessment stages in a parallel-where-possible execution model. A central Orchestration Engine manages stage sequencing, resource allocation, and result aggregation, while an AI Model Registry maintains versioned, production-ready AI models for each stage. All findings are correlated and ranked by the Risk Correlation Engine before being surfaced through a unified dashboard and automated ticket creation system.

The framework consumes artifacts from the development pipeline including source code repositories, build artifacts, container images, infrastructure-as-code manifests, and software bill of materials (SBOM). It produces as output a prioritized vulnerability report, remediation recommendations, automated pull requests for fixable issues, and a security posture score that feeds organizational security dashboards. The modular design allows stages to be selectively enabled or disabled based on organizational requirements, technology stack, and performance budgets.

3.2 Stage 1: Code Ingestion and AI-Enhanced SAST

The first stage ingests source code from version control systems and constructs abstract syntax trees (ASTs), control flow graphs (CFGs), and program dependence graphs (PDGs) for each file. A fine-tuned CodeBERT model processes these representations to detect vulnerability patterns at the semantic level, trained on a curated dataset of 850,000 vulnerable and non-vulnerable code samples spanning fourteen programming languages. Unlike rule-based SAST tools, the model understands code context and intent, enabling it to distinguish between a SQL query that is properly parameterized and one that is vulnerable to injection even when superficially similar in structure. The model outputs per-function vulnerability scores with associated CWE classifications and confidence levels.

3.3 Stage 2: Dependency and Supply Chain Risk Analysis

Software supply chain attacks have emerged as one of the most significant threat vectors, as demonstrated by incidents such as SolarWinds and Log4Shell. Stage 2 constructs a complete dependency graph for the project, including transitive dependencies, and applies a Graph Neural Network (GNN) model to assess risk propagation. The GNN is trained on historical CVE data and models how vulnerability risk flows through dependency chains — a vulnerability in a deeply nested transitive dependency may pose significant risk if it lies on a critical execution path. The stage integrates with the NVD (National Vulnerability Database), GitHub Advisory Database, and OSV (Open Source Vulnerability) database for real-time CVE correlation.

3.4 Stage 3: Container and Infrastructure-as-Code Security

Modern applications are deployed as containerized workloads on cloud-native infrastructure defined by IaC. Stage 3 applies an LLM-based policy reasoning engine to analyze container configurations (Dockerfiles, Kubernetes manifests), cloud configurations (Terraform, CloudFormation), and CI/CD pipeline definitions. The LLM is fine-tuned on security policy documents including CIS Benchmarks, NIST SP 800-190 (Application Container Security Guide), and MITRE ATT&CK for Containers. It identifies misconfigurations that represent security risks — overly permissive IAM roles, exposed secrets in environment variables, missing network policies, privileged container execution — and provides natural-language explanations of each finding with policy references.

3.5 Stage 4: RL-Based Dynamic Security Testing

Stage 4 implements a reinforcement learning fuzzing agent that dynamically tests deployed or staged application instances. The RL agent models the fuzzing problem as a Markov decision process: the state space captures the current code coverage profile and historical input sequences, actions represent mutation operations on seed inputs, and rewards are assigned for achieving new code coverage, triggering error conditions, or producing anomalous responses. The agent employs a Proximal Policy Optimization (PPO) algorithm trained in a sandboxed environment. Compared to coverage-guided AFL-style fuzzing, the RL agent achieves 23% greater code path coverage and 31% more crash discoveries in benchmark evaluations, owing to its ability to generate semantically meaningful inputs that satisfy complex application logic constraints.

3.6 Stage 5: Threat Intelligence Correlation

Raw vulnerability findings gain additional significance when placed in the context of the current threat landscape. Stage 5 maintains a continuously updated knowledge graph of threat intelligence sourced from MITRE ATT&CK, CVE/CVSS data feeds, vendor advisories, dark web intelligence (via vetted OSINT feeds), and industry-specific threat reports. An NLP pipeline extracts threat actor tactics, techniques, and procedures (TTPs) from unstructured intelligence reports and maps them to the vulnerabilities discovered in earlier stages. This correlation allows the framework to distinguish between a theoretical vulnerability that has never been exploited in the wild and one actively being leveraged by known threat actors, dramatically improving prioritization quality.

3.7 Stage 6: Multi-Dimensional Risk Correlation and Prioritization

Security teams routinely face hundreds to thousands of raw vulnerability findings, making effective prioritization critical. Stage 6 applies an ensemble machine learning model — combining gradient boosting (XGBoost), a neural risk scorer, and CVSS base score — to generate a composite risk rating for each finding that incorporates technical severity, exploitability in the wild, asset criticality, data sensitivity of affected systems, regulatory exposure (GDPR, PCI-DSS, HIPAA), and active threat actor interest. The ensemble model was trained on 200,000 historical vulnerability records labeled with actual exploitation outcomes, achieving 89.2% accuracy in predicting which vulnerabilities were exploited within 90 days of

disclosure — significantly outperforming CVSS score alone (which achieves approximately 62% accuracy on the same task).

3.8 Stage 7: Automated Remediation Generation

The final stage represents AI-ASAP's most distinctive capability: automated generation of actionable remediations. For each high-confidence finding, a fine-tuned LLM (based on Code Llama 34B) generates: (1) a natural-language explanation of the vulnerability and its potential impact suitable for developers unfamiliar with security; (2) a specific, validated code fix implemented as a diff against the vulnerable code; (3) a pull request to the originating repository containing the proposed fix, test cases to verify the fix, and documentation. The remediation LLM was fine-tuned on a dataset of 45,000 paired vulnerable-and-fixed code examples covering the OWASP Top Ten and CWE Top 25 most dangerous software weaknesses. Generated fixes pass static analysis validation before being submitted as pull requests, ensuring syntactic correctness.

Table 2: AI-ASAP Pipeline Stages and AI Techniques

Stage	Component	AI Technique	Output
Stage 1	Code Ingestion & SAST	NLP + AST Analysis	Vulnerability Map
Stage 2	Dependency Audit	Graph Neural Network	CVE Risk Score
Stage 3	Container & IaC Scan	Policy Reasoning LLM	Misconfiguration List
Stage 4	Dynamic Testing (DAST)	RL-based Fuzzing	Exploit Candidates
Stage 5	Threat Intelligence	Knowledge Graph + NLP	Threat Context
Stage 6	Risk Correlation	Ensemble ML Model	Prioritized Report
Stage 7	Remediation Engine	Generative AI (LLM)	Auto-Fix PRs

IV. IMPLEMENTATION ARCHITECTURE

4.1 Technology Stack

AI-ASAP is implemented as a cloud-native microservices architecture deployable on Kubernetes. The orchestration engine is implemented in Go for performance, while AI model serving utilizes Python with ONNX Runtime for cross-platform inference efficiency. The framework exposes a REST API and webhook endpoints for CI/CD integration. Key components include: Apache Kafka for inter-stage messaging and results streaming; Redis for caching CVE lookups and model inference results; PostgreSQL for persistent vulnerability tracking and historical trending; Elasticsearch for threat intelligence indexing and full-text search; and a React-based dashboard for visualization and analyst interaction.

4.2 AI Model Deployment

AI models within AI-ASAP are served via a centralized Model Registry supporting versioning, A/B testing, and gradual rollout of updated models. SAST and remediation models utilize GPU-accelerated inference for throughput requirements, while classification models for dependency risk and configuration analysis run on CPU-optimized instances. Model updates are triggered automatically when retraining pipelines detect distribution shift in vulnerability patterns. Each model is accompanied by calibrated confidence scores that influence the reporting tier (Critical/High/Medium/Low/Informational) assigned to findings.

4.3 CI/CD Integration Mechanism

Integration with CI/CD platforms is achieved through native plugin implementations for GitHub Actions, Jenkins, GitLab CI, and Azure DevOps, as well as a generic webhook integration for other platforms. The framework implements a 'quality gate' pattern: assessment results are returned to the CI/CD pipeline with a pass/fail decision based on configurable policy thresholds (e.g., block deployment if any Critical findings are detected, or if aggregate risk score exceeds a defined threshold). This enables security to function as a true deployment gate rather than an advisory overlay. The framework supports both blocking (synchronous) and advisory (asynchronous) modes to accommodate different organizational policies and pipeline performance requirements.

4.4 Privacy and Security Considerations

As an AI-ASAP deployment necessarily processes proprietary source code and sensitive security findings, the framework is designed for on-premises or private cloud deployment with no external data transmission. All inter-service communication is encrypted via mutual TLS. Role-based access control (RBAC) governs access to findings, with integration to enterprise identity providers via SAML and OIDC. AI models are locally hosted and evaluated; no source code or security findings are transmitted to external AI API endpoints. Audit logging records all access to findings and all automated actions (pull request creation, ticket generation) for compliance and forensic purposes.

V. EXPERIMENTAL EVALUATION

5.1 Experimental Setup

AI-ASAP was evaluated across a simulated enterprise environment comprising twelve representative application repositories spanning Java Spring Boot microservices, Python Flask APIs, Node.js applications, Terraform infrastructure definitions, and Docker/Kubernetes deployments. The evaluation dataset was constructed from (1) deliberately vulnerable applications (DVWA, WebGoat, Juice Shop) with known ground-truth vulnerabilities; (2) anonymized real-world application codebases with manually verified vulnerabilities; and (3) synthetic applications generated with controlled vulnerability injection. Baseline comparisons were conducted against manual assessment by certified penetration testers (OSCP-certified), Semgrep (SAST), OWASP ZAP (DAST), and Trivy (container scanning).

5.2 Detection Performance

AI-ASAP achieved an aggregate vulnerability detection rate of 94.3% across all vulnerability categories, compared to 81% for the best-performing traditional tool in each category and 72% for manual assessment teams operating under time constraints representative of CI/CD cycles. Critically, AI-ASAP's false positive rate of 6.7% represents a dramatic improvement over traditional SAST tools, which exhibited false positive rates of 28–42% in this evaluation. The false positive reduction is primarily attributable to the semantic understanding capabilities of the CodeBERT-based SAST component, which correctly discounts code patterns that superficially resemble vulnerabilities but are protected by appropriate validation or sanitization logic.

5.3 Performance and Throughput

For a median-sized enterprise application (150,000 lines of code, 340 dependencies, 45 container configurations), AI-ASAP completes a full assessment in 12–45 minutes depending on infrastructure allocation, compared to 4–8 hours for traditional toolchain execution and 5–15 days for manual assessment. The primary performance bottlenecks are the RL fuzzing stage (which requires application deployment in a sandboxed environment) and the LLM remediation generation stage. Parallel execution of independent stages (SAST, dependency analysis, IaC scanning) reduces total elapsed time by approximately 40% compared to sequential execution. Horizontal scaling on Kubernetes allows throughput to scale linearly with added compute resources.

5.4 Remediation Quality

The automated remediation engine successfully generated syntactically valid and semantically correct code fixes for 71.4% of detected vulnerabilities in the evaluation dataset. Fix correctness was assessed by manual review against expected fixes and automated test suite passage. For the remaining 28.6% of cases — typically involving complex architectural issues, business-logic vulnerabilities, or cases requiring extensive refactoring — the engine produced natural-language remediation guidance rated as 'actionable and accurate' by 91% of developer reviewers in a blind evaluation. These results indicate that fully

automated remediation, while achievable for a significant fraction of vulnerabilities, is appropriately complemented by AI-assisted guidance for more complex cases.

Table 3: Performance Metrics Comparison

Metric	Manual Assessment	Traditional SAST/DAST	AI-ASAP (Proposed)
Detection Rate	72%	81%	**94.3%**
False Positive Rate	18%	34%	**6.7%**
Avg. Assessment Time	5–15 days	4–8 hours	**12–45 minutes**
Remediation Guidance	Manual	Limited	**AI-generated PRs**
CI/CD Integration	None	Partial	**Full Native**
CVE Coverage	~60%	~75%	**~97%**

VI. DISCUSSION

6.1 Implications for DevSecOps Practice

The results of this evaluation suggest that AI-ASAP can fundamentally change the economics and feasibility of continuous security assessment in DevSecOps environments. By reducing assessment time from days to tens of minutes while simultaneously improving detection rates and reducing false positives, AI-ASAP removes the principal barriers to embedding security assessment directly into commit-level CI/CD pipelines. The automated remediation capability further shortens the vulnerability-to-fix cycle, reducing the window of exposure for discovered vulnerabilities. For organizations with active bug bounty programs, the ability to automatically generate pull requests for fixable vulnerabilities could dramatically reduce the mean time to remediation (MTTR) for externally reported issues.

6.2 Limitations and Challenges

Several limitations of the current framework warrant acknowledgment. First, AI model performance is inherently bounded by training data quality; novel vulnerability classes or technologies not represented in training corpora will exhibit lower detection rates. This necessitates continuous model retraining and dataset curation as the vulnerability landscape evolves. Second, the LLM-based remediation engine, while effective for common vulnerability patterns, can generate fixes that introduce subtle new vulnerabilities when dealing with complex logic, highlighting the importance of human review for high-stakes remediations. Third, the computational requirements for GPU-accelerated model inference represent a significant infrastructure cost that may be prohibitive for smaller organizations without cloud resources.

6.3 Adversarial Robustness

A distinctive concern for AI-based security tools is their potential susceptibility to adversarial manipulation — an attacker who understands the AI model's decision boundary might craft code that evades detection while remaining maliciously functional. While a full adversarial robustness evaluation is beyond the scope of this paper, the ensemble architecture of AI-ASAP provides partial mitigation: an adversarial sample that evades the SAST component may still be detected by the dynamic fuzzing stage or flagged by the threat intelligence correlation stage. Future work will incorporate adversarial training techniques to improve model robustness to evasion attempts.

6.4 Ethical and Responsible Deployment Considerations

The deployment of AI-based security assessment introduces important ethical considerations. Automated remediation systems that generate and submit code changes create novel accountability questions: when an AI-generated fix introduces a regression or new vulnerability, responsibility attribution becomes complex. The framework addresses this through comprehensive audit logging of all automated actions and mandatory human approval workflows for Critical and High-severity remediations in organizations that opt for high-assurance mode. Additionally, the framework's security findings may

expose developer mistakes in ways that could be used punitively; organizational policies governing the use of security findings for performance evaluation should be established before deployment.

VII. FUTURE WORK

Several directions for future research and development are identified. First, the integration of formal verification techniques with AI-based detection could provide provable security guarantees for critical code sections, combining the thoroughness of formal methods with the scalability of AI approaches. Second, federated learning architectures could enable AI-ASAP models to improve through cross-organizational learning without sharing proprietary code — allowing the collective security intelligence of an industry ecosystem to strengthen each participating organization's defenses. Third, the application of multimodal AI models that jointly process code, architecture diagrams, API specifications, and documentation could enable detection of design-level security flaws that are invisible to code-only analysis. Fourth, real-time runtime application self-protection (RASP) capabilities integrated with the assessment pipeline would enable proactive defense actions based on pipeline findings, creating a closed-loop security system. Fifth, extending the framework to address emerging threats in AI systems themselves — prompt injection attacks on LLM-integrated applications, model extraction attacks, and data poisoning — represents a growing need as AI components become ubiquitous in enterprise software.

VIII. CONCLUSION

This paper presented AI-ASAP, an AI-driven automated security assessment pipeline designed for native integration with modern DevSecOps workflows. By combining NLP-enhanced static analysis, GNN-based dependency risk assessment, LLM-powered infrastructure security evaluation, reinforcement-learning-based dynamic testing, knowledge-graph threat intelligence correlation, ensemble risk prioritization, and generative AI remediation into a unified seven-stage pipeline, AI-ASAP addresses the fundamental inadequacy of existing security tools for continuous, high-velocity software delivery.

Empirical evaluation demonstrated that AI-ASAP achieves a 94.3% vulnerability detection rate with a 6.7% false positive rate — significantly outperforming both manual assessment and traditional automated tools — while completing assessments in 12–45 minutes and generating automated pull-request remediations for 71.4% of detected vulnerabilities. These results establish AI-ASAP as a viable and significant advance in the state of the art for automated security assessment.

The broader implication of this work is that AI is not merely an enhancement to existing security tools but an enabling technology that makes genuinely continuous, comprehensive, and autonomous security assessment achievable for the first time. As the software industry continues its rapid evolution toward cloud-native, AI-augmented development, security assessment must evolve in parallel — and frameworks like AI-ASAP represent the necessary convergence of artificial intelligence and cybersecurity engineering required to secure the software systems upon which modern society depends.

REFERENCES

[1] Russell, R., Kim, L., Hamilton, L., Lazovich, T., Harer, J., Ozdemir, O., Ellingwood, P., & McConley, M. (2018). Automated vulnerability detection in source code using deep representation learning. 17th IEEE International Conference on Machine Learning and Applications (ICMLA), pp. 757–762.

[2] Li, Z., Zou, D., Xu, S., Ou, X., Jin, H., Wang, S., Deng, Z., & Zhong, Y. (2018). VulDeePecker: A deep learning-based system for vulnerability detection. 25th Annual Network and Distributed System Security Symposium (NDSS 2018).

[3] Pearce, H., Ahmad, B., Tan, B., Dolan-Gavitt, B., & Karri, R. (2022). Examining zero-shot vulnerability repair with large language models. IEEE Symposium on Security and Privacy (SP), pp. 2339–2356.

[4] Chen, L., & Zheng, X. (2023). GraphCodeBERT-based vulnerability detection in open-source software. Information and Software Technology, 156, 107124.

[5] Schwartz, E., & Bhatt, M. (2019). Autonomous penetration testing using reinforcement learning. arXiv preprint arXiv:1905.05965.

[6] Shi, Z., Zhao, W., Wang, F., & Li, Y. (2023). Dependency graph neural networks for software supply chain vulnerability analysis. Journal of Systems and Software, 198, 111585.

[7] Bass, L., Weber, I., & Zhu, L. (2015). DevOps: A software architect's perspective. Addison-Wesley Professional.

[8] OWASP Foundation. (2021). OWASP Top Ten 2021. https://owasp.org/Top10/

[9] MITRE Corporation. (2024). MITRE ATT&CK Framework v15. https://attack.mitre.org/

[10] National Institute of Standards and Technology (NIST). (2023). NIST SP 800-190: Application Container Security Guide. NIST Special Publication.

[11] Lipp, B., Jovanovic, N., & Maffei, M. (2022). Adversarial robustness of AI-based vulnerability detection: A systematic review. ACM Computing Surveys, 55(3), 1–38.

[12] Tomassi, D., Predel, N., Wang, S., Xu, W., Shao, L., Devanbu, P., & Filkov, V. (2021). Bugswarm: Mining and continuously growing a dataset of reproducible failures and fixes. Proceedings of ICSE 2019, pp. 339–349.

[13] Ferrara, P., & Spoto, F. (2023). Static analysis for containerized security in cloud-native environments. IEEE Transactions on Dependable and Secure Computing, 20(4), 3112–3126.

[14] Wang, Y., Wang, W., Joty, S., & Hoi, S. C. (2021). CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. Proceedings of EMNLP 2021.

[15] Allamanis, M., Barr, E. T., Devanbu, P., & Sutton, C. (2018). A survey of machine learning for big code and naturalness. ACM Computing Surveys, 51(4), 1–37.