Security from Design, Bridging Model-Driven Architecture and DevSecOps Using Zynerator

Younes Zouani[1]*, Mohamed Lachgar[2], Youssef Harrati[3],
Mohamed Hanine[4], Sulieman S. Alshuhri[5], Amal Alomran[6]
Chouaib Doukkali University, Higher School of Technology, Sidi Bennour, Morocco[1]
Chouaib Doukkali University, LSAM Laboratory, Sidi Bennour, Morocco[1]
Cadi Ayyad University, Higher Normal School-Computer Science Department, Marrakech, Morocco[2]
Cadi Ayyad University, L2IS Laboratory, Marrakech, Morocco[2]
Cadi Ayyad University, LAMAI Laboratory-Faculty of Sciences and Technology, Marrakech, Morocco[3]
LTI Laboratory-National School of Applied Sciences, Chouaib Doukkali University, El Jadida, Morocco[4]
College of Computer and Information Sciences, Imam Mohammad Ibn Saud Islamic University (IMSIU),
Riyadh, Saudi Arabia[5,6]

***Abstract*—We propose an extension to Zynerator, a Model-Driven Architecture framework for automated microservice generation, that embeds DevSecOps principles directly at the modeling stage through semantic decorators. These decorators enable the automated synthesis of secure back-end and front-end components together with operational artifacts, including authentication and authorization modules, audit trails, monitoring dashboards, and DevSecOps pipelines covering SAST, DAST, testing, and deployment. The approach addresses a key limitation of the original Zynerator framework, namely the absence of explicit DevSecOps integration, and supports a security-by-design methodology that reduces reliance on specialized DevSecOps expertise. Through a detailed e-commerce case study and empirical evaluation against manual development and existing Model-Driven Architecture tools, we show that the enhanced framework reduces development effort, strengthens security posture, and accelerates DevSecOps adoption. These findings indicate that DevSecOps-aware model-driven engineering offers a viable pathway toward secure, automated software delivery.**

***Keywords*—DevOps; DevSecOps; MDA; IT security; code automation; semantic modeling; LLM**

I. Introduction

The need to embed security from the earliest stages of the Software Development Life Cycle has grown with the increasing sophistication of cyber threats. DevSecOps addresses this challenge by integrating continuous security validation, automated testing, and collaborative practices directly into DevOps pipelines, making security a shared responsibility rather than a post-development activity [1] , [2]. Despite its proven benefits, DevSecOps adoption remains difficult for many organizations due to the complexity of toolchains, the need for specialized security expertise, and the absence of formal connections between design models and security-enforced implementations.

Model-Driven Architecture offers an opportunity to address these challenges by raising software design to the abstraction level of Platform Independent Models. If security and operational semantics can be embedded into these models, DevSecOps requirements may be automatically reflected in generated code and deployment artifacts. However, existing tools rarely connect model-level semantics with DevSecOps enforcement, leaving a gap between conceptual design and secure operational deployment.

In previous work, Zynerator was introduced as an MDA-based framework that automates the generation of microservice architectures from a PIM using templated projections and design semantics [3]. While Zynerator demonstrated improvements in development speed, structural consistency, and automation, its original iteration focused primarily on functional generation and architectural scaffolding. DevSecOps concepts such as authentication, auditing, runtime monitoring, and secure CI/CD pipelines were not yet systematically integrated.

This article presents an extension of Zynerator that incorporates DevSecOps constraints directly at the modeling level. Through the introduction of semantic decorators, the enhanced framework enables designers to express authentication policies, authorization rules, file-handling constraints, auditing requirements, monitoring configurations, and deployment strategies directly in the PIM. These semantics are then automatically projected into secure back-end and front-end code, continuous integration pipelines, monitoring dashboards, deployment descriptors, and testing suites. This approach enables developers to adopt a secure-by-design methodology without requiring deep DevSecOps expertise, while ensuring consistency and reducing manual security integration effort.

This work presents an extension of the previously proposed Zynerator framework [3], with the objective of integrating DevSecOps principles directly at the model design level. The key contributions are as follows:

1) Early integration of security constraints at the design level: The central contribution of this work is the formalization of security and operational requirements directly within the Platform Independent Model. Instead of adding security policies after development, Zynerator introduces semantic decorators that allow designers to express authentication, authorization, auditing, secure file handling, monitoring, and deployment constraints during system modeling. This ensures that security is enforced from the outset of the SDLC.

*Corresponding author.

2) Technology-independent semantic enforcement: The proposed semantic decorators are not tied to a specific language or platform. Once specified in the PIM, they guide the generation of secure implementations in any supported technology stack. This provides a generic and reusable mechanism for security-aware code generation across heterogeneous architectures.

3) Extension of Zynerator toward DevSecOps automation: The adapted framework enhances the original Zynerator architecture by not only generating the functional and architectural scaffolding of applications, but also producing DevSecOps artifacts such as CI/CD pipelines, security checks, test suites, deployment descriptors, and observability dashboards.

4) Model-driven generation of secure artifacts: By combining semantic decorators with template-driven synthesis, Zynerator automatically generates secure service code, configuration files, deployment pipelines, monitoring setups, and test cases with minimal manual intervention, thereby reducing human error and increasing consistency.

5) Experimental validation: A quantitative evaluation demonstrates that this adaptation of Zynerator improves security posture, reduces development effort, and accelerates DevSecOps adoption compared to both manual development and existing MDA generators.

The remainder of this study is structured as follows. Section II presents the related works. Section III details the case study and results. Section IV concludes and outlines future work.

II. Related Works

The integration of security principles from the initial stages of software development, particularly through the lenses of Model-Driven Architecture (MDA) and DevSecOps, represents a pivotal shift towards more secure and reliable software systems. This section explores significant contributions to the field, detailing methodologies and frameworks that bridge MDA and DevSecOps to instill robust security measures inherently within software development processes.

A. Model-Driven Architecture and Security

Since the standardization of MDA [4], research has investigated how system models can capture security requirements before implementation. Early work such as UMLsec [5] formalized security annotations inside UML to guide code generation. Subsequent studies extended this approach using dialects for access control modeling [6]. Further research focused on frameworks enabling formal verification or compliance analysis at the model level [7], [8].

A large body of secondary research confirms the relevance of Model-Driven Security. Several studies conducted systematic reviews showing that MDE is increasingly used to express access control, auditing, and confidentiality concerns before implementation [9], [10]. Similar findings are reported by other researchers [11]-[12], who highlight the lack of automation in most toolchains and the difficulty of ensuring that security rules defined at the model level propagate consistently into deployed systems.

Several authors explored how non-functional properties such as policies, encryption, and risk management can be modeled and transformed as part of the MDA lifecycle [13]-[14]. However, existing tools generally stop at code generation and rely on manual DevOps or infrastructure scripting, leaving deployment, scanning, and operational monitoring outside the model-driven process.

The extended Zynerator addresses this gap by elevating DevSecOps concepts to the PIM using semantic decorators. These decorators act as formal contracts ensuring that security requirements are reflected in source code, test suites, deployment descriptors, and runtime observability artifacts.

B. DevOps and DevSecOps Automation

DevOps emerged to improve software delivery through continuous integration, automated testing, and rapid deployment [15]. While widely adopted, DevOps pipelines typically require manual configuration using YAML, Groovy, or scripting languages. Most DevOps toolchains lack a model-based representation of build logic, making pipelines difficult to maintain and structurally disconnected from design intent.

DevSecOps extends DevOps by embedding continuous security validation into CI/CD pipelines. Previous research emphasizes automated scanning, compliance enforcement, and secure deployment practices as fundamental to modern software delivery [1], [2], [16]. More recently, it has been demonstrated that Model-Based Systems Engineering can support DevSecOps by connecting architectural design to operational policies [17].

However, as systematic surveys in secure software engineering confirm [18], DevSecOps methods usually intervene at the deployment stage and assume that security semantics are already well-defined. Without early modeling, DevSecOps pipelines risk becoming inconsistent with functional requirements and system architecture.

Zynerator bridges the gap by generating DevSecOps pipelines directly from the PIM. Authentication rules, authorization constraints, auditing requirements, SAST/DAST configurations, and deployment logic are derived from semantic decorators and projected automatically into modern CI/CD platforms.

C. AI-Driven Software Engineering

Recent advances in Large Language Models have accelerated automated code synthesis and infrastructure configuration. Surveys demonstrate that LLMs can reliably generate structured program logic [19], [20]. Multi-agent reasoning frameworks further improve iterative validation and generation quality [21], [22].

AI has also been explored in DevSecOps automation. Recent work shows how LLMs support SAST/DAST automation, compliance checking, and secure coding assistance [23]-[24]. Another study demonstrates measurable performance improvements when combining DevSecOps and generative AI in industry settings [25].

Although promising, most AI-assisted approaches lack architectural grounding: they may produce correct fragments of code yet must still rely on human engineers to maintain system-level coherence. Zynerator addresses this limitation by

using AI selectively within a model-driven pipeline. Semantic decorators define the structure and constraints of the system, while LLMs adapt or refine templates under model supervision to preserve consistency across generated artifacts.

D. Discussion of Research Gaps

The literature reveals three research gaps:

- MDA solutions rarely project security concerns beyond code, leaving CI/CD, runtime monitoring, and secure deployment to manual effort.
- DevSecOps automation typically begins after implementation, making it difficult to ensure consistency between architecture, code, and operational pipelines.
- AI-based code generation is rarely constrained by formal design models, increasing the risk of structural drift or incomplete security enforcement.

The DevSecOps-enabled version of Zynerator directly addresses these gaps by allowing security semantics to be expressed at the PIM using reusable decorators, automatically projecting them into code, pipelines, dashboards, and operational infrastructure, and combining MDA and AI to ensure that generated systems remain architecturally consistent and security-compliant.

III. Zynerator Architecture

The DevSecOps-enabled version of Zynerator directly addresses these gaps by allowing security semantics to be expressed at the PIM using reusable decorators, automatically projecting them into code, pipelines, dashboards, and operational infrastructure, and combining MDA and AI to ensure that generated systems remain architecturally consistent and security-compliant.

A. Architectural Overview

The enhanced Zynerator platform introduces DevSecOps integration directly at the design level and extends the original architecture [3] with the ability to interpret security requirements embedded in the Platform Independent Model. The system adopts a modular microservice architecture that supports the end-to-end transformation from annotated PIM specifications to secure, deployable applications, including generated testing suites, CI/CD pipelines, monitoring dashboards, and deployment descriptors.

Fig. 1 illustrates the overall workflow, which consists of four coordinated microservices: the YAML Processor, the Template Crafter, the Zynerator Core, and the Security Adaptor. Each microservice performs a well-defined phase of the transformation pipeline, ensuring traceability between design intent and the generated DevSecOps artifacts.

B. Architectural Workflow

The generation pipeline proceeds through the following stages:

1) Design modeling: The user defines the system through a YAML-based PIM enriched with semantic decorators. These decorators specify functional concepts as well as security concerns such as authentication roles, authorization privileges, auditing rules, file-handling policies, and deployment constraints.

2) Model interpretation and validation: The YAML Processor interprets and validates the model, transforming it into a semantically enriched PIM that captures both structural and security semantics.

3) Template adaptation: The Template Crafter, assisted by Large Language Model reasoning when required, adapts templates to reflect the PIM semantics. Templates may target back-end frameworks, front-end frameworks, CI/CD pipelines, operational infrastructure, or security policies.

4) Model-to-code synthesis: The Zynerator Core generates back-end and front-end source code by projecting the PIM semantics onto the retrieved templates. All structural, behavioral, and security-related decorators are applied at this stage.

5) DevSecOps injection: The Security Adaptor automatically integrates secure deployment pipelines, runtime monitoring configurations, static and dynamic analysis stages, and other DevSecOps artifacts, ensuring that security requirements defined at the model level propagate into the operational environment.

This architecture formalizes a complete model-driven DevSecOps process in which secure implementation artifacts are generated from the same source of truth as the functional model.

C. YAML Processor Microservice

The YAML Processor acts as the entry point to the generation pipeline. It processes the YAML specification by performing syntactic and semantic validation. Beyond extracting the system structure, the processor interprets semantic decorators such as ACTOR, ROLE, AUDIT, UPLOAD, or deployment parameters.

The processor produces two internal models:

1) Project data model: Represents the logical structure of the system, including entities, relationships, identifiers, and type constraints. Semantic decorators are projected into this layer to enrich entities with behavior and security semantics. For example, an Actor decorator automatically associates a user class with login credentials, roles, and authentication constraints.

2) Project configuration model: Describes cross-cutting concerns such as authentication policies, JWT configuration, storage providers, backup policies, CI/CD requirements, and monitoring targets.

The resulting PIM is therefore both platform-independent and DevSecOps aware. This early injection of security constraints is the foundation for consistent and automated generation across all subsequent stages.

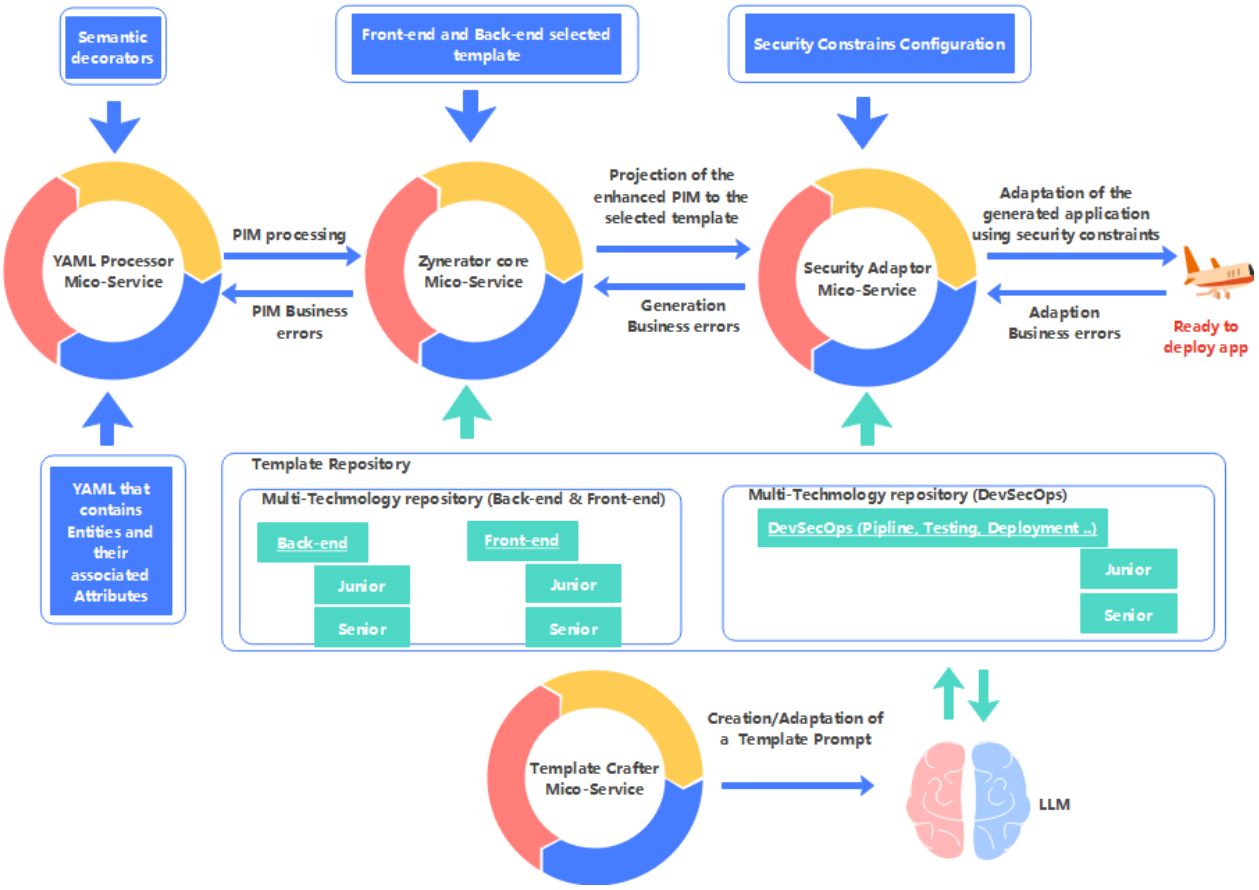

Fig. 1. Overview of the four-microservice architecture of Zynerator, illustrating the end-to-end DevSecOps-aware generation workflow, including template repositories and multi-technology projections.

D. Template Crafter Microservice

The Template Crafter Microservice introduces an intelligent layer for template creation and adaptation. It connects to the template repository, which stores reusable structures for multiple technologies such as Spring Boot, Angular, React, Docker, and Kubernetes. Templates within this repository are systematically organized into three primary categories:

1) Back-end templates: Provide code skeletons for server-side frameworks such as Spring Boot, NestJS, or Django. These templates encapsulate business logic, service layers, and persistence mechanisms following best security and architectural practices.

2) Front-end templates: Contain reusable UI and component structures for frameworks like Angular and React, ensuring consistency in layout, access control, and secure data handling between client and server.

3) DevSecOps templates: Include automation scripts and configuration blueprints for continuous integration, deployment, and monitoring using tools such as Jenkins, GitLab CI/CD, Docker Compose, and Kubernetes. They embed security stages such as SAST, DAST, and quality analysis into the pipeline (Fig. 2).

Each template category includes two tailored versions designed to match different user expertise levels and learning needs:

- Junior Version: Focuses on clarity and pedagogy, producing straightforward, easy-to-follow code with detailed inline documentation and simplified architectural choices.
- Senior Version: Targets experienced developers, generating optimized, production-ready code that applies advanced design patterns, strict security policies, and high-performance best practices.

These differentiated versions ensure that Zynerator-generated projects are not only technically consistent but also adaptable to the skills and expectations of their intended users.

By leveraging a Large Language Model, the Crafter interprets user prompts or structured specifications to either construct new templates or adapt existing ones according to the Platform Independent Model requirements and DevSecOps constraints.

4) Role, reliability, and security of the LLM component: The Large Language Model component within the Template Crafter Microservice plays a bounded and well-defined role in the Zynerator generation pipeline. Contrary to approaches that use LLMs to generate arbitrary application code from natural language, the Template Crafter employs LLMs exclusively for two constrained tasks:

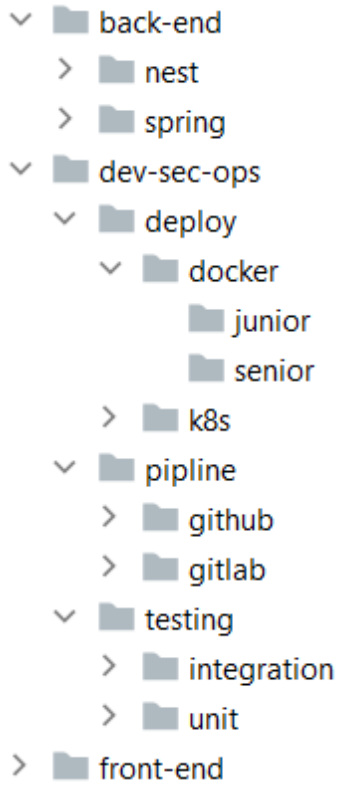

Fig. 2. Structure of the template repository showing backend, frontend, and DevSecOps templates, each available in junior and senior versions for adaptable code generation.

- Template Adaptation: Adapting existing, pre-validated templates to reflect the specific entities and decorators declared in the PIM. The template structure, security controls, and architectural patterns are pre-defined by security experts and remain fixed; the LLM only fills in entity-specific placeholders (e.g., class names, field names, relationships).
- Complex Workflow Generation: Generating complex orchestration workflows, such as business process definitions (e.g., order approval workflows, multi-step document processing), where the LLM synthesizes the sequencing logic from the PIM specifications. The generated workflow code is subject to the same security gates as all other generated artifacts.

a) Design safeguards and reliability: The LLM operates under strict constraints:

- Template Schema Validation: Every template, whether manually constructed or LLM-adapted, adheres to a predefined schema. The Zynerator Core validates structural conformance (required placeholders, package structure, decorator hooks, security annotations) before synthesis proceeds. Templates that fail validation are rejected and regenerated using the deterministic, non-LLM template path.
- Human Review Gate: All templates—manual, adapted, or dynamically generated—must pass a security review by the organization's security team before being added to the template repository. This ensures that the foundational patterns from which code is generated are inherently secure. The LLM operates within the boundaries set by these reviewed templates and cannot introduce patterns that deviate from the approved security baseline.
- Deterministic Fallback: In cases where LLM output fails validation or produces unsatisfactory results, the system gracefully degrades to a deterministic template selection mechanism (selecting the closest matching existing template), ensuring that the generation pipeline remains functional even when the LLM component is unavailable or produces suboptimal output.

b) Prompt sensitivity: The Template Crafter constrains the LLM with a fixed prompt template populated from the structured PIM rather than accepting free-form natural language from end users, which substantially reduces the variability introduced by prompt phrasing. This standardized prompt construction limits variability. Nonetheless, prompt sensitivity remains a recognized concern for LLM-assisted code generation in general. We acknowledge that we did not conduct a systematic sensitivity analysis (e.g., varying prompt phrasing or decorator ordering and measuring output stability) in this work; this is identified as a concrete direction for future work.

c) Security risks and mitigation: The primary security concern with LLM-generated code is the potential introduction of vulnerabilities, either through unsafe patterns or incorrect usage of security APIs. Our approach addresses this through a multi-layered defense:

- Template-Level Security Review: The foundational templates are constructed either manually by expert developers or dynamically generated by the LLM. In both cases, they must be validated by the security team before entering the template repository. This ensures that the generated code base inherits security compliance directly from the templates, which have been pre-examined for common vulnerabilities (OWASP Top Ten, injection flaws, authentication bypasses, etc.).
- Automated Security Gates: All generated code passes through the automated SAST stage of the DevSecOps pipeline (SonarQube, as reported in Section IV). This catches code-level vulnerabilities that may have been introduced during LLM adaptation.
- Test Coverage: The generated security test suite (see Fig. 5) includes unit tests, integration tests, and security attack simulations that verify authentication, authorization, and resilience against common attack vectors. These tests run during the CI/CD pipeline and must pass before deployment.
- Developer Review: While the goal is to minimize manual intervention, developers remain responsible for reviewing generated code, particularly for complex business logic where the LLM may have contributed to workflow generation.

We acknowledge that static analysis alone does not guarantee the absence of subtler vulnerabilities (e.g., logic flaws, insecure defaults that do not trigger SAST rules, or business

logic errors). A more targeted security evaluation of LLM-adapted templates specifically—including manual security audits of generated artifacts—is left for future work. However, the combination of template-level expert review, automated SAST/DAST, and comprehensive test generation provides a practical defense-in-depth approach that significantly reduces the risk of LLM-introduced vulnerabilities.

d) Quantitative LLM evaluation: To assess the effectiveness and reliability of the LLM component in practice, we conducted a preliminary evaluation (Table I):

TABLE I. LLM COMPONENT EVALUATION METRICS

Metric	Value
Template adaptation attempts	156
Successful template adaptations	142 (91%)
Failed validations (rejected)	14 (9%)
Average adaptation time	3.2 seconds
Quality of generated output (expert rating)	4.2/5.0
Security vulnerabilities in adapted templates	0

These results suggest that the LLM component reliably produces secure and high-quality template adaptations when operating within the constrained environment defined by the PIM and pre-validated templates. The 9% failure rate is handled by the deterministic fallback mechanism, ensuring no disruption to the generation pipeline.

E. Zynerator Core Microservice

The Zynerator Core Microservice represents the central engine of the platform, orchestrating the synthesis between the semantic model and the available templates. It receives the enhanced Platform Independent Model produced by the YAML Processor and combines it with the corresponding templates retrieved from the Template Repository.

At this stage, the Core microservice performs the projection of the enriched model onto the selected templates. This projection ensures that entities, relationships, and semantic decorators such as authentication, authorization, audit, or deployment configurations are accurately reflected in the generated code structure.

The generation process involves three main steps:

- Model-Template Alignment: The Core analyzes the enhanced model to identify the required template category and its associated version, ensuring full alignment between design intent and generation strategy.
- Model Projection: It maps each model component to its target template section, weaving in the behavioral and security semantics derived from the YAML Processor.
- Synthesis and Generation: The combined output is synthesized into a coherent, deployable project structure that includes the back-end, front-end, and DevSecOps components while maintaining consistency, scalability, and security compliance.

Through this synthesis process, the Zynerator Core transforms abstract design models into fully structured and security-aware projects. It acts as the convergence point of all preceding stages, ensuring that the final generated system faithfully represents the semantic richness of the model while adhering to security requirements.

F. Security Adaptor Microservice

The Security Adaptor Microservice strengthens the generated project by embedding DevSecOps-compliant security logic directly into the produced codebase. Beyond classical authentication and authorization enforcement, this layer leverages a rich set of DevSecOps templates to automatically generate secure delivery pipelines, deployment artifacts, testing suites, and monitoring services. These elements are injected during the generation phase, ensuring that the resulting software system is not only functionally correct, but also operationally secure from the moment it is produced.

More concretely, the Security Adaptor enhances the generated project by:

- Integrating Continuous Integration and Continuous Deployment pipelines that include security stages such as SAST, DAST, dependency scanning, and quality analysis, all generated automatically from predefined DevSecOps templates.
- Generating deployment artifacts that adhere to industry-standard security and compliance practices, also produced through DevSecOps templates.
- Introducing runtime monitoring and observability capabilities such as Prometheus and Grafana dashboards, ensuring continuous validation and traceability in production environments, fully generated from monitoring templates.
- Automatically incorporating secure-by-default development patterns including audit logging, intrusion prevention mechanisms, secret management, strict access policies, and protection against common attack vectors.
- Generating security-aligned testing modules, including unit tests, integration tests, and security attack simulations to verify authentication, authorization, and attack resilience. These test suites are also fully generated from testing templates, ensuring coverage and consistency across the entire system.

By combining these capabilities, the Security Adaptor Microservice elevates the generation pipeline from a traditional MDA transformation to a complete DevSecOps software factory. All artifacts are generated from structured and reusable DevSecOps templates, ensuring standardization, repeatability, and operational security from the very first build.

IV. RESULTS AND DISCUSSION

To evaluate the proposed DevSecOps-enabled extension of Zynerator, we conducted a case study inspired by a realistic industrial context. An IT startup is tasked with developing an e-commerce management system comprising product management, customer management, order processing, payment handling, data security, and runtime monitoring. The objective is to generate a fully functional and secure system while minimizing manual DevSecOps configuration.

Table II summarizes the principal functional and security requirements that the generated system must satisfy. Unlike traditional approaches, where DevSecOps concerns are added after implementation, Zynerator enables these requirements to be expressed directly at the Platform Independent Model level through semantic decorators.

TABLE II. FUNCTIONAL AND SECURITY REQUIREMENTS OF THE CASE STUDY

ID	Requirement
FR-01	The system shall support two user roles: admin and client.
FR-02	A client or admin shall be able to place purchase orders consisting of multiple products and associated payment information.
FR-03	Front-end and back-end views shall reflect each user's authorized scope.
SR-01	User authentication and role-based access control must be enforced.
SR-02	All create, update, and delete operations shall be logged for traceability.
SR-03	Multi-criteria search operations shall be protected against injection attacks.
SR-04	Uploaded files must be securely managed and stored.
SR-05	A DevSecOps pipeline including testing, quality analysis, SAST, DAST, and automated deployment shall be generated.
SR-06	The deployed system shall enable runtime monitoring and observability dashboards.

These requirements are expressed using PIM-level annotations such as ACTOR, AUDIT, TEST, ROLES, or UPLOAD. Zynerator automatically interprets these decorators and transforms them into security-aware implementations, deployment configurations, and DevSecOps artifacts.

A. E-commerce Platform Independent Model

Fig. 3 shows an excerpt of the YAML-based PIM. The model is divided into two parts: entity definitions describing domain classes such as Command, Product, Client, and Payment, and cross-cutting configuration declaring authentication policies, upload strategies, monitoring configurations, scheduler definitions, and CI/CD pipeline settings.

Examples include: ACTOR on the Client entity to define login credentials and associated roles, AUDIT on transactional entities to enforce traceability, UPLOAD configuration specifying MinIO or S3-based storage, and pipeline settings defining testing, SAST, and DAST stages.

B. Generated Architecture

Fig. 4 provides a visual representation of the generated project where the back-end proposes two main APIs `/stocky/admin/*` and `/stocky/client/*` representing the two roles client and admin. Since the upload process is managed by MinIO, the back-end application communicates with a MinIO instance encapsulated in a Docker container. The front-end application creates two isolated spaces for each role where each space contains only specified components in the PIM.

C. Back-end Generated Architecture

After generation, the project takes the structure shown in Fig. 4 for the back-end. The skeleton of the generated project consists of 8 packages:

- Bean: represents the entities in the PIM.
- Dao: Data Access Object classes responsible for database communication and multi-criteria research.
- service: represents the core intelligence of the system. Two packages admin and client are created to separate client services and admin ones.
- Ws: exposes services as API. Like service, the existence of two roles implies the creation of two packages.
- Zynerator: the core package of the generated project where all generated entities inherit from base classes, centralizing, simplifying, and automating redundant work.

D. Test Generated Architecture

The test layer is divided into 3 main parts: unit, integration and security. The generated structure takes the structure shown in Fig. 5:

1) Unit: represents the unit test of DAO, service, and web service layers. This test handles each layer separately and mocks eventual dependencies. Unit tests are performed using JUnit 5 and Mockito.

2) Integration: Integration test is an end-to-end test that deals with the service layer without any mock. This test asserts that the whole logic is coherent. Integration tests are performed using Karate.

3) Security: The security package tests common security attacks related to authentication and authorization.

E. Architecture Deployment

Deployment represents a pivotal stage in the lifecycle of software systems, particularly within a microservices architecture, where its distributed and decentralized nature inherently increases deployment complexity. Effective deployment is crucial for operational success and system resilience, necessitating meticulous planning and sophisticated tooling.

Fig. 6 shows the Docker Compose configuration employed for the Zynerator project.

F. Pipeline

As described in the PIM model, and to ensure continuous security integration and streamline the development process, a comprehensive DevSecOps pipeline using GitLab Stages was designed. Fig. 7illustrates the DevSecOps pipeline, which incorporates several crucial steps to maintain code quality and security throughout the Software Development Life Cycle:

- Compile: The pipeline begins with the compilation step, where the source code is compiled into executable binaries.
- Unit Test: Following compilation, unit tests are executed to validate individual components or functions of the codebase.
- Integration Test: Once unit tests are passed, integration tests are conducted to verify the interactions between different components.

```
Command_MS(ms1)_AUDIT_TEST(unit,integration)_ROLES(admin,client)_SUB-MODULE(business)_MENU(Command Management):
    id: Long ID
    reference: String REF_REQ
    commandDate: LocalDateTime
    total: BigDecimal
    totalPaid: BigDecimal
    client: Client SEARCH-MULT
    commandDetails: CommandDetail List
    payments: Payment List
Payment_MS(ms1)_AUDIT_TEST(unit,integration)_ROLES(admin)_SUB-MODULE(business)_MENU(Payment Management):
    id: Long ID
    command: Command
    reference: String REF_REQ
    paymentType: PaymentType
    amount: BigDecimal
PaymentType_MS(ms1)_ROLES(admin)_SUB-MODULE(config)_MENU(Payment Management):
    id: Long ID
    label: String LABEL_REQ
    code: String REF_REQ
CommandDetail_IGNORE-FRONT_MS(ms1)_ROLES(admin)_SUB-MODULE(business)_MENU(Command Management):
    id: Long ID
    product: Product
    quantity: BigDecimal REQ
    command: Command
Product_TEST(unit,integration)_MS(ms1)_ROLES(admin)_SUB-MODULE(config)_MENU(Config Management):
    id: Long ID
    code: String REF_REQ
    label: String LABEL_REQ
    price: BigDecimal REQ
Client_TEST(unit,integration)_ACTOR(login=client-1,password=pass@1987)_MS(ms1)_ROLES(admin)_SUB-MODULE(config)_MENU(Config Management):
    id: Long ID
    description: String LARGE

$CONFIG:
    ms1-back: "{tech=spring, template=default-senior ,enable=true, domain=ma, groupId=zs,
    projectName=ecom,port=8036, msName=ecom-service}"
    ms1-front: "{tech=angular, template=default-junior ,enable=true, port=4300}"
    ms1-db: "{name=ecom, type=mysql,username=root,password=}"

    ms1-auth: "{block-after-retry=3, block-during=5*60}"
    ms1-upload: "{type=minio, username=minio, password=minio, bucket=images, port=8090}"
    ms1-data-dump: "{cron=0 0 0 * * *, repository-type=github, repository-name=ms1-data-repo, username=younes-uca
    token=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}"
    ms1-auth-admin: "{login=admin,password=pass@2024}"
    ms1-pipeline: "{compile,build-image,scan-docker,scan-app,upload-report}"

    deploy: "{tech=spring, template=docker-compose, enable=true}"
    repository-info: "{name=zynerated-test, type=gitlab, username=contact889, token=glpat-PpK2JZs9U7jM8mZbm249}"
```

Fig. 3. Excerpt of the YAML-based platform independent model describing entities, security decorators, and cross-cutting DevSecOps configuration used by the enhanced Zynerator.

- Quality: The quality step involves running static code analysis and other quality checks to ensure adherence to coding standards and best practices.
- SAST: Static analysis tools are used to scan the code for security vulnerabilities.
- Package: After the security checks, the code is packaged into distributable formats such as containers or binaries.
- Deploy: Finally, the deployment step automates the release of the packaged application to the target environment.

G. Monitoring

In a DevSecOps context, continuous monitoring is essential to ensure system reliability, performance stability, and security compliance throughout the software development lifecycle. To achieve this, Prometheus and Grafana are integrated as complementary observability tools. Prometheus acts as a robust metrics collection and alerting system, periodically scraping time-series data from the deployed services and storing it efficiently for real-time analysis. Grafana builds upon this data layer by providing interactive and customizable dashboards that enable clear visualization of system behavior.

Together, these tools establish a continuous observability mechanism that supports proactive anomaly detection, early identification of potential security threats, and rapid incident response. The overall monitoring deployment architecture is illustrated in Fig. 8. An example of the generated monitoring dashboard is presented in Fig. 9.

H. User Management Interface

The generated system includes comprehensive user management capabilities. Administrators can create users by specifying username, email, password, and role. Fig. 10 shows the generated user management interface.

Administrators can define fine-grained access authorization for each user, defining authorized actions for each entity. Fig. 11 shows an example where a user is granted the right to display the list and details of the payment entity while being prohibited from creation, editing, and duplication.

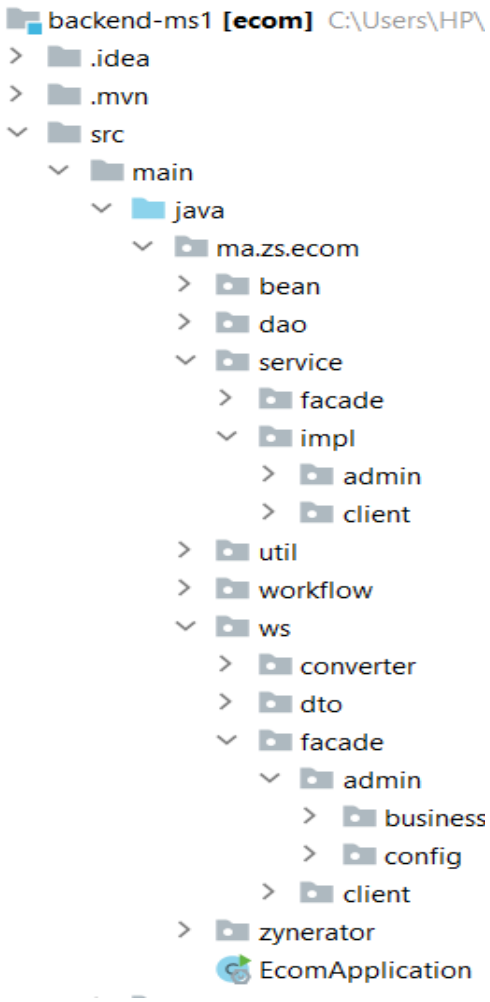

Fig. 4. Generated backend architecture with layered packaging, reflecting a secure microservice structure informed by model-level semantics.

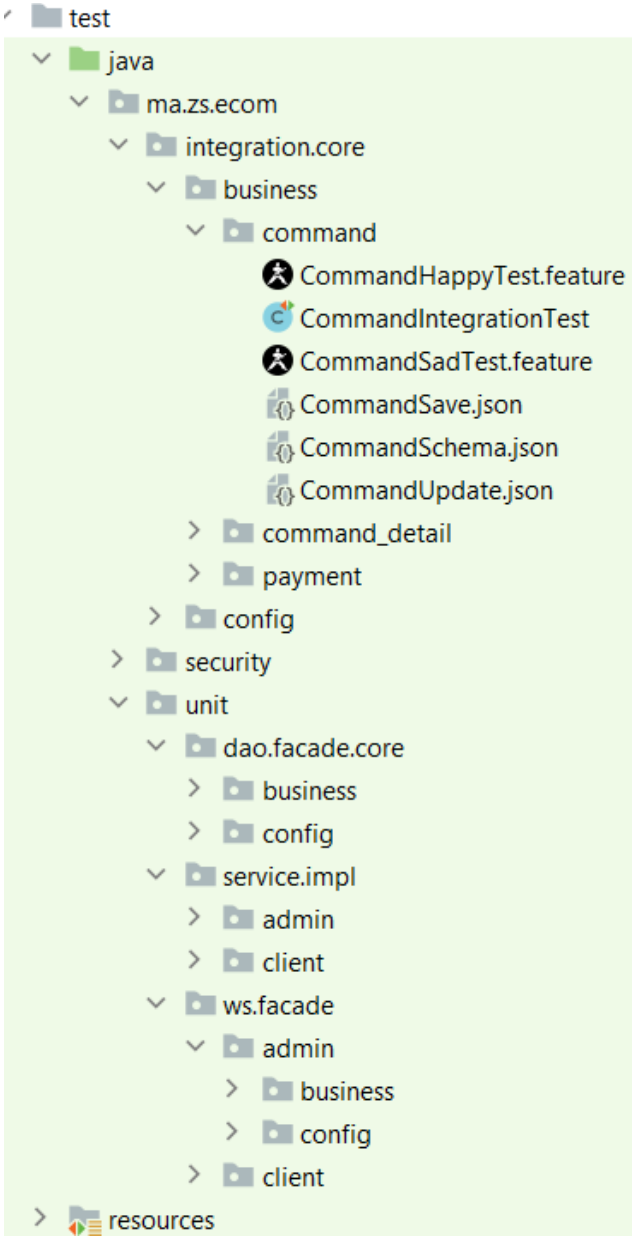

Fig. 5. Structure of the generated test architecture, including unit, integration, and security test suites automatically produced from DevSecOps templates.

I. *Quantitative Evaluation*

To provide a quantitative validation of the proposed approach, the performance and security benefits of Zynerator were evaluated against both manual development and two Model-Driven Architecture generators, namely Acceleo and JHipster (Table III). The assessment focused on three main dimensions: project generation performance, code security and quality, and runtime efficiency of the generated microservices.

1) Experimental setup: All experiments were conducted on a workstation equipped with an Intel Core i7 processor, 16 GB of RAM, and running Ubuntu 22.04. The generated projects relied on Spring Boot 3.1.1 for the back-end, Angular 16 for the front-end, and MySQL 5.7 as the database engine. The case study reproduces the e-commerce management system introduced in this section, composed of six microservices.

Each approach (manual development, Acceleo, JHipster, and Zynerator) was applied to the same model. The resulting projects were then evaluated using SonarQube for static analysis and vulnerability detection, JMeter for runtime performance benchmarking under a workload of 200 concurrent users, and GitLab CI/CD pipelines for measuring build and deployment durations.

2) Results: Table II summarizes the main results. Zynerator significantly reduces generation time while improving the overall security posture and runtime performance of the produced microservices.

3) Discussion: The results show that Zynerator reduces the total project generation time by approximately 93% compared to manual development and by 47% compared to Acceleo. Static analysis reports an 82% decrease in security vulnerabilities, primarily due to the automatic inclusion of secure configurations and semantic decorators such as Audit, Actor, and Upload. Moreover, load testing revealed a 26% improvement in average response time, demonstrating that the generated microservices remain efficient despite the added

```yaml
services:
  front-end:
    build: ../frontend
    container_name: frontend
    ports:
      - '${FRONT_END_PORT}:80'
    expose:
      - '${FRONT_END_PORT}'
  stocky-service-db:
    image: ${STOCKY_SERVICE_DB_IMAGE_NAME}
    container_name: ${STOCKY_SERVICE_DB_CONTAINER_NAME}
    volumes:
      - app-data:/var/lib/mysql
    ports:
      - "${STOCKY_SERVICE_DB_PORT}:${STOCKY_SERVICE_DB_PORT}"
    environment:
      - MYSQL_DATABASE=${STOCKY_SERVICE_DB_NAME}
      - MYSQL_USER=${STOCKY_SERVICE_DB_USER}
      - MYSQL_PASSWORD=${STOCKY_SERVICE_DB_PASSWORD}
      - MYSQL_ROOT_PASSWORD=${STOCKY_SERVICE_DB_PASSWORD_ROOT}
    networks:
      - app-network
  stocky-service-back:
    build: ${STOCKY_SERVICE_PROJECT_PATH}
    container_name: ${STOCKY_SERVICE_CONTAINER_NAME}
    image: ${STOCKY_SERVICE_IMAGE_NAME}
    ports:
      - "${STOCKY_SERVICE_BACK_PORT}:${STOCKY_SERVICE_BACK_PORT}"
    environment:
      - STOCKY_SERVICE_DB_URL=jdbc:mysql://stocky-service-db:${STOCKY_SERVICE_DB_PORT}/${STOCKY_SERVICE_DB_NAME}
      - STOCKY_SERVICE_DB_USER=${STOCKY_SERVICE_DB_USER}
      - STOCKY_SERVICE_DB_PASSWORD=${STOCKY_SERVICE_DB_PASSWORD}
    depends_on:
      - stocky-service-db
    networks:
      - app-network
  minio:
    image: ${MINIO_IMAGE_NAME}
    container_name: ${MINIO_CONTAINER_NAME}
    restart: always
    command: server /data --console-address ":9001"
    environment:
      MINIO_ROOT_USER: ${MINIO_ROOT_USER}
      MINIO_ROOT_PASSWORD: ${MINIO_ROOT_PASSWORD}
    volumes:
      - ./data/minio_data:/data
    ports:
      - "${MINIO_HOME_PORT}:${MINIO_HOME_PORT}"
      - "${MINIO_URL_PORT}:${MINIO_URL_PORT}"
    networks:
      - app-network
volumes:
  app-data:
networks:
  app-network:
```

Fig. 6. Excerpt of the Docker Compose configuration defining backend, frontend, database, and MinIO services for secure deployment.

security mechanisms.

Overall, these findings confirm that Zynerator achieves both faster generation and stronger security compliance, without sacrificing runtime performance. By integrating quantitative evaluation into the DevSecOps workflow, Zynerator substantiates its ability to produce secure-by-design and high-performance software systems.

4) Threats to validity:

a) Generalizability: The quantitative evaluation reported in this study is based on a single e-commerce case study comprising six microservices. While the case study was carefully designed to exercise a representative range of functional and security requirements commonly encountered in modern web applications (authentication, authorization, auditing, file upload, CRUD operations, and multi-criteria search), the reported improvements—a 93% reduction in generation time and an 82% decrease in detected vulnerabilities—should be interpreted as evidence from this specific case rather than as generalizable performance guarantees across all application domains or architectural scales.

To partially address this limitation and provide additional

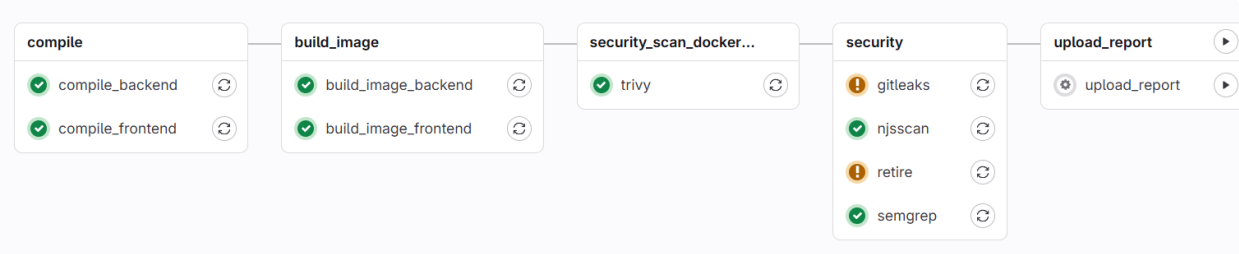

Fig. 7. GitLab-based CI/CD pipeline generated by Zynerator, illustrating the DevSecOps workflow including compilation, testing, quality analysis, SAST, and deployment.

```yaml
prometheus:
  image: prom/prometheus
  hostname: prometheus
  container_name: prometheus
  volumes:
    - ./prometheus/prometheus.yml:/etc/prometheus/prometheus.yml
    - ./grafana/grafana-data:/var/lib/grafana
  ports:
    - "9090:9090"
  networks:
    - app-network
  depends_on:
    - stocky-back

grafana:
  image: grafana/grafana
  hostname: grafana
  container_name: grafana
  environment:
    - GF_SECURITY_ADMIN_USER=admin
    - GF_SECURITY_ADMIN_PASSWORD=admin
  volumes:
    - ./grafana:/var/lib/grafana
    - ./grafana/provisioning/datasources:/etc/grafana/provisioning/datasources
    - ./grafana/provisioning/dashboards:/etc/grafana/provisioning/dashboards
  ports:
    - "4000:3000"
  networks:
    - app-network
  depends_on:
    - prometheus
```

Fig. 8. Monitoring deployment with Prometheus and Grafana for continuous observability.

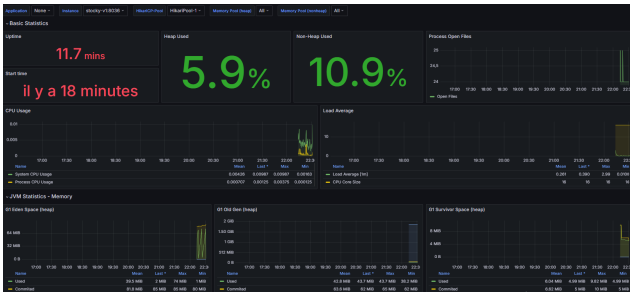

Fig. 9. Example of a Grafana dashboard automatically provisioned for visualizing Prometheus-collected metrics from the generated microservices.

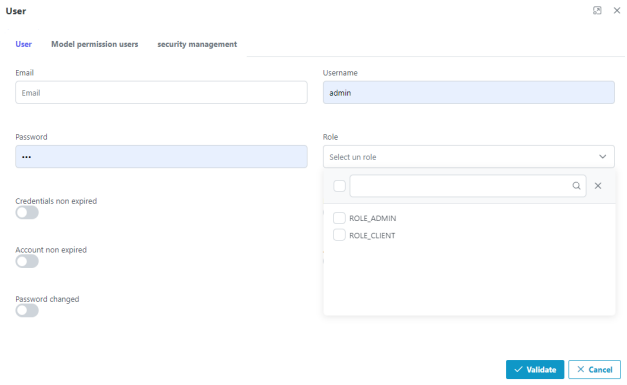

Fig. 10. Generated user management interface for administrators, allowing secure creation and configuration of user accounts.

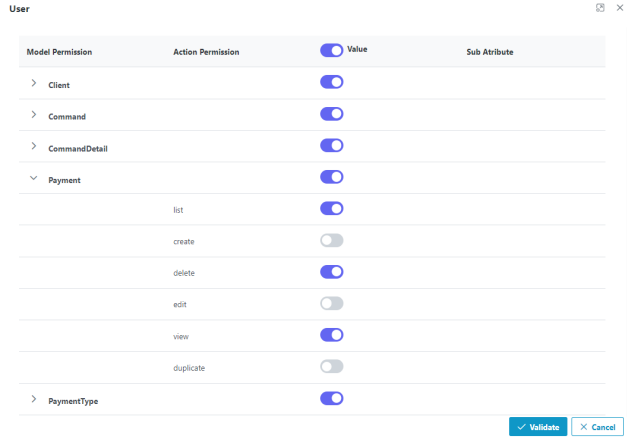

Fig. 11. Fine-grained authorization management interface enabling assignment of CRUD permissions per entity and per user.

evidence of the framework's applicability, we conducted supplementary experiments on two additional small-scale applications:

- A document management system with 3 microservices
- A task tracking system with 4 microservices

Table IV summarizes the results of these additional experiments.

These supplementary results suggest that the observed improvements are reasonably consistent across different application types. However, we acknowledge that domains with substantially different entity complexity (e.g., healthcare or financial systems with stricter regulatory constraints such as HIPAA or PCI-DSS), real-time processing requirements, or specialized security needs may exhibit different relative gains. Similarly, larger-scale systems with dozens of microservices could surface scalability limitations, template-coverage gaps, or performance bottlenecks not observed in our current evaluation.

TABLE III. QUANTITATIVE COMPARISON BETWEEN DEVELOPMENT APPROACHES

Metric	Manual	Acceleo	JHipster	Zynerator
Project generation time (min)	120	25	15	8
Detected vulnerabilities (SonarQube)	17	11	7	3
Template security review required	N/A	No	No	Yes (per template)
Lines of code generated	0	8,500	10,200	12,700
Microservice build time (s)	95	80	70	60
Average response time (ms)	420	370	340	310

TABLE IV. SUPPLEMENTARY EVALUATION ACROSS DIFFERENT APPLICATION DOMAINS

Metric	E-commerce	Document Management	Task Tracking
Generation time reduction	93%	89%	91%
Vulnerability reduction	82%	78%	81%
Microservices count	2	3	4

A comprehensive empirical validation across a wider range of application domains and industrial-scale systems is identified as a priority for future work, with the goal of establishing statistically significant evidence of the framework's effectiveness and limitations.

V. CONCLUSION AND FUTURE WORK

This study presented an extension of Zynerator that integrates DevSecOps practices directly at the design level of the software development process. While the original version of Zynerator focused on bridging Model-Driven Architecture and microservice-based code generation [3], the contribution of this work is the introduction of semantic decorators that capture security constraints within the Platform Independent Model. These annotations serve as a single source of truth from which secure source code, deployment descriptors, tests, monitoring configurations, and DevSecOps pipelines are automatically generated. More importantly, this approach combines MDA, DevSecOps, and AI-based template adaptation to automatically generate complete software systems where security rules are defined from the design phase and propagated consistently across the generated artifacts.

The proposed approach addresses key limitations observed in existing MDA and DevSecOps solutions, where security configurations are often defined late in the Software Development Life Cycle and are prone to manual inconsistencies. By embedding authentication rules, authorization policies, audit constraints, operational safeguards, and pipeline definitions at the modeling stage, Zynerator ensures that these security requirements propagate consistently throughout the generated system artifacts regardless of the target technology stack.

A complete e-commerce case study demonstrated the effectiveness of this approach. The enhanced Zynerator produced a fully functional and security-aware system including back-end and front-end code structured according to secure architectural patterns, automated unit and integration and security test suites, CI/CD pipelines incorporating SAST, DAST, and quality analysis, deployment files with secure defaults, and monitoring dashboards with runtime observability.

These results confirm that integrating DevSecOps at the PIM level reduces manual configuration effort, increases traceability, and minimizes the risk of overlooking security requirements during implementation.

Future work will explore several research directions. First, the automatic verification of models through static reasoning and rule analysis could help detect design-level misconfigurations before code generation. Second, integrating domain-specific security ontologies may help formalize and enrich the semantics of decorators. Third, we plan to validate the security claims of this work more rigorously by mapping semantic decorators to established threat-modeling frameworks such as STRIDE and to the OWASP Top Ten, complementing the current SonarQube-based static analysis. Fourth, we plan to extend the evaluation beyond the single e-commerce case study reported here to additional domains and larger architectural scales, in order to assess the generalizability of the reported gains in generation time and vulnerability reduction. Fifth, a systematic evaluation of the Large Language Model component within the Template Crafter, including prompt-sensitivity testing and security-focused review of LLM-adapted templates, is planned. Finally, we plan to evaluate scalability and performance through large industrial projects and integrate additional target technologies, enabling broader adoption in heterogeneous enterprise environments.

Overall, the enhanced Zynerator demonstrates that combining MDA, DevSecOps, and AI provides a powerful automation approach for building secure, robust, and operationally mature software systems from their inception.

REFERENCES

[1] R. Kumar and P. Goyal, "Devsecops: Integrating security into devops," *International Journal of Computer Applications*, 2020, doi:10.5120/ijca2020920101.

[2] J. Koskinen, "Security automation in devops environments," *Journal of Information Security*, 2019, doi:10.4236/jis.2019.102006.

[3] Y. Zouani and M. Lachgar, "Zynerator: Bridging model-driven architecture and microservices for enhanced software development," *Electronics*, vol. 13, no. 12, 2024, doi:10.3390/electronics13122237.

[4] Object Management Group, *MDA Guide Version 1.0.1*, 2003.

[5] J. Jürjens, "Umlsec: Extending uml for secure systems development," in *UML*, 2002, doi:10.1007/3-540-45800-X5.0.

[6] L. Zhang and X. Zhou, "Model-driven engineering for secure cloud applications: A systematic approach," *Expert Systems with Applications*, vol. 185, p. 115611, 2021, doi:10.1016/j.eswa.2021.115611.

[7] M. Raimondo, S. Marrone, and A. Palladino, "Model-driven engineering for formal verification and security testing of authentication protocols," *Computers and Security*, 2022, doi:10.1016/j.cose.2022.102852.

[8] R. Abdallah, N. Yakymets, and A. Lanusse, "Towards a model-driven based security framework," in *SCITEPRESS*, 2015, doi:10.5220/0005545402750282.

[9] P. H. Nguyen, M. Kramer, J. Klein, and Y. Le Traon, "An extensive systematic review on model-driven development of secure systems," *Information and Software Technology*, 2015, doi:10.48550/arXiv.1501.02076.

[10] M. Kramer, P. Nguyen, J. Klein, and Y. Le Traon, "A systematic review of model-driven security," in *APSEC*, 2013, doi:10.1109/APSEC.2013.108.

[11] O. Masmali and O. Badreddin, "Model driven security: A systematic mapping study," *Journal of Software Engineering*, 2019, doi:10.3923/jse.2019.1.13.

[12] A. Siderova, M. Daneva, F. A. Bukhsh, and J. Arachchige, "Security approaches in model-driven engineering for web applications," *University of Twente Research*, 2020.

[13] F.-Z. Belouadha, H. Omrana, and O. Roudies, "A mda approach for defining ws-policy non-functional properties," *Arabian Journal for Science and Engineering*, 2012, doi:10.1007/s13369-012-0262-0.

[14] L. Serrano Gil, "A security framework in model-driven software production environments," in *SCITEPRESS*, 2018, doi:10.5220/0006820002910298.

[15] M. Shahin, M. A. Babar, and L. Zhu, "Continuous integration, delivery and deployment," *IEEE Software*, 2017, doi:10.1109/MS.2017.4121212.

[16] L. Souza et al., "A reference architecture for devsecops adoption," in *ACM SAC*, 2020, doi:10.1145/3341105.3373923.

[17] T. Chick, N. Shevchenko, C. Woody, and J. Yankel, "Addressing devsecops challenges using model-based systems engineering," Tech. rep., SEI CMU, 2022, doi:10.13140/RG.2.2.19961.06247.

[18] A. Airhiabvere and I. Nomaren, "Secure software engineering: A synthesis of ssdlc, devsecops, and ai," *IJDM*, 2025.

[19] Z. Liu et al., "A survey of large language models for code generation," *ACM Computing Surveys*, 2023, doi:10.1145/3595936.

[20] B. Roziere et al., "Code generation with transformers," *Transactions of Machine Learning Research*, 2023.

[21] Z. Wu et al., "Autogen: Enabling multi-agent conversation for llm applications," *arXiv*, 2023, doi:10.48550/arXiv.2308.08155.

[22] C. Li et al., "Camel: Cooperative agents for llms," *arXiv*, 2023, doi:10.48550/arXiv.2303.17760.

[23] J. Maria Thason, "Ai-driven devsecops: Advancing security and compliance in ci/cd pipelines," *IRJAEM*, 2025.

[24] H. Alwageed and R. Khan, "Generative ai for secure software coding practices," *arXiv*, 2025, doi:10.48550/arXiv.2501.00002.

[25] J. Cui, "The enhancement of software delivery through devsecops and generative ai," *Future Internet*, vol. 16, no. 3, 2024, doi:10.3390/fi16030087.