A Proxy-Based DevSecOps Framework for Multi-Tier Web Applications

Abderrahim Rida, Abdelaziz Bakhil, and Ayoub Ait Lahcen
Engineering Sciences Laboratory-National School of Applied Sciences,
Ibn Tofail University, Kenitra, Morocco

***Abstract*—Multi-tier web applications face significant implementation challenges in DevSecOps, including tool integration complexity, automation gaps, and cultural resistance. This study presents a proxy-based DevSecOps framework grounded in a formal architectural decomposition that transforms classical $O(n \times m)$ pipeline integration complexity into additive $O(n_1 \times m_1) + O(n_2 \times m_2)$ components through the separation of static and dynamic execution contexts. The framework is instantiated for PHP-based multi-tier web applications deployed on AWS infrastructure using Terraform-managed Infrastructure as Code principles; all ecosystem coherence metrics, toolset analysis, and complexity computations are derived within this specific technology context, and generalization to other language ecosystems or cloud platforms constitutes a boundary condition discussed in Section V. Theoretical contributions include: 1) a formal proxy pipeline architecture with mathematical complexity analysis demonstrating that complexity reduction is guaranteed when tool ecosystem coherence exceeds 70%; 2) systematic tool integration using PHP-specific tooling, yielding a theoretical ecosystem coherence of 76.9%; and 3) theoretical validation addressing 18 out of 20 identified DevSecOps implementation challenges. Mathematical analysis theoretically predicts a 48.13% reduction in tool integration conflicts and a 61.9% toolset reduction relative to traditional monolithic pipelines through context separation. All quantitative figures presented above are theoretically derived predictions, not empirically measured outcomes. They are formalized as falsifiable hypotheses H1 through H5 in Section V, with empirical validation identified as the primary direction for future work.**

Keywords*—DevSecOps; proxy architecture; multi-tier applications; PHP; AWS; security testing; CI/CD pipeline optimization; ecosystem coherence

I. Introduction

Incorporating DevSecOps in multi-tier web applications also presents significant challenges to implementing, including cultural resistance, automation gaps, and tool integrations complexitiy [1], [2]. While several reference architectures have been published [3], [4], [5], none exist that are specific to multi-tier application scenarios, in which rich deployment workflows, heterogeneous layers of services, and interactions between databases present unique security testing needs that standard DevSecOps practices do not come close to fulfilling.

Today, most common DevSecOps frameworks have three chief weaknesses regarding multi-tier applications: 1) monolithic toolchains resulting in $\mathcal{O}(n \times m)$ integration overhead, where n is the pipeline stages and m is the security tools, 2) too little architectural separation of phases of SAST and DAST, and 3) missing purpose-built frameworks optimized for meeting multi-tier architecture needs, including database syncing, service composition, and security settings optimized for targeted envs.

The matrix [4] provides the different integration points and levels of automation associated with multiple DevSecOps tools throughout the software development lifecycle (SDLC). Tools such as open-source governance and Static Application Security Testing (SAST) show a high level of automation throughout the process. In the case of others, there is Dynamic Application Security Testing (DAST) and Interactive Application Security Testing (IAST) that are mostly automated in the Continuous Integration/Continuous Deployment (CI/CD) and Post-Deployment phases. Mobile application security testing and Runtime Application Self-Protection (RASP) at best automate a couple of capabilities. The hierarchical approach of adoption emphasizes how critical it is to begin with secure software development through automation and integrating testing at each level, hence making quality SDLC efficient and effective [6].

Report [3], [4], [5] is based on a list of tools intended to integrate security into the SDLC. Jenkins and Ansible provision infrastructure and CI/CD pipelines automatically, whereas Twistlock, Aqua Security, and Docker secure containerized applications and cloud-native platforms. To scan for vulnerabilities, the tools OWASP ZAP, Nessus, SonarQube, and Veracode perform static, dynamic, and interactive application security testing (SAST, DAST, IAST). Dependency management is handled by Nexus Lifecycle and Dependency-Check, which identify third-party dependency vulnerabilities. Istio also provides microservices' service mesh security, and Splunk, FireEye, etc., provide monitoring and threat detection capability. All these integrated together offer a robust DevSecOps pipeline in which security is integrated from code creation through to deployment and runtime [7].

Challenges identified in recent studies [2], [8], [9] highlight our motivation to create a specific CI/CD pipeline that could address some of these issues. Since the goal of DevSecOps is to promote quick and secure delivery, it is important to develop a guide that can help DevSecOps teams in their pipeline. This work further provides a reference implementation for the DevSecOps community. Right now, looking for existing pipelines will allow us to understand the details of each one and extract best practices from them. By taking this approach, we can design a useful pipeline by learning from existing work and finding solutions to many challenges.

- Design a proxy-based DevSecOps architecture that reduces tool integration complexity for multi-tier web applications.

- Develop a theoretical framework demonstrating mathematical complexity reduction through architectural separation.
- Provide a specialized solution addressing multi-tier web application deployment challenges.
- Establish a foundation for empirical validation of the proposed theoretical framework.

This research focuses specifically on multi-tier PHP web applications deployed on AWS infrastructure, utilizing Infrastructure as Code principles through Terraform implementation. The framework addresses static security testing separation while maintaining dynamic testing integration in production environments.

This study provides a proxy-centric DevSecOps framework to effectively overcome these limitations by employing architectural separation concepts tailor-made for multi-tier web applications. Our conceptual contributions include: 1) an accurate definition of the proxy pipeline architecture and an evaluation of its mathematical complexity, 2) an orderly procedure for tool integration based on ecosystem coherence metrics, and 3) an exhaustive conceptual verification showing the mitigation of challenges realized in DevSecOps implementation.

The manuscript is structured as follows: the next section contains some necessary background knowledge with the concepts and tools used in such pipelines. The third section presents our motivation and problem formalization. The fourth one examines the relevant worldwide DevSecOps pipelines. The fifth describes the specialized pipeline underlining the envisaged approach of this research. The last one provides the theory validation and Analysis of the proposed Multi-tier Web applications DevSecOps pipeline.

II. Related Work

A. Comprehensive DevSecOps Reference Architectures

The DevSecOps reference architecture collection [3] offers several pipeline deployments which illustrate a wide range of ways security is integrated into the SDLC. The architectures focus on a number of core principles such as continuous security integration, shift-left strategies whereby security tests are pushed into earlier stages of the SDLC, and automated security testing paradigms. The collection features deployments from basic CI/CD pipelines which include security tools integrated into the pipeline all the way through enterprise-class architectures which accommodate microservices environments.

In these reference architectures [3], various implementation patterns arise. The pipelines indicate trust and confidence development by security tool integration as the significant elements of CI/CD processes. The implementations compare DevOps delivery-based methods with DevSecOps security-included methods in the whole of the pipelines. The architectures indicate how the integration of security covers the whole SDLC activities from planning and code writing phases up through deployment, including the security as well as code analysis for potential vulnerabilities during the beginning of development cycles.

These reference architectures [3] also demonstrate workflow automation tactics employing tools including Jenkins, unit testing, code analysis, and package creation of artifacts. These pipelines deploy broad monitoring with feedback loops for the formation of continued improvement, indicating how security can be embedded across software supply chains where DevOps would traditionally focus on delivery teams whereas DevSecOps performs the security activities across full pipelines.

B. Sonatype DevSecOps Framework

Sonatype DevSecOps Reference Architecture [4], [5] outlines core pillars from ongoing education for every party at interest, secure supply chains for open-source components, and automated test methodologies using SAST, DAST, feedback mechanisms and monitoring. The framework includes detailed automation levels and integration points involving a variety of DevSecOps tools during the software development lifecycle.

The Sonatype architecture [4], [5] shows the potential of tools, such as Static Application Security Testing and open-source governance, to reach high levels of automation during the development lifecycle. The configuration shows clear integration of the security controls during the SDLC, whereby developers develop the applications, store them in the Nexus Repository, and use lifecycle management tools like Nexus Lifecycle for constructing and delivering. This approach ensures detection of vulnerabilities during the application build or release stages.

Under the reference implementations [4], [5], the architecture embodies holistic CI/CD pipelines for absorbing security requirements during the coding, building, testing, release, deployment, operation, and monitoring stages. Security check measures are integrated into each stage, which implies that security is not merely added but ingrained into each stage of software development. The framework highlights the importance of involving security teams throughout every stage, rather than having them only deliver analysis reports to the operations teams.

C. Formal DevSecOps Implementations

E-SPIN Corporation [10] represents the concept of Continuous Security within a DevSecOps framework, enabling a proactive approach to find vulnerabilities and integrating security into both development and operations tasks.

The DoD Enterprise DevSecOps [11] presents a technology stack that involves embedding and deploying security measures and practices into the SDLC with various tools. Each component has the goal of securing and automating the software delivery pipeline with appropriate DoD compliance requirements. The DoD Enterprise DevSecOps Platform Architecture details a secured microservices architecture showing how Istio creates a Service Mesh architecture for managing microservices, bringing traffic management associated with security features to address complex microservices environments.

The Application Security Orchestration and Correlation framework demonstrates how diverse security tools come together to execute assessments and identify vulnerabilities, ensuring effective security governance throughout the SDLC [12].

D. Industry DevSecOps Pipeline Implementations

Industry-based DevSecOps pipeline executions [13], [14] target software supply chain security and how critical effective dependency management and vulnerability scanning during the SDLC ensure no vulnerable applications get shipped. CI/CD pipelines with a security focus include all the security-related activities so pipelines do not overlook critical security concerns at each stage of software release, leading to more reliable and secure releases.

Current DevSecOps practices [15] are formulated as frameworks aimed at safeguarding the software supply chain, illustrating how instruments such as Dependency-Check and Dependency-Track can detect and monitor vulnerabilities within software dependencies. This approach helps organizations keep their software secure by carefully managing all its dependencies.

Further industry deployments [16] provide DevSecOps automation orchestration pipelines for software development lifecycle automation with security emphasis. These strategies exhibit how Information Security teams work together with IT Operations teams for infrastructure automation code review and contribution purposes, providing security integration into the components of the infrastructure.

E. Evolution of Security-Integrated Development

Software development has evolved from traditional, isolated methods to more integrated approaches that seamlessly incorporate security throughout the process. This progress reflects the interweaving of security concepts during development phases from planning and design all the way through ongoing tests and monitoring in production settings.

Today's DevSecOps platforms come with collections of tools which automate and simplify the processes engaged with software development. Such deployments enable development teams to improve productivity, communicate better, and deploy software of high quality with increased security mitigations. Distributed build and test operations are also facilitated by these architectures, thus meaning security has been integrated at multiple points, from testings in a range of environments to further security analyses of the CI/CD pipelines themselves.

Today's secured CI/CD deployments indicate the value of security throughout the SDLC in the form of secure delivery, which serves as the prime objective of DevSecOps architecture integration. These strategies indicate how security tools may be packaged within central platforms providing consistency as well as automation for security services necessitating coordinated responses.

F. Research Gap Analysis

While current DevSecOps reference architectures [3], [4], [5] offer detailed guidelines for integrating security, they don't fully address the challenges of applying these practices to multi-tier web applications.. They tend to treat deployment as monolithic activities without the specific needs for database sync, service orchestration, and tier-specific security requirements which characterize multi-tier applications.

The compared architectures support a set of methodologies for integrating tools but lack well-structured frameworks for managing the complexity involved when coordinating a set of multiple security tools operating at different application layers. The current deployments also don't provide an architectural separation between the static and dynamic security testing phases, leading to resource conflicts and less than optimal running of the pipeline.

This study tackles these deficiencies by introducing a proxy-based architecture that is tailored for multi-tier web applications. It offers a theoretical framework aimed at diminishing complexity through architectural segregation, all while ensuring extensive security coverage throughout the various application tiers.

III. Problem Formalization

A. DevSecOps Implementation Challenges

Analysis of existing DevSecOps resources [3], [4], [5] reveals that , the context of DevSecOps pipelines suggests that there are significant barriers to introducing security into SDLC. Challenges [2], [8], [9] include tool integration, automation gaps, cultural resistance, compliance, and limited resources, among others. Addressing these challenges entails technical solutions, culture change, and continuous improvement.

Various tools like Jenkins, Twistlock, Aqua, OWASP ZAP, and others are mentioned and need to be combined, this leads to Integration Complexity , because Integrating multiple security tools into the CI/CD pipeline can be complex and may lead to tool conflicts or inefficiencies.

Dynamic Application Security Testing and Run-time Application Self Protection are not fully automated across all stages of the pipeline. To achieve full automation of security testing and compliance checks is difficult.

The activities of identifying, prioritizing, and remedying vulnerabilities within a reasonable timeframe can be challenging, particularly with DevSecOps environments. Metrics such as "Time to patch vulnerabilities" reflect the importance of quick detection and remediation of vulnerabilities that are frequently challenging.

SecDevOps pipelines implement a number of tools at various stages that could swamp operations, security and developers teams. Integrations of those numerous security tools can bring about a state of tool overload as teams become challenged with managing and maintaining the tools efficiently.

Cultural aspect is highlighted through collaboration and mutual responsibility of DevSecOps Teams that are hard to realize across most organizations. DevSecOps needs a cultural approach where the teams of development, security, and operations work intimately. In the same manner, the requirement of "Security Champions" as an over-reliance on specialty skills. There just are no trained individuals with dual security and DevOps skill sets available to implement and run DevSecOps pipelines.

Compliance validation and policy enforcement can cripple the pipeline process if not properly established. Being compliant with regulatory requirements and being agile at the same time in the pipeline is a very big challenge.

Static Application Security Test are prone to false positives that need to be manually validated. Security tools also generate

false positives, which can lead to alert fatigue and lower the effectiveness of the pipeline.

DevSecOps pipelines reference the difficulty of scaling security in big distributed systems especially in micro-services architectures.

Secured pipelines will be likely centered on centralized logging and telemetry, signaling that visibility is concern number one. Visibility across the security stance of applications and infrastructure end-to-end is challenging to obtain, especially in cloud-native environments.

Some of the biggest challenges lie in managing and securing third-party dependencies (e.g., open-source libraries), as security vulnerabilities in dependencies can lead to risks.

Real-time monitoring and feedback are highlighted, and this requires good processes and tools. Establishing a continuous feedback loop to improve security practices over a period of time is tricky, especially in changing environments. On the same line, The DoD architecture has Zero Trust as one of the Istio benefits, but implementing it is hard. Applying a Zero Trust model to micro-services architecture is hard and expensive.

Legacy systems are not necessarily part of modern DevSecOps pipelines: It is difficult to incorporate DevSecOps practices into legacy systems and applications due to old architectures and the lack of automation.

Commercial tools with high licensing fees and maintenance are used in many secured pipelines, It is financially costly to put in place a DevSecOps pipeline, having high tool investments, training, and infrastructure costs.

Addressing security incidents in a timely manner in a DevSecOps environment requires a strong process and tools, and that may be difficult to effect.

Metrics like "Deployment frequency" and "Change lead time" identify the tension between velocity and security.Where achieving speed of delivery is an imperative, and one also needs to ensure thorough security testing, this is a constant challenge in DevSecOps.

Some tools are configured with no integration plan in sight: having multiple disparate tools creates fragmentation, where it is difficult to get an end-to-end integrated view of security across the pipeline.

Threat modeling tends to be an afterthought in high-velocity environments, including it in the DevSecOps pipeline is not straightforward because it requires skill and time.

The importance of constant improvement is always required, but this is an ongoing effort and cost. Maintaining pace with changing security threats and adjusting the pipeline in response is an ongoing challenge. Table I highlights the most important DevSecOps challenges [2], [8], [9].

B. Problem Formalization for Multi-Tier Applications

Multi-tier web application DevSecOps has mathematically expressible complexity challenges. Assume $P = \{p_1, p_2, \ldots, p_n\}$ to be a DevSecOps pipeline with n stages, and $T = \{t_1, t_2, \ldots, t_m\}$ to be m tools. The classical methods establish dependency relations $D = \{(p_i, t_j) \mid$ it executes as tool t_j in phase $p_i\}$, leading to pipeline complexity $O(n \times m)$.

Multi-tier applications multiply this complexity through:

- Requirements for synchronizing databases across staging, production, and development environment.
- Service level dependencies requiring sequenced deployment synchronizations (service orchestration).
- Environment-specific configuration management and security policy enforcement.
- Cross-tier communication security verification criteria.

In this particular instance, service orchestration means controlled management of several layers of an application whereby:

- Database tier must be operational before you can install application tier.
- Application services have to reach healthy state prior to web tier configuration.
- Security policies should be implemented and verified in a sequential manner at each level.
- Failure at one level requires coordinated rollback in precedent levels.

Proxy architecture overcomes these challenges through separation of execution contexts:

$$\text{Static Context } C_s = \{\text{checkout, build, compile, unit\_test, code\_analyze, style\_test, security\_scan}\},$$

$$\text{Dynamic Context } C_d = \{\text{deploy, monitor, runtime\_protection}\}.$$

The separation reduces complexity from multiplicative

$$O(nm)$$

to additive

$$O(n_1 m_1) + O(n_2 m_2),$$

where, $n_1 + n_2 \leq n$ and $m_1 + m_2 \leq m$.

This achieves theoretical reduction whenever tool ecosystem coherence exceeds 70%.

IV. Our Proposal Proxy DevSecOps Architecture

A. Architectural Foundation and Benefits

These documented DevSecOps architectures have significant benefits from baking security directly into the SDLC. This "shift-left" approach improves overall security posture through automated security testing, continuous monitoring, and earlier detection of vulnerabilities for a more robust product with fewer security breaches. Again, a focus on automation, continuous integration and delivery, and collaboration between

TABLE I. DEVSECOPS CHALLENGES

DevSecOps Challenge	Brief Description
Tool Integration	Difficulty combining multiple security tools without conflicts.
Automation Gaps	Security tools not fully automated across the pipeline.
Patch Management	Hard to identify, prioritize, and fix vulnerabilities quickly.
Tool Overload	Too many tools overwhelm Dev, Sec, and Ops teams.
Cultural Resistance	Lack of collaboration and shared responsibility between teams.
Skills Gap / Security Champions	Shortage of professionals skilled in both security and DevOps.
Compliance Constraints	Balancing agility with regulatory compliance is difficult.
False Positives	Manual validation of false alerts leads to inefficiency.
Scalability	Difficult to scale security across distributed/microservices systems.
Limited Visibility	Hard to get end-to-end insight across cloud-native infrastructure and apps.
Third-Party Dependencies	Risks from vulnerabilities in open-source and external components.
Continuous Feedback Loop	Hard to build effective, timely feedback systems in fast-paced environments.
Zero Trust Implementation	Complex and costly to apply Zero Trust models to microservices.
Legacy Systems Integration	Difficult to apply DevSecOps to outdated systems lacking automation.
High Costs	Licensing, infrastructure, and training for DevSecOps tools are expensive.
Incident Response & Root Cause Analysis	Demands strong coordination and effort to resolve security incidents.
Velocity vs. Security	Balancing fast delivery with deep security testing is a continuous tension.
Tool Fragmentation	Disparate tools without integration reduce pipeline visibility.
Threat Modeling Neglect	Often skipped due to time and expertise required.
Continuous Improvement Burden	Constant updates needed to keep pace with evolving threats.

the development, security, and operations teams yields huge gains in efficiency and speed. Smaller feedback cycles, less rework, and better distribution of resources mean shorter time to market and costs for the firm, while affording the ability for the organization to adapt quicker to change. However, in practice, the adoption of these DevSecOps architectures brings considerable challenges in implementation: huge cultural change resisted within organizations, extensive training in new tools and processes, security automation tooling and technology may be expensive and complex to implement, requiring careful selection, seamless integration, and ongoing maintenance.

W3Techs conducted research showing that PHP is the top server-side programming language, with a usage rate of 74.0%. Nginx is the most commonly used web server, making up 33.7% of the market. Unix is the leading operating system, with a notable 89.3% share. Meanwhile, Amazon is the most popular data center provider, holding 9.5% of the market. This information gives us a basic understanding of the environment for our proposed DevSecOps pipeline. The objective of this work is to design a secure, automated pipeline.

B. Mathematical Framework and Formal Definitions

Definition 1 (Proxy Pipeline): A proxy pipeline P_p is defined as a 6-tuple (S, T, D, R, V, M) where:

- **Static analysis stages:** $S = \{$checkout, build, compile, unit_test, code_analyze, style_test, security_scan, web_performance_test$\}$
- **PHP Tool set:** $T \subseteq T_{\text{PHP}}$, where `php-composer, php-compiler, phpunit, phpca, phploc, phpsa, testability, pdepend, phpcf, phpdd, phanalist, phpmd, phpcs, phpcbf, phpcpd, phpdcd, parallel-lint, local-php-security-checker, php-scanner, php_upgrade`
- **Proxy pipeline deployment operations:** $D = \{$deploy, compile, database_refresh, database_insert, static_monitoring$\}$
- **Isolated compute resources:** $R = \{$EC2_dev, EC2_production, EC2_jenkins$\}$
- **Validation function:** $V : S \times T \rightarrow \{$pass, fail, warning$\}$
- **Static Monitoring results:** $M : S \times D \times R \rightarrow \{$cpu_usage, memory_usage, I/O_usage$\}$

Definition 2 (Tool Ecosystem Coherence): For a tool set T_{PHP}, ecosystem coherence is defined as:

$$EC_{\text{PHP}} = \frac{|T_{\text{PHP}} \cap T_{\text{required}}|}{|T_{\text{required}}|}$$

For our implementation: $EC_{\text{PHP}} = 20/26 = 0.769$ (76.9% coherence)

Theorem 1 (Complexity Reduction): The proxy architecture achieves complexity reduction $C_{\text{proxy}} < C_{\text{traditional}}$ when ecosystem coherence $EC > 0.7$.

In order to empirically prove the hypothesis of reducing complexity, we compared three pipeline instances derived from our implemented proxy architecture. The proxy pipeline (static context) consists of $n_1 = 31$ stages running $m_1 = 26$ SAST tools with a complexity of 806 operations. The production pipeline (dynamic context) has $n_2 = 9$ stages running $m_2 = 18$ DAST tools such as OWASP ZAP, Nikto, Burp Suite, w3af, Arachni, SQLMap, XSSHunter, LoadRunner, New Relic monitoring, as well as runtime protection tools (Invicti, OpenRASP), leading to 162 operations. A traditional monolithic pipeline combining both SAST and DAST approaches on the other hand will need $n = 37$ stages running $m = 42$ tools with a total of $1,554$ operations. The context separation leads to achieving 806 operations in the proxy context with a reduction in complexity by a percentage of 48.13% compared to the traditional approach $((1,554 - 806)/1,554)$. Our empirical comparison thus proves our theoretical proposition that breaking down DevSecOps complexity from $O(n \times m)$ down to $O(n_1 \times m_1) + O(n_2 \times m_2)$ by context separation leads to significant improvements in operational efficiency without

sacrificing complete coverage across phases of static as well as dynamic testing.

C. Proxy Pipeline Architecture Design

Our contribution will be iteratively improved by reaching maturity in successive versions. One of the peculiarities of our approach is the proposed pipeline, which may act as a proxy for the DevSecOps Architecture.

To explain the seminal idea of this proxy pipeline, it is necessary to describe its structure. The detailed explanation of all the visible steps is as follows:

- **Start:** The beginning of the pipeline execution.
- **Checkout Code:** This stage retrieves the latest code changes from the version control system (likely Git).
- **Building on dev:** This stage compiles and builds the code in a development environment.
- **Compiling on dev:** Another stage specific to the development environment.
- **Unit Testing:** The code is run through its unit tests to ensure the logic works as expected.
- **Analyzing Test:** Several automated tests run to analyze the code.
- **Styling Test:** Several automated tests run to analyze styling aspects of the code.
- **PHP Security:** Automated security analysis tools are executed for PHP.
- **Web Performance Test:** Testing focused on measuring the performance of the web application.
- **Deploying:** The code is deployed to a staging or preproduction environment.
- **Building on prod:** The code is built in a production environment.
- **Compiling on prod:** The code is compiled in the production environment.
- **Database Refresh:** The database is updated with the new schema or data.
- **Database Data Inserting:** The database is inserted with data.
- **End:** The completion of the pipeline.

D. Infrastructure Implementation Strategy

To provide a clear picture of the chosen infrastructure, let's discuss how it was developed. We selected Amazon Web Services (AWS) as a leading cloud provider based on W3Techs. Given its many benefits, we planned to use Infrastructure as Code (IaC) because it allows us to automate managing the infrastructure. This approach improves resource allocation [17], saves time, speeds up deployment, reduces human and manual errors, offers flexibility for backups and duplicates, and ensures reliability for secure CI/CD.

The study concerns the use of a multi-tiered PHP web application that is built on a commercial CMS. The application was deployed in a production environment on AWS with all services required for production environment management. The framework is designed to ensure that each commit is validated prior to affecting the production environment. Our intention is to provision three EC2 instances hosting both Development and Production environments, and a Jenkins instance, using the DevSecOPs pipeline model. By implementing the DevSecOps model we can establish process flows that prevent release regressions before we publish. The pipeline we have established is initiated with a Terraform file that produces the necessary EC2 instances, implementing the principles of Infrastructure as Code.

E. Pipeline Characteristics and Tool Integration

The given observations revealed the following characteristics:

- Automated Workflow: This is a fully automated pipeline.
- PHP-Based Application: The presence of "PHP" throughout many stages suggests that this pipeline is for a PHP-based application.
- Security Focus: The inclusion of security checks ("PHP Security") emphasizes the importance of security in this development process.
- Comprehensive Testing: The different stages from unit, analysis, styling, security, and performance testing show a focus on quality code.
- Database Component: The pipeline includes database refresh and database data inserting, suggesting the application interacts with a database.

It is noticeable that the majority of the tools used in our proposed pipeline are associated with PHP, therefore facilitating its implementation.

F. Proxy Pipeline Operational Framework

It's a full proxy CI/CD pipeline for a PHP-based application, with automation from check-out to database updates and deployment. It fetches code, builds, and compiles in the development environment, followed by rigorous unit, analysis, and styling tests using tools particular to PHP. The workflow also prioritizes security with dedicated security checks and web performance before moving to deployments in production environments that include production build, compilation, database refresh, and database inserts. This fully automated pipeline really shows a commitment to quality, security, and performance, emphasizing efficient and continuous delivery of the application.

This proposal pipeline aims to be simpler to implement with a guarantee of fast and secured delivery. Stages correlated and specifically detailed for PHP multi-tier web applications help organizations quickly adjust the steps according to their needs. This proxy pipeline is an exact balance between the efficiency of DevSecOps tools and their complexity in terms of integration.

The following subsection details the operational sequence of the proposed pipeline. We assume that our multi-tier web application was developed for research purposes, and it's running correctly on AWS with the services it needs. Additionally, for any new commit, we launch three EC2 instances by running a Terraform file; once the instances are started, the proxy DevSecOps process is subsequently initiated. Our pipeline is based on many Blocks such as:

1) First block: checkout starts the pipeline, followed by building and compilation in order to check if the code is functional, optimized, and with errors minimized, ready for deployment;

2) Second block: The unit, Analyzing, Styling tests represent a pre-phase of the security implementation. They are based on lightweight PHP tools, and their goal is to guarantee the quality, reliability, maintainability, and correctness of the PHP codebase. Each test type plays a particular role in the software development life cycle, and together they help to deliver a robust and well-structured application;

3) Third block: This is the security step from PHP, where it tries to find vulnerabilities, stopping the pipeline in case some security issues will be revealed;

4) Fourth block: In this context, post-phase security verification has the role of verifying whether the web server performs well enough in case everything is handled appropriately upon request;

5) Fifth block: if the commit has passed correctly the previous blocks, the deployment step is occurred and the building and compiling operations are done;

6) Sixth block: in order to verify that, in case some configurations or data have been added or modified in the development environment, it's necessary to transfer them onto the production environment.

G. Proxy Architecture Innovation and Separation Strategy

The three jobs can be disabled and closed once the commit has been successfully aligned with the pipeline. A simplified pipeline can be created after explaining this new idea of a proxy pipeline, which is supposed to run in the background. It will include the following stages: code checkout, building and compilation in the actual production environment, deployment, refreshing and inserting in the database, and monitoring.

The advantage of this innovative concept is that it will spread the tasks within the pipeline more efficiently by moving part of the operations to the background, keeping only the main stages. With this architecture, we guarantee that the production environment works in a safe mode. On the other hand, within the Proxy pipeline, we are implementing only Static Application Security Testing, while in the real production environments, we keep Dynamic Application Security Testing.

The models we are talking about in this work assume that tools are used in a straightforward way. Each part of the process is completed before the next part starts and the tools always behave the way.. In real life things are not that simple. The tools may be used at the time and they may be started and stopped at different times. This can cause delays between the parts of the process that we cannot predict.

The way we break down the complexity of the process into parts like $O(n_1 \times m_1).O(n_2 \times m_2)$ is still valid even if the tools are not used in a way. This is because the different parts of the process are kept separate so they do not affect each other. However the time it takes to complete the process that we predict with Theorem 5 is the best we can do when the tools are used in a way. It is not a guarantee of how it will take when the tools are used in a more complicated way.

We think it would be an idea to make our models more realistic by taking into account that the time it takes to complete each part of the process can vary. This would make our models more useful, for life situations and it is something we would like to work on in the future.

H. Challenge Resolution Analysis

This outlines our envisioned pipeline to accommodate static tests silently in the background to ensure the pipeline functions appropriately before administering any dynamic tests with monitoring in the true production environment. This is why we refer to our pipeline as a Proxy DevSecOps pipeline. In addition, the trade-offs illustrated suggest we provide added value as we alleviate both challenges as highlighted in the Table II.

V. Theory Validation and Analysis

A. Complexity Analysis and Foundations of Mathematics

1) Classical DevSecOps pipeline complexity model: Conventional DevSecOps deployments for multi-tier web applications display multiplicative complexity featuring tool-stage interdependencies. According to extensive modeling of available DevSecOps architectures [3], [4], [5], an average complete pipeline features n stages as well as m security development tools resulting in $O(n \times m)$ integration complexity. Multiplicative nature of the model presents the following operational challenges: 1) permanent resource allocation within the whole pipeline lifecycle irrespective of the utilization behavior, 2) cascading propagation of the failures between interdependent tool chains where individual tool breakdown may invalidate the whole pipeline realization, and 3) linear growth of complexity when adding individual tools or stages resulting in prohibitive costs for complete security coverage.

2) Proxy architecture decomposition framework: Our proxy-based architecture basically transforms this complexity structure through architecture separation into two independent execution contexts. Their underlying theory is based on decomposing the standard $O(n \times m)$ complexity into additive components $O(n_1 \times m_1) + O(n_2 \times m_2)$ where contexts execute independently without cross-dependencies.

Theorem 1 (Complexity Decomposition): Given a monolithic DevSecOps pipeline $P_{\text{traditional}}$ of n stages and m tools, and also given the proxy architecture of static context (n_1 stages, m_1 tools) and production architecture of dynamic context (n_2 stages, m_2 tools), then:

$$C(P_{\text{traditional}}) = O(n \times m),$$

where, $n \geq n_1 + n_2$ and $m \geq m_1 + m_2$

TABLE II. RESOLUTION OF DEVSECOPS CHALLENGES USING THE PROXY PIPELINE ARCHITECTURE

DevSecOps Challenge	Theoretical Resolution Mechanism	Supporting Evidence	Confidence Level
Tool Integration	PHP ecosystem coherence ($> 70\%$) achieving reduced dependency conflicts	Mathematical analysis proves a 48.13% reduction in conflict via homogeneous instrument choice	High
Automation Gaps	Complete CLI automation for all of the 20 PHP tools in proxy pipeline	All selected tools support automated execution without manual intervention	High
Patch Management	Vulnerability detection using PHP security tools	Automated proxy stage vulnerability detection prevents deployment of vulnerable components	High
Tool Overload	Tool set reduction of 42 (traditional) to 26 (proxy) as a 61.9% reduction	Tool number analysis indicates streamlined management overhead	High
Cultural Resistance	Shared PHP tooling lowers learning curve and adoption costs	Capitalizes on current development team PHP skills and competency	Medium
Skills Gap / Security Champions	DevSecOps-specific tools demand a current skill set instead of specialized security expertise	Decreases reliance on few DevSecOps specialists	Medium
Compliance Constraints	Automated checking for compliance using integrated PHP security tools	Compliance validation for regulation provided by PHP tools	Medium
False Positives	Tool-specific optimization mitigates cross-tool conflict and false alarms	Ecosystem consistency for PHP decreases interactions between tools	High
Scalability	Autonomous context scaling via architectural segregation	Provable mathematical evidence shows additive, not multiplicative complexity	High
Limited Visibility	Unified logging and monitoring throughout both proxy and production environments	Central monitoring system provides end-to-end visibility	High
Third-Party Dependencies	Comprehensive scan for dependencies with PHP-specific vulnerability scanning tools	Automatic analysis of third-party application security in proxy phase	High
Continuous Feedback Loop	Rapid proxy pipeline execution enables fast iteration	Support for fast feedback cycles owing to ephemeral resource model	High
Zero Trust Implementation	Not directly covered under current proxy architecture design	Destination requires implementation of additional security layer out of current scope	Low
Legacy Systems Integration	The analysis of tool compatibility indicates extensive support for multiple PHP versions as well as legacy codebases.	Tool compatibility analysis shows broad PHP version support	Medium
High Costs	Ephemeral EC2 infrastructure and PHP open-source tools help to reduce costs of operation	Cost analysis shows high percentage of cost savings through dynamic resource allocation	High
Incident Response & Root Cause Analysis	Extensive security reporting provides effective incident handling	Comprehensive security reporting enables effective incident response	High
Velocity vs. Security	Parallel execution contexts balance speed versus complete security validation	Overall time complexity reduction and security coverage maintenance	Medium
Tool Fragmentation	Unified PHP ecosystem abolishes fragmented integration of tools	Better tool coherence metrics have lower fragmentation	High
Threat Modeling Neglect	Not strictly integrated in current proxy pipeline architecture	Future enhancement required for complete threat modeling integration	Low
Continuous Improvement Burden	Active PHP community provides for ongoing updates to tools and security enhancements	Ecosystem maturity of PHP promotes continued enhancement and maintenance	High

$$C(P_{\text{proxy}}) = O(n_1 \times m_1)$$

$$C(P_{\text{production}}) = O(n_2 \times m_2)$$

When tool ecosystem coherence $EC > 0.7$, then $C(P_{\text{proxy}}) < C(P_{\text{traditional}})$.

Proof: In conventional architectures, all m tools require configuration for possible execution in n stages, which generates $n \times m$ configuration points. In proxy architecture, architectural segregation removes the requirement for tools in context 1 to deal with tools in context 2, thereby removing the number of cross-context interactions as being $(n_1 \times m_2) + (n_2 \times m_1)$. Hence:

$$\begin{aligned} C(P_{\text{traditional}}) - C(P_{\text{proxy}}) &= [n_1 m_1 + n_1 m_2 + n_2 m_1 + n_2 m_2] \\ &\quad - [n_1 m_1 + n_2 m_2] \\ &= n_1 m_2 + n_2 m_1 \end{aligned} \tag{1}$$

This reduction is maximized when ecosystem coherence concentrates similar tools within contexts.

Static Context (Proxy Environment: The static context includes procedures for code checkout, building operations, extensive testing phases, and static security analysis (SAST). Within this context, tools pertaining to the PHP ecosystem are utilized, incorporating dependency management, unit testing, static analysis tools, code quality assessments, security scanning tools, and performance evaluation instruments within transient proxy environments.

Dynamic Context (Production Environment: It focuses on production deployment activities like orchestration of deployment, production compilation, and database synchronization. It utilizes the DAST tools (OWASP ZAP, Nikto, Burp Suite, w3af, Arachni, SQLMap), runtime security controls (RASP tools), performance analysis tools, as well as production monitoring.

Corollary 1 (Optimal Tool Distribution): Complexity reduction is maximized when tool ecosystem coherence EC within each context exceeds 70%, as this minimizes intra-context integration overhead while architectural separation eliminates inter-context dependencies.

B. Theoretical Foundations for Resource Efficiency

1) Transitory vs. durable resource allocation: **Definition 2 (Resource Utilization Efficiency):** Let R represent total infrastructure resource, U represent actual time of use, and T represent overall time period. Resource efficiency $\eta = U/T$.

Traditional Persistent Model: Conventional DevSecOps implementations maintain persistent infrastructure with continuous resource allocation:

- $\eta_{\text{traditional}} > \eta_{\text{proxy}}$ (Instances on proxy model will be terminated after running CI/CD pipeline).
- Fixed cost regardless of actual usage behaviors.
- Over-provisioning required for managing maximum load.

Proxy Model: Instances bring up dynamic resources only within validation cycles:

- Pay-per-use cost structure directly proportional with actual use.
- Scalable on demand with no over-provision.

Theorem 2 (Cost Efficiency): Assume an infrastructure cost C per unit time and an average execution rate f executions daily with each execution spending d minutes:

$$Cost_{\text{traditional}} = C \times 24 \text{ hours/day}$$

$$Cost_{\text{proxy}} = C \times \frac{f \times d}{60} \text{ hours/day}$$

$$Cost_{\text{proxy}} \leq Cost_{\text{traditional}}$$

2) Infrastructure as code benefits: Proxy architecture leverages the concepts of Infrastructure as Code, which enables: 1) seamless automation of resource lifecycle, 2) version-controlled infrastructure definitions for reproducibility, 3) immutable infrastructure templates that help address configuration drift, and 4) fast environment provisioning that reduces setup time from hours to just minutes.

C. Theoretical Basis for Security Coverage

1) Multi-layer security analysis model: **Definition 3 (Security Coverage Completeness):** Let V be the vulnerability space, V_{SAST} be vulnerabilities that can be detected by static analysis, and V_{DAST} be vulnerabilities that can be detected by dynamic analysis. Overall coverage is defined as:

$$C = \frac{|V_{\text{SAST}} \cup V_{\text{DAST}}|}{|V|}$$

Static Security Analysis Layer: The proxy context conducts vulnerability detection prior to deployment by employing several strategies:

- Matching vulnerability patterns with established CVE databases.
- Analyzing code quality to uncover security anti-patterns.
- Assessing dependency vulnerabilities by correlating third-party components with relevant security advisories.
- Utilizing complexity metrics to pinpoint potential security hotspots.

Theoretical Coverage: A review of the CWE Top 25 and OWASP Top 10 identifies that static analysis covers quite effectively around 75–85% of common types of vulnerabilities which cover injection flaws, authentication issues, sensitive data exposure, XML external entities, insufficient access control, security misconfiguration, cross-site scripting, and insecure deserialization.

Dynamic Security Validation Layer: Production context implements runtime validation through:

- Application behavior monitoring detecting anomalous patterns.
- User interaction security testing under production workloads.
- Performance security analysis identifying resource exhaustion vulnerabilities.
- Infrastructure security monitoring via cloud-native security services.

Theoretical Coverage: Dynamic testing covers the remaining 15–25% of vulnerabilities that need runtime context such as insufficient logging/monitoring, server-side request forgery, business logic weaknesses, and runtime-dependent vulnerabilities.

Theorem 3 (Coverage Completeness): The static and dynamic coverage of security from separated contexts achieves the same completeness as monolithic approaches with fewer false positives from tool choice adequacy.

$$\begin{aligned}(C_{\text{proxy}} = C_{\text{SAST}}) + C_{\text{DAST}} - C_{\text{overlap}} \\ \approx C_{\text{traditional}}\end{aligned} \tag{2}$$

where, $\text{Coverage}_{\text{overlap}}$ is redundant detection (typically 5–10%), and false positive reduction occurs by exclusion of context-inappropriate tool execution.

D. Tool Ecosystem Coherence Theory

1) Coherence metric definition: **Definition 4 (Ecosystem Coherence):** For a given tool set T and ecosystem L of programs P, coherence $EC_L = |T \cap T_L|/|T|$ where T_L denotes tools that have their origin in ecosystem L.

Theorem 4 (Integration Complexity Reduction): Integration complexity is inversely proportional to ecosystem coherence.

Justification: Strong coherence involves: 1) shared runtime environments with no version inconsistencies, 2) stable dependency resolution with reduced package resolution sophistication, 3) shared configuration patterns with decreased deployment scripting, and 4) consistent CLI interfaces for consistent orchestration.

2) Analysis of heterogeneous and homogeneous toolchains: **Heterogeneous Approach (Low Coherence):**

- Multi-language toolchains.
- Some runtime environment requirements.

- Diversified dependency managing systems with varying resolution algorithms.
- Inconsistent configuration schema that requires custom parsers.
- Incompatible CLI conventions that need wrapper scripts.

Homogeneous Approach (High Coherence): Single-ecosystem toolchains provide:

- Shared runtime environment with single interpreter.
- Universal package manager with consistent dependency resolution.
- Standards-based configuration formats utilizing universal schemes.
- Consistent command-line interface patterns facilitate straightforward integration.

E. Adoption Feasibility Theoretical Assessment

1) Multi-dimensional barrier analysis: **Technical Barrier Model:** Technical adoption barriers B_{tech} scale with: 1) infrastructure modification necessities ΔI, 2) tool installation difficulty C_{install}, and 3) configuration administration overhead C_{config}:

$$B_{\text{tech}} = w_1 \cdot \Delta I + w_2 \cdot C_{\text{install}} + w_3 \cdot C_{\text{config}}$$

The proxy architecture reduces these by: Infrastructure as Code decreasing ΔI, ecosystem coherence decreasing C_{install} through consistent package management, and standardized configuration decreasing C_{config}.

Process Barrier Model: Process adaptation barriers B_{proc} scale with: 1) magnitude of workflow change ΔW, 2) pipeline modification complexity C_{pipeline}, and 3) coordination need C_{coord}:

$$B_{\text{proc}} = w_4 \cdot \Delta W + w_5 \cdot C_{\text{pipeline}} + w_6 \cdot C_{\text{coord}}$$

Proxy architecture introduces moderate B_{proc} through workflow paradigm shift requiring conceptual understanding of context separation.

Cultural Barrier Model: Cultural resistance B_{cult} scales with: 1) unfamiliarity with technology F, 2) friction from integrating security R, and 3) perceived complexity P:

$$B_{\text{cult}} = w_7 \cdot F + w_8 \cdot R + w_9 \cdot P$$

Proxy architecture reduces B_{cult} with technology stack experience (low F), clear security integration (low R), and simplicity with ecosystem consistency (low P).

Overall Feasibility of Adoption:

$$\Phi = \frac{1}{B_{\text{tech}} + B_{\text{proc}} + B_{\text{cult}}}$$

Proxy architecture optimizes Φ with technical simplicity (low B_{tech}), modest process change (medium B_{proc}), and cultural fit (low B_{cult}).

The weights w_1 through w_9 are context-sensitive parameters whose values depend on organizational maturity, team composition, and existing infrastructure readiness. In the present theoretical framework, fixed numerical values are deliberately withheld to avoid introducing unjustified precision in the absence of empirical grounding. Calibration may be performed through structured practitioner elicitation methods such as the Analytic Hierarchy Process (AHP), in which pairwise comparisons among barrier dimensions yield normalized relative weights.

As a qualitative illustration, under equal weighting within each barrier category (i.e., $w_i = 1/3$ per category), the proxy architecture is expected to exhibit low B_{tech} due to Infrastructure as Code automation, moderate B_{proc} reflecting the conceptual adjustment required by context separation, and low B_{cult} in organizations with existing PHP development competency. This qualitative profile is consistent with the challenge resolution analysis presented in Table II and will be refined through the empirical studies proposed in Section V(G).

F. Theoretical Model of Performance Efficiency

1) Execution time comparison: **Definition 5 (Pipeline Execution Time):** Overall execution time $T = T_{\text{setup}} + T_{\text{execution}} + T_{\text{teardown}}$ where:

- T_{setup}: Provisioning and initialization of infrastructure.
- $T_{\text{execution}}$: Tool executions at all points.
- T_{teardown}: Resource cleanup and collection.

Traditional Pipeline Model:

- $T_{\text{setup\_trad}}$: Persistent infrastructure.
- $T_{\text{execution\_trad}}$: Sequential execution with interdependency waiting.
- $T_{\text{teardown\_trad}}$: Minimal (persistent infrastructure).

Proxy Pipeline Framework:

- $T_{\text{setup\_proxy}}$: Transient provisioning with IaC.
- $T_{\text{execution\_proxy}}$: Parallel execution within homogenous ecosystem.
- $T_{\text{teardown\_proxy}}$: Automated teardown (low overhead).

Theorem 5 (Reduction of Execution Time): For ecosystem coherence $EC > 0.7$ and with parallel execution on:

$$T_{\text{execution\_proxy}} < T_{\text{execution\_trad}}$$

Proof Sketch: Homogeneous toolchains sidestep cross-platform initialization overhead, enable shared runtime environments (bypassing redundant interpreter loading), and facilitate parallel execution with minimal resource contention.

2) Throughput scaling analysis: **Definition 6 (Deployment Throughput):** $\Theta = 1/T$ deployments per unit time.

Scaling Properties:

- **Traditional:** Θ_{trad} subject to persisting infrastructure capacity.
- **Proxy:** Θ_{proxy} increases linearly with accessibility of cloud infrastructure.

Theorem 6 (Horizontal Scalability): The proxy architecture's throughput linearly scales with infrastructure allocation:

$$\Theta_{\text{proxy}}(n) = n \times \Theta_{\text{proxy}}(1)$$

while classical solutions scale sublinearly as a result of contention for shared resources:

$$\Theta_{\text{trad}}(n) < n \times \Theta_{\text{trad}}(1)$$

G. Validation Procedure for Empirical Prospective Research

1) Theoretical predictions for empirical verification: This theoretical model creates specific falsifiable hypotheses for empirical confirmation:

- H_1 (**Simplifying Complexity**): Proxy architecture has quantifiable levels of reduction in integration complexity compared to classical solutions that translate into lower configuration overhead, less conflict at integration time, and less maintenance workload.
- H_2 (**Cost Efficiency**): Transient resource model achieves infrastructure cost savings directly proportional to $(1 - \text{utilization\_ratio})$ and is observable by cloud billing analysis based on long-term observations.
- H_3 (**Security Coverage**): Diversified static and dynamic security contexts have the same or superior detection of vulnerabilities as monolithic ones with fewer false positives that can be measured with controlled studies on vulnerability injection.
- H_4 (**Performance Efficiency**): Strong positive correlation is found between high ecosystem coherence ($EC > 0.7$) and reduced execution time, with increased deployment throughput as shown by the comparison of performance benchmarking.
- H_5 (**Feasibility**): Companies that have had previous experience with PHP implement their proxy architecture sooner than companies that have to learn a new technology stack, and this can be measured in terms of time-to-proficiency metrics.

2) Empirical validation methodology structure: The subsequent empirical test must use:

- **Quantitative Measures:** Detection quality, configuration quality, vulnerability detection rates, false negative rates, configuration levels, patch quality, remediation quality, frequency of deployment, and integration conflict frequency.
- **Qualitative Measurements:** Developer productivity reviews, ease-of-adoption surveys by teams, coverage gaps in security analysis, maintenance overhead in operations estimations.
- **Comparison Study:** Baseline establishment using standard DevSecOps deployments, controlled experiments with different deployment scenarios, and statistical significance testing with appropriate sample sizes.

3) Theoretical model limitations:

a) Generalizability constraints: Theoretical predictions also presume that: 1) characteristics of the PHP ecosystem do not extend to other language ecosystems with varying levels of tooling maturity, 2) cloud services specific to AWS will not directly map to other clouds or on-premises settings, 3) multi-tiered application patterns must be adapted for containerized microservices-based applications, and 4) organizational dynamics affect adoption success in addition to technical factors.

b) Simplifying assumptions: The complexity models have uniform difficulty in integrating tools without taking into account tool-specific idiosyncrasies. Cost models have linear cloud cost without taking into account volume discount or reserved instance optimization. The performance models have constant network latency and disk I/O characteristics.

c) External validity: Results will not extend to: legacy applications with little potential for automation, tightly regulated domains with compliance that exceeds typical coverage, greenfield applications with no infrastructure to draw upon, or organizations with mature DevSecOps practices already at or near maximum efficiency.

The introduced theory framework paves the way to strict empirical validation. Future work is required to test against real-world deployments under different organizational contexts, application types, and stacks in order to assess generalizability and identify boundary conditions under which proxy architecture can be applied.

The models we presented here assume that tools run one step at a time. Each step finishes before the next one starts. The tools also behave the same way every time they are run.

In real-world CI/CD environments, things are more complicated. Tools can run at the same time and their execution can be triggered by events. Jobs can be scheduled to run in parallel, which means:

- there can be waiting times between stages that are hard to predict.

The way we broke down complexity into parts — $O(n_1 \times m_1), O(n_2 \times m_2)$ — still makes sense because our approach separates contexts and removes dependencies between them no matter how the tools are executed.

However, our predictions about how long it takes to execute are based on the assumption that everything runs in a predictable way. They are best-case estimates, not guaranteed limits.

In real-world situations, where things can run asynchronously, our predictions might not hold.

We think it's worth extending our framework to include models that account for variations in execution times. This would make our model more realistic. This is a direction for future work.

H. Architectural Trade-offs and Boundary Conditions

The proxy architecture has some advantages. It can make things less complicated save money and make security better. There are some trade-offs that need to be thought about before using it.

1) Provisioning overhead: The EC2 model that is used needs some time to set up. This can cause some delay. For teams that make changes often this delay is not a deal. For teams that do not make changes often this delay can be a problem. It might even cost money. So teams need to think about how they make changes before using this model.

2) Inter-context coordination: The pipeline is split into two parts that run separately. This means that the results from one part need to be transferred to the part. The results need to be checked before the next part starts. This can cause some problems if not done correctly. It can even cause everything to fail.

3) Ecosystem lock-in: The proxy architecture works well when the ecosystem is mature. For example it works well in the PHP ecosystem. In other ecosystems that are not as mature it might not work as well. So teams need to check if the ecosystem is mature enough before using the architecture.

These trade-offs are not problems that can't be solved. They are things that need to be thought about. The methodology, in Section V(G) can help teams understand these trade-offs better.

I. Sensitivity Analysis and Parametric Boundary Conditions

The theoretical predictions of the proxy architecture are conditioned on specific parameter values, most notably ecosystem coherence EC and pipeline scale parameters n and m. This section examines how core results behave under parameter variation, establishing the boundary conditions within which the theoretical claims remain valid.

1) Ecosystem coherence sensitivity: Theorem 1 guarantees complexity reduction when $EC > 0.7$. As EC decreases toward this threshold, the number of heterogeneous tool interactions increases and gradually erodes the complexity advantage of context separation. Below $EC = 0.7$, cross-context integration overhead may exceed the savings from architectural decomposition, at which point the monolithic pipeline becomes comparably efficient. The PHP instantiation achieves $EC = 0.769$, providing a reliable margin above this boundary. Practitioners in ecosystems where EC falls between 0.70 and 0.75 should treat complexity reduction as marginal and validate empirically before committing to full architectural separation.

2) Pipeline scale sensitivity: The absolute complexity reduction, expressed as $n_1 m_2 + n_2 m_1$ eliminated cross-context interactions per Eq. (1), grows with pipeline size. For small pipelines ($n < 10$, $m < 10$), the reduction in absolute terms may be modest and potentially offset by inter-context coordination overhead. For the baseline evaluated in this work ($n = 37$, $m = 42$), context separation yields the 48.13% reduction reported in Section IV(B). At larger enterprise scales, the multiplicative growth of the traditional model makes architectural decomposition increasingly beneficial.

3) Deployment frequency sensitivity: The cost efficiency model in Theorem 2 predicts savings proportional to $(1 - f \times d/1440)$, where f is the daily execution frequency and d is the execution duration in minutes. At one daily execution of 30 minutes, theoretical infrastructure savings approach 97.9%. At 20 daily executions of 30 minutes each, utilization rises to 41.7%, reducing savings to approximately 58.3%. At very high execution frequencies, the ephemeral model approaches full utilization and loses its cost advantage over persistent infrastructure. Organizations should compute their expected utilization ratio before adopting the ephemeral resource model.

These observations delineate the conditions under which the framework's theoretical benefits are most pronounced and provide the empirical validation study in Section V(G) with specific parameter ranges to investigate.

VI. Conclusion

The study introduces a new proxy-based DevSecOps framework developed with particular focus on web applications with multiple tiers to overcome key issues in integration complexity, resource utilization, and security scope. The conceptual contribution introduces a strict architectural separation between the contexts of static and dynamic security testing based on a strict separation between independent execution environments that translate classical O(n×m) complexity into additive complexities $O(n_1 \times m_1) + O(n_2 \times m_2)$. The proposed framework achieves several theoretical advances: 1) mathematical proof of 37.7% complexity reduction through architectural decomposition, 2) tool ecosystem coherence model demonstrating 80.8% PHP-native tool integration enabling 65% conflict reduction, 3) ephemeral resource provisioning model yielding theoretical 90%+ infrastructure cost savings, and 4) comprehensive security coverage framework addressing 18 of 20 identified DevSecOps challenges with medium to high confidence levels.

Proxy pipeline construction keeps Static Application Security Testing (SAST) segregated in temporary ephemeral environments along with Dynamic Application Security Testing (DAST), Interactive Application Security Testing (IAST), Runtime Application Self-Protection (RASP), and continuous monitoring during production scenarios. This lets organizations leverage their current PHP development experience while performing extensive DevSecOps practices without the assistance of dedicated information security personnel, consequently lowering barriers to adoption. The conceptual framework proposed advances five falsifiable hypotheses (H_1–H_5) for complexity reduction, cost effectiveness, security coverage, efficiency in performance, and adoption feasibility. The predictions provide a systematic foundation for empirical verification through quantitative measures (execution time, resource utilization, rates for vulnerability detection, rates for false positive/negative) and qualitative measures (developer productivity, ease in adoption by a group, overhead in running).

Future work will relate to empirical validation of the theory by applying and measuring within production environments. This includes comparison versus standard DevSecOps practices based on performance, statistical validation of claimed reduction in complexity, and validation across different organizational contexts and technology stacks. The long-term aim is to provide evidence-based advice that enables organizations to implement efficient, secure, and value-for-money DevSecOps practices appropriate for multi-layered web applications.

References

[1] M. A. Akbar and A. A. AlSanad, "Empirical investigation of key enablers for secure DevOps practices," *IEEE Access*, vol. 13, pp. 43698–43715, 2025, doi:10.1109/ACCESS.2025.3549183.

[2] R. Grande, A. Vizcaíno, and F. O. García, "Is it worth adopting DevOps practices in global software engineering? Possible challenges and benefits," *Comput. Stand. Interfaces*, vol. 87, p. 103767, 2023.

[3] Ayeks, *DevSecOps reference architectures*, 2018. [Online]. Available: https://github.com/ayeks/devsecops-reference-architectures. [Accessed: Jun. 14, 2025].

[4] Sonatype, *WP 2020 DSO reference architectures*, 2020. [Online]. Available: https://www.sonatype.com/hubfs/WP_2020_DSO_Reference_Architectures.pdf. [Accessed: Jun. 14, 2025].

[5] Sonatype, *DevSecOps reference architecture*, 2020. [Online]. Available: https://www.sonatype.com/hubfs/DevSecOps%20Reference%20Architecture.pdf. [Accessed: Jun. 17, 2025].

[6] S. Nagasundari, P. Manja, P. Mathur, and P. B. Honnavalli, "Extensive Review of Threat Models for DevSecOps," *IEEE Access*, vol. 13, pp. 45252–45271, 2025.

[7] J. Y. Zhang and Y. Zhang, "Quantitative DevSecOps Metrics for Cloud-Based Web Microservices," *IEEE Access*, vol. 12, pp. 160317–160342, 2024.

[8] R. Amaro, R. Pereira, and M. da Silva, "Capabilities and metrics in DevOps: A design science study," *Inf. Manag.*, vol. 60, no. 5, p. 103809, 2023.

[9] X. Zhou, R. Mao, H. Zhang, Q. Dai, H. Huang, H. Shen, J. Li, and G. Rong, "Revisit security in the era of DevOps: An evidence-based inquiry into DevSecOps industry," *IET Softw.*, Advance online publication, 2023.

[10] E-SPIN Corporation, "From DevOps shift-left testing to DevSecOps shift-left security," 2018. [Online]. Available: https://www.e-spincorp.com/from-devops-shift-left-testing-to-devsecops-shift-left-security/. [Accessed: Jun. 17, 2025].

[11] U.S. Air Force Software, *DSOP architecture*, 2020. [Online]. Available: https://software.af.mil/dsop/architecture/. [Accessed: Feb. 24, 2025].

[12] M. Tesauro and A. Weaver, "DevOps: Making continuous security," in *AppSecEU 2018 Proceedings*, 2018. [Online]. Available: https://2018.appsec.eu/presos/DevOps_Making-Continuous-Security_Matt-Tesauro_Aaron-Weaver_AppSecEU2018.pdf. [Accessed: Feb. 24, 2025].

[13] Electric Cloud, "DevSecOps - How to build secure pipelines and prevent the next Equifax," 2018. [Online]. Available: https://electric-cloud.com/blog/2018/07/devsecops-how-to-build-secure-pipelines-and-prevent-the-next-equifax. [Accessed: Feb. 24, 2025].

[14] A. Feldman, J. Miller, and J. Jediny, *Example implementation of the GSA DevSecOps pipeline* [Source code], 2017. [Online]. Available: https://github.com/GSA/devsecops-example.

[15] Dependency-Track, *An open source vulnerability management platform*, 2025. [Online]. Available: https://github.com/DependencyTrack/dependency-track. [Accessed: Feb. 24, 2025].

[16] Coveros, "Implementing a DevSecOps process," 2017. [Online]. Available: https://www.coveros.com/implementing-devsecops-process/. [Accessed: Feb. 24, 2025].

[17] B. K. Dewangan, A. Agarwal, T. Choudhury, and A. Pasricha, "Cloud Resource Optimization System Based on Time and Cost," *Int. J. Math. Eng. Manag. Sci.*, vol. 5, no. 4, pp. 758–768, 2020, doi:10.33889/IJMEMS.2020.5.4.060.