AI-Driven Security Testing in DevSecOps: Integrating Cross-Project Vulnerability Discovery with Real-Time Network Intrusion Detection

Lawal G. Anand
Independent Researcher

Date: 17-10-2026

***Abstract*—**DevSecOps seeks to embed security into every stage of software delivery, yet its security testing remains divided between build-time tools that search code for weaknesses and run-time tools that watch networks for attacks. This review examines how artificial intelligence can connect these two ends of the pipeline. It analyzes what each pipeline stage requires of an AI-based tester, surveys cross-project vulnerability discovery and learning-based real-time network intrusion detection, and critically assesses the distance between benchmark results and operational performance. It then proposes a reference architecture in which both detectors share a risk and evidence store, common identifiers, and a feedback loop, and it describes integration patterns, risk-scoring and gate policies, and use cases such as pre-merge review with exposure context, attack replay in staging, and incident-to-regression conversion. Pipeline-level evaluation measures are discussed alongside model-level practice. Open challenges include false-positive burden, model drift and maintenance, adversarial and supply chain risks, privacy, and latency. The review concludes that integrated, evidence-driven security testing is feasible but depends on linked benchmarks, realistic evaluation, and attention to developer trust.

***Keywords*—**DevSecOps, security testing, vulnerability discovery, cross-project learning, intrusion detection, network traffic analysis, contrastive learning, continuous delivery, machine learning.

I. INTRODUCTION

Software is now released at a pace that would have seemed implausible two decades ago. Teams practicing continuous integration and delivery routinely ship changes many times per day, and the engineering culture that supports this pace has been associated with stronger organizational performance [1]. Security, however, has historically been a gate at the end of the process rather than a property of the process itself. DevSecOps is the movement to correct this by embedding security activities, responsibilities, and automation throughout the delivery lifecycle, from planning and coding through deployment and operations [2], [3]. The idea is simple to state and difficult to execute: every automated stage must now answer a security question quickly enough that it does not slow the pipeline it protects.

Two families of security testing sit at opposite ends of that pipeline and have developed largely apart. At the build side, vulnerability discovery tools examine source code, dependencies, and binaries for weaknesses before release. At the run side, intrusion detection systems examine live network behavior for evidence that something, possibly one of those unfixed weaknesses, is being attacked. Each side produces findings in its own format, on its own schedule, for its own audience. The developer who receives a static analysis warning rarely learns whether the affected function is being probed in production, and the operations analyst who receives an intrusion alert rarely learns which code change introduced the exposed component.

Artificial intelligence changes the economics of both activities. Learned models can scan code at the speed of a commit and can classify traffic at the speed of a link, and they can do so without the exhaustive hand-written rules that earlier tools required [4], [5]. Equally important, learned representations offer a common language in which findings from both ends might be compared, ranked, and connected. This review examines that opportunity. It focuses on two capabilities in particular: cross-project vulnerability

discovery, in which a model trained on some software projects is expected to find weaknesses in others [6], and real-time network intrusion detection, in which models classify traffic as it is observed. It asks how the two can be integrated into a DevSecOps pipeline so that each strengthens the other.

The review is organized around practical questions rather than around algorithms alone. What does a pipeline need from an AI vulnerability detector, and how well do current models meet those needs? What does a run-time detector need in order to function inside containerized, frequently changing infrastructure? What shared data and decision logic would let the two reinforce one another? And how should the combined system be evaluated so that reported benefits survive contact with production? The paper is a structured narrative review: sources were drawn from security, software engineering, and machine learning venues and selected to represent influential methods and recurring design patterns, and the aim is to synthesize rather than to enumerate exhaustively.

The main contributions are the following:

- An analysis of what each pipeline stage requires from AI-based security testing, expressed in terms of latency, granularity, and tolerance for error.
- A critical account of cross-project vulnerability discovery and of learning-based network intrusion detection, with emphasis on the gap between benchmark results and operational reality.
- A reference architecture and a set of integration patterns for coupling build-time and run-time detection through a shared evidence store and feedback loop.
- Guidance on pipeline-level evaluation and a prioritized list of open problems.

Section II reviews DevSecOps and the security testing landscape. Section III covers AI-based vulnerability discovery and Section IV covers learning-based intrusion detection. Section V presents the integration architecture, Section VI describes use cases, and Section VII addresses evaluation. Section VIII discusses challenges, Section IX outlines future directions, and Section X concludes.

II. DEVSECOPS AND THE SECURITY TESTING LANDSCAPE

A. Security as a Property of the Pipeline

A delivery pipeline is a sequence of automated stages that transforms a code change into a running service. The discipline of continuous delivery treats this pipeline as the single path to production and insists that every change pass through it [7]. DevSecOps extends that principle by attaching security checks to the same path. Practitioner studies describe recurring themes in this effort, including the need to automate security activities so that they scale with delivery speed, the importance of shared responsibility between development, security, and operations, and persistent friction when security tooling produces slow or noisy results [2], [3]. These themes translate directly into requirements for any AI component: it must be fast, it must be trusted by the people who receive its output, and it must fit the way teams already work.

B. Conventional Security Testing Techniques

Existing practice combines several complementary techniques, summarized in Table I. Static analysis examines code without running it and can run early, but it is well known for producing large numbers of warnings that developers learn to ignore, and studies of tool adoption identify false positives and unclear warnings as leading reasons developers abandon such tools [8], [9]. Dynamic testing exercises a running application and finds problems that only appear at run time, though it requires a deployed environment and tends to run later. Fuzzing generates large numbers of unusual inputs to trigger faults, and its techniques have been systematically surveyed as a powerful but resource-intensive approach [10]. Software

composition analysis checks third-party dependencies against known vulnerability records. Run-time monitoring, including network intrusion detection, observes behavior after release.

TABLE I
SECURITY TESTING TECHNIQUES MAPPED TO PIPELINE STAGES

Technique	Typical Stage	Strengths	Weaknesses
Static analysis	Commit, build	Early feedback; no execution needed	High false-positive volume [8], [9]; rule maintenance
Dynamic testing	Test, staging	Finds run-time and configuration flaws	Needs a running environment; limited code coverage
Fuzzing	Test	Finds unexpected faults with little prior knowledge [10]	Compute-intensive; harness effort; slow convergence
Composition analysis	Build	Fast matching against known vulnerable components	Cannot find unknown flaws; depends on database quality
AI vulnerability discovery	Commit, build	Learns patterns beyond hand-written rules [4]	Generalization and data-quality concerns [11]
Network intrusion detection	Staging, production	Observes real attacks and probing as they occur [5]	False alarms; drift; limited code context [12]

C. Where AI Fits

AI methods are best understood as additions to this toolkit that target its characteristic weak points. Learning-based vulnerability detectors aim to generalize beyond fixed rules, so that variants of a flaw can be recognized without a hand-written pattern. Learning-based intrusion detectors aim to recognize attack behavior in traffic without exhaustive signatures. In both cases the same cautions apply: a model is only as good as its training data, its performance in the laboratory often exceeds its performance in production, and its outputs must be calibrated against costs that differ sharply between a failed build and a missed breach. Subsequent sections examine each capability against these cautions before turning to their combination.

III. AI-DRIVEN VULNERABILITY DISCOVERY IN CODE

A. Problem Formulations

Learning-based vulnerability discovery is usually framed as classification at some level of granularity. At the coarsest level, a model labels a file or a commit as likely vulnerable or not. At function level, it labels each function, which matches how many datasets are constructed. At the finest level, it ranks or marks individual statements, which is closer to what a developer needs in order to act. Surveys of the area trace the progression from data-mining approaches using handcrafted metrics to deep neural architectures that learn features directly from code [4], [13]. For pipeline integration the choice of granularity matters, since a commit-level flag is cheap to compute but difficult to act on, whereas a statement-level flag is actionable but harder to produce reliably.

B. Representation and Model Families

Models differ chiefly in how they read code. Sequence models consume token streams and were among the earliest deep learning approaches to outperform feature-engineered baselines [14]. Slice-based systems focus on code fragments around sensitive operations [15]. Graph-based systems operate on structures that combine syntax, control flow, and data dependencies, building on the code property graph [16] and on graph neural networks that learn program semantics from it [17]. More recently, pretrained transformers, including models trained on both code and natural language and models that incorporate data-flow

information, have provided strong starting points that can be fine-tuned for vulnerability tasks [18], [19], and line-level transformers have been developed to localize the vulnerable statements within a function [20].

C. Cross-Project Generalization

The property most relevant to DevSecOps is generalization across projects. A pipeline protects many repositories, including new ones for which no labeled vulnerability history exists, so a detector that works only on the project it was trained on offers little value. Cross-project evaluation holds out entire projects at test time, and it consistently reveals a gap relative to within-project evaluation, because naming conventions, libraries, coding idioms, and vulnerability types differ from project to project. Representation learning with contrastive objectives is a promising response: by training an encoder to bring together code that shares vulnerability-relevant characteristics and to separate code that does not, regardless of project, such methods aim to learn features that transfer [21], [22]. Recent work applies contrastive representation learning directly to cross-project software vulnerability discovery [6], a direction that fits the needs of multi-repository pipelines.

D. Data Quality and Realism

Reported performance must be read with care. An influential study showed that deep learning vulnerability detectors that score well on existing benchmarks lose much of that performance on realistic data, and traced the loss to issues such as duplicated samples, unrealistic class balance, and models that learn dataset artifacts rather than vulnerability semantics [11]. Datasets assembled from public vulnerability fixes, such as large collections of C and C++ functions linked to vulnerability records, have enabled progress but carry label noise, since a function changed in a fix is not necessarily the only vulnerable one [23]. Later work on the quality of vulnerability datasets examines these problems systematically and cautions that noisy labels and duplicates distort conclusions [24]. For practitioners the implication is that a vendor claim of high accuracy on a public benchmark says little about behavior on the organization's own code, and that local validation is essential.

TABLE II
REQUIREMENTS OF PIPELINE STAGES AND THE FIT OF CURRENT VULNERABILITY MODELS

Pipeline Concern	What the Pipeline Needs	Current Capability	Remaining Gap
Latency	Results within the commit or build time budget	Small and medium models can score changes quickly	Large pretrained models need acceleration or caching [18], [19]
Granularity	Statement- or hunk-level pointers	Line-level localization emerging [20]	Reliability still below function-level detection
Generalization	Useful on new, unlabeled projects	Contrastive and pretrained approaches [6], [21]	Performance drop on realistic data [11]
Precision	Few false alarms to preserve developer trust	Threshold tuning and ranking	Calibration rarely validated on local code
Data quality	Trustworthy labels and fresh examples	Large public datasets [23]	Label noise and duplication [24]

IV. REAL-TIME NETWORK INTRUSION DETECTION

A. Detection in Modern Delivery Environments

Network intrusion detection predates DevSecOps, but its operating context has changed. Applications are decomposed into many services that communicate across internal networks, so most traffic moves

laterally between services rather than crossing a single perimeter. Infrastructure is ephemeral: containers and virtual machines appear and disappear with each deployment, so addresses and baselines shift continuously. Traffic is increasingly encrypted, limiting payload inspection. These conditions favor detectors that learn from flow-level statistics and timing, adapt quickly to new deployments, and can be provisioned automatically alongside the services they watch.

B. Learning Approaches

Machine learning has long been applied to intrusion detection, and broad surveys catalog methods ranging from decision trees and support vector machines to deep neural networks [5], [25]. Deep architectures learn representations from flow features or raw byte sequences, while unsupervised approaches model normal behavior and flag deviations, which is attractive when attack labels are scarce. A representative design uses an ensemble of small autoencoders to score network traffic online with modest resources [26]. Recent work explores how artificial intelligence can be applied to intrusion detection for network traffic analysis and improved cybersecurity [27], reflecting continuing interest in learned detectors that adapt to changing conditions. For DevSecOps, the key requirements are low latency, bounded resource use, and the ability to update models as services change, all of which favor compact models and incremental training.

C. Benchmarks and Operational Reality

The gap between benchmark and deployment is as important here as in code analysis. Public datasets differ widely in age, realism, and construction, and a survey of network-based intrusion detection data sets documents the variety of attack types, labeling methods, and limitations across them [28]. A frequently used dataset was generated from multi-day captures with a feature extraction tool [29], and another was produced in a laboratory testbed to include contemporary synthetic attacks [30], yet later analysis found labeling and flow-construction problems in the former [31]. More fundamentally, intrusion detection is an open-world problem in which benign traffic is vast and varied and attacks are rare, so even a low false-positive rate can generate an unmanageable number of alerts [12]. Studies of common mistakes in security machine learning add that unrealistic data splits and sampling bias routinely inflate reported performance [32].

TABLE III
CHARACTERISTICS OF LEARNING-BASED INTRUSION DETECTION APPROACHES

Approach	Supervision	Strengths	Limitations in Delivery Pipelines
Classical classifiers on flow features	Supervised	Fast; interpretable; low cost [5]	Needs labeled attacks; limited with encrypted or novel behavior
Deep supervised models	Supervised	Learn rich features from flows or bytes [25]	Label dependence; retraining cost; benchmark overfitting [32]
Autoencoder ensembles	Unsupervised	Online, lightweight, no attack labels [26]	Baseline drift when services redeploy; tuning of alert thresholds
Graph-based host models	Mixed	Capture lateral movement across services	Graph maintenance in ephemeral infrastructure

V. AN INTEGRATED ARCHITECTURE

A. Why Integrate

Build-time and run-time detection answer complementary questions. The first asks where weaknesses might exist in what is being shipped; the second asks what is actually being attacked in what is running.

Each has a characteristic blind spot. Code-level models flag far more potential weaknesses than any team can address, and they cannot tell which matter in practice. Traffic-level models can observe real attacks, but they cannot say which component was targeted or where the fix belongs. Joining them turns two noisy signals into mutually checking evidence: a flagged function that is also the target of observed exploitation traffic deserves immediate attention, while a flagged function behind an unreachable interface can wait.

B. Reference Architecture

Fig. 1 shows a reference architecture. The pipeline stages run from code to operation. A vulnerability discovery service scans commits and builds, and an intrusion detection service watches traffic in test, staging, and production environments. Both write to a shared risk and evidence store and read from it. The store holds findings, alerts, an inventory of deployed services and their versions, and metadata about known vulnerabilities and threats. A feedback path returns operational findings to development as new regression tests, labeled training examples, and revised gate policies.

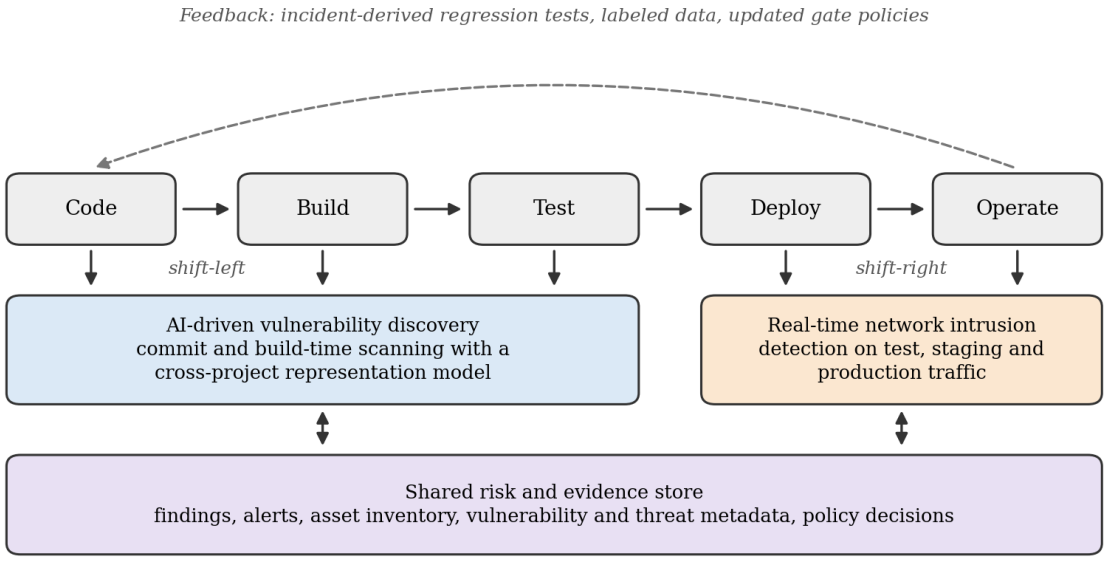


Fig. 1. Reference architecture for AI-driven security testing in DevSecOps. Build-time vulnerability discovery and run-time intrusion detection share a risk and evidence store, and a feedback path returns operational findings to the development side.

C. Shared Identifiers and the Evidence Store

The integration depends on being able to say that a finding in code and an alert in traffic concern the same thing. This requires a common vocabulary for components and versions, so that a build artifact can be mapped to the running service that carries it, and a service endpoint can be mapped back to the source that implements it. Vulnerability identifiers and affected-product records provide a further bridge between a code weakness and public knowledge about it, while a behavioral knowledge base such as MITRE ATT&CK offers shared labels for adversary techniques [33]. In practice the hardest engineering work lies in maintaining accurate links between commits, builds, deployments, and network entities, rather than in the models themselves.

D. Integration Patterns

Several patterns for coupling the two detectors can be distinguished, and Table IV compares them. In the loosely coupled pattern, each tool reports to a common dashboard and correlation is left to humans. In the correlation pattern, an engine joins findings by component and time and raises combined risk scores. In

the feedback pattern, run-time observations are used to retrain or reprioritize build-time models and to generate tests. In the joint modeling pattern, a single model consumes both code-derived and traffic-derived features, which offers the highest potential but also the highest data demands.

TABLE IV
PATTERNS FOR COUPLING BUILD-TIME AND RUN-TIME DETECTION

Pattern	Mechanism	Benefit	Cost and Risk
Loose coupling	Common dashboard; manual correlation	Easy to adopt; tool-agnostic	Analyst burden; slow response
Rule-based correlation	Join findings by component, version, and time	Transparent; quick to deploy	Depends on accurate asset and version mapping
Feedback loop	Alerts drive tests, labels, and retraining [34]	Adapts to drift; improves over time	Governance for retraining; label quality
Joint modeling	One model over code and traffic features	Can learn cross-source interactions	Needs linked data that rarely exists; harder to explain

E. Risk Scoring and Gate Policy

A combined risk score should reflect three things: the likelihood that a weakness is real, the likelihood that it is reachable and exploitable in the deployed configuration, and the evidence that it is being targeted. The first comes from the code model, the second from architecture and exposure data, and the third from traffic observations and external signals about exploitation in the wild [35]. Gate policies then translate scores into actions. A reasonable policy blocks a release only for findings that are both high-confidence and high-exposure, files lower-priority findings as tracked work, and escalates immediately when run-time evidence contradicts the assumptions that allowed a release to proceed. Policies expressed in this way are easier to defend to development teams than blanket thresholds on a single detector.

VI. USE CASES

A. Pre-Merge Review with Exposure Context

When a change is proposed, the vulnerability model can flag suspicious hunks, and the evidence store can supply context about the affected service: whether it is internet-facing, whether it has recently been probed, and whether similar flaws were previously exploited. Presenting that context alongside the finding helps a reviewer decide quickly whether the warning is worth interrupting work for. This directly addresses the observed tendency of developers to ignore warnings that arrive without context [9].

B. Attack Replay in Staging

Pre-production environments offer an opportunity to test both detectors together. Known exploit traffic for a newly flagged weakness can be replayed against a staging deployment, and the intrusion detector can be checked to confirm that it raises an alert. If it does not, the gap is found before release rather than after an incident. Conversely, if replayed traffic succeeds against the application, the corresponding code finding gains confidence. This closes a verification loop that neither detector can close alone.

C. Runtime-Informed Test Prioritization

Fuzzing and dynamic testing are expensive, and effort is wasted when spent on components that attackers ignore [10]. Observations of which services and interfaces attract scanning or exploitation attempts can guide where testing resources go, and findings from the code model can guide where fuzzing

harnesses should focus. The result is a test plan driven by both predicted weakness and observed adversary interest.

D. Incident-to-Regression Conversion

When an incident occurs, the traffic evidence, the affected code path, and the eventual fix can be packaged as a regression test and as labeled training data. Over time, this converts the organization's own history into a dataset tailored to its code and traffic, which partially addresses the realism gap discussed in Sections III and IV. Care is needed to ensure the labels are accurate and that retraining is governed, but the potential to build locally relevant datasets is among the strongest arguments for integration.

E. Supply Chain and Third-Party Components

Modern services embed many third-party components whose source may be unavailable or unreviewed. Composition analysis identifies known vulnerable versions, while the intrusion detector can reveal anomalous behavior from those components at run time, such as unexpected outbound connections. Combined, they provide protection against both known and previously unknown problems in code the organization did not write.

VII. EVALUATION

A. Model-Level Evaluation

Model-level evaluation should follow practices that guard against inflated results. For vulnerability models, this means cross-project splits, deduplication, realistic class ratios, and testing on code that postdates the training data [11], [24]. For intrusion detection, it means temporal splits, realistic base rates, and reporting of false-positive burden in alerts per unit time rather than only accuracy [12], [32]. Precision-recall analysis is more informative than aggregate accuracy for both tasks because positives are rare. Explanations of individual predictions, for example using feature attribution methods, should be assessed for faithfulness as well as readability [36].

B. Pipeline-Level Evaluation

The value of integration shows up at the level of the pipeline, where the relevant measures are operational. Useful indicators include the proportion of findings that developers act on, the time from detection to remediation, the number of releases delayed by security gates and the proportion of those delays judged justified in retrospect, the time to detect and triage run-time incidents, and the number of incidents whose root cause had been flagged before release. Because software delivery performance is commonly assessed through measures of throughput and stability [1], security automation should be evaluated for its effect on those measures too: a control that improves detection but doubles lead time may not be sustainable.

C. Experimental Designs

Rigorous evaluation of an integrated system is difficult because ground truth is scarce. Three designs are practical. Retrospective replay re-runs historical commits and traffic through the system and compares its output with known incidents and fixes. Controlled injection plants known weaknesses and attack traffic in a staging environment and measures detection and correlation. Longitudinal field studies compare pipeline metrics before and after adoption, acknowledging confounding factors. Using more than one design, and reporting negative results, would strengthen the evidence base considerably.

VIII. CHALLENGES

A. False Positives and Developer Trust

The most consistent theme in the evidence on security tooling is that noise erodes trust [8], [9]. AI components can aggravate or relieve this problem depending on calibration. Models should output calibrated confidence, thresholds should be set on local data, and the system should learn from developer dismissals rather than repeatedly presenting the same rejected finding. Suppression must be auditable so that it does not become a way to hide real problems.

B. Drift and Model Maintenance

Both code and traffic distributions change continuously, as languages, frameworks, and architectures evolve and as services are redeployed. Surveys of concept drift describe methods for detecting and adapting to such change [34], and the integration architecture should include drift monitoring on both detectors. More broadly, machine learning systems carry maintenance burdens beyond those of ordinary software, including data dependencies, entangled components, and configuration debt [37], and these burdens must be planned for when models become part of the delivery pipeline.

C. Adversarial and Supply Chain Risks

Learned detectors can themselves be attacked. Adversaries may craft code that evades vulnerability models, or traffic that evades intrusion detectors, and attacks against security models must produce inputs that remain valid in the problem domain, a constraint that shapes both attack and defense [38]. Training data, pretrained models, and the pipeline's own tooling form a supply chain that can be poisoned. Defenses include provenance tracking for data and models, adversarial testing as part of model release, and diversity among detectors so that no single evasion defeats the system.

D. Data Sharing and Privacy

Cross-project learning benefits from data across many repositories and organizations, but code is proprietary and traffic can contain personal information. Federated learning, in which models are trained across data holders without centralizing raw data, offers one route [39], although it introduces coordination overhead and does not remove the need for governance. Organizations should also consider what information the evidence store retains and who can query it, since it concentrates sensitive knowledge about both weaknesses and live exposure.

E. Resource Constraints and Latency

Pipelines have strict time budgets and run-time detectors have strict throughput demands. Large pretrained models can be impractical to run on every commit without optimization, and deep traffic models may not keep pace on high-speed links. Techniques such as incremental scanning of only changed code, caching of embeddings, model distillation, and sampling of traffic can help, but each trades some accuracy for speed. These trade-offs should be made explicit and tuned per environment.

IX. FUTURE DIRECTIONS

Several lines of research would advance the field most. Linked benchmarks are the first priority: datasets that connect vulnerable code, the deployments that run it, and the traffic that targets it would allow integrated systems to be compared rigorously, and none currently exists in public form. Second, better cross-project methods are needed, including representation learning that remains robust under realistic data, along with training sets that reflect real vulnerability distributions rather than curated samples [6], [11].

Third, run-time detectors should become deployment-aware, using knowledge of service topology and release events to reset baselines instead of misreading each redeployment as an anomaly.

Further work should explore large language models as components in the pipeline, for example to explain findings in developer-friendly terms, to draft candidate fixes, or to generate exploit-like test traffic, with careful attention to reliability and misuse. Causal and counterfactual reasoning could help determine whether an observed attack actually depends on a flagged weakness. Finally, human factors deserve study equal to algorithmic work: how developers and analysts perceive, trust, and act on AI-generated findings will determine whether the technology changes practice, and controlled studies of these interactions remain rare.

X. CONCLUSION

DevSecOps asks security to move at the speed of delivery, and AI offers the means to automate two of its most important activities: finding vulnerabilities before release and detecting intrusions after it. Treated separately, each yields a stream of uncertain findings. Treated together, through a shared evidence store, common identifiers, and a feedback loop, they can validate and prioritize one another, directing scarce attention toward weaknesses that are both likely and actually under attack. Progress in cross-project representation learning for code [6] and in learned traffic analysis [27] supplies the technical foundation.

The path to realizing this promise runs through practical rather than purely algorithmic obstacles. Models must be evaluated on realistic data and judged by pipeline outcomes, not only by benchmark scores. Findings must be calibrated and contextualized so that developers trust them. Models must be maintained as code and traffic evolve and defended against adversaries who target them. And the field needs shared, linked datasets so that claims can be tested. Where these conditions are met, AI-driven security testing can move DevSecOps from a collection of gates toward a continuous, evidence-driven control loop.

DECLARATIONS

Conflict of Interest. The authors declare that they have no known financial or personal relationships that could have appeared to influence the work reported in this paper.

Funding. This research received no specific grant from any funding agency in the public, commercial, or not-for-profit sectors.

Ethical Approval. Not applicable. This article is a literature review and does not involve human participants, animals, or personal data.

Informed Consent. Not applicable.

Use of Artificial Intelligence. Generative AI tools were used to assist with drafting, structuring, and language editing of this manuscript. The authors reviewed and edited all content and take full responsibility for the accuracy, originality, and integrity of the published work, including the verification of all cited references.

Data Availability. No new datasets were generated or analyzed in this study. All sources discussed are cited in the reference list.

REFERENCES

[1] N. Forsgren, J. Humble, and G. Kim, *Accelerate: The Science of Lean Software and DevOps*. Portland, OR, USA: IT Revolution Press, 2018.

[2] A. A. U. Rahman and L. Williams, "Software security in DevOps: Synthesizing practitioners' perceptions and practices," in *Proc. Int. Workshop Continuous Softw. Evol. Delivery (CSED)*, 2016, pp. 70–76.

[3] H. Myrbakken and R. Colomo-Palacios, "DevSecOps: A multivocal literature review," in *Proc. Int. Conf. Softw. Process Improvement Capability Determination (SPICE)*, 2017, pp. 17–29.

[4] G. Lin, S. Wen, Q.-L. Han, J. Zhang, and Y. Xiang, "Software vulnerability detection using deep neural networks: A survey," *Proc. IEEE*, vol. 108, no. 10, pp. 1825–1848, Oct. 2020.

[5] A. L. Buczak and E. Guven, "A survey of data mining and machine learning methods for cyber security intrusion detection," *IEEE Commun. Surveys Tuts.*, vol. 18, no. 2, pp. 1153–1176, 2nd Quart., 2016.

[6] R. Purushothaman, N. S. P. K. Yadati, P. Nutalapati, S. M. Nethi, S. Dodda, and A. K. Thimmapuram, "Contrastive representation learning for cross-project software vulnerability discovery," in *Proc. 2026 Int. Conf. Artif. Intell., Syst., Emerg. Technol. (ICAISET)*, Apr. 2026, pp. 1–6.

[7] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Boston, MA, USA: Addison-Wesley, 2010.

[8] A. Bessey *et al.*, "A few billion lines of code later: Using static analysis to find bugs in the real world," *Commun. ACM*, vol. 53, no. 2, pp. 66–75, Feb. 2010.

[9] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?," in *Proc. 35th Int. Conf. Softw. Eng. (ICSE)*, 2013, pp. 672–681.

[10] V. J. M. Manès *et al.*, "The art, science, and engineering of fuzzing: A survey," *IEEE Trans. Softw. Eng.*, vol. 47, no. 11, pp. 2312–2331, Nov. 2021.

[11] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, "Deep learning based vulnerability detection: Are we there yet?," *IEEE Trans. Softw. Eng.*, vol. 48, no. 9, pp. 3280–3296, Sep. 2022.

[12] R. Sommer and V. Paxson, "Outside the closed world: On using machine learning for network intrusion detection," in *Proc. IEEE Symp. Secur. Privacy (S&P)*, 2010, pp. 305–316.

[13] S. M. Ghaffarian and H. R. Shahriari, "Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey," *ACM Comput. Surv.*, vol. 50, no. 4, Art. no. 56, 2017.

[14] R. Russell *et al.*, "Automated vulnerability detection in source code using deep representation learning," in *Proc. 17th IEEE Int. Conf. Mach. Learn. Appl. (ICMLA)*, 2018, pp. 757–762.

[15] Z. Li *et al.*, "VulDeePecker: A deep learning-based system for vulnerability detection," in *Proc. Netw. Distrib. Syst. Secur. Symp. (NDSS)*, 2018.

[16] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *Proc. IEEE Symp. Secur. Privacy (S&P)*, 2014, pp. 590–604.

[17] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," in *Proc. Adv. Neural Inf. Process. Syst. (NeurIPS)*, vol. 32, 2019.

[18] Z. Feng *et al.*, "CodeBERT: A pre-trained model for programming and natural languages," in *Findings Assoc. Comput. Linguistics: EMNLP*, 2020, pp. 1536–1547.

[19] D. Guo *et al.*, "GraphCodeBERT: Pre-training code representations with data flow," in *Proc. Int. Conf. Learn. Represent. (ICLR)*, 2021.

[20] M. Fu and C. Tantithamthavorn, "LineVul: A transformer-based line-level vulnerability prediction," in *Proc. 19th Int. Conf. Mining Softw. Repositories (MSR)*, 2022, pp. 608–620.

[21] T. Chen, S. Kornblith, M. Norouzi, and G. Hinton, "A simple framework for contrastive learning of visual representations," in *Proc. 37th Int. Conf. Mach. Learn. (ICML)*, 2020, pp. 1597–1607.

[22] P. Khosla *et al.*, "Supervised contrastive learning," in *Proc. Adv. Neural Inf. Process. Syst. (NeurIPS)*, vol. 33, 2020, pp. 18661–18673.

[23] J. Fan, Y. Li, S. Wang, and T. N. Nguyen, "A C/C++ code vulnerability dataset with code changes and CVE summaries," in *Proc. 17th Int. Conf. Mining Softw. Repositories (MSR)*, 2020, pp. 508–512.

[24] R. Croft, M. A. Babar, and M. M. Kholoosi, "Data quality for software vulnerability datasets," in *Proc. 45th Int. Conf. Softw. Eng. (ICSE)*, 2023.

[25] M. A. Ferrag, L. Maglaras, S. Moschoyiannis, and H. Janicke, "Deep learning for cyber security intrusion detection: Approaches, datasets, and comparative study," *J. Inf. Secur. Appl.*, vol. 50, Art. no. 102419, 2020.

[26] Y. Mirsky, T. Doitshman, Y. Elovici, and A. Shabtai, "Kitsune: An ensemble of autoencoders for online network intrusion detection," in *Proc. Netw. Distrib. Syst. Secur. Symp. (NDSS)*, 2018.

[27] M. Kumar, S. Janarthanan, S. Dodda, P. Parida, and M. S. Soumik, "Exploring the artificial intelligence (AI) for intrusion detection system in network traffic analysis for improved cybersecurity," in *Proc. 2026 IEEE Int. Conf. AI Eng. Innov. (AIEI)*, Mar. 2026, pp. 1–6.

[28] M. Ring, S. Wunderlich, D. Scheuring, D. Landes, and A. Hotho, "A survey of network-based intrusion detection data sets," *Comput. Secur.*, vol. 86, pp. 147–167, Sep. 2019.

[29] I. Sharafaldin, A. H. Lashkari, and A. A. Ghorbani, "Toward generating a new intrusion detection dataset and intrusion traffic characterization," in *Proc. 4th Int. Conf. Inf. Syst. Secur. Privacy (ICISSP)*, 2018, pp. 108–116.

[30] N. Moustafa and J. Slay, "UNSW-NB15: A comprehensive data set for network intrusion detection systems (UNSW-NB15 network data set)," in *Proc. Mil. Commun. Inf. Syst. Conf. (MilCIS)*, 2015, pp. 1–6.

[31] G. Engelen, V. Rimmer, and W. Joosen, "Troubleshooting an intrusion detection dataset: The CICIDS2017 case study," in *Proc. IEEE Secur. Privacy Workshops (SPW)*, 2021, pp. 7–12.

[32] D. Arp *et al.*, "Dos and don'ts of machine learning in computer security," in *Proc. 31st USENIX Secur. Symp.*, 2022, pp. 3971–3988.

[33] B. E. Strom *et al.*, "MITRE ATT&CK: Design and philosophy," The MITRE Corporation, McLean, VA, USA, Tech. Rep., 2018.

[34] J. Gama, I. Žliobaitė, A. Bifet, M. Pechenizkiy, and A. Bouchachia, "A survey on concept drift adaptation," *ACM Comput. Surv.*, vol. 46, no. 4, Art. no. 44, 2014.

[35] C. Sabottke, O. Suciu, and T. Dumitraș, "Vulnerability disclosure in the age of social media: Exploiting Twitter for predicting real-world exploits," in *Proc. 24th USENIX Secur. Symp.*, 2015, pp. 1041–1056.

[36] S. M. Lundberg and S.-I. Lee, "A unified approach to interpreting model predictions," in *Proc. Adv. Neural Inf. Process. Syst. (NeurIPS)*, vol. 30, 2017, pp. 4765–4774.

[37] D. Sculley *et al.*, "Hidden technical debt in machine learning systems," in *Proc. Adv. Neural Inf. Process. Syst. (NeurIPS)*, vol. 28, 2015, pp. 2503–2511.

[38] F. Pierazzi, F. Pendlebury, J. Cortellazzi, and L. Cavallaro, "Intriguing properties of adversarial ML attacks in the problem space," in *Proc. IEEE Symp. Secur. Privacy (S&P)*, 2020, pp. 1332–1349.

[39] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y Arcas, "Communication-efficient learning of deep networks from decentralized data," in *Proc. 20th Int. Conf. Artif. Intell. Statist. (AISTATS)*, 2017, pp. 1273–1282.