Politecnico di Torino

Politecnico di Torino

Master's Degree in Computer Engineering

A.a. 2024/2025

Graduation Session October 2025

APIOps and DevSecOps

Automating API Deployment and Security Scanning in Cloud Environments

Referees:
Andrea Atzeni
Leonardo Frioli

Candidate:
Alekos Interrante Bonadia

Abstract

This thesis presents the development and implementation of an APIOps and DevSecOps platform for automated API release and security analysis in cloud environments. The research was conducted at Cloud9 Reply to address a key enterprise challenge: securing and managing APIs effectively.

The initial research provides a comprehensive review of API architectures, platform maturity models, and core DevOps principles. It includes a comparative analysis of leading API management solutions (including Kong API Gateway, Azure API Management, Tyk API Gateway, and Layer7 API Management Platform) based on critical features such as security, scalability, and cloud integration, justifying the selection of the optimal technology stack for the enterprise context.

The primary contribution is an integrated APIOps platform built on Microsoft Azure. The solution leverages Azure API Management, Azure DevOps with self-hosted agents, Azure Key Vault for secure secret management, and immutable Azure Blob Storage (WORM) to ensure forensic readiness.

The core of the platform is structured around six specialized pipelines.

The Extractor pipeline initiates a GitOps workflow: after extracting configurations from Azure API Management, it automatically opens a Pull Request. This mechanism ensures that every change is discussed, reviewed, and explicitly approved by the team before being integrated into the official codebase, guaranteeing a collaborative quality control process.

Next, the Static Security Analysis pipeline inspects the APIs' source code and their specifications using tools such as SonarQube, Spectral, and TruffleHog.

The Publisher pipeline manages the controlled promotion of APIs between environments such as development and production, ensuring secure handling of API secrets to prevent data exposure.

Post-deployment, the Reachability and Rollback pipeline performs health checks and, in case of failure, triggers an automatic rollback based on the latest Git tag. This tag represents a known, stable version of the overall API configuration within APIM, thus ensuring service continuity.

The Dynamic Security Analysis pipeline conducts DAST (Dynamic Application Security Testing) using the APIsec tool and evaluates the results through an advanced security gate, which automatically blocks non-compliant releases. This gate is designed for high flexibility, supporting four distinct operational modes. This allows failure criteria to be configured based on specific rules, such as CVSS score thresholds, severity levels, critical vulnerability types, or a combination thereof, adapting the risk assessment to different contexts.

Finally, the Master Orchestrator coordinates the entire process, offering crucial operational flexibility. It can operate in a blocking mode, which waits for the outcome of the security scans to proceed with a controlled release only upon success, or in an

informational (non-blocking) mode, which initiates the scans to provide rapid feedback to developers without interrupting their workflow.

The role of ITSM systems in coordinating remediation workflows was also analyzed during the research. Instead of a full ITSM integration, a more direct feedback loop was implemented. To streamline the remediation workflow, an email-based ticketing system is configured directly within SonarQube and APIsec, which automatically alerts the designated teams responsible for review when vulnerabilities are detected.

To ensure forensic readiness, the platform implements an automated system that collects, organizes, and loads all logs and artifacts produced during the DevSecOps workflow. All security reports, logs, and pipeline outputs are digitally signed, timestamped, and preserved in immutable WORM storage, providing tamper-evident evidence and a verifiable chain of custody.

After deployment, the platform was assessed against relevant security standards, regulations, directives, and frameworks such as ISO/IEC 27001:2022, PCI DSS, NIST SP 800-92, HIPAA, NIS2, and GDPR, demonstrating its alignment with key compliance requirements.

Results show that the integrated APIOps approach improves API security, reducing critical vulnerabilities detected in production, maintains development speed and operational efficiency, and provides detailed audit trails to support compliance. Long-term analysis demonstrates substantial improvements in enterprise API management capabilities.

This work contributes a practical framework for applying modern APIOps and DevSecOps principles to enterprise API management. By effectively shifting security checks earlier into the development lifecycle (shift-left), the proposed solution reduces long-term costs and streamlines processes. The platform's modular design provides a flexible foundation that can evolve with emerging technologies, ensuring long-term viability and adaptability.

Table of Contents

List of Figures

Chapter 1

Introduction

The management and deployment of *Application Programming Interfaces* (APIs) in modern software systems has become increasingly complex, especially in cloud-native and enterprise environments where reliability, security, and compliance are essential. APIs are no longer peripheral components but represent the backbone of digital transformation strategies, enabling interoperability, modularity, and service composition.

In recent years, the number of publicly available APIs has increased significantly. According to Postman's[1] *State of the API Report 2024*, the platform community surpassed **35 million developers**, up from 25 million in 2023 [1], confirming the rapid expansion of the API ecosystem and its growing relevance in software development.

To address the increasing complexity and scale of API ecosystems, organizations are adopting specialized operational paradigms designed to improve automation, security, and consistency. One such approach is *APIOps*, which focuses specifically on automating the entire API lifecycle, from design and testing to deployment, versioning, and governance.

In parallel, the broader paradigm of *DevSecOps* has emerged as an evolution of DevOps, with the primary goal of integrating security controls and validations throughout all stages of the software development and delivery pipeline. Although not limited to APIs, DevSecOps complements APIOps by ensuring that security is embedded by design within automated API workflows.

Together, APIOps and DevSecOps provide a cohesive framework for managing APIs at scale: they enable consistent deployment practices across environments, reduce manual errors, and address security requirements proactively and systematically.

This industry trend is directly reflected in the business context at *Cloud9 Reply*. The company is experiencing a growing demand from corporate clients for DevSecOps platforms that can be seamlessly integrated into their existing workflows. This demand

[1]Postman is a unified platform for designing, building, testing, and scaling APIs collaboratively.

has increased recently, highlighting a practical need in the market. Such market signals provide a strong practical motivation for this thesis, grounding its theoretical framework in real-world engineering challenges.

This work introduces an automated solution for managing the release of APIs in a cloud environment, fully integrated with an API-Management system. A central element of the solution is the inclusion of security scans, executed through DevSecOps tools capable of detecting vulnerabilities and policy violations in a continuous and repeatable manner.

1.1 Thesis Objectives

The overarching objective of this thesis is to demonstrate that a fully automated, security-aware API release pipeline can be designed, implemented, and operated within a modern enterprise cloud environment.

To support this goal, the work is structured around the following specific objectives:

1. **Platform Architecture and Integration**: Build a complete solution that integrates Azure API Management with Azure DevOps to enable seamless API workflows. The architecture will be modular, allowing integration with emerging technologies and advanced security tools.

2. **Security Integration and Automation**: Embed static and dynamic security checks directly in the deployment pipeline, ensuring that vulnerabilities are identified earlier, reducing maintenance costs, and enabling continuous security monitoring throughout the API lifecycle.

3. **Operational Excellence and Reliability**: Automate deployment processes to support multi-environment API releases, provide rollback mechanisms and reachability verification to ensure resilience, and create audit trails that support compliance needs.

4. **Compliance and Standards Alignment**: Assess the platform's alignment with major cybersecurity standards and regulations, including ISO/IEC 27001:2022, PCI DSS, NIST SP 800-92, HIPAA, NIS2 Directive, and GDPR, showing how the solution addresses key security controls and regulatory requirements.

5. **Performance and Scalability Validation**: Evaluate the impact of the automated approach on development speed and operational efficiency, verifying that it improves security without compromising performance or scalability.

Achieving these objectives will demonstrate how APIOps and DevSecOps practices can be effectively applied to enterprise API management.

1.2 Structure of the Thesis

The thesis is organized into six chapters, each focusing on a key part of the platform design and implementation:

Chapter 2: Background and Key Concepts sets the foundation of this work. It explains the basics of APIs, their main architectural styles, and how API platforms evolve through different maturity levels. The chapter also covers core DevOps principles, the role of IT Service Management (ITSM), and the adoption of DevSecOps in cloud environments. Finally, it compares API Management platforms such as *Kong, Azure API Management, Tyk, and Layer7*.

Chapter 3: Project Development and Implementation describes the design and implementation of the APIOps platform. It begins with an overview of DevSecOps tools and explains the design choices made for the Azure-based platform. It details the architecture, execution environment, and Azure DevOps project structure. The chapter then presents the pipeline orchestration, focusing on six key components:

1. Extractor Pipeline,
2. Static Security Analysis Pipeline,
3. Publisher Pipeline,
4. Reachability and Rollback Pipeline,
5. Dynamic Security Analysis Pipeline,
6. Master Orchestrator Pipeline.

Chapter 4: Testing and Evaluation describes the strategy used to test both security and functionality. It introduces the testing environment, including the OWASP API testing framework, and the use of SonarQube for static analysis. The chapter also covers compliance and audit checks, and compares tools such as Spectral, TruffleHog, and DAST solutions, with detailed results from the APIsec tool.

Chapter 5: Compliance with Cybersecurity Standards, Directives and Regulations evaluates the platform against major frameworks and regulations. It shows how the solution meets core security requirements and how its architecture can adapt to different compliance needs across various regulatory environments.

Chapter 6: Conclusions and Future Works summarizes the main results and compares them with the initial objectives. It reviews the effectiveness of the APIOps and DevSecOps approach, discusses its impact on enterprise API management, and outlines possible future improvements.

Chapter 2

Background and Key Concepts

2.1 APIs

Application Programming Interfaces (APIs) are at the core of modern software architecture. They enable communication and data exchange between different systems, applications, and services regardless of the programming languages or platforms used. As digital ecosystems become more complex, APIs have become the backbone of integration strategies across Web, mobile, and cloud-based environments.

An API defines a set of rules and protocols for accessing a software component or service. These interfaces allow developers to abstract the underlying implementations and interact with the functionalities through well-defined endpoints. Instead of rewriting entire systems, organizations can leverage APIs to reuse, extend, and scale their services effectively.

The role of APIs has evolved significantly. Initially used for basic internal communication, they are now critical for exposing services to external partners, building ecosystems, and enabling business models based on data and service monetization. This transformation has led to the emergence of *API-first* approaches, where the design and management of APIs precede the implementation of the software itself.

From a business view, APIs help companies be fast and innovate. They cut the time to launch new services, make third-party integrations easier, and provide core parts for microservices and serverless architectures. In fast-changing fields like finance, telecommunications, and healthcare, API ecosystems allow real-time data sharing and ensure compliance with open standards and regulations.

2.1.1 API Types and Styles

As outlined in Erik Wilde's overview on API styles [2], APIs can be considered as the "languages" that software systems use to communicate and share information.

Over the years, a wide variety of technologies have been developed to support API implementation, but the design of an API always starts with a key decision: which API style should be adopted?

An API style defines the general model and structure used to expose functionality or data. It shapes how API consumers interact with the system and what kind of abstractions are available to them. While many technical implementations are possible, most API designs today are built around one of five major styles:

- **Tunnel Style:** The API is viewed as a remote interface that allows calling functions or methods on the server, much like invoking local procedures. This model is often used in Remote Procedure Call (RPC) systems, including technologies such as gRPC[1]. It is useful when performance and precise control over function calls are required.

- **Resource Style:** The API is seen as a collection of addressable resources that clients can manipulate using standard HTTP methods like GET, POST, PUT, and DELETE. This is the foundation of RESTful APIs and is typically documented using OpenAPI specifications. It promotes simplicity, scalability, and interoperability.

- **Hypermedia Style:** This style treats APIs as interconnected web-like resources. Each response may include links to related actions, guiding the client dynamically. Though less common in practice, this style aligns closely with the original vision of REST.

- **Query Style:** In this model, the API exposes a structured data model that clients can query and manipulate using a general-purpose query language. GraphQL[2] is a well-known implementation of this style, allowing clients to request exactly the data they need and reduce over-fetching or under-fetching issues.

- **Event-based Style:** Instead of offering direct interactions, the API is designed around events that are published by providers and consumed by subscribers. This model is essential for asynchronous and real-time architectures, and is often implemented using technologies like Kafka[3] or WebSockets[4].

[1]gRPC is a high-performance RPC framework developed by Google that uses HTTP/2 and Protocol Buffers for efficient communication.

[2]GraphQL is a query language for APIs developed by Facebook that allows clients to request only the data they need, improving efficiency and flexibility.

[3]Apache Kafka is a distributed event streaming platform used for high-performance data pipelines, streaming analytics, data integration, and mission-critical applications.

[4]WebSocket is a communication protocol that enables full-duplex, bidirectional data exchange between a client and a server over a single, long-lived connection, ideal for real-time applications.

Each API style reflects different architectural choices and is suitable for specific use cases. Choosing the most appropriate style depends on several factors, such as the nature of the data, communication patterns, expected latency, scalability requirements, and the preferences of API consumers and producers. Understanding these different styles is essential to make informed design decisions when building or evolving an API ecosystem.

2.2 API Platform Maturity Models

As organizations increase their reliance on APIs, they often face operational challenges such as inconsistent governance, fragmented infrastructure, and security vulnerabilities. To address these issues systematically, maturity models are commonly used to assess current capabilities and define a roadmap for incremental improvement. Many of these models are inspired by well-established frameworks from the software engineering field.

One of the earliest and most influential is the *Capability Maturity Model (CMM)* [3], initiated in 1986 as part of a research initiative funded by the *U.S. Department of Defense*. Developed at the *Software Engineering Institute of Carnegie Mellon University*, its original purpose was to evaluate the capability of software contractors to deliver on complex development projects reliably and predictably. The CMM defined process maturity as clear levels that help organizations improve quality, efficiency, and control.

The model defines five levels of maturity:

1. **Initial**: Processes are ad hoc, undocumented, and reactive. Project success often relies on individual effort and heroics rather than consistent methods.

2. **Repeatable**: Basic project management processes are established. Some repeatability is achieved, although rigor and standardization may still be limited.

3. **Defined**: Processes are documented, standardized, and institutionalized across the organization. A shared process framework is adapted to specific contexts.

4. **Managed**: Quantitative metrics are introduced to monitor performance, allowing for data-driven process control and predictability.

5. **Optimizing**: Continuous improvement becomes the strategic focus. Feedback loops and innovation are used to refine and enhance processes proactively.

Each maturity level in the Capability Maturity Model is defined by a set of *Key Process Areas (KPAs)* clusters of related activities that are critical for achieving the objectives of that level. KPAs represent core focus areas such as project planning, configuration management, or quality assurance, and their successful implementation is considered essential for process capability at that stage. Each KPA is further characterized by specific goals, measurement criteria, institutionalization practices,

and verification mechanisms. Advancement through the levels is strictly sequential, as each stage builds upon the process foundations established in the preceding one.

Although the CMM was originally designed for evaluating software development processes in defense contracting, its structured approach has since been widely adopted in other sectors, including IT, government, and commercial domains. To address the challenges of maintaining multiple models, the framework was eventually consolidated into the *Capability Maturity Model Integration (CMMI)*. Introduced in the early 2000s and maintained today by the Information Systems Audit and Control Association (ISACA). The most recent version CMMI v3.0 was released in 2023.

The base ideas from the CMM are still important. Its step-by-step, process-focused method has shaped many models, including those for API platform maturity. In API management, using a maturity model helps check and improve areas like lifecycle automation, developer enablement, security governance, and operational scalability. This supports a continuous improvement strategy designed for API ecosystems.

Kong API Maturity Levels

In the context of API infrastructure, Kong[5] proposes a domain-specific maturity model called the *API Platform Engineering Maturity Model*[4] which defines five maturity levels: **Basic**, **Foundational**, **Intermediate**, **Advanced**, and **Differentiated**. Each level is evaluated in three dimensions (Figure 2.1):

- *Runtime Platform*: architectural deployment options, gateway scalability, and observability;
- *Discovery and Documentation*: catalogue quality, developer portals, and API discoverability;
- *APIOps*: automation of the API life-cycle, including linting, testing, CI/CD and policy governance.

This layered approach allows organizations to evaluate not only the technical aspects of their API infrastructure, but also the cultural and process maturity of their teams.

[5]Kong is a cloud-native API gateway provider known for its open-source and enterprise solutions for managing, securing, and scaling APIs.

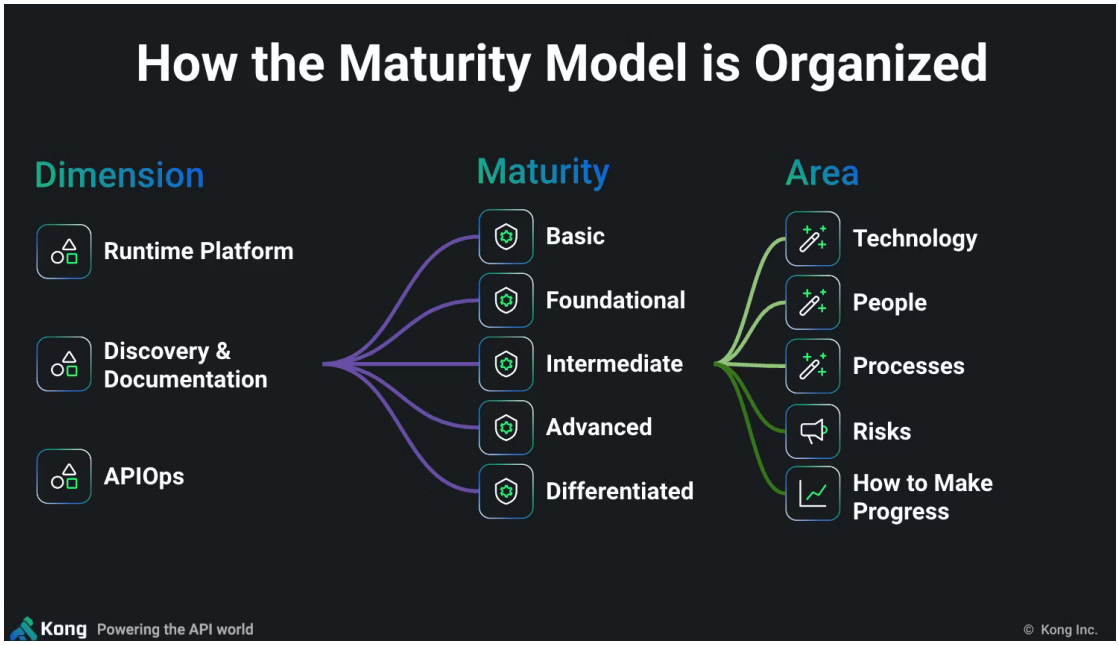


Figure 2.1: Organizational maturity progression in API platform engineering, as illustrated by Kong[4].

	Technology	People	Process
Basic	Lacks an API gateway, leading to APIs being exposed directly without mediation.	API awareness and responsibility rest solely with application developers.	Relies on existing SDLC processes, with no specialized approach for API deployments.
Foundational	Individual teams start adopting their own API gateways, leading to inconsistency.	Certain team members begin to specialize in managing API gateways.	Informal process for API gateway selection and configuration, often based on available tools.
Intermediate	Formal API gateways established, but limited in deployment styles and multi-tenancy capabilities.	Dedicated team for API gateway management emerges, with increased collaboration across teams.	Formal procedures for accessing the API platform are introduced, involving coordination across various teams.
Advanced	API gateways are consolidated and support a broad spectrum of deployment styles.	Roles expand to include SREs and specialized support functions for the API platform.	Self-service access to the API platform is enabled, enhancing autonomy and efficiency.
Differentiated	API gateway evolves into a sophisticated platform supporting extensive customization.	High degree of collaboration among various specialized roles in API management and governance.	Seamless and transparent API management processes, with governance and customization driven by API metadata.

Figure 2.2: Stages of maturity across Technology, People, and Process based on Kong's API Platform Engineering Maturity Model[4]

Discovery and Documentation

Another important aspect of working with APIs is how they are described and made discoverable. The Discovery and Documentation dimension focuses on how APIs are organized, documented, and made available to both internal and external users. This includes not only the presence of up-to-date documentation, but also the tools and processes that help developers easily find, understand, and use APIs.

As organizations move through the maturity stages, from **Basic** to **Differentiated**, they go from having little or no documentation to building complete, automated portals and marketplaces for their APIs. In the early stages, API documentation is often static, incomplete, and manually written by developers. In more advanced stages, documentation becomes interactive, is automatically published, connected to version control, and includes examples based on real use cases. This evolution helps transform APIs into well-documented and easy-to-use products.

	Technology	People	Process
Basic	No API portal product, with OpenAPI specifications often found in Git repositories.	API developers handle documentation on a best-effort basis, without specialized training.	No formal process for API documentation, leading to inconsistent and ad-hoc documentation practices.
Foundational	Wiki-style tools used as informal API documentation repositories.	API developers document APIs on a best-effort basis; some begin to specialize in this area.	Awareness of API documentation's importance grows, but processes remain informal and unstandardized.
Intermediate	Official API portal product introduced, supporting basic OpenAPI specification rendering.	API Platform team maintains the API portal, with developers responsible for their documentation.	Formal process for publishing to the API portal established, though still manual.
Advanced	API portal with SSO integration, self-registration, and comprehensive analytics.	Involvement of technical writers and support agents enhances documentation quality and support.	Self-registration and automated publishing to the API portal streamline the documentation process.
Differentiated	API Portal evolves into a marketplace with advanced analytics and SDKs for various languages.	Collaboration among API teams, technical writers, and data engineers for tailored documentation.	Seamless, metadata-driven documentation process with integrated analytics and governance adherence.

Figure 2.3: Maturity progression for the Discovery and Documentation dimension, based on Kong's API Platform Engineering Maturity Model [4].

2.3 DevOps Principles and Practices

The implementation of DevOps practices represents a crucial factor in controlling contemporary API platforms. *DevOps* is a set of principles, cultural practices, and

technical approaches that aim to bridge the gap between software development (Dev) and IT operations (Ops). Its core goal is to accelerate software delivery while maintaining high quality, through enhanced collaboration, shared responsibilities, and process automation.

DevOps includes Continuous Integration (CI) together with Continuous Delivery (CD) and Infrastructure as Code (IaC) as well as automated testing and real-time monitoring as its main practices. These practices reduce the time needed to bring changes into production, increase reliability, and foster a feedback-oriented development culture. As illustrated in Figure 2.4, the DevOps workflow is typically represented as an infinite loop that includes planning, coding, building, testing, releasing, deploying, operating and monitoring.

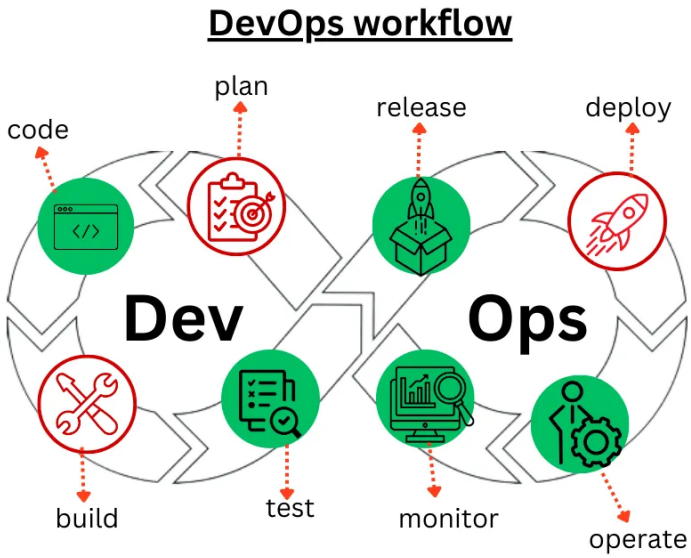


Figure 2.4: DevOps workflow. [5].

Several tools are available to support the implementation of DevOps workflows, enabling automation, traceability, and scalability across the software delivery pipeline. Among the most widely adopted are **Azure DevOps** and **Jenkins**, which offer different levels of integration and flexibility for managing CI/CD processes.

2.3.1 Azure DevOps

Among the most widely adopted platforms for implementing DevOps workflows is **Azure DevOps**, a comprehensive suite of development services provided by Microsoft. It enables teams to plan, build, test, and release software in a collaborative and automated manner. Supporting both Microsoft and open-source technologies,

Azure DevOps is particularly suited for enterprise environments and cloud-native architectures.

The platform offers a complete set of integrated services that span the entire software development lifecycle. Each component is designed to foster automation, collaboration, and continuous improvement across development and operations teams.

- **Azure Repos** provides cloud-hosted Git repositories with advanced collaboration features. It is used to manage version control for source code, infrastructure configurations, and API assets such as OpenAPI specifications and policy files. The service supports pull requests, branch policies, and code reviews to ensure quality and traceability.
- **Azure Pipelines** is a CI/CD service that automates the building, testing, and deployment of code to various environments. It integrates continuous integration (CI), testing, and delivery (CD) to streamline the release process. Pipelines are defined as code using YAML and support parallel jobs, containers, virtual machines, and Kubernetes-based [6] workloads.
- **Azure Boards** is a work tracking system that supports Agile project management[7]. It enables teams to manage tasks, backlogs, and development progress using customizable dashboards and workflows.
- **Azure Artifacts** allows teams to manage and share software components and build outputs across projects. It facilitates reuse, versioning, and consistency of dependencies without relying on external package repositories.
- **Azure Test Plans** offers a suite of tools for managing test cases and collecting structured feedback. It supports manual and exploratory testing, helping teams maintain quality standards throughout the application lifecycle.

Thanks to its native integration with Azure services and support for modern DevOps practices, Azure DevOps provides a reliable foundation for implementing secure and scalable APIOps pipelines in enterprise cloud environments.

2.3.2 Jenkins

"Jenkins is a self-contained, open-source automation server which can be used to automate all sorts of tasks related to building, testing, and delivering or deploying software." [6].

[6]Kubernetes is an open-source system for automating the deployment, scaling, and management of containerized applications.

[7]Agile project management is a flexible and iterative approach to planning and managing software development projects, emphasizing continuous feedback, adaptive planning, and collaboration between cross-functional teams.

It supports the implementation of CI/CD pipelines and can be installed on various operating systems using native packages, Docker containers, or run directly using the Java Runtime Environment. Its plugin-based architecture allows users to tailor the platform to specific needs by integrating with a wide range of tools and services, including Git, Docker, Kubernetes, and cloud providers.

One of its most powerful features is the *Pipeline* system, which enables developers to define build, test, and deployment workflows as code. Jenkins Pipelines are written in a domain-specific language based on Groovy, allowing both simple and complex automation scenarios to be expressed declaratively. This approach promotes transparency, repeatability, and version control of the automation process.

Jenkins also supports user interfaces such as Blue Ocean, which provides a modern visualization of pipelines and builds, improving usability for both developers and DevOps engineers. Thanks to its flexibility, community support, and mature ecosystem, Jenkins remains a reference solution for implementing DevOps practices in both small-scale and enterprise environments.

2.4 Coordinating Enterprise Workflows through ITSM and a Structured Process Model

Organizations of the enterprise scale need to resolve the essential problem of combining DevOps rapid delivery methods with organizational governance requirements. A strong **IT Service Management (ITSM)** framework establishes the fundamental structure to achieve balanced operations. The IT Service Management framework delivers value through the complete delivery and management of IT services which support business operations. A service-centered approach ensures every IT task including development and operations and support connects to business goals and customer requirements.

While ITSM defines the *"what"* and *"why"* of service delivery, **ITIL** complements it by providing the operational *"how"*. ITIL is the most widely adopted collection of best practices for implementing ITSM, offering standardized processes and guidelines to support change management, incident response, and problem resolution.

ITSM platforms like *ServiceNow*[8] serve as essential tools to enable this model. ServiceNow provides structured workflows for change management through automated approval processes and cross-team orchestration capabilities. The affected infrastructure components connect to change requests (CRQ) which include impact evaluations and rollback plans that automatically start stakeholder validation. Every stage of this process remains trackable and continues to be operational throughout its lifetime.

The following operational model represents a practical ITSM implementation, structured to manage the lifecycle of API and application releases.

[8]ServiceNow is a platform-as-a-service designed to support IT service management and help desk operations through automated workflows.

The workflow depicted in Figure 2.5 starts through a continuous team discussion between **Business Stakeholders** and the **Application Team Lead** before any official ticket exists. The business requirements undergo thorough understanding and refinement through this dialogue before they move into formal governance procedures. The Application Team Lead functions as a vital bridge to transform functional requirements into achievable technical objectives.

After this early exchange, the Application Team Lead formalizes the proposal by creating a **Change Request** within the central **ITSM System**. This action elevates the request from an informal idea to a formally governed entity. The ITSM platform then assumes the role of process orchestrator: it not only distributes tasks to the **Infrastructure Team**, but also serves as a dynamic workspace for the **Infrastructure Team Lead**, who manages and updates all aspects of their contribution to the workflow.

The model focuses on two main areas of collaboration. The first is the **Technical Coordination** between the Application Team Lead and the Infrastructure Team Lead; this relationship is vital for creating a clear and integrated solution. The second area, equally important, is the **Security Review**. This is a structured conversation between the **Infrastructure Team Lead** and the **Cybersecurity Team**. This step is not a strict barrier but a team effort to identify risks and develop strategies to reduce them, ensuring that security is incorporated right from the design phase.

A key aspect of this model is giving power to the technical teams. The **Application Team** is responsible for keeping the CRQ in the ITSM system, documenting technical progress and decisions. This strengthens accountability and ensures that information stays centralized and current. Smooth coordination within and between teams happens through regular communication among the leads and their members.

While maintaining a distinction between development and operational roles, this approach is inspired by the core principles of DevOps. In this model, **shared responsibility** is a central pillar, fostering collaboration between teams. It clarifies ownership, reduces handoffs, and integrates security as an active part of the process from the earliest stages. Through an ITSM framework, organizations can establish clear roles and structured approval workflows, enabling API evolution that is both rapid and traceable, without compromising security, reliability, and alignment with business goals

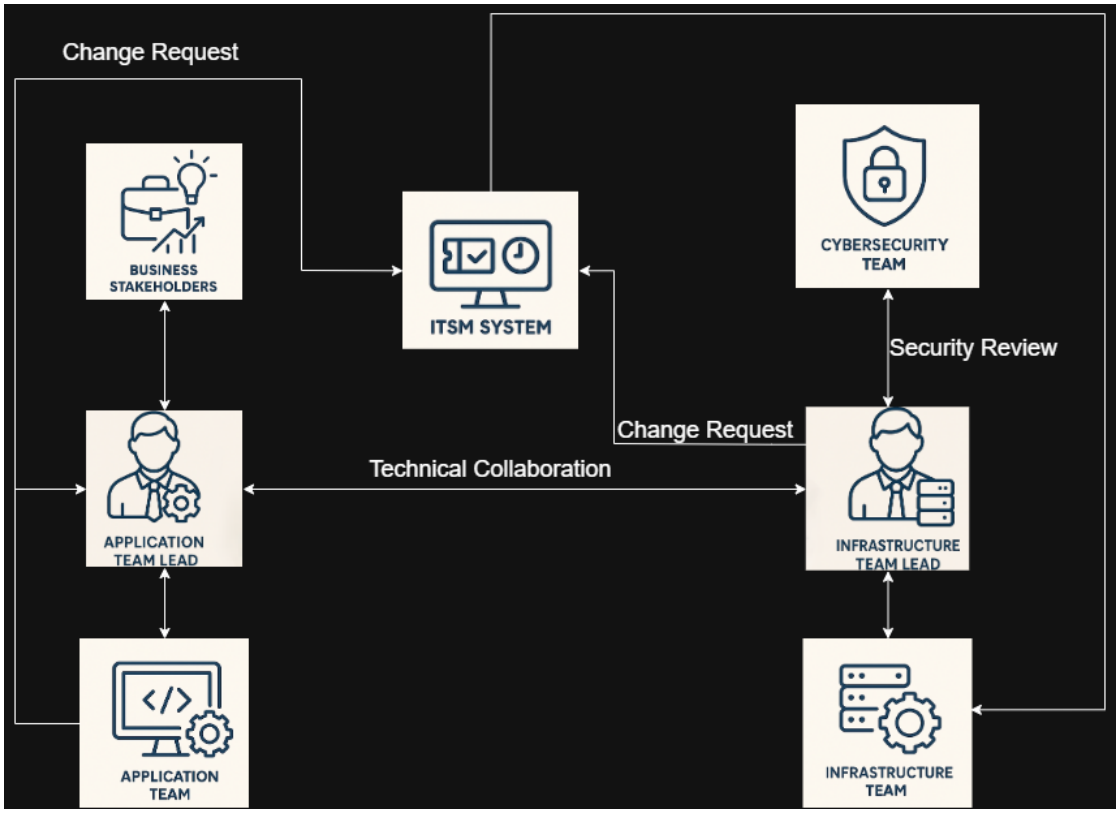


Figure 2.5: Stakeholder interactions and workflow integration via ITSM during API release lifecycle.

Practical Scenario: Application Service Evolution via API

To illustrate this operational model with a practical example, let's consider a common scenario: a company needs to modernize an older and monolithic service. The goal is to expose its functions through a new secure Application Programming Interface (API) to allow for new business integrations

Requirement Analysis and Definition: The process starts from a business need. The **Business Stakeholders** (e.g. a business unit head) engage with the **Application Team Lead**. The goal of this preliminary phase is not technical, but it must define the functional scope, business objectives (e.g., reduced time-to-market, operational efficiency), and success criteria.

Formalization and Process Initiation: Once the requirements are established, the Application Team Lead translates the business need into a formal proposal. That is done by creating a **Change Request (CRQ)** within the **ITSM** platform. The CRQ is not a simple ticket, but a structured artifact containing:

- A description of the change and its business justification;

- A list of impacted assets and Configuration Items (CIs);
- A preliminary assessment of service risk and impact;
- High-level functional and non-functional requirements for the new API.

Technical Coordination and Solution Design: When the CRQ is created, the system automatically gives tasks to the right teams. The **Infrastructure Team Lead** is brought in to check the impact on the infrastructure. Close technical teamwork between the application and infrastructure leads is critical here to write the *solution design* document. This document explains the technical architecture, including the endpoints, the security protocol for users (e.g., OAuth 2.0), required network setups (like placing it in a DMZ or setting firewall rules), and plans for future growth.

Integration of Security Governance (Security by Design): Next, the *solution design* is reviewed by the security experts. The Infrastructure Team Lead takes the plan to the **Cybersecurity Team**. It's important to note that their job isn't to just approve or deny the project at the end. Instead, they work with the team to decide on the right security protections (like encrypting data, preventing common attacks, and setting up monitoring). This makes sure security is a part of the original plan, not a last-minute fix.

Implementation and Documentation Lifecycle: With the design approved, the teams proceed with implementation. The CRQ serves as the "single source of truth" for the entire change lifecycle. All produced artifacts like source code, scripts for resource provisioning, test plans, and results are systematically linked to the CRQ. This ensures complete traceability and auditability of the changes from the initial requirement to its deployment in production.

This approach allows companies to ensure that all new changes and innovations are done in a controlled and secure way. It provides a reliable process for transforming a business requirement into a well-documented service, ready for enterprise use.

2.5 APIOps

APIOps is an extension of DevOps principles, adapted specifically to the management of APIs throughout their entire lifecycle. As APIs have become foundational components of modern integration and digital services, ensuring their consistency, security, and operational reliability has become a strategic concern. APIOps introduces automation and control into key phases of the API lifecycle, including design, development, testing, deployment, versioning, documentation, and monitoring.

By integrating APIs into CI/CD pipelines and adopting practices such as contract testing, linting, and automated publishing, APIOps enables teams to manage APIs as reliable, reusable digital products, rather than isolated technical artifacts. This product-centric perspective encourages closer collaboration among developers, platform engineers, and security teams, and aligns API delivery with broader organizational objectives related to scalability, standardization, and policy enforcement.

Figure 2.6 illustrates the evolution of APIOps capabilities across different maturity levels, as represented in Kong's API Platform Engineering Maturity Model [4].

	Technology	People	Process
Basic	No proper GitOps automation tool in place; golden image and API proxy configuration are manual.	API developers assume APIOps tasks on a best-effort basis, mostly based on manual steps.	Ad-hoc manual processes for APIOps activities like golden image creation and gateway configuration.
Foundational	Introduction of an APIOps toolchain; still lacking tools like API mocking servers and linting.	Specialized roles emerge in the platform team for APIOps toolchain, but not as a full service.	Simple processes around API gateway configuration and infrastructure provisioning start to get implemented.
Intermediate	APIOps pipeline offers consistent functionality like automatic proxy configuration and linting.	APIOps engineer is a formal role; other areas like security and IaC get involved in pipeline design.	API teams work at the OpenAPI specification level; the pipeline handles processing and validation.
Advanced	Pipeline supports custom plugin configuration and integrates with API portal for documentation.	APIOps engineer adopts an APIOps-as-product mindset; enterprise architecture gets involved.	Seamless process for API teams, with the pipeline taking care of most aspects after receiving OpenAPI specs.
Differentiated	Customizable APIOps pipeline setup tailored to specific API profiles based on metadata.	High collaboration among APIOps engineers, infrastructure engineers, security, observability, etc.	API teams enjoy a transparent process; providing OpenAPI specs and metadata is sufficient for compliant deployment.

Figure 2.6: Maturity progression for the APIOps dimension, based on Kong's API Platform Engineering Maturity Model [4].

2.5.1 Analysis of APIOps Methodologies and Frameworks

APIOps represents more than a single methodology; it encompasses a variety of approaches, philosophies, and standards that organizations must carefully evaluate to make informed strategic decisions about API management. A foundational element of this evaluation is the cultural shift toward treating APIs as *digital products* rather than as mere technical endpoints. As Kin Lane discusses in an interview with Luke Seelenbinder [7], this perspective requires APIs to be supported with proper pricing models, comprehensive documentation, and dedicated support, just like any product offered to customers. This paradigm shift carries profound organizational implications. Traditional IT departments, historically focused on internal system integration, must now adopt product management methodologies typically associated with customer-facing offerings. This transition demands new organizational structures, including the emergence of API Product Managers who bridge technical implementation with

business strategy, and API Designer roles that focus on developer experience rather than just functional correctness. The resistance to this change often stems from deeply embedded cultural factors: engineering teams may view product-centric approaches as bureaucratic overhead, while business stakeholders might struggle to quantify the ROI (*return on investment*) of improved API design.

The economic impact of this shift is tangible. Organizations adopting API-as-product strategies report a significant trend toward faster partner integration times and a notable reduction in support costs due to better documentation and design consistency. These benefits, however, are contingent upon upfront investments in tooling, training, and organizational restructuring, which can constitute a substantial portion of the total IT budget during the transformation phase. The fundamental challenge, therefore, lies not in the adoption of this paradigm, but rather in the effective management of the transition cost while ensuring operational continuity.

This product-centric view directly influences the technical development lifecycle, leading to a critical decision between a **Design-First** and a **Code-First** approach. In Design-First development, teams create an API contract using a format like the OpenAPI Specification (OAS)[9] before writing any code, establishing a single *source of truth*. This method offers numerous advantages, such as enabling parallel development between frontend and backend teams using mock servers[10], allowing for early validation of design choices and facilitating 'shift-left' security by enabling automated vulnerability scanning and compliance checks on the API contract before a single line of implementation code is written. This proactive approach to security is reflected in the industry's significant progress in making software more secure over the past year. Despite these strengths, it introduces initial overhead in design time and can create project bottlenecks if the design process is not managed efficiently.

Conversely, the Code-First approach prioritizes implementation speed by generating API specifications directly from the written code. This accelerates rapid prototyping and ensures documentation always matches the implementation, but it often leads to API sprawl[11], incurs significant governance debt, and creates integration challenges for consumer teams. The industry's convergence on OAS, while alternatives like RAML[12] and API Blueprint[13] have declined, suggests that Design-First approaches provide

[9]OpenAPI Specification (OAS) is a standard format for describing REST APIs. It allows both humans and computers to understand API capabilities without accessing the actual code.

[10]Mock servers generate responses based on the specification, enabling frontend development before backend implementation

[11]API sprawl refers to the uncontrolled proliferation of APIs within an organization, often developed without consistent standards or governance, leading to increased complexity, integration difficulties, and long-term maintenance challenges.

[12]RAML (RESTful API Modeling Language) was an earlier specification format that lost adoption to OpenAPI.

[13]API Blueprint used Markdown for API descriptions but has largely been replaced by OpenAPI.

greater long-term value despite their initial costs. Market research indicates that the global open API market, largely driven by OpenAPI Specification adoption, was valued at $4.53 billion in 2024 and is projected to reach $31.03 billion by 2033 [8].

The success of any API approach depends on strong standards and active community support. Well-known specifications such as OpenAPI (OAS), AsyncAPI[14], and JSON Schema[15] offer stability and mature tools. Newer specifications like TypeSpec[16] and MCP[17] bring innovation but may also cause fragmentation. As noted in the discussion on *solutionism* [9], organizations should ask whether these tools solve real problems or simply add complexity. When chosen carefully, they can still improve governance and drive progress.

Beyond the development philosophy, two distinct paradigms for operational management have emerged: the APIOps Cycles framework and the GitOps paradigm. APIOps Cycles provides a business-oriented model for managing APIs through iterative phases. As Lane explores [9], this model supports advanced concepts like *Arazzo* [18] for defining multi-operation workflows and *Overlays* [19] for applying concerns like security as separate layers. Its main strengths are that it is easy for different stakeholders to use and flexible to adapt, but because it gives fewer strict rules, it can become inconsistent if not supported by strong governance.

In contrast, GitOps, heavily promoted by the Cloud Native Computing Foundation (CNCF)[20], mandates that Git repositories serve as the single source of truth for all configurations, with every change following a rigorous pull request workflow. The CNCF Annual Survey 2024 found that 77% of respondents have adopted GitOps methodology to some degree [10], with organizations reporting benefits such as reduced deployment failures, improved audit compliance, and faster disaster recovery. These advantages directly support the observed industry trend of accelerating release cycles, where 29% of organizations now release code multiple times a day, and a growing emphasis on automating security and compliance checks throughout the development lifecycle. At the same time, challenges remain, including a steep learning curve and the need for a culture that fully trusts automation. In practice, many organizations combine the strategic orientation of APIOps Cycles with the operational discipline of GitOps, thereby integrating flexibility with reliability.

---

[14]AsyncAPI uses events or messages to trigger actions, instead of direct API calls.

[15]JSON Schema defines how JSON data should be structured and validated.

[16]TypeSpec is a flexible language for designing APIs in a clear and extensible way.

[17]MCP (Model Context Protocol) aims to standardize how AI models interact with external systems.

[18]Arazzo is a specification for describing API workflows and sequences, showing how multiple API calls work together to accomplish business goals.

[19]Overlays allow teams to apply different configurations (like security policies or regional settings) to a base API definition without modifying the core specification.

[20]CNCF is a Linux Foundation project that promotes cloud-native computing, including technologies like Kubernetes and standardized practices like GitOps.

For the context of this thesis the adoption of an **API-centric** approach that integrates **Design-First** and **GitOps** methodologies emerges as the most effective strategy.

This choice is not dictated by technical considerations alone, but reflects a profound commitment to treating APIs as **strategic business assets**. The **Design-First** model, based on the **OpenAPI Specification**, ensures that APIs are designed as products before being developed, guaranteeing consistency, stability, and compliance with regulations from the earliest stages of their lifecycle.

The integration with **GitOps** practices in which the Git repository becomes the single source of truth for API configuration extends this rigorous governance to the production phase. In this way, every change, policy update, and deployment is fully automated, traceable, and reversible.

This approach offers the perfect balance between methodological rigor, tooling support (with automated pipelines), providing the necessary foundation for a solid, secure, and compliant API management.

2.5.2 APIOps in Azure Environments

As described in the maturity model, the *APIOps* dimension reflects the shift from manual, ad hoc API delivery processes to fully automated, policy-driven workflows. In the Microsoft Azure ecosystem, this paradigm is implemented by combining Azure DevOps services with Azure API Management (APIM) to enable a GitOps-style API lifecycle [21]. This approach ensures that APIs can be versioned, validated, and deployed through repeatable, auditable pipelines, significantly increasing reliability and governance.

The workflow illustrated in Figure 2.7 includes the following main stages:

- **Synchronization:** API operators initiate an extractor pipeline that synchronizes the current state of the APIM instance with a Git repository. This exports all relevant configuration data, such as API definitions, policies, diagnostic settings, and service metadata, into version-controlled files.
- **Pull request generation:** When differences between the Git repository and the APIM instance are detected, a pull request (PR) is automatically created. This initiates a review process to validate the changes before they are merged.
- **API development and contribution:** API developers clone the repository, make changes locally (typically using OpenAPI specifications), and submit their contributions via additional PRs. These changes may involve the design of new APIs or updates to existing ones.

[21] GitOps is an operational framework that applies DevOps best practices, such as version control, collaboration, compliance, and CI/CD, to infrastructure automation.

- **Review and approval:** All proposed changes undergo peer review through the pull request mechanism, ensuring compliance with internal guidelines and early detection of potential issues.
- **Automated deployment:** Once approved, the publisher pipeline is triggered. This pipeline applies the validated configuration to the APIM instance, effectively deploying the APIs, policies, and settings across the target environment.

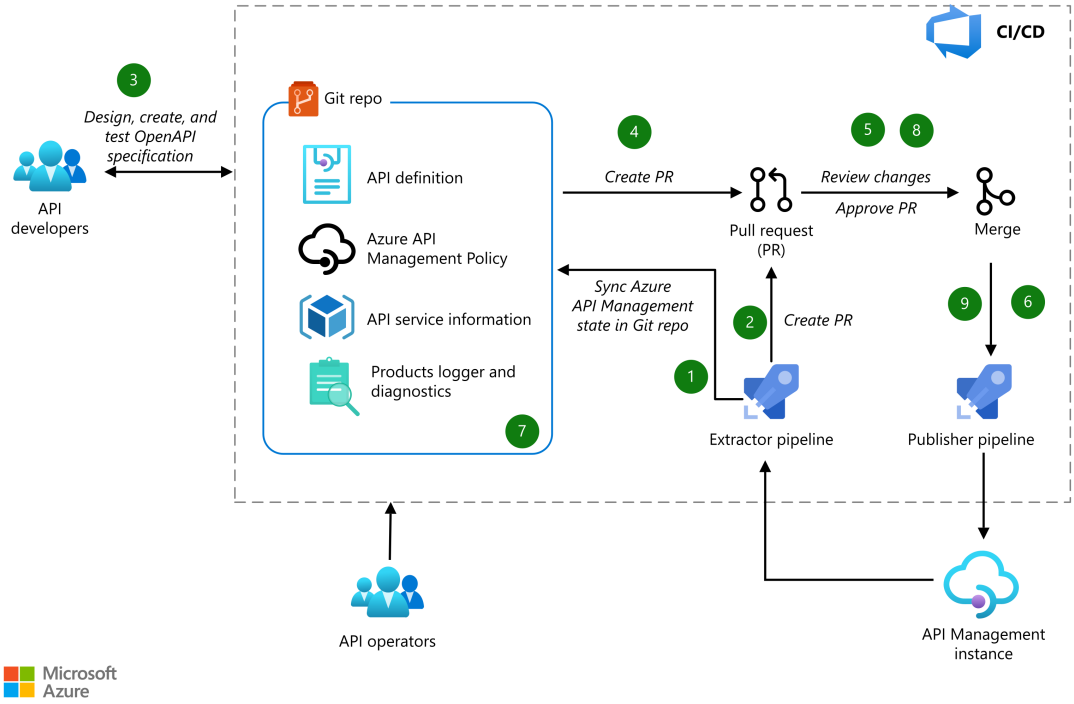


Figure 2.7: APIOps reference architecture in Azure, adapted from [11].

This architecture enables several key benefits. All API-related changes are tracked through Git, which enhances auditability and facilitates rollback. The use of pull requests introduces checkpoints for validation, reducing the likelihood of misconfigurations or policy violations. Furthermore, the automation of deployment ensures consistency across environments, while limiting direct access to production resources, reinforces security best practices and adheres to the principle of least privilege.

Thanks to these capabilities, the Azure APIOps model is particularly well suited for organizations operating in regulated sectors where compliance, traceability, and operational rigor are essential.

Building on this operational foundation, it becomes equally important to address the strategic and technical aspects of how APIs are governed and exposed. This is the

role of *API Management* (APIM), a key layer that enables centralized control over the full API lifecycle, from publication to monitoring and policy enforcement.

2.6 API Management (APIM)

API Management (APIM) refers to the set of processes, tools, and technologies used to publish, secure, monitor, and govern APIs in a scalable and controlled manner.

An API Management solution typically includes the following core components:

- **API Gateway**: Acts as an intermediary between clients and backend services, handling request routing, rate limiting, caching, and authentication policies.
- **Management Layer**: Tools for admins and platform teams (web portals, APIs, automation scripts) to define, set up, and manage APIs and their policies.
- **Developer Portal**: Offers a self-service interface for internal and external developers to explore, test, and subscribe to APIs. It often includes auto-generated documentation, onboarding tools, and usage analytics.
- **Analytics and Monitoring**: Enables real-time visibility into API usage, performance metrics, and error rates. These insights support decisions regarding scaling, security, and optimization.
- **Security and Governance**: Ensures consistent enforcement of policies such as authentication, access control, quota enforcement, and request validation.

The goal of API Management is to remove operational and governance work from development teams. This lets them focus on the business logic of APIs, while still meeting compliance, reliability, and performance standards.

Centralizing API governance ensures consistency in distributed systems, makes partner integration easier, and supports building scalable, secure digital platforms.

APIM solutions are especially useful in enterprises. Here, APIs are used for internal integration and for offering services to third-party developers, business partners, or the public.

2.6.1 Kong API Gateway and Management Platform

Kong provides a lightweight, high-performance, and extensible API gateway designed for cloud-native, hybrid, and multi-cloud deployments. At the core of its offering is **Kong Gateway**, a reverse proxy that manages, routes, and secures API traffic across distributed architectures and microservices environments [12].

Kong Gateway is implemented as a Lua[22] application that runs within NGINX[23]. It is distributed with OpenResty, a pre-packaged bundle of NGINX modules that enables efficient Lua scripting and runtime extensibility. This architecture supports a modular plugin system that allows custom logic to be injected at various stages of the API request lifecycle.

Services and Routes

Kong Gateway defines API traffic behavior through a declarative object model, where two key building blocks are **Services** and **Routes** [13].

A *Service* in Kong is an abstraction of an upstream application, API, or microservice. It encapsulates the connection configuration necessary to forward traffic such as the protocol, host, port, and path of the target system. Each service acts as a logical representation of a backend system that can be reused across multiple API endpoints.

A *Route* defines how incoming client requests are matched and forwarded to a service. Routes contain matching criteria such as request paths, HTTP methods, headers, protocols, and hostnames. One service may be exposed through multiple routes, allowing administrators to implement different access policies, plugins, or user experiences for the same underlying backend.

Figure 2.8 illustrates this pattern: a client sends a request to a public-facing route (e.g., `/mock`), which Kong uses to locate and forward the request to the associated service (e.g., `example_service`), which in turn communicates with the upstream application.

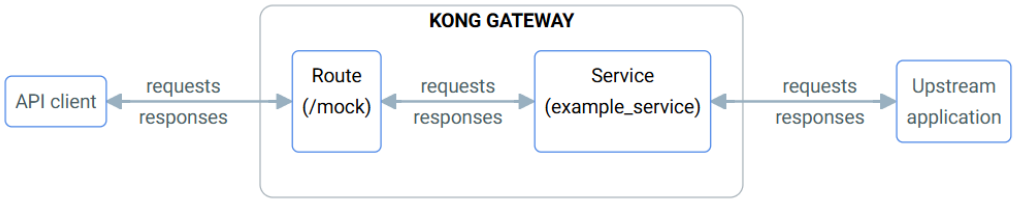


Figure 2.8: Kong Gateway routing model: routes match incoming requests and forward them to services, which proxy upstream applications.

[22]Lua is a lightweight, high-level scripting language designed for embedding into applications.

[23]NGINX ("engine x") is a high-performance HTTP server, reverse proxy, content cache, and load balancer.

Plugins and Deployment Ecosystem

Beyond traffic routing, Kong Gateway offers a rich and extensible plugin system that enables fine-grained control over API behavior. Plugins, whether built-in or custom-developed, can be applied to services, routes, or consumers to implement policies such as authentication, rate limiting, traffic transformations, logging, and other runtime operations. Kong provides a wide selection of official plugins and also supports the development of custom plugins.

Kong Gateway supports **multiple deployment models**. In a self-managed environment, users have full control over configuration using the Kong Admin API, the Kong Manager graphical interface, or the declarative configuration tool `decK`. Alternatively, Kong Gateway is available as a managed service through **Kong Konnect**, a SaaS-based control plane hosted by Kong Inc. Konnect simplifies the administration of multi-node and multi-region deployments by centralizing policy management, analytics, and API lifecycle operations. It is offered in both Plus (pay-as-you-go) and Enterprise subscription tiers [14].

For containerized environments, Kong offers native integration with Kubernetes through its Ingress Controller. This enables configuration of Kong Gateway using Kubernetes-native resources such as Ingress and Gateway APIs. Helm charts[24] and an Operator[25] are also available to streamline deployment and management in DevOps and GitOps workflows.

In addition to its core gateway capabilities, Kong provides a suite of tools that support the broader API lifecycle.

Kong Manager is a web-based graphical interface for administrating Kong Gateway. It allows users to manage services, routes, plugins, and consumers through an intuitive UI, simplifying daily operational tasks without requiring direct interaction with the Admin API.

`decK` is a command-line tool for managing Kong configurations in a declarative manner using YAML files. It enables automation and version control, making it well-suited for GitOps and APIOps workflows. decK supports tasks such as synchronization, drift detection, and modularization of complex configurations.

Insomnia is a desktop application focused on API-first design and testing. It allows developers to define OpenAPI or GraphQL specifications, send test requests, validate responses, and organize environments. Its collaborative features and spec-first approach make it a powerful tool for developing and maintaining APIs across their entire lifecycle.

---

[24]Helm charts are reusable, versioned packages that define, install, and upgrade Kubernetes applications using YAML configuration templates. Helm simplifies the deployment and management of complex Kubernetes resources.

[25]A Kubernetes Operator is a software extension that automates the management of complex applications on Kubernetes by encoding operational knowledge into custom controllers. It uses Custom Resource Definitions (CRDs) to extend the Kubernetes API.

2.6.2 Azure API Management

Azure API Management (APIM) is a fully managed platform provided by Microsoft for publishing, securing, and monitoring APIs. It enables organizations to centralize control over their APIs while providing developers and partners with secure and consistent access to backend services. Designed for both internal and external API exposure, APIM integrates natively with the broader Azure ecosystem and supports enterprise-grade deployment scenarios.

At its core, Azure API Management acts as a gateway that sits between API consumers and backend services. It processes incoming requests, applies policies such as rate limiting or authentication, and forwards them to the appropriate services. By doing so, it decouples API consumers from backend implementations and provides a unified entry point for managing access, enforcing security, and collecting telemetry.

Azure APIM supports multiple API types, including REST, SOAP, and GraphQL. APIs can be imported from OpenAPI specifications, WSDL documents, or manually defined through the Azure Portal or infrastructure-as-code tools. Once defined, APIs can be grouped into products and associated with subscriptions, allowing fine-grained control over who can access which APIs and under what conditions.

Some of the key features offered by Azure API Management include:

- **Developer portal:** A customizable, auto-generated website where developers can discover, subscribe to, and test APIs. It supports authentication, access control, and integrates with the subscription management workflow.

- **Policy engine:** A powerful XML-based policy framework allows applying transformations, validations, filtering, caching, or authentication policies at various scopes (global, API, operation).

- **Security integration:** Native support for API keys, JWT validation, OAuth 2.0, and mutual TLS. Integration with Azure Active Directory and Microsoft Entra provides enterprise-grade identity management.

- **Monitoring and analytics:** Detailed logging and metrics are available via Azure Monitor[26], Log Analytics[27], and Application Insights[28]. These tools provide visibility into usage patterns, errors, and performance bottlenecks.

[26] A comprehensive monitoring solution for applications, infrastructure, and networks on Azure. It collects and analyzes performance metrics and activity logs.

[27] A service within Azure Monitor that allows you to query, analyze, and visualize log data collected from various sources.

[28] An application performance management (APM) tool that helps developers monitor their live web applications. It detects performance anomalies and provides powerful analytics to diagnose issues.

- **Versioning and revisions:** Azure APIM allows maintaining multiple versions and revisions of APIs simultaneously, facilitating smooth transitions and backward compatibility during updates.

- **CI/CD and IaC support:** APIs, policies, and configurations can be managed declaratively using ARM templates[29], Bicep[30], Terraform[31], or GitHub Actions[32], making the platform compatible with modern DevOps and GitOps workflows.

Thanks to its deep integration with Azure services such as Azure Functions, Logic Apps, Key Vault, and Azure DevOps, Azure API Management is a suitable choice for organizations building secure, scalable, and automated API ecosystems.

2.6.3 Tyk API Gateway and Management Platform

Tyk is an open-source API gateway and management platform designed to deliver control, security, and observability across APIs in modern distributed environments. Initially released under an open-source license, Tyk has matured into a modular and extensible solution that supports both community-driven use cases and enterprise-scale deployments.

In contrast to fully managed API platforms, Tyk follows a self-hosted, service-oriented architecture composed of several core components:

- **Tyk Gateway:** The traffic-processing engine responsible for request routing, authentication, rate limiting, quota enforcement, and payload transformations. It functions as a lightweight reverse proxy and can be deployed in both cloud-native and on-premises environments.

- **Tyk Dashboard:** A web-based administrative interface used to configure APIs, manage access control policies, view analytics, and define developer workflows. Available in the commercial distribution, it complements the gateway with visual management capabilities.

- **Developer Portal:** A customizable front-end interface for API consumers, enabling them to explore documentation, register applications, and manage subscriptions. It supports workflows based on API keys and token-based authentication.

[29]JSON-based templates that define the infrastructure and configuration for a deployment to Azure.

[30]A domain-specific language for declaratively deploying Azure resources.

[31]An open-source Infrastructure as Code (IaC) tool that allows you to define and manage infrastructure using configuration files.

[32]A workflow automation tool integrated into GitHub for continuous integration (CI) and continuous deployment (CD).

- **Tyk Pump:** A background service that extracts metrics and usage logs from the gateway and exports them to external monitoring and analytics platforms such as MongoDB[33], InfluxDB[34], Prometheus[35], or ElasticSearch[36].

Tyk supports a variety of communication protocols, including REST, GraphQL, and gRPC, and offers multiple authentication schemes such as API keys, JWT, HMAC, and OAuth 2.0. Access and transformation policies can be defined through JSON configuration files or managed via the graphical dashboard.

Configuration is fully declarative and can be automated using APIs, static files, or orchestration platforms. Integration with Kubernetes is supported through the Tyk Operator, enabling GitOps-style workflows for managing API deployments in containerized environments.

Although Tyk is not offered as a fully managed SaaS platform by default, it can be deployed in cloud environments using Helm charts or accessed through *Tyk Cloud*, the vendor-hosted version of the product.

Due to its flexibility, transparency, and automation-friendly design, Tyk is often chosen by organizations looking for an open and extensible alternative to proprietary API management platforms, particularly in DevOps-driven environments where customization and control are strategic priorities.

2.6.4 Layer7 API Management Platform

Developed by Broadcom, **Layer7 API Management** is a comprehensive enterprise-grade solution designed to control, secure, and monitor APIs across on-premises, hybrid, and private cloud environments. Layer7 is particularly focused on security, compliance, and integration within complex enterprise IT landscapes.

At the core of the platform lies the **Layer7 API Gateway**, a high-performance reverse proxy responsible for request routing, identity enforcement, message transformation, and deep security inspection. The gateway supports a wide range of authentication and authorization standards. It is configured and managed through the **Policy Manager**, a graphical interface that enables administrators to visually compose service-level policies using a rich set of configurable assertions[37].

Complementing the gateway is the **Layer7 Developer Portal**, which provides API consumers with a customizable interface to discover, test, and subscribe to available

[33]A popular, open-source NoSQL database that uses a document-oriented data model.

[34]A time series database developed for storage and retrieval of time series data.

[35]An open-source systems monitoring and alerting toolkit built for the cloud native world.

[36]An open-source, distributed search and analytics engine built on Apache Lucene, used to store, search, and analyze large volumes of structured and unstructured data in near real-time.

[37]In Layer7, an *assertion* represents a modular policy building block that performs a specific function, such as authentication, data transformation, or logging. Assertions can be combined visually to define complex service behavior without writing code.

APIs. It supports role-based access control, API key provisioning and management, interactive documentation, and integration with external analytics and monitoring systems. Deployment options include on-premises installation, hybrid proxy clustering, and private cloud environments.

The architecture of Layer7 is modular and layered. It features a dedicated processing layer for policy resolution, a database layer, typically based on MySQL, and a system layer that supports Linux-based deployment on virtual machines or physical appliances. The platform also supports integration with Hardware Security Modules (HSMs) to ensure secure storage and processing of cryptographic keys.

In summary, Layer7 offers the following key capabilities:

- A **graphical policy modeling environment** (Policy Manager), which replaces web dashboards or YAML/JSON-based infrastructure-as-code approaches;
- Deep protocol-level inspection and security enforcement features, specifically designed for sensitive or highly regulated environments;
- **Policy-as-Code** capabilities via Graphman[38], with native integration into Git-based automation pipelines;
- Multi-protocol support, including SOAP, REST, GraphQL, and gRPC, with advanced request and response transformation options;
- Built-in support for clustering, session persistence, and advanced load balancing strategies to ensure high availability and scalability;
- Enterprise grade audit logging, role based access control, and real-time policy enforcement mechanisms aligned with governance and compliance frameworks.

2.7 From DevOps to DevSecOps

The transition from DevOps to DevSecOps reflects the growing awareness that **security** must be embedded throughout the *software development lifecycle (SDLC)*, rather than treated as an isolated or final step. DevSecOps is defined as the integration of security practices into DevOps workflows, ensuring that security becomes a *shared responsibility* across development, operations, and security teams [15].

At its core, DevOps is guided by the CAMS model *Culture, Automation, Measurement, and Sharing*, a conceptual framework that emphasizes collaboration, toolchain integration, continuous improvement, and knowledge transfer [15]. Table 2.1 summarises how each pillar is extended when security is embedded by design.

[38]Graphman is a GraphQL-based management API introduced in the Layer7 Gateway to support Policy as Code and automation workflows.

Principle	DevOps	DevSecOps
Culture	Collaboration between Dev and Ops	Security specialists fully integrated in cross-functional teams
Automation	Build, test, release, deploy	Security scanning, policy enforcement and compliance-as-code
Measurement	Business and system indicators	Metrics on threats and vulnerabilities
Sharing	Sharing tools and knowledge	Sharing security issues and solutions

Table 2.1: DevOps vs DevSecOps Principles [15]

A key principle of DevSecOps is the concept of *"shifting security to the left"*, which means incorporating security controls as early as possible in the development pipeline [15]. This reduces the cost and complexity of fixing vulnerabilities later in the process.

Figure 2.9 illustrates a generic DevSecOps infinity loop in which every phase is surrounded by continuous security activities.

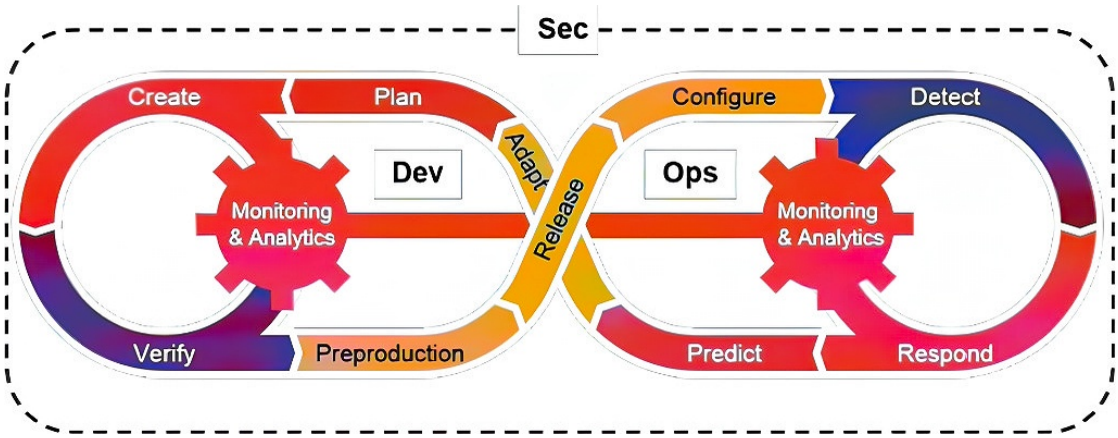


Figure 2.9: DevSecOps lifecycle with integrated security practices across the SDLC [15]

According to the lifecycle model proposed [15], the DevSecOps process is structured into eleven interconnected stages:

- **Create:** Emphasizes secure code development and team collaboration from the start.
- **Plan:** Involves defining security requirements, assessing risks, and regulatory compliance.

- **Adapt:** Ensures the ability to respond quickly to change while maintaining security controls.
- **Verify:** Conducts security testing and vulnerability assessments.
- **Preproduction:** Performs final checks before production deployment.
- **Release:** Handles secure and compliant continuous delivery to production.
- **Predict:** Introduces proactive detection of potential threats.
- **Respond:** Defines incident response strategies to minimize impact.
- **Detect:** Provides real-time monitoring of security events.
- **Configure:** Focuses on secure infrastructure and application configuration management.
- **Monitor:** Enables analytics-driven insights into system health and security posture.

In the context of cloud-native and API-centric architectures, integrating DevSecOps becomes especially important. APIs often serve as entry points to critical systems, and misconfigurations or insecure deployments can expose an organization to significant risk. Embedding security scanning, threat modeling, and policy validation within automated pipelines ensures that APIs are not only fast to deploy, but also secure by design.

2.7.1 Tooling Requirements and Platform Capabilities

The tools most widely adopted in the DevSecOps field aim to satisfy three intertwined objectives:

(i) *reduce the time required to move code from development to production by automating builds, tests, and approvals;*

(ii) *integrate security checks early in the software lifecycle to identify and fix issues before deployment;*

(iii) *maintain end-to-end visibility over all artifacts, source code, containers, configurations, by tracking versions, signatures, and approvals throughout the pipeline.*

In the *DoD Enterprise DevSecOps Fundamentals* [16], the U.S. Department of Defense (DoD) organizes tools into functional categories, rather than prescribing specific vendors or products. This approach enables each software factory to assemble a tailored toolchain suited to its architecture, mission, and classification level.

Each software factory must implement a standard set of *common* tools that span all phases of the development lifecycle. As the document notes:

> *"Use of the word common is indicative of a class of tooling; it does not, nor should it be construed that the same tool must be used across every implementation."* [16, p. 14]

This allows for architectural flexibility while maintaining alignment with a baseline set of capabilities.

The DoD model separates a central **DevSecOps platform** that handles provisioning, compliance rules, and secure CI/CD pipelines ,from mission-specific **software factories** that build and deploy software on it. This separation promotes standardization and reuse while enabling delivery teams to operate with controlled autonomy.

A core principle of DevSecOps is the tight integration of development (Dev), operations (Ops), and security (Sec) activities across the pipeline. Accordingly, the toolchain must support the following:

- **Automated testing**: includes unit, integration, regression, and performance tests automatically executed within continuous integration and delivery pipelines (CI/CD).
- **Security scans**: This category encompasses multiple types of automated analysis. Key activities include:
  - **SAST (Static Application Security Testing):** Analysis of source code or binaries for security vulnerabilities without executing the program.
  - **DAST (Dynamic Application Security Testing):** Inspection of running applications to detect vulnerabilities through simulated attacks.
  - **IAST (Interactive Application Security Testing):** A hybrid approach combining SAST and DAST elements by analyzing applications during runtime while observing internal behavior.
  - **Component Analysis and Hardening:** Validation of the Software Bill of Materials (SBOM)[39] and execution of post-deployment hardening processes.
- **Logging and telemetry**: refers to the centralized collection of logs, metrics, and traces, which are often analyzed through SIEM[40] and SOAR[41] systems for effective threat detection and forensic auditing.

[39] A Software Bill of Materials (SBOM) is a structured inventory of components and dependencies used in a software system.

[40] Security Information and Event Management (SIEM) platforms aggregate and correlate security data to detect threats in real time.

[41] Security Orchestration, Automation and Response (SOAR) tools help coordinate, automate, and respond to security incidents across systems.

- **Artifact provenance**: involves tracking the origin and history of software artifacts, ensuring each component is versioned, digitally signed, and subject to gated promotion workflows before reaching production.
- **Zero-Trust enforcement**: implements security measures based on the principle that no user, system, or process should be trusted by default. This includes mutual TLS (mTLS)[42], deny-by-default policies, and runtime security constraints integrated directly into the pipeline.

These tools and capabilities are organized according to the **ten-phase lifecycle** defined by the DoD: *Plan, Develop, Build, Test, Release, Deliver, Deploy, Operate, Monitor,* and *Feedback* [16]. Each phase is associated with specific tool classes and security practices, ensuring that protection, compliance, and traceability are embedded continuously throughout the software delivery process.

Table 2.2 summaries the key tool classes and security-related capabilities associated with each of these phases.

[42]Mutual TLS (mTLS) ensures that both the client and the server authenticate each other using digital certificates, providing two-way encrypted communication.

Phase	Tool Categories	Security Capabilities
Plan	Collaboration platforms, Threat modeling tools	Risk assessment, security requirements management, threat modeling
Develop	Secure IDEs, code repositories, code review tools	Inline SAST, secret detection, enforcement of secure coding policies
Build	CI pipelines, dependency checkers	SBOM creation and validation, signature verification, scanning of third-party libraries
Test	Automated test frameworks, DAST/IAST tools	Security regression testing, vulnerability identification
Release	Release management tools, digital signing services	Release integrity, provenance tracking, security approval gates
Deliver	Artifact repositories (e.g., Iron Bank, Azure Artifacts)	Integrity verification, controlled distribution
Deploy	IaC tools (e.g., Terraform, Ansible), orchestrators (e.g., Kubernetes)	Policy-as-Code, environment hardening, automatic rollback triggers
Operate	Runtime managers, service meshes, autoscalers	Secure runtime configuration, traffic isolation, enforcement of runtime policies
Monitor	Logging and monitoring stacks, SIEM/SOAR systems	Anomaly detection, behavioral analytics, compliance monitoring
Feedback	Metrics dashboards, issue tracking platforms	DevSecOps KPIs[a](e.g., DORA,[b] DoD4[c]), feedback for continuous improvement

[a] **DevSecOps KPIs** include metrics such as deployment frequency, vulnerability count, and remediation time.

[b] **DORA** metrics include deployment frequency, lead time for changes, mean time to restore, and change failure rate.

[c] **DoD4** is a metric model introduced by the U.S. Department of Defense, focusing on delivery speed, system stability, mission value, and cyber-resilience.

Table 2.2: Tool categories and related security capabilities across DevSecOps phases.

Chapter 3

Project Development and Implementation

3.1 Overview of Today's DevSecOps Tools

Modern DevSecOps practices rely on combining specialised tools rather than adopting a single, all-in-one solution. This modular approach enables teams to cover the full application lifecycle, from code analysis to runtime protection, using components that are easier to integrate, update, and replace as needs evolve. This trend is confirmed by a 2025 comparative analysis of DevSecOps platforms [17], which shows that pipelines are typically composed of modular tools, each of which solves a specific problem. To understand the comprehensive security approach required in modern API management, this analysis follows the natural progression of software delivery, examining tools and practices across four critical phases of the DevSecOps pipeline. Each phase addresses distinct security challenges and employs specialized tools designed for that stage of the lifecycle.

Develop and Build: Analyzing Static Artifacts

This stage focuses on analyzing static assets, which includes everything that can be checked before the application is actually run. The strategic objective is to identify and remediate vulnerabilities at their point of origin, long before they can escalate into production threats. This approach embodies the core principles of the shift-left philosophy, bringing security considerations directly into the development workflow.

The security of an application begins with its very blueprint, the API specification itself. Before any code is written, linters like **Spectral** can validate OpenAPI files. They enforce consistent naming, proper versioning, and fundamental security practices, providing a solid foundation from the start [18].

Once development starts, the focus shifts to the source code. Static Application

Security Testing (SAST) tools scan this first-party code for flaws. A platform like **SonarQube** offers deep and continuous checks to keep code quality high over time [19]. For quick, targeted checks inside a CI pipeline, a lightweight tool like **Semgrep** often excels, using simple YAML rules to spot dangerous patterns [20].

A particularly critical vulnerability, however, requires a specialist: the hardcoded secret. This is where a tool like **TruffleHog** comes in. Its sole purpose is to dig through a project's entire history to hunt for anything that looks like an API key or a private token [21].

But an application's security is not just about its own code. It also depends on the vast ecosystem of third-party libraries it consumes and the infrastructure it will run on. Software Composition Analysis (SCA) tools address the library risk. **Snyk** provides broad scans of dependencies and container images, while a faster tool like **Trivy** is often used for quick validation in CI workflows.

Similarly, the infrastructure definitions must be validated. Since cloud environments are defined as code, scanners like **Checkov** are essential for parsing Terraform or Kubernetes files to find insecure configurations, such as open network ports or weak access roles.

Once all checks are passed, it is essential to protect the artifacts[1] created during the pipeline. This is where **Cosign** is used:

`Cosign` is a supply chain security tool designed to ensure the *integrity, provenance, and authenticity* of software artifacts and other build outputs [22].

By default, Cosign uses the `ECDSA-P256` [2] signature scheme and the `SHA-256` hash function, aligning with the OCI registry specification[3] Signatures are linked to artifacts via their SHA-256 digest[4] and stored as separate objects in the registry, enabling signature discovery and verification without altering the original image.

Cosign supports both **key-based** and **keyless** signing workflows:

- In *key-based mode*, developers generate a persistent key pair locally (e.g., via `cosign generate-key-pair`) and use the private key to sign the artifact digest. The resulting signature is uploaded to an OCI registry and optionally recorded in *Rekor*, a tamper-evident transparency log that facilitates auditability and accountability.

---

[1] **Artifact:** Any file produced by the build or deployment pipeline, such as container images, binaries, reports, or API specifications, that is subsequently deployed or archived.

[2] ECDSA-P256 refers to the Elliptic Curve Digital Signature Algorithm (ECDSA) that uses the P-256 curve, also known as NIST P-256, for public key cryptography.

[3] An OCI registry is a content-addressable storage system that follows the Open Container Initiative Distribution Specification. It is used to store and distribute container images and related artifacts by referencing their cryptographic digests.

[4] **Digest:** A cryptographic hash (e.g., SHA-256) of a file or artifact, used to uniquely identify its content. If the file changes, even slightly, its digest also changes, making it useful for integrity verification.

- In *keyless mode*, Cosign generates an ephemeral in-memory key pair and obtains a short-lived X.509 certificate from Sigstore's *Fulcio* certificate authority, binding the public key to the developer's *OIDC identity* (e.g., GitHub Actions or Azure Pipelines). The signature and certificate are always recorded in *Rekor*, which is backed by a Merkle tree and guarantees immutability.

Each signature is cryptographically bound to the artifact's digest and, in keyless workflows, also to the signer's identity and CI context. Verification can be performed using the Cosign CLI ensuring that only trusted, signed artifacts are deployed into production environments.

By leveraging standardized registries and public transparency logs, Cosign provides a foundation for tamper-resistant, verifiable, and identity-bound artifact delivery pipelines.

This guarantees:

- **Provenance** : the artifact was built by a specific CI workflow;
- **Integrity** : the artifact has not been tampered with post-build;
- **Non-repudiation** : the signature can be traced to a verifiable identity.

Pre-production: Staging Validation

The pre-production stage is a critical phase where the running application is tested before final deployment. This typically involves Dynamic Application Security Testing (DAST) to identify vulnerabilities that are not visible in static code.

A common approach is to employ a general-purpose DAST scanner. A tool like **OWASP ZAP**, for example, can automatically scan an application's endpoints to find a broad range of common web vulnerabilities and security misconfigurations [23].

However, for applications heavily reliant on APIs, a deeper level of testing is often necessary to cover risks specific to their architecture. This is where specialized tools such as **APIsec** offer significant advantages. Unlike generic scanners, platforms like APIsec are designed to ingest an API's OpenAPI specification, allowing them to understand its intended logic and data models.[24]. Based on this context, they can generate targeted tests for complex flaws, such as those listed in the **OWASP API Security Top 10**, including Broken Object Level Authorization (BOLA)[5]

[5]**BOLA (Broken Object Level Authorization)** is a common security vulnerability in APIs, where access controls are not properly enforced at the object level. This allows attackers to manipulate object identifiers in requests, gaining unauthorized access to data they shouldn't retrieve.

Deploy: Policy and Runtime Enforcement

The deploy phase ensures that only secure and compliant artifact[6] reach production. This involves both *pre-deployment policy enforcement* and *post-deployment runtime protection*.

CI/CD Platforms. Modern CI/CD tools provide protected environments and conditional stages to control where and how code is released. Role-based permissions and approval gates help prevent unauthorized changes in sensitive environments. For instance, in Azure DevOps, *Environments* support *Approvals and Checks*, allowing organizations to introduce manual approval steps or automated verifications before promoting artifacts to the next stage of the pipeline, such as a production deployment.

Infrastructure-as-Code Policies. Tools like *Spacelift* integrate with IaC workflows to evaluate policy compliance before provisioning. By leveraging the *Open Policy Agent (OPA)* and its *Rego* language, Spacelift can automatically detect violations such as open network ports, insecure storage accounts, or API gateways provisioned without TLS. These checks ensure that deployment environments for APIs are secure by design. In the Azure ecosystem, the same principle is realized through *Azure Policy*, which can validate ARM or Bicep templates before deployment and, via the *Azure Policy for Kubernetes* add-on, enforce admission control directly in AKS clusters. This prevents non-compliant workloads or insecure configurations from ever reaching production.

Admission Control in Kubernetes. When APIs are deployed in Kubernetes clusters, *Gatekeeper* extends OPA to enforce policies directly at the admission level. Rules expressed as *Custom Resource Definitions (CRDs)* can block configurations that violate security requirements, such as exposing APIs without authentication or missing encryption policies.

API Gateway and Edge Protection. Once deployed, APIs require strong runtime enforcement. *Azure API Management* provides built-in policy enforcement mechanisms such as authentication, authorization, rate limiting, request validation, and logging. At the network edge, services like *Cloudflare API Shield* complement this by applying Web Application Firewall (WAF) rules, mutual TLS authentication, and DDoS mitigation, ensuring that external traffic is filtered and controlled before reaching internal workloads.

Access Control and Network Restrictions. Zero-trust principles can be applied within the cluster or infrastructure to limit API exposure. Micro-segmentation tools such as *Calico* restrict communication between services, ensuring that sensitive backend APIs are only reachable through approved gateways. Additionally, Kubernetes-native mechanisms such as *Role-Based Access Control (RBAC)* and *Pod Security Standards (PSS)* enforce the principle of least privilege across workloads and users. In cloud

[6] In this context, the term "artifact" refers to any output of the CI/CD process that is destined for deployment, such as container images, API definitions, infrastructure plans, or runtime configurations.

environments like Azure, network-level controls such as *Virtual Network segmentation* and *private endpoints* complement container-level policies.

Operate: Observability and Continuous Testing

Once APIs are live, continuous monitoring is essential for detecting anomalies and ensuring compliance with performance and security requirements.

Prometheus collects metrics such as request volume, error rates, and response times from API services and Kubernetes nodes, while *Grafana* provides visual dashboards and alerts for deviations such as sudden spikes in 5XX errors or authentication failures. This observability layer enables early detection of potential denial-of-service attacks or misconfigurations.

From an API-specific perspective, platforms like *Azure API Management* also provide native analytics and integration with *Azure Monitor*, delivering insights into request patterns, latency, and consumption per client or product. These capabilities not only support operational resilience but also reinforce security by highlighting suspicious traffic patterns or abuse of exposed endpoints.

This landscape of tools illustrates the essential capabilities for a secure pipeline. The following section will now focus on the project's specific architecture, showing how these same principles were realized within the integrated environment of Microsoft Azure.

The entire solution was developed within the Microsoft Azure ecosystem, leveraging Azure DevOps for orchestration. A key decision in this implementation was the use of a self-hosted agent to ensure a controlled and private testing environment.

3.1.1 Core Infrastructure and Execution Environment

The foundation of the CI/CD process is a virtual machine (VM) hosted in Azure, which runs a dedicated **self-hosted agent**. This approach was chosen over Microsoft-hosted agents to allow for the local installation and configuration of specific security tools, such as SonarQube and the APIsec agent, that both runs in a Docker container, creating an isolated and consistent testing environment.

To ensure secure and private communication, the agent's VM was placed in a dedicated Azure Virtual Network (VNet). This VNet was then connected to the VNets of the Azure API Management (APIM) instances using **VNet Peering**. The VNet peering enables resources in two separate virtual networks to communicate with each other as if they were in the same network. All traffic between the peered VNets is routed through the Microsoft private backbone infrastructure, not the public internet. This design choice provides significant benefits in terms of security and performance, establishing a low-latency, private network path between the CI/CD agent and the API gateways.

Two distinct APIM instances were provisioned to represent separate environments:

- **Development (dev):** Used for initial deployments, testing, and security scanning.
- **Production (prod):** The final, stable environment for released APIs.

Secrets management is centralized using **Azure Key Vault**. The Key Vault stores critical secrets, such as the subscription keys or credentials associated with the APIs themselves. This practice is fundamental to security, as pipelines reference secrets by name rather than exposing them in code. This allows for seamless promotion between `dev` and `prod` environments by simply referencing the same secret name, while the underlying value is managed securely within the Key Vault.

For logging and auditing, an **Azure Blob Storage** account was configured with a **WORM (Write-Once, Read-Many)** policy. This creates an immutable storage container where access logs and security scan reports are automatically uploaded by the agent. Access control is tightly configured to grant write permissions only to the agent's Managed Identity, ensuring the integrity of the audit trail.

3.1.2 Azure DevOps Project and Pipeline Orchestration

The orchestration of the entire workflow is managed within an Azure DevOps project. Secure, authenticated integration with Azure resources is achieved through **Service Connections**. A Service Connection acts as a secure wrapper within Azure DevOps that stores the credentials needed to connect to an external service like Azure.

For this project, the connection to the Azure subscription is based on an **App Registration** in Microsoft Entra ID . This process involves:

1. Creating an App Registration in Entra ID, which acts as the application's identity.
2. This registration generates a corresponding **Service Principal**, which is the security identity that can be assigned permissions on Azure resources (e.g., a "Reader" role on a resource group).
3. The Service Connection in Azure DevOps is then configured with the Service Principal's credentials.

When a pipeline runs, it uses this Service Connection to authenticate with Azure, effectively acting with the permissions granted to the Service Principal. This provides a secure and auditable mechanism for the pipeline to manage Azure resources.

The CI/CD process was intentionally designed for modularity and reusability, structured as a master pipeline coordinating five specialized "worker" pipelines. This flexibility is further enhanced by extensive use of:

- **Runtime Parameters:** Each pipeline is parameterized, allowing for dynamic inputs during execution (e.g., specifying the target environment or enabling/disabling certain checks).

- **Library Variable Groups:** A core design principle is the strict separation of configuration from pipeline logic, a practice that enhances both security and reusability. This is achieved through Azure DevOps Libraries, with variable management structured around two distinct categories:
  - **Configuration Parameters:** Non-sensitive variables that define the operational context are stored in standard variable groups. These include resource identifiers and environment-specific settings (e.g., for `dev` and `prod`), allowing a single pipeline template to be dynamically adapted to different environments.
  - **Centralized Secret Management:** Sensitive data is handled through a more robust mechanism that leverages a direct integration between secure variable groups and **Azure Key Vault**. This approach provides a secure-by-design method for managing credentials such as API tokens and the cryptographic keys and passwords. Secrets are fetched securely at runtime, ensuring they are never hard-coded or exposed in logs, in adherence with security best practices.

3.1.3 The DevSecOps Workflow: An Overview of the Pipelines

The core logic is driven by a master pipeline that orchestrates the execution of five specialized pipelines in a coordinated sequence. The master pipeline manages the overall flow using control flags and passes parameters to the individual worker pipelines, allowing for a highly customizable DevSecOps process.

The workflow is composed of the following pipelines:

1. **Extractor:** Adapted from the official Azure APIOps repository[25], this pipeline extracts the current configurations of deployed APIs from a target APIM instance. It then translates these configurations into code and submits them to the Azure Repos via a pull request (PR), enabling a GitOps-style management approach.

2. **Static Analysis:** This pipeline focuses on pre-deployment security checks. It integrates multiple tools to scan the API configuration files and source code for vulnerabilities: **SonarQube Scanner** for static code analysis, **Spectral** for linting OpenAPI specifications, and **TruffleHog** for detecting hardcoded secrets. After the checks, **Cosign** is used to digitally sign the reports generated during the security scans.

3. **Publisher:** Responsible for the deployment process, this pipeline takes the validated API configurations from the repository and publishes them to the target APIM environment (`dev` or `prod`).

4. **Reachability Test:** Immediately following a deployment, this pipeline performs a basic health check to ensure the newly deployed API endpoints are reachable

and responsive. In the event of a failure, it triggers an automatic rollback to the previously known stable version. This versioning is managed through a combination of Git tagging in the repository and the use of Named Values within APIM.

5. **Dynamic Test:** This pipeline executes dynamic application security testing (DAST) against the running API. It leverages the **APIsec** agent installed on the self-hosted VM to perform in-depth security scans, identifying runtime vulnerabilities that are not visible during static analysis.

3.2 Project Structure and Pipelines implementation

3.2.1 Extractor Pipeline

The **Extractor** pipeline serves as the critical bridge between the live Azure API Management (APIM) instance and the Git repository, automating the process of translating deployed API configurations into code. Its primary goal is to enable a **GitOps** workflow, where the repository becomes the single source of truth for all API definitions.

The process begins with a high degree of flexibility, allowing the user to specify a target environment, either `dev` or `prod`, through a runtime parameter. Based on this choice, the pipeline dynamically selects the appropriate APIM instance and resource group to connect to. The core of its operation relies on the official **Azure APIOps** extractor tool. This tool connects to the target APIM instance and meticulously translating its current state, including APIs, operations, policies, and named values, into a structured directory of human-readable configuration files.

As an immediate quality check on the extracted assets, the pipeline integrates **Spectral** to lint the OpenAPI specifications. This step ensures that the generated configurations adhere to predefined quality and style rules before they are even proposed for inclusion in the repository. Once the extraction and linting are complete, all the generated files are packaged into a pipeline artifact. For enhanced visibility, a summary of the extracted directory structure is automatically generated and displayed in the pipeline's summary tab, offering a clear overview of the changes.

With the API configurations captured as code, the pipeline's second stage initiates the GitOps workflow. Instead of pushing changes directly to the main branch, it follows a safer, more controlled practice. The pipeline automatically creates a new temporary branch, commits the extracted files, and, as shown in Figure 3.1, then opens a **pull request** (PR) to merge these changes into the main branch. This crucial step enforces a formal review process, allowing team members to inspect and approve any modifications to the API landscape before they become part of the official codebase.

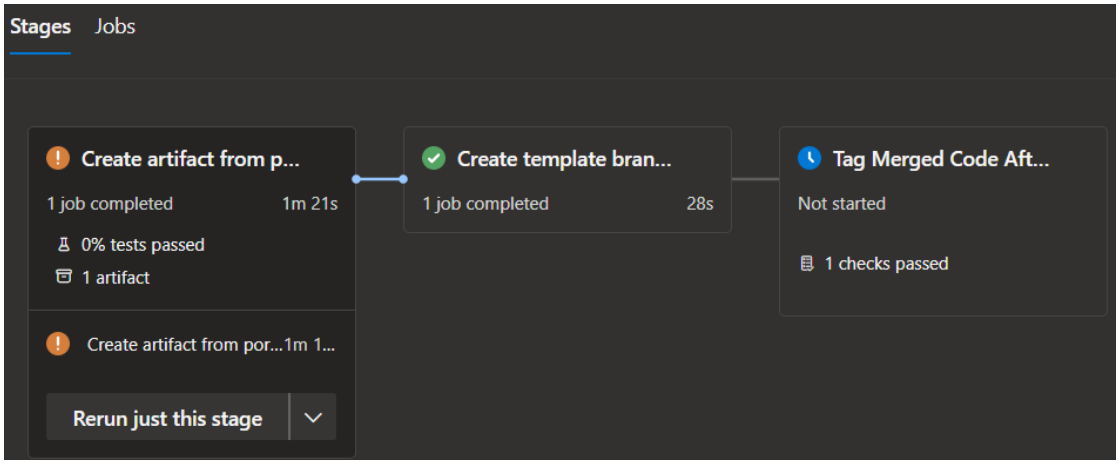


Figure 3.1: Extractor Pipeline execution stages: (1) Create artifact from portal extraction, (2) Create template branch with pull request, and (3) Tag merged code after approval.

The final stage of the workflow executes only after the pull request has been manually approved and merged. This stage acts as a final confirmation step, creating an immutable, versioned snapshot of the API configurations. To determine the correct version, it reads a specific named value from the extracted `apim-version`. It then applies this version as a **Git tag** to the main branch of the repository, officially marking a new, stable release of the API configurations. This tagging mechanism is fundamental for the subsequent deployment and rollback pipelines.

3.2.2 Static Security Analysis Pipeline

This pipeline checks the code for security issues and keeps a safe record of the findings. The workflow is logically divided into two distinct phases: a primary security scanning phase and a final archival phase for immutable storage. The process begins with the core analysis tasks. The job starts by performing a clean checkout of the source code. Immediately following this, it establishes a secure context by connecting to **Azure Key Vault** to fetch the necessary cryptographic secrets. These secrets, including the keys required for digital signing, are loaded into the pipeline's environment, a practice that avoids hard-coding sensitive information. With the credentials loaded, a setup script is executed to install the necessary toolchain for the upcoming scans.

The first analysis tool to run is **SonarQube**. The pipeline conducts a Static Application Security Test (SAST) on the designated source code, publishing the results and quality gate status back to the SonarQube server for review. After the first scan, the pipeline carries out checks to ensure the integrity of the reports, which is a top priority.

The pipeline then runs the tool **Spectral**, which validates the OpenAPI specification files and produces a report. A crucial element of this design is that immediately after

the report is generated, it is digitally signed. The pipeline then verifies this signature to ensure the signing process was successful.

This same sign-and-verify pattern is repeated for the next tool, **TruffleHog**, which scans the project for hard-coded secrets and generates its own report.

Once all scans are complete, the generated reports and their corresponding digital signatures are packaged into a single, timestamped archive. This archive is then published as a pipeline artifact. As a final security measure within this phase, the cryptographic key files are securely erased from the agent's filesystem to prevent any potential exposure.

This leads to the final archival stage, which is dependent on the successful completion of the first. First, the artifact with the security reports is downloaded. Using a managed identity for authentication, the archive is uploaded to the pre-configured `WORM` (Write-Once, Read-Many) storage container. This keeps the reports in an immutable state, creating a permanent and auditable record of the analysis. After the upload is confirmed, the downloaded artifact is removed from the agent.

3.2.3 Publisher Pipeline

The **Publisher** pipeline is the operational counterpart to the Extractor, designed to automate the deployment of API configurations from the Git repository to the live Azure API Management (APIM) instances. It completes the GitOps cycle by translating the version-controlled code back into a functioning API landscape.

The workflow is initiated by selecting a target environment, such as development or production. This crucial choice dynamically configures the pipeline to connect to the appropriate APIM instance and its corresponding resources. The process is organized into distinct, environment-specific stages, allowing for independent deployment workflows and the potential integration of manual approval gates through Azure DevOps Environments.

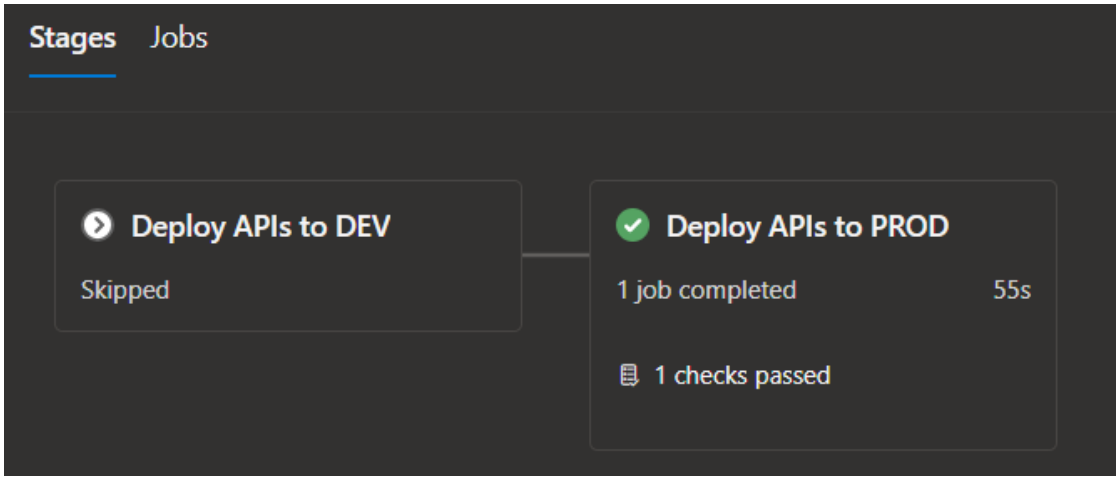


Figure 3.2: Example of environment selection in the Publisher pipeline, where the deployment was directed to the PRODUCTION environment.

The core deployment logic systematically processes the contents of the repository. First, it iterates through each API definition. For each one, it reads the necessary configuration files to determine its structure and backend service endpoint. The pipeline then imports the API into the target APIM instance and associates it with the correct **environment-specific product**.

Next, the pipeline handles the deployment of named values. A critical security feature of this process is its **ability to manage secrets**. If a named value is designated as sensitive, its actual value is not read from the repository. Instead, the pipeline securely connects to **Azure Key Vault** to retrieve a reference to the secret, ensuring that sensitive information is never exposed in the codebase. As a final, critical step, the pipeline **tags the deployed environment with a specific version identifier**, passed by runtime parameter. By updating a specific named value `apim-version` within APIM, this step ensures clear traceability for each deployment and provides a solid foundation for auditability and future rollbacks.

3.2.4 Reachability and Rollback Pipeline

This pipeline serves as a critical post-deployment validation mechanism, designed to ensure service continuity by verifying the health of deployed APIs and providing an automated rollback capability in case of failure. Its workflow is divided into two distinct, conditionally executed jobs: a primary validation job that performs reachability tests, and a secondary rollback job that acts as an automated safety net.

The process begins with the validation job, which is manually triggered against a specific target environment, either development or production. The pipeline first determines the precise version of the APIs under test, either by automatically detecting the current **Git tag** of the repository or by using a version explicitly provided at

runtime. The core of this job is a test runner that systematically iterates through a dedicated directory containing test definitions. For each API, it executes a series of pre-defined HTTP requests designed to simulate real-world interactions. The process is sophisticated enough to handle chained API calls, automatically extracting authentication tokens from one response to use in subsequent requests.

A test is considered successful only if the API endpoint returns an expected HTTP status code. If any API fails to respond correctly, the entire validation job is marked as failed. Regardless of the outcome, the pipeline meticulously logs the results of every test and generates artifacts that clearly distinguish between the APIs that passed and those that failed. These artifacts are crucial for providing immediate diagnostic information to developers, allowing them to quickly identify the source of the failure.

The second job, which handles the rollback, is designed to execute *only* if the preceding validation job fails. This conditional execution is the cornerstone of the pipeline's automated recovery strategy. Upon activation, this job's objective is to restore the entire environment to its last known-good state. It checks out the source code from a previously designated stable **Git tag**, which represents the last known-good state of the system.

Using the code from this stable tag, the pipeline performs a complete rollback, redeploying all APIs and their configurations as defined in that version. This holistic approach ensures that the environment returns to a fully consistent and predictable state, eliminating any problematic changes introduced in the failed deployment.

As a final, crucial step, the pipeline updates the environment's version identifier within API Management to reflect the stable tag that was just restored. This action ensures that the recorded version of the environment accurately matches its deployed state, maintaining the integrity of the audit trail and providing clear traceability for the recovery action.

3.2.5 Dynamic Security Analysis Pipeline

The dynamic security analysis pipeline performs Dynamic Application Security Testing (DAST) to check API security while the system is in a running state. The pipeline accepts various parameters at runtime to specify which environments and APIs should be tested, making it possible to run the same security checks across development and production stages.

When the pipeline starts, it retrieves the necessary credentials from Azure Key Vault, including the **APIsec** platform token and **Cosign** signing keys. This approach keeps sensitive data secure and prevents it from being exposed in the pipeline's code. The pipeline then connects to Azure API Management to get the latest OpenAPI specifications for each target API. These specifications inform the security scanner which endpoints exist, what methods they support, and how they are expected to behave during testing.

Using these specifications, the pipeline runs security scans through the **APIsec** platform against the live API gateway. APIsec was selected for this role because it

uses AI to find more vulnerabilities, provides an easy-to-use dashboard for reviewing results, and includes advanced features like Role-Based Access Control (RBAC) and Broken Object Level Authorization (BOLA) detection.

Since dynamic security scans can be time-consuming, the pipeline uses a **polling** approach to manage this process. It periodically queries the APIsec API to check the scan's status, waiting until the platform reports that all tests are complete.

When the scan finishes, the pipeline downloads a report containing all identified vulnerabilities. This report is processed by an automated security gate that applies a configurable set of rules to assess risk.

The security gate is designed to be highly flexible, allowing the failure logic to be defined through a primary parameter, `SECURITY_GATE_MODE`, which supports four distinct modes:

- **`cvss_only`**: The pipeline fails only if a vulnerability exceeds a predefined CVSS[7] score threshold.
- **`severity_only`**: Failure is determined exclusively by severity levels ('Critical', 'High', 'Medium', 'Low'), regardless of the CVSS score.
- **`flexible`**: The pipeline fails if **any** of the configured rules are violated. This includes exceeding the CVSS threshold, detecting specific severity levels, or identifying vulnerability categories considered critical (e.g., `authentication`, `injection`, `Denial Of Service`, `Information Leak`).
- **`strict`**: This mode applies all rules from the `flexible` mode and introduces stricter checks, such as a lower CVSS threshold for high-risk HTTP methods (e.g., 'GET', 'POST', 'PUT', 'DELETE', etc).

The security gate evaluates each detected vulnerability against the configured rules in a specific order of precedence.

This hierarchical logic ensures that the most critical findings trigger a failure immediately, without needing to evaluate subsequent, less critical rules. The evaluation process is as follows:

1. **Critical Vulnerability Type Check:** First, the gate checks if the vulnerability's name matches any of the keywords listed in the `CRITICAL_VULNERABILITY_TYPES` parameter. This rule acts as an immediate failure condition in `flexible` and `strict` modes. If a match is found, the vulnerability fails the gate without any consideration of its CVSS score or severity level. This ensures that certain classes of vulnerabilities are never permitted, regardless of their calculated risk score.

[7]The Common Vulnerability Scoring System (CVSS) is a framework for assessing and communicating the severity of software vulnerabilities. It assigns a numerical score, typically from 0.0 to 10.0, to indicate the level of risk, with higher scores representing more critical vulnerabilities.

2. **Severity Level Check:** If the vulnerability is not flagged as a critical type, the gate then checks its severity level. This check is active in `severity_only`, `flexible`, and `strict` modes. If the severity is found in the list defined by the `FAIL_ON_SEVERITY_LEVELS` parameter, the vulnerability fails the gate.
3. **CVSS Score Check:** Finally, if the vulnerability passes the first two checks, its CVSS score is evaluated. This check is active in `cvss_only`, `flexible`, and `strict` modes. The gate determines the appropriate threshold to use:
   - By default it uses the general threshold defined in `CVSS_SCORE_THRESHOLD`.
   - However, if the `SECURITY_GATE_MODE` is set to `strict` and the vulnerability's HTTP method is listed in the `STRICT_METHODS` parameter, the gate applies the more stringent (lower) threshold from `STRICT_METHODS_CVSS_THRESHOLD`.

Figure 3.3 provides a practical example of this hierarchical logic in action. It shows the security gate's output when running in `strict` mode. The gate fails because the vulnerability's severity ("Critical") is included in the `Fail on Severities` list. This demonstrates the hierarchical evaluation logic, where the severity check takes precedence and causes failure before the CVSS score check is evaluated.

```
================================================================
Running Enhanced Security Gate Analysis
Mode: strict
================================================================
Configuration:
  CVSS Threshold: 7
  Strict Methods CVSS: 5
  Fail on Severities: Critical High
  Strict Methods: GET POST DELETE
  Critical Vuln Types: authentication authorization injection

Analyzing: 0198f011-91eb-78d3-8d9d-a1be3752c142 (Score: 9.0, Severity: Critical, Method: get)
 SEVERITY EXCEEDED: Critical is in fail list
   FAILED SECURITY GATE
================================================================
 SECURITY GATE FAILED for API: test
================================================================
```

Figure 3.3: An example of the security gate failing in `strict` mode.

If the vulnerability's score is greater than or equal to the selected threshold, it fails the gate. This advanced logic allows for customized risk assessment based on context.

For example, the system can be configured to automatically fail when critical "authentication" vulnerabilities are found, regardless of their CVSS score, or to apply different risk thresholds depending on the HTTP method being analyzed.

While using CVSS provides a reliable and industry-standard vulnerability assessment, the system integrates multiple criteria to avoid relying solely on it.

If the scan results show vulnerabilities that exceed the defined limits, the pipeline automatically fails and stops the deployment process. This automatic quality gate ensures that APIs with serious security problems cannot be deployed without manual review.

The pipeline follows the same process for handling evidence as the static analysis pipeline. Each vulnerability report is digitally signed using **Cosign** to guarantee its integrity and authenticity. The signed report and its corresponding signature are packaged with a timestamp and uploaded to a central **WORM storage** for immutable retention.

3.2.6 The Master Orchestrator Pipeline

This Master Pipeline acts as the central control system for the entire DevSecOps workflow. Its primary role is not to perform operational tasks itself, but to orchestrate the sequence of the specialized child pipelines using the Azure DevOps REST API. It connects everything from code extraction to security testing and final deployment, passing the required parameters to each component.

The key to its adaptability lies in its use of distinct boolean flags, namely `waitFor_Static_SecurityScan` and `waitFor_Dynamic_SecurityScan`. These flags allow the operator to choose between two fundamentally different execution models, providing a powerful balance between rigorous governance and development agility.

- **Integrated Deployment with Manual Approval:**

  When the `waitFor_StaticSecurity_Scan` flag is set to `true`, the pipeline executes a fully integrated and controlled workflow. It triggers the static security scan and then actively monitors its progress by **polling** the API until the scan is complete. If the scan succeeds, the orchestrator proceeds to the **Publisher** stage. Crucially, this stage is configured to target a specific Azure DevOps Environment, which is protected by a manual approval check. This serves as a critical human checkpoint, requiring an authorized user to explicitly approve the deployment before the scanned APIs are published.

- **Decoupled Informational Scan:**

  When the flag is set to `false`, the pipeline's behavior shifts to provide an informational scan without forcing a subsequent deployment. It initiates the static scan, but after doing so, the orchestrator's main workflow for this run concludes. It does not proceed to the publishing stage. This approach effectively decouples the scanning phase from the deployment phase. It empowers the development team to review the findings on their own timeline and decide independently when, or if, they wish to proceed with a manual publication.

The same logic applies to the dynamic scan flag. It allows the final DAST scan to act either as a blocking step that stops the pipeline on failure or as a non-blocking step that only provides information. This behavior is controlled through simple parameters, making the pipeline flexible enough to support both strict release workflows and fast, iterative development.

This design choice allows teams to adapt the pipeline's behavior to their specific needs, whether they require strict governance with manual approvals or prefer a more agile, iterative development process.

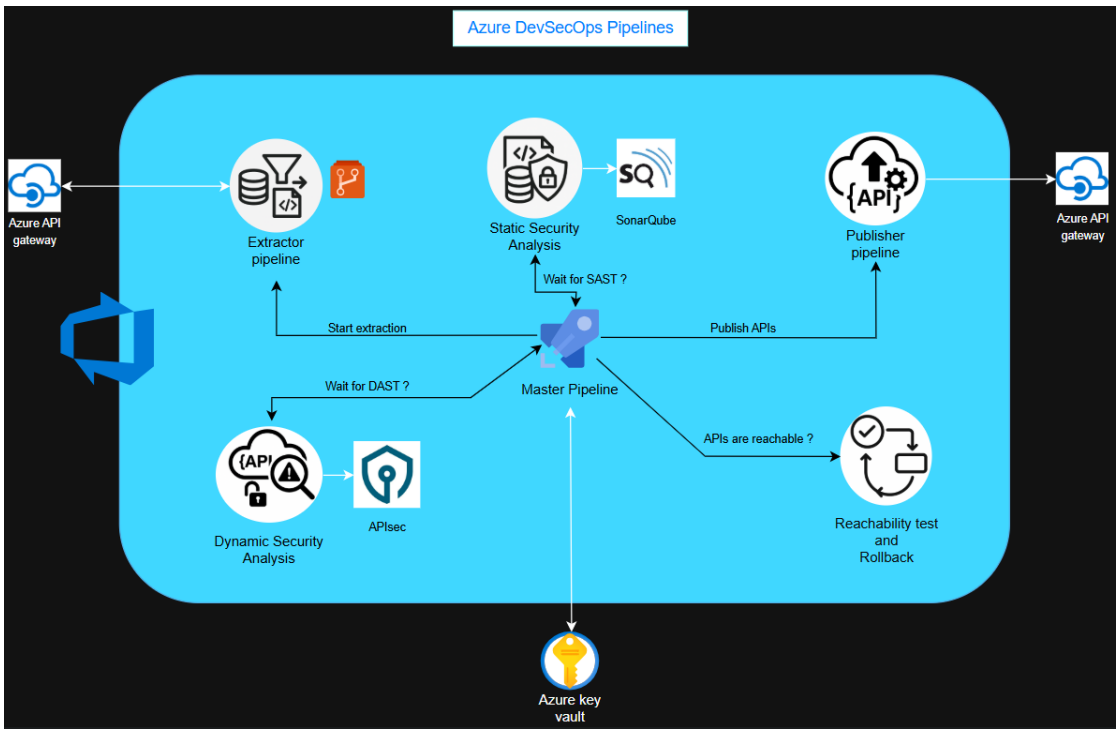


Figure 3.4: Pipelines overview: the master pipeline orchestrates the execution of the five specialized pipelines

Chapter 4

Testing and Evaluation

4.1 Testing Environment

4.1.1 OWASP and crAPI

To evaluate the effectiveness and real-world applicability of the proposed DevSecOps pipeline, this thesis adopts the ***completely ridiculous API*** (crAPI) [26] as a reference test environment. The crAPI project is a purposely vulnerable API developed under the umbrella of the OWASP Foundation. It simulates a modern microservice-based application composed of insecure components and flawed business logic, making it a valuable resource for assessing the security tooling integrated into the CI/CD lifecycle.

The repository includes a fully containerized architecture, typically deployed using Docker Compose, exposing RESTful endpoints across multiple domains such as user management, authentication, vehicle registration, and more. These APIs are intentionally designed with common vulnerabilities, reflecting the risk categories identified in the OWASP Top Ten. The evolution of these critical risks, as shown in the comparison between the 2017 and 2021 lists (see Figure 4.1), highlights the dynamic nature of application security and the need for continuous adaptation of security controls.

The use of crAPI allows for consistent, repeatable testing of the pipeline's security stages; it serves as a baseline to analyze coverage metrics, evaluate false positives, and assess the integration of DevSecOps tools under realistic conditions.

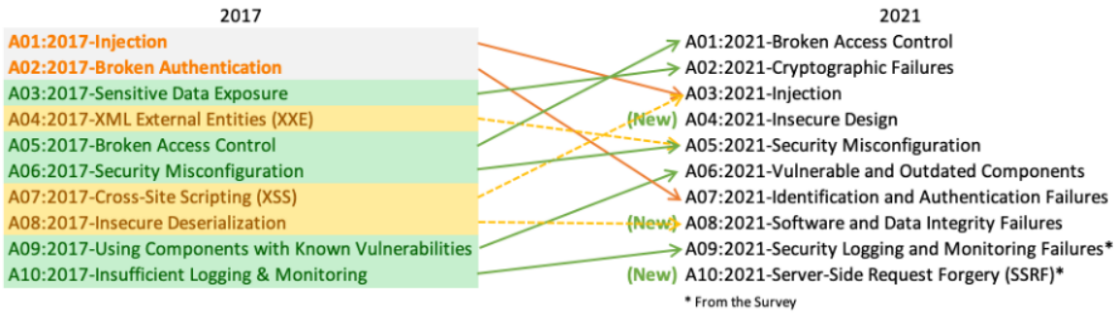


Figure 4.1: Comparison of OWASP Top 10 risks between the 2017 and 2021 editions, showing the re-prioritization and introduction of new categories. [27].

4.1.2 OWASP overview

The Open Worldwide Application Security Project (OWASP) is a nonprofit foundation committed to improving software security through open-source projects, community collaboration, and global knowledge sharing. Founded in 2001, OWASP is widely regarded as an authoritative source in the cybersecurity domain, offering freely accessible standards, tools, educational materials, and awareness programs [28].

Among its most influential contributions is the OWASP Top Ten, a regularly updated list that highlights the most critical risks to web application security. This document has become a foundational reference for security professionals, developers, auditors, and educators worldwide.

Beyond the Top Ten, OWASP maintains and supports a large ecosystem of security tools and initiatives, like **OWASP ZAP**: A powerful open-source penetration testing tool for web applications, **Security Knowledge Framework (SKF)**: An educational tool supporting secure coding and threat modeling, **Cheat Sheet Series**: Practical guidance on common security topics for developers.

OWASP also promotes industry best practices in Secure Software Development Lifecycle (SSDLC), and its standards are often integrated into corporate policies, development guidelines, and regulatory compliance frameworks.

In the context of this thesis, OWASP serves both as a theoretical reference and as a provider of practical resources used to assess the security posture of API-centric applications.

4.2 Tools selection and configuration

Choosing DevSecOps pipeline tools is a matter of hitting the right balance between detection capabilities, ease of integration, long-term maintainability, and good community support.

Static application security testing (SAST) tool comparison is important and should

go beyond detection capabilities. Ease of integration with development workflows, transparency in reporting, support for multiple programming languages, community involvement, and long-term maintainability should also be considered.

On this front, *SonarQube* CE is an ideal choice for companies that want to introduce code quality and security checks in a sustainable and scalable setup.

While *Semgrep* CE was not the tool chosen for the code analysis, it was tested and evaluated. It offers good performance and flexibility. Developers can create custom rules to detect specific code patterns, allowing greater control over what to scan and how to respond.

One of the most useful features of Semgrep CE is its dashboard. Even in the Community Edition, the UI is clean, responsive, and intuitive. As shown in Figure 4.2, the dashboard provides a list of results grouped by category, such as secret leaks, insecure coding practices, or missing validations. Each result includes key details like the filename, line number, triggered rule, and severity level. Clicking on a result shows a code snippet, making it easy to see what caused the alert and why.

The interface also provides strong filtering features. Developers can sort issues by project, severity, rule ID, or status (open, fixed, to fix, ignored). This helps prioritize the most important findings and reduce noise from less significant ones. Tagging and grouping are also supported by the dashboard, which is especially useful for large repositories or when multiple teams work on the same codebase.

A timeline view in the chart shows how findings change over time. This makes it easier to track whether security and code quality are improving. It's also clear when new issues are introduced or if recent changes have caused regressions.

Another advantage is the dashboard's responsiveness. Compared to heavier platforms, Semgrep CE updates results almost instantly after a scan. This was very effective during the testing phase, where fast feedback was needed. Developers could iterate on rule sets and immediately see the results without waiting for long pipeline runs.

Even though Semgrep was not integrated into the final CI/CD pipeline, its dashboard played a key role during experimentation. It helped validate the effectiveness of custom rules, test team workflows, and identify risky patterns early in the development process. Testing these features proved valuable for a comparative analysis within the overall DevSecOps strategy.

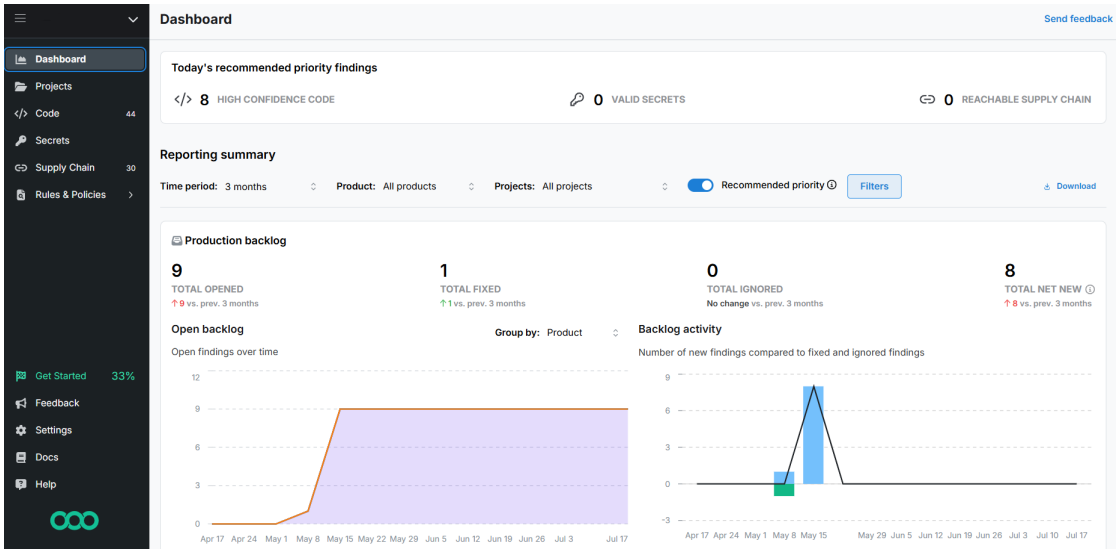


Figure 4.2: Semgrep CE dashboard: example of code findings and security backlog

SonarQube tool, on the other hand, makes a better structured and more comprehensive code analysis. It has the ability to review over 20 programming languages using pre-configured rule sets that detect bugs, code smells[1], and vulnerabilities. These features are tightly integrated into a single centralized dashboard that allows teams to track technical debt[2] project health, and plot quality trends over time. The ability to quantify technical debt and apply quality gates[3] is most appropriate for enterprise governance models and risk management procedures.

Furthermore, *SonarQube*'s native plugin support for typical CI/CD systems[4] such as Jenkins, GitHub Actions, GitLab CI, and Azure DevOps, which enables quality checks to be plugged into existing pipelines with minimal effort. This native integration has the potential to provide smooth, continuous feedback loops so that issues get identified early in the software development cycle.

From maintenance and support perspectives, *SonarQube* also benefits from a large and active community, along with extensive documentation and plugin availability.

[1]A code smell is any symptom in the source code that possibly indicates a deeper problem, such as poor design or maintainability issues.

[2]Technical debt refers to the future rework cost incurred by taking an easy solution today instead of a better one.

[3]A quality gate is a set of conditions that must be met by a project before it is released or promoted. Typically includes coverage, bugs, vulnerability, and duplications thresholds. [29]

[4]CI/CD (Continuous Integration / Continuous Delivery) is a set of software development practices that enables teams to get code changes into production frequently and reliably through automated pipelines.

Though the Community Edition doesn't come with all the advanced security rules (which are available in commercial plans), it does give a solid foundation to identify many standard coding and security issues in a maintainable, centralized, and auditable way.

These capabilities make SonarQube a sound choice for companies looking to improve software quality with secure development practices at scale.

4.2.1 SonarQube

SonarQube is a key tool in the static analysis phase of the pipeline. The Community Edition supports over 20 programming languages, frameworks, and infrastructure-as-code tools. It helps detect bugs, code smells, and security issues using a rule-based SAST engine

This makes it a good fit for projects that use multiple languages, which is common in modern API development.

One important reason for its adoption is the **built-in secret detection engine** added in recent versions. It can find more than 60 types of sensitive data, such as AWS credentials, private keys, and hardcoded passwords, inside supported source files. [19].

Thanks to this feature, SonarQube can catch critical issues early, without needing commercial plugins.

SonarQube also helps developers manage and resolve them in an organized way thanks to a granular **issue lifecycle model**. Each issue is assigned one of these statuses:

- `Open`: the issue was just found and hasn't been reviewed yet;
- `Confirmed`: someone checked it and agreed it's valid;
- `Accepted`: it needs to be fixed;
- `False Positive`: it doesn't require any action;
- `Fixed`: the issue was solved and verified.

Projects can use rules to assign new issues to the right person automatically and when that happens, the developer gets an email. From there, they can review and manage the issue directly in the web interface. This helps keep things clear and speeds up the response.

Each codebase is linked to one or more **Quality Profiles**. These define which rules apply to each language. Teams can change the profiles to reflect their own coding style or standards.

A **Quality Gate** defines a set of conditions that code must meet to be considered production-ready, such as zero blocker issues, minimum test coverage, and duplication thresholds.

In addition, each issue can be connected to the principles of the **Clean Code model**, which emphasizes **Consistency**, **Intentionality**, **Adaptability**, and **Responsibility**[30]. This is applied through the **Clean as You Code** practice, which focuses on keeping new or modified code clean while progressively improving the overall codebase. Such an approach helps teams prioritize issues based on their real impact on software quality and long-term maintainability.

SonarQube integrates well with Azure DevOps, enabling **pull request blocking** when Quality Gate conditions are not met. This forms a critical pillar of the project's automated DevSecOps enforcement strategy.

Instance configuration

The SonarQube instance is configured in *Multi-Quality Rule (MQR) Mode*. In this mode, each rule is linked to **all the quality attributes it affects**, such as *security*, *reliability*, and *maintainability*, instead of being limited to just one, as in the standard configuration.

Each quality attribute is also assigned a specific severity level for the same rule. This gives a clearer picture of the issue's overall impact. For example, a misconfigured cloud resource in an IaC file might reduce security, affect deployment stability, and increase maintenance costs.

This approach improves the **accuracy of risk assessment** in projects that use different languages and technologies. It helps both developers and security teams **prioritize fixes** based on real impact, rather than treating each issue separately.

In this project, which includes code, infrastructure, and API definitions, MQR Mode was useful for revealing problems with effects across multiple areas of quality.

Technical-debt model

In this project, technical debt is estimated using the default model provided by SonarQube. The tool assigns a time-based cost to each issue found in the source code. For example, fixing a minor code smell might take 5 minutes, while addressing a potential vulnerability could require 30 minutes. These values are added up and shown as total technical debt, measured in hours or days of developer effort.

To relate this effort to the size of the codebase, SonarQube uses a simple rule: writing one line of code is assumed to take **30 minutes**. Using this estimate, the total development effort of a project can be calculated and compared with the time needed for remediation. This comparison produces a metric called the *debt ratio*. SonarQube uses this ratio to rate maintainability on a scale from **A (excellent)** to **E (poor)**. This helps teams focus on the most important technical improvements.

The debt model is in line with the broader view described by Ariadi Nugroho, Joost Visser and Tobias Kuipers [31]. Their study presents a method to estimate both the "principal" of technical debt (the cost to fix problems) and the "interest" (the extra work caused by those problems over time). Their results show that systems with low

code quality tend to have much higher long-term maintenance costs. This highlights the value of tracking and reducing technical debt early in the development process.

SCM sensor and Azure DevOps integration

The *Source Control Management (SCM) sensor* in SonarQube collects version control metadata during code analysis. It uses the `git blame` command to link each line of code to the last author and commit. This data allows SonarQube to:

- assign issues to the developer who last modified the affected line;
- detect new or changed code, supporting the "Clean as You Code" practice;
- show detailed annotations in the code viewer, such as author, commit ID, and date.

In this project, the SCM sensor was enabled to improve traceability and developer accountability.

SonarQube's integration with Azure DevOps enabled the following features:

- automatic pull request decoration with Quality Gate status and issue details;
- enforcement of merge rules that block pull requests if the Quality Gate fails;
- direct links from SonarQube issues to the related files and commits in Azure Repos.

This integration makes the feedback loop faster. It helps developers fix problems quickly and keep code quality high throughout the project.

Tailored quality profiles

The default *Sonar way* rule sets were used as a base, and custom quality profiles were created by inheriting and extending them for the languages and technologies most relevant to the API delivery pipeline.

These profiles keep the structure and updates of the upstream rule sets, but include project-specific changes to better match the operational and security context. Key adjustments include:

1. **Deactivating rules** that caused false positives or conflicted with accepted design patterns used during implementation.
2. **Raising severity levels** for rules related to API exposure, authentication, and misconfigured infrastructure components.
3. **Enabling preview rules** for secret detection, to improve alignment with tools like TruffleHog.

This inheritance-based approach ensures that future changes to the upstream profiles (such as new rules or better detections) are applied automatically. It reduces maintenance effort and helps keep long-term compatibility. All custom rules remain visible in the *Quality Profiles* dashboard, supporting governance and audit over time.

Quality Gate rationale

All projects are subject to a custom Quality Gate named *test-api-quality-gate*, which follows the principles of the *Clean as You Code* methodology. This strategy focuses on enforcing quality standards exclusively on newly added or modified code, avoiding retroactive penalties on legacy components.

The gate applies the following thresholds to each new commit or pull request:

Metric (on new code)	Operator	Threshold
Issues	>	0
Security Hotspots Reviewed	<	100 %
Coverage	<	**0 %**
Duplicated Lines	>	3 %

The `<0%` coverage threshold is a deliberate choice: this condition is always true, effectively disabling coverage as a gating factor. This configuration shifts the pipeline's focus toward API definitions and post-deployment security analysis, rather than traditional unit testing. To ensure a minimum quality baseline, one condition still applies globally: the presence of *zero blocker issues* across the entire codebase, regardless of whether the issues affect only new or existing code.

Developer notifications

To deliver useful feedback without overwhelming users, each developer receives email alerts only for specific events:

- new issues assigned to them;
- Quality Gate failures on projects they follow;
- failed background tasks, such as interrupted analyses.

This notification policy follows a *least-noise principle*, sending alerts only to people who can actually act on them. It avoids, for instance, notifying the same user who just triggered the event.

With pull request **annotations** in Azure DevOps, serious code quality issues are sent immediately to the right person.

Dashboard and Visualization

The SonarQube web interface helps to have a complete view about code quality, security, and reliability metrics. It has different levels of detail tailored to various roles, from project managers to individual developers.

The main **Projects dashboard** (Fig. 4.3) provides a high-level overview. From this page, it is possible to see the **Quality Gate status** and check issues by severity for **Security**, **Reliability**,

Maintainability[5], and **Security Review**[6]. Each one is rated from **A (best)** to **E (worst)**, helping teams spot problems quickly. The dashboard also shows metrics for **Coverage**, **Duplication percentage**, and a **Languages list**, allowing teams to explore the overall state of each project.

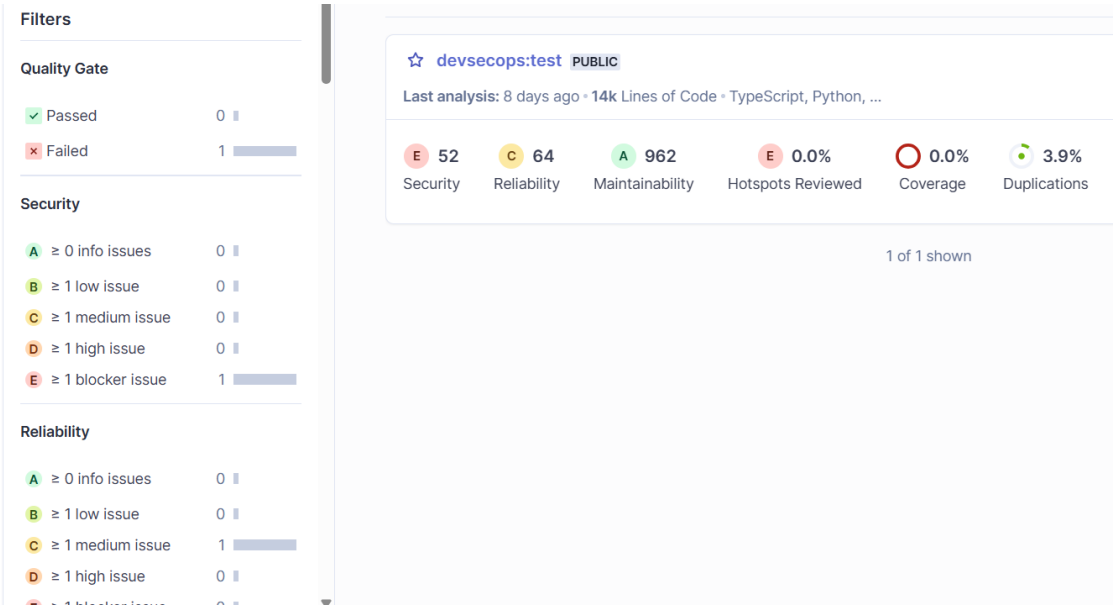


Figure 4.3: The main SonarQube dashboard, providing a portfolio-level view of project status.

For a deeper analysis, the **Project Overview** page (Fig. 4.4) presents detailed metrics for a single codebase. This dashboard clearly indicates whether the project passed or failed its Quality Gate and provides the exact reason for the failure ; for example: "51 Blocker Severity Issues is greater than 0". It also distinguishes between issues found in **New Code** versus the **Overall Code**, which is central to the "Clean as You Code" philosophy. This granular view gives developers actionable feedback, allowing them to focus directly on the conditions that are blocking code promotion.

[5]Ratio of the estimated time needed to fix all outstanding maintainability issues to the size of the project.

[6]The percentage of reviewed (fixed or safe) security hotspots.

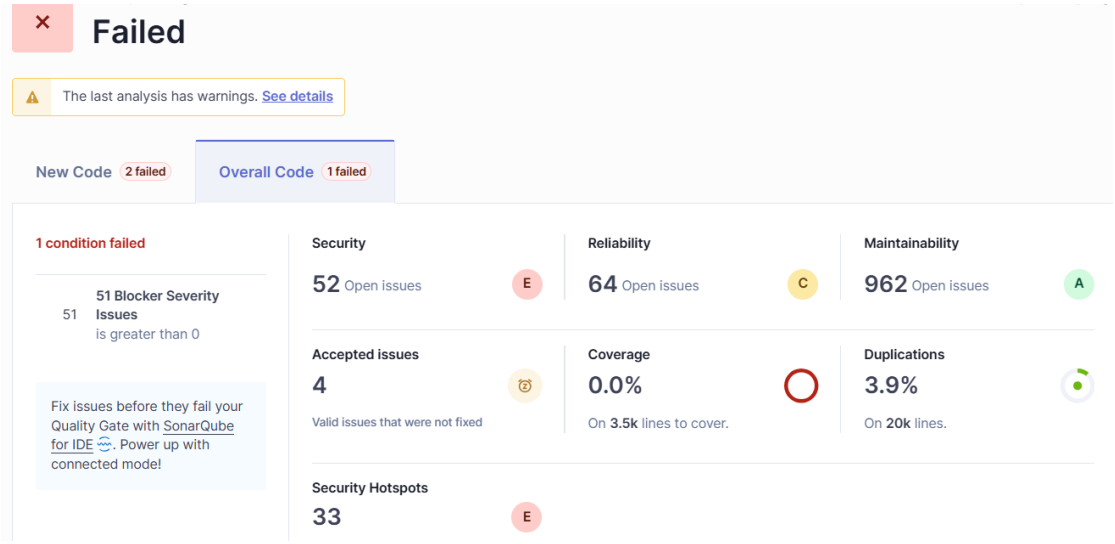


Figure 4.4: The detailed project overview, highlighting the specific reason for the Quality Gate failure.

4.2.2 Spectral

Spectral is used as a dedicated linter for API specification files. While tools like SonarQube focus on implementation (source code and infrastructure), Spectral checks the structure and meaning of API contracts. It ensures that all API descriptions follow organizational style guides and industry standards, promoting clarity, maintainability, and interoperability.

Spectral is a general-purpose rules engine that can analyze any YAML or JSON file. However, it is tailored for API use cases and includes built-in rulesets such as:

- `spectral:oas` – for OpenAPI v2/v3 validation;
- `spectral:asyncapi` – for AsyncAPI v2.x schemas;
- `@stoplight/spectral-owasp-ruleset` – for applying OWASP API security best practices.

These rulesets can be customized or extended to enforce internal design rules, such as naming patterns, required metadata fields, consistent versioning, or banned patterns (e.g., numeric IDs in URLs). Rules are written in YAML and use JSONPath-like expressions, making them easy to understand and lightweight to execute. This allows Spectral to run efficiently in both local development and CI pipelines.

In this project, Spectral is integrated into the Azure DevOps pipeline. It scans all `.yaml`, `.yml`, and `.json` files in the API directory, using a shared `.spectral.yaml` config file that extends multiple base rulesets and includes custom rules. The analysis results are exported in HTML format and then **digitally signed with Cosign** to ensure integrity and authenticity. The signed report is saved as a pipeline artifact, creating a tamper-evident audit trail for API validation as part of the DevSecOps process.

4.2.3 TruffleHog

TruffleHog is used in `filesystem` mode to scan the extracted working directory for secrets. Even though SonarQube already checks source files for exposed secrets, TruffleHog uses a different detection

engine based on entropy checks and custom patterns. This helps find secrets in files that might be ignored or misconfigured in other scans.

TruffleHog can also be configured to scan Git history or remote branches, which is one of its most powerful features.

4.2.4 Dynamic Application Security Testing Tool Comparison

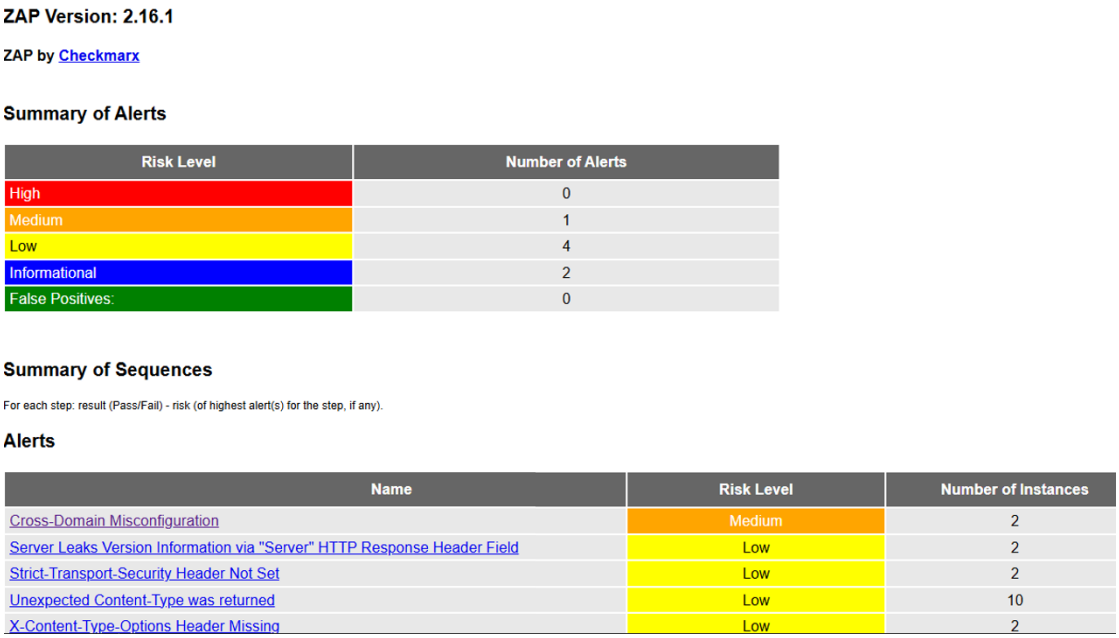
The evaluation and selection of Dynamic Application Security Testing (DAST) tools plays a crucial role in the design of a secure and automated DevSecOps pipeline. Among the wide range of available solutions, two tools were identified as particularly relevant for this thesis: **OWASP ZAP**, a widely adopted open-source web application scanner, and **APIsec**, a platform developed specifically for API security testing. Although both tools share the common objective of detecting vulnerabilities during runtime, they adopt very different approaches in terms of methodology, depth of analysis, and integration capabilities.

In order to provide a concrete basis for comparison, both solutions were implemented and tested within the same pipeline environment. Their outputs and dashboards were then analyzed, not only in terms of the vulnerabilities identified, but also with respect to the usability of their interfaces, the detail of the reports produced, and their ability to handle API-specific attack scenarios. The following sections present an in-depth discussion of the characteristics, strengths, and limitations of each tool.

4.2.5 OWASP ZAP

OWASP ZAP (Zed Attack Proxy) is one of the most widely used open source security scanners. It is developed and maintained by the OWASP community and is designed to help both developers and security teams find vulnerabilities in web applications and APIs. Its popularity comes from the fact that it is free, easy to start with, and flexible enough to be extended for more advanced testing needs.

The dashboard, shown in Figure 4.5, presents the results of a scan . At the top there is a summary table that groups findings by severity level such as *High, Medium, Low and Informational*. Below this summary each vulnerability is described in detail with its name, the level of risk, and the number of times it was found. This structure makes it possible to start from a quick overview of the application risk and then drill down into the technical details. The tool also supports exporting reports in formats like HTML or JSON, which can be shared with developers or processed automatically in security workflows.



ZAP Version: 2.16.1

ZAP by Checkmarx

Summary of Alerts

Risk Level	Number of Alerts
High	0
Medium	1
Low	4
Informational	2
False Positives:	0

Summary of Sequences

For each step: result (Pass/Fail) - risk (of highest alert(s) for the step, if any).

Alerts

Name	Risk Level	Number of Instances
Cross-Domain Misconfiguration	Medium	2
Server Leaks Version Information via "Server" HTTP Response Header Field	Low	2
Strict-Transport-Security Header Not Set	Low	2
Unexpected Content-Type was returned	Low	10
X-Content-Type-Options Header Missing	Low	2

Figure 4.5: OWASP ZAP vulnerability report dashboard

One of the main strengths of ZAP is its ability to act as a proxy between the client and the server. It stands between the tester's browser and the web application so that it can intercept and inspect messages sent between browser and web application, modify the contents if needed, and then forward those packets on to the destination. It can be used as a stand-alone application, and as a daemon process.

This means that it can capture traffic, show the requests and responses, and even allow testers to modify them to explore potential weaknesses. ZAP also includes automated scanning functions such as the Spider[7] and the AJAX Spider[8]. These modules explore the target application to discover both static and dynamic endpoints. Scans can be passive, where the tool only observes and analyzes traffic, or active, where it tries specific attacks like SQL injection or cross site scripting. In addition, users can extend ZAP through the Marketplace which provides extra add ons such as advanced authentication support, custom scan rules, or integrations with continuous integration and delivery pipelines.

Despite these advantages, ZAP is still a general purpose scanner. It is very effective at detecting common technical problems such as missing security headers or cross domain misconfigurations, but it has clear limitations with modern API security. Handling authentication methods like OAuth 2.0 or JWT tokens often requires complex manual configuration, and the tool is not well suited to find business logic flaws or authorization problems that depend on the specific design of the API. For this reason, while ZAP remains a strong and reliable choice for basic vulnerability detection, it cannot provide the deeper and more context aware analysis offered by tools created specifically for API

[7]The Spider automatically explores the target application by following all the links it finds, similar to a user who clicks through every page. This process helps discover hidden or less obvious endpoints.

[8]The AJAX Spider is designed to explore modern web applications that load content dynamically using JavaScript. It interacts with the application like a real browser to uncover endpoints that may not appear in static links.

security testing.

4.2.6 The APIsec Platform

APIsec is a specialized platform designed for the automated security testing of APIs. Unlike general-purpose scanners, it employs a model-driven approach. The platform first ingests and analyzes an API's specification to build a comprehensive model of its structure, business logic, and data handling processes. This methodology facilitates more intelligent and context-aware security testing. The main interface for managing this process is the APIsec dashboard, which provides a centralized view of testing activities and results, as shown in Figure 4.6.

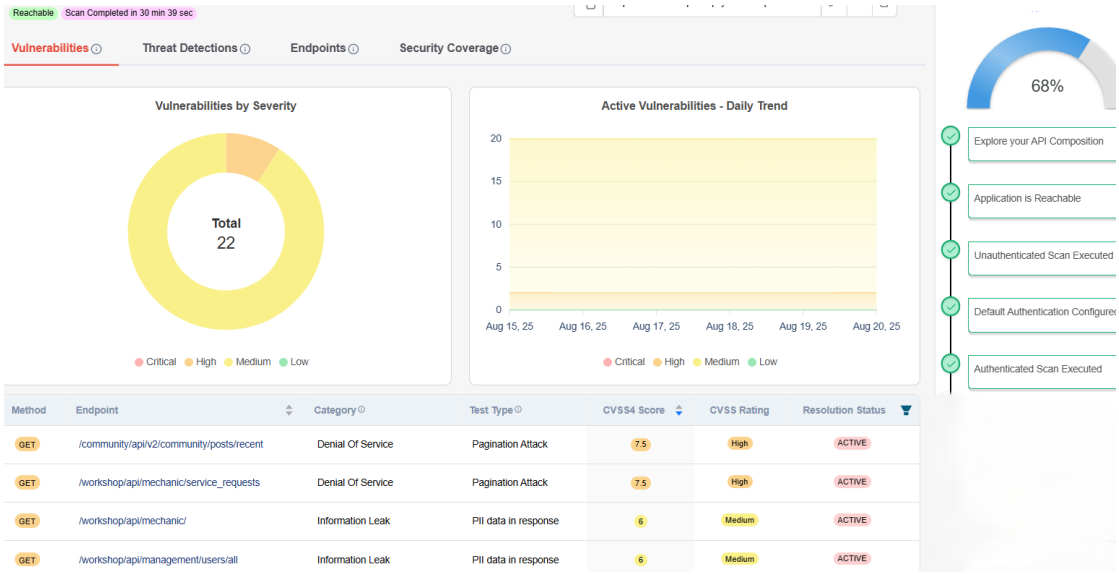


Figure 4.6: The APIsec dashboard, illustrating vulnerability overviews and the application model.

Dashboard and Security Insights

The dashboard functions as a centralized interface, providing a real-time overview of the organization's API security posture. It aggregates data from all tested APIs and presents key metrics to facilitate risk assessment and prioritization.

Key information displayed on the dashboard includes:

- **API Inventory and Risk Exposure:** It provides a summary of all onboarded applications and the total number of API instances under assessment.
- **Security Analytics:** The interface includes graphical representations of weekly security trends, showing the evolution of identified vulnerabilities and overall risk levels.
- **Categorized Risks:** Vulnerabilities are classified according to severity *(Critical, High, Medium, or Low)*, enabling teams to prioritize remediation efforts based on potential impact.
- **Security Posture Analysis:** The dashboard continuously evaluates APIs, highlighting the percentage of vulnerable endpoints and providing a breakdown of critical and active issues that require immediate attention. It also maintains a history of scans for each API.

Core Platform Components

The APIsec platform is built around several key components that together enable complete and automated API security testing.

- **Applications** act as the central hub for managing the API inventory. Each registered API becomes an entry point for configuring test profiles, authentication settings, and security assessments. From here, users can initiate scans and track their results across different environments.
- **Hosted Agents** are optional dedicated scanning nodes that can be deployed inside private networks, data centers, or cloud environments. They allow security tests to be executed from *specific network locations*, making it possible to analyze APIs that are not publicly exposed or that sit behind corporate firewalls.
- **Integrations** extend the platform into existing DevSecOps workflows. APIsec provides native connectors for continuous integration systems such as Azure DevOps and Jenkins, as well as for issue tracking platforms like *Jira* and *Azure DevOps Boards*, where detected vulnerabilities can be automatically logged as work items. Notification tools including Slack and Microsoft Teams are also supported, ensuring that security alerts reach the right teams in real time. In addition, API specifications from sources such as SwaggerHub or API gateways can be imported directly to streamline the setup process.

  A practical example of this integration is the automatic creation of remediation tickets: when a vulnerability is identified during a scan, APIsec can generate a Jira issue or an Azure DevOps work item, assign it directly to the responsible developer, and keep track of its remediation status until closure. This automation reduces manual overhead and ensures vulnerabilities are systematically managed in the same workflow as other development tasks.
- **API Tokens** provide a secure way to interact programmatically with the platform. They are essential for automation, enabling pipelines and scripts to trigger scans, retrieve reports, or manage configurations without manual intervention.

The App Model Workflow

The platform use a structured, multi-stage workflow known as the "App Model" to guide users through the process of progressively configuring and executing API security tests. This incremental approach ensures that functional and security validations are properly established.

The App Model consists of six distinct phases:

1. **Define the API Application:** An application entry is created to serve as the baseline for all subsequent testing configurations. The API's endpoint URL must be reachable to proceed.
2. **Run an Initial Unauthenticated Scan:** A preliminary security scan is performed without authentication credentials. This step assesses publicly accessible endpoints for common misconfigurations.
3. **Configure API Authentication:** Authentication credentials (e.g., API keys, OAuth tokens) are configured to permit deeper testing of protected API endpoints.
4. **Execute an Authenticated Scan:** With authentication enabled, a comprehensive security analysis is performed on restricted endpoints to identify authorization flaws or potential privilege escalation vulnerabilities.
5. **Validate RBAC Configuration:** Role-Based Access Control (RBAC) policies are tested to verify that users can only access resources and perform actions permitted by their assigned roles.
6. **Refine API Test Coverage:** The user can supplement the model with missing parameter values derived from API specifications to improve test coverage, particularly for dynamic API behaviors.

Analysis and Reporting Sections

APIsec goes beyond simple vulnerability detection by providing advanced capabilities for both interactive analysis and structured reporting. These two dimensions complement each other: the dashboard allows security teams to explore findings in real time, while the generated reports serve as formal documentation for audits, compliance, and long-term tracking.

Analysis

The interactive analysis features are accessible directly from the platform's dashboard.

The **Vulnerabilities** section presents each identified issue in detail, linked to the exact endpoint and HTTP method. Every entry contains a category (such as Authentication , Authorization , ...), a CVSS-based severity score, and complete technical evidence, including the HTTP request and response. This level of detail allows developers to reproduce the problem and apply precise fixes. From the same view, vulnerabilities can be marked as false positives, accepted as risk, or directly converted into remediation tickets in integrated platforms such as Azure DevOps or Jira.

The **Threat Detections** view groups findings by vulnerability type, for example SQL Injection or Broken Authentication. This grouping highlights patterns and shows which endpoints are affected by the same class of issue, enabling teams to prioritize fixes across the API surface rather than addressing each alert in isolation.

The **Endpoints** section provides a complete map of the API structure as derived from its specification. Endpoints that handle sensitive information, such as personal or financial data, are automatically flagged, while unauthenticated endpoints are highlighted to provide an immediate overview of potential exposure.

Finally, the **Security Coverage** section offers transparency into the performed tests. It maps each category of attack to the relevant endpoints, showing exactly what has been tested and what has not. This feature is especially useful in audit scenarios, since it proves the extent of the security testing and its alignment with standards such as the OWASP API Top 10.

Reports

Beyond the dashboard, APIsec automatically generates reports that summarize and formalize the testing activity. These reports can be exported in different formats and serve both technical and managerial purposes.

The **OWASP Report** focuses on coverage against the OWASP API Security Top 10. It provides a high-level summary of how many endpoints were tested, how many test scenarios were executed, and what vulnerabilities were found. The report is designed to give stakeholders a quick but reliable assessment of API security posture. It shows the percentage of endpoints covered by tests, the number of vulnerabilities per category, and the most affected areas of the API.

The **Penetration Test Report** is more detailed and resembles the documentation produced by a manual penetration testing engagement. It lists every vulnerability with its endpoint, method, category, severity, and status. Each entry also includes detailed evidence and remediation guidance, for example suggesting better handling of sensitive data in API responses. This report is particularly valuable for development teams, as it provides concrete steps for addressing issues and can be used to track remediation progress over time.

Together, the analysis dashboard and the exported reports create a complete feedback loop. The dashboard supports day-to-day security monitoring and integration with development workflows, while the reports provide structured documentation that can be shared with auditors, managers, or compliance officers. This dual approach ensures that the results of APIsec scans are not only actionable for developers but also formally documented for governance and regulatory needs.

Advanced Testing Capabilities with Playbooks

APIsec provides several advanced testing capabilities that go beyond traditional scanners, and these are organized into **Playbooks**. In this context, a Playbook is a structured collection of automated tests designed specifically for APIs. Instead of running only generic checks, Playbooks take into account the API specification, its workflows, and the way data and permissions are managed. This approach allows APIsec to simulate realistic attack scenarios and uncover both technical and business-level issues.

One important capability enabled by Playbooks is the detection of **business logic flaws**. By modeling the intended behavior of the API, the platform can identify weaknesses that appear when endpoints are used in an unexpected sequence or when workflows can be bypassed. These flaws are difficult to detect with simple fuzzing or static scans because they depend on how the application logic is implemented.

Another core example is the identification of **Broken Object Level Authorization (BOLA)** vulnerabilities. Here APIsec tests whether users can improperly access data that belongs to others by manipulating identifiers, such as order IDs or file references. This type of test reproduces real-world abuse cases and addresses one of the most critical risks listed in the OWASP API Security Top 10.

APIsec automates BOLA detection through a three-step workflow:

1. **Define Endpoints and Object Identifiers:** The user specifies which API endpoints handle user-specific objects and maps the identifiers used in API requests (e.g., `userId`, `orderId`).
2. **Configure Authorization Rules:** The expected access control behavior is defined, establishing which users or roles should have permission to access specific objects.
3. **Execute BOLA Attack Scenarios:** The platform automatically generates and executes test cases that simulate unauthorized access attempts, such as a user trying to access another user's data. Endpoints with BOLA vulnerabilities are flagged for remediation.

Finally, Playbooks also cover **role-based access control (RBAC)** verification. By using credentials for different roles (such as administrator, employee, or customer), APIsec checks whether each role can only access the endpoints and data that are meant for it. This helps to uncover authorization weaknesses, for example cases where a low-privileged user can gain access to administrator functionality.

This is also a three-step process:

1. **Configure User Roles and Permissions:** The user defines different roles (e.g., `admin`, `standard_user`) and provides corresponding authentication credentials for each.
2. **Perform a Dry Run:** APIsec simulates API calls using the credentials for each defined role to evaluate access control behavior against the API specification.
3. **Analyze the RBAC Map:** The system generates a visual RBAC Map that shows which roles have access to specific API endpoints. This allows teams to compare the expected access permissions against the actual, observed behavior.

4.2.7 Evaluation and Tool Selection

The comparison of OWASP ZAP and APIsec shows a key difference: a general-purpose web scanner versus a specialized, context-aware API security platform. This was the deciding factor for this thesis.

OWASP ZAP is a solid tool for finding common web vulnerabilities. It works as an HTTP proxy, making it good at spotting technical flaws that are independent of the app's purpose. However, it does not fully understand the API's business logic, its workflows, or its authorization model.

APIsec works at a higher level. It reads the API specification to build a model of the app's intended behavior. This lets it find complex API-specific issues, such as Broken Object Level Authorization (BOLA) and flawed Role-Based Access Control (RBAC), which other scanners may miss.

Because this project requires automated, in-depth security tests in a modern DevSecOps pipeline, this type of analysis is essential. ZAP is still useful in any security toolkit, but APIsec was chosen for its ability to address critical security needs in API-based systems.

4.3 Forensic Readiness and Digital Evidence

The DevSecOps platform was designed not only to integrate multiple security tools but also to guarantee the preservation of their outputs for digital forensics. The implemented architecture establishes forensic readiness by ensuring that all relevant evidence is collected, signed, and stored in an immutable manner, thereby supporting future investigations with reliable and verifiable data.

4.3.1 Evidence Sources

The system integrates four primary evidence sources that reflect the different stages of the DevSecOps workflow:

1. **Static security analysis pipeline.** The results produced by *Spectral* and *TruffleHog* are exported directly by the pipeline, digitally signed, and stored in WORM storage.
2. **Dynamic security analysis pipeline.** The *APIsec* agent generates logs and reports during dynamic API testing. These artifacts are signed by the pipeline and stored in the same immutable storage.
3. **SonarQube analysis and access logs.** To overcome the lack of native audit export in SonarQube Community Edition (CE), a custom mechanism was developed. A scheduled cron job retrieves access logs and analysis results directly from the host VM; these artifacts are then individually signed and stored.
4. **Azure DevOps agent execution logs.** Execution logs of pipeline runs are signed and stored to provide a timeline of CI/CD activities.

4.3.2 Integrity and Immutability

All artifacts are individually signed with `cosign`. Each artifact is assigned a creation timestamp at the moment it is generated. This timestamp provides precise temporal context for forensic analysis, allowing investigators to reconstruct the exact sequence of events and correlate data across different sources. The signing key is retrieved securely at runtime from *Azure Key Vault* over a private endpoint. This guarantees cryptographic integrity and enables independent verification of each artifact using the corresponding public key.

The final storage location is an **Azure Blob container configured in immutable WORM (Write-Once, Read-Many) storage**, which enforces immutability for the full retention period. Once written, data cannot be modified or deleted, even by administrators. Role-based access control ensures separation of duties: service identities that collect evidence, such as the pipelines and the scheduled log collector, have write-only permissions. Auditor and investigator identities are limited to read-only access.

4.3.3 Forensic Value

From a digital forensics perspective, this design provides the following guarantees:

1. **Tamper-evident evidence:** digital signatures ensure that any modification attempt is detectable.

2. **Immutable retention:** WORM storage enforces non-repudiation of logs and reports for the entire retention period.
3. **Cross-source correlation:** investigators can correlate results across static analysis (SonarQube, Spectral, TruffleHog), dynamic analysis (APIsec), and execution traces (Azure DevOps agent logs).
4. **Chain of custody:** every step of evidence collection, signing, and storage is auditable. Signing events are tied to controlled Key Vault access, while all signed artifacts are preserved in immutable WORM storage, ensuring verifiable integrity and traceability.
5. **Independent verification:** artifacts can be retrieved directly from immutable storage and validated against the public key without relying on the original CI/CD infrastructure.

By combining cryptographic integrity, immutable storage, and multi-source evidence collection, the platform establishes a verifiable chain of custody. This ensures that investigators can reliably reconstruct events with confidence in the authenticity and integrity of all preserved artifacts.

4.3.4 Automated Evidence Collection from SonarQube

To ensure forensic readiness despite the lack of native auditing in the SonarQube CE, a custom evidence collection pipeline was implemented on the DevOps agent VM. This pipeline guarantees that access logs and analysis results are collected, signed, and preserved in immutable storage, thus transforming operational data into verifiable forensic artifacts.(Fig. 4.7).

The **Access logs** are captured inside the SonarQube container by enabling the native HTTP logging subsystem; the format is extended to include IP address, HTTP verb, user agent and authenticated user. The **Static-analysis results** (*measures*, *issues*, and quality-gate status) are extracted via the SonarQube REST API, authenticated with a scoped project token.

A systemd-timer triggers a `cron` job every day on the VM. The job orchestrates three Bash scripts:

1. `access_log_to_csv.sh`: converts the raw log to CSV and signs both artifacts using **Cosign**. It securely retrieves the private key and its passphrase from *Azure Key Vault*, handling the key in-memory and clearing all secrets from the environment after use.
2. `export_sonarqube_data.sh`: exports metrics and issues in JSON format and signs the JSON with the same Cosign key.
3. `generate_and_upload.sh`: packages *all* artifacts (`*.csv`, `*.log`, `*.json`) and their `*.sig` files into a single ZIP archive, then uploads the archive to an Azure Blob container configured in **WORM (Write-Once, Read-Many)** mode.

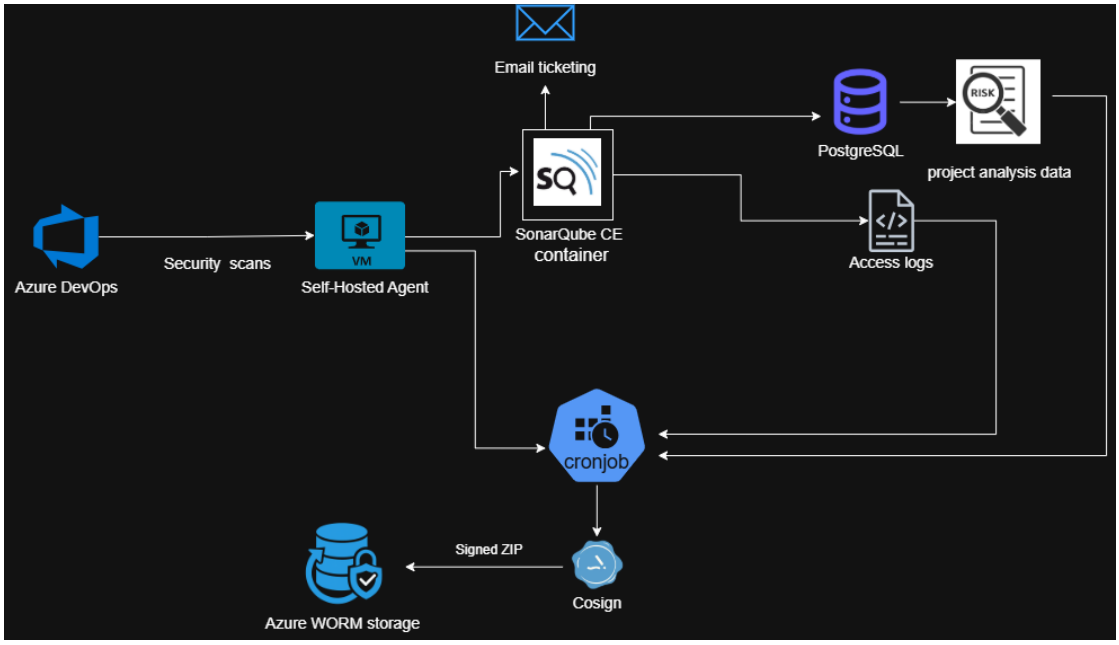


Figure 4.7: Automated evidence pipeline: collection, signing and immutable archiving.

Immutable Storage Configuration and Access Control

The WORM functionality in Azure Blob Storage, also known as immutable storage, is critical for guaranteeing the **integrity** and **non-repudiation** of audit evidence. Immutable storage for Azure Blob Storage supports two main types of policies: **time-based retention policies**, which store data for a specified interval, and **legal hold policies**, which preserve data indefinitely until the hold is explicitly cleared. When a policy is active, objects can be created and read, but are protected from modification or deletion for its duration [32].

For the audit trail to be meaningful, the evidence required absolute trustworthiness. The central challenge was to protect the collected artifacts from any form of modification or premature deletion. The solution was found in a specific feature within **Azure Blob Storage**: its immutable storage capability. The implementation employs a **time-based retention policy**, configured in a **locked** state. The exact duration of the retention period is fully **customizable** according to compliance or organizational requirements. The 'locked' configuration is the most critical detail, as it renders the policy irreversible for its entire duration, creating a powerful safeguard that not even an administrator can bypass. This mechanism was adopted to guarantee data integrity and long-term trustworthiness of the stored evidence. While Azure also offers a 'legal hold' for indefinite data preservation, that approach is designed for specific legal cases and was considered unnecessary for this project's scope. The time-based model proved a much better fit for meeting standard, cyclical retention requirements. Finally, this data-level protection was reinforced with strict access controls. The pipeline's identity was granted the `Storage Blob Data Contributor` role, while auditors were assigned the `Storage Blob Data Reader` role. This setup is a direct application of the **principle of least privilege**, ensuring a robust separation of duties.

Security hardening

To ensure the cryptographic integrity of the evidence collection process, additional security measures were implemented throughout the signing workflow. The entire signing process relies on Azure Key Vault to securely manage and provide the cryptographic keys used for signing each artifact.

Access to the secrets themselves is also tightly restricted. A dedicated Service Principal was created with a single, minimal permission: the "Key Vault Secrets Officer" role. This grants just enough privilege to read the secrets needed for the signing operation, and nothing more.

Special attention was given to the handling of the private signing key itself. To minimize its exposure, the key is never written to disk. Instead, it is injected directly into the `cosign` process at runtime and subsequently shredded from memory upon completion of the task.

This approach of signing each artifact individually is fundamental. The resulting ZIP archive serves as a convenient transport mechanism, but it is not a single point of trust. The integrity of the evidence relies on the cryptographic signatures attached to each file within it.

The verification process for an auditor is therefore direct and reliable. An investigator can download any artifact from the immutable WORM storage and validate its integrity by checking the attached `.sig` file against the public key.

This design not only compensates for missing audit features but also ensures compliance, integrity, and forensic value of SonarQube-related evidence within the broader DevSecOps framework.

Chapter 5

Compliance with Cybersecurity Standards, Directives and Regulations

The implemented architecture was built to meet functional needs for secure audit trail management in a DevSecOps environment. It follows best practices like immutable storage, cryptographic integrity, and the principle of least privilege.

A review of the system indicates that these measures align with key requirements from recognized cybersecurity frameworks and regulatory standards. This section examines how the implemented security controls map to various standards and regulations, with particular attention to data retention requirements and their regulatory implications.

5.1 Foundational Security Controls Implemented

Before mapping to individual standards, it is useful to highlight the foundational controls that underpin the entire architecture:

- **Immutable storage:** Audit evidence is stored in Azure Blob Storage with WORM (Write Once, Read Many) configuration, preventing deletion or modification for the defined retention period.
- **Cryptographic integrity:** Each artifact is digitally signed at creation with `cosign`, ensuring verifiable authenticity and resistance to tampering.
- **Granular access control:** Keys and credentials are securely managed via Azure Key Vault, restricted by scoped identities and private endpoints, enforcing the principle of least privilege.

These core mechanisms provide the foundation upon which compliance with multiple frameworks is demonstrated.

5.2 ISO/IEC 27001:2022

ISO/IEC 27001:2022 emphasizes the CIA triad (confidentiality, integrity, availability) and requires logging of relevant activities for future analysis. The implemented architecture supports these objectives by:

- Generating structured security reports from static and dynamic analyses.
- Preserving their immutability in WORM storage.
- Ensuring traceability through digital signatures and controlled access.

Together, these measures strengthen the organization's ability to detect, investigate, and respond to incidents—objectives central to an ISMS.

5.3 PCI DSS

The **Payment Card Industry Data Security Standard** (PCI DSS) [33] is a well-established security framework aimed at protecting cardholder data and sensitive authentication information in payment systems. While the architecture described in this thesis supports a generalized DevSecOps flow, one not limited to payment processing, the principles outlined by PCI DSS serve as a valuable benchmark for ensuring integrity, retention, and auditability of security evidence.

The implemented architecture demonstrates alignment with PCI DSS particularly with respect to audit trail protection and data retention. According to PCI DSS requirement: "Retain audit trail history for at least one year; at least three months of history must be immediately available for analysis" [33].

PCI DSS prescribes that audit evidence should be retained for an extended period and safeguarded against tampering. This requirement is addressed through the use of WORM storage, which ensures that once audit evidence is written, it cannot be modified or deleted for the specified retention period. In addition, the assignment of granular roles enforces the principle of least privilege and guarantees a clear separation of duties, both of which are strongly emphasized in PCI DSS.

Furthermore, PCI DSS highlights the need for mechanisms that detect unauthorized changes to critical files. This objective is inherently fulfilled by the cryptographic signing process implemented via `cosign`, which assigns a verifiable signature to each artifact. These signatures act as immutable fingerprints, allowing auditors to independently validate the integrity of every file. The current retention policy is time-based and configurable, with a locked configuration that ensures immutability for the chosen duration. The system is designed to allow for the extension of the retention period without any architectural changes, thereby ensuring long-term compliance and providing a robust foundation for alignment with broader regulatory frameworks.

5.4 NIST SP 800-92 Guidelines

The National Institute of Standards and Technology Special Publication 800-92 provides comprehensive guidance for log management in cybersecurity contexts. The framework identifies four primary functions for log management infrastructure: log generation and collection, log analysis and correlation, log storage and retention, and log disposal [34].

The architecture presented in this thesis reflects these principles in a concrete DevSecOps workflow:

- **Log generation and collection:** Security scan reports and access logs from security platforms are consistently generated and collected as part of the automated pipeline, ensuring that no activity or result is left undocumented.
- **Log analysis and correlation:** Reports are automatically evaluated against defined security policies. This transforms the pipeline into an active security control that can stop deployments when risk thresholds are exceeded.
- **Log storage and retention:** All artifacts are digitally signed at creation and stored in Azure Blob Storage configured in WORM mode. The retention period is configurable and the locked policy guarantees immutability for its entire duration.

- **Log disposal:** NIST highlights the importance of properly disposing of logs once their retention period has expired, to avoid unnecessary accumulation and reduce the risk of exposure of sensitive information. In the implemented architecture, this requirement is addressed through Azure Blob Storage's time-based retention policy in WORM mode. Once the configured retention period ends, log data is automatically and irreversibly deleted by the platform, ensuring a secure and compliant disposal process without the need for manual intervention.

5.5 HIPAA

Another category of data that requires a distinct management approach is patient health information. Due to its sensitive and personal nature, this data is protected by specific regulations. In the United States, the primary framework is the Health Insurance Portability and Accountability Act (HIPAA)[35], which sets the standard for protecting patient data.

At its core, the HIPAA Security Rule requires organizations to ensure the confidentiality, integrity, and availability of all electronic patient information. A key part of HIPAA is not just being secure, but being able to prove it through careful documentation. Organizations must keep detailed records of their security efforts, such as risk assessments, security policies, and logs of system activity. These documents serve as the official evidence during an audit.

The data retention requirements under the Health Insurance Portability and Accountability Act (HIPAA) are nuanced, with different rules applying to distinct categories of information. Specifically, HIPAA mandates a **six-year retention period** for a specific set of documents related to compliance and administrative oversight. This category includes the organization's security policies, risk analysis documentation, incident response reports, and the audit logs generated by the system.

It is crucial to distinguish this requirement from the retention of the patient's Protected Health Information (PHI) itself. HIPAA does not define a federal retention period for PHI. Instead, the retention timeline for medical records is governed by a complex matrix of **state-level laws and regulations**. These state laws often vary significantly and may require retention periods far longer than six years, sometimes based on factors like the patient's age or the last date of treatment. Consequently, any compliant data management architecture must be capable of handling multiple, variable retention policies, applying the six-year rule to its operational and security logs while deferring to state-specific mandates for the underlying patient data.

State laws often impose stricter and longer retention requirements for medical records, which can range from seven to ten years or even longer, especially for minors' records. Furthermore, other federal regulations, like those for Medicare providers, mandate their own specific timeframes. This creates a significant challenge for organizations, which must navigate a patchwork of rules to remain compliant.

The implemented architecture is uniquely suited to address this complexity. Its use of Azure Blob Storage with a configurable WORM policy allows an organization to set and lock retention periods on a granular basis, aligning with the strictest applicable state or federal law.

5.6 GDPR

The General Data Protection Regulation (GDPR) introduces complex considerations for vulnerability management systems. Article 4(1) defines personal data as "any information relating to an identified or identifiable natural person" [36]. Moreover, the Court of Justice of the European Union has clarified that even dynamic IP addresses can constitute personal data when they allow the identification of individuals (Case C-582/14, Breyer v. Germany).

In addition, Article 9 establishes "special categories of personal data", including information on racial or ethnic origin, political opinions, religious or philosophical beliefs, trade union membership, genetic and biometric data, health data, and data concerning sex life or sexual orientation. The

processing of such data is prohibited by default and permitted only under strict conditions, requiring the highest security safeguards.

This has direct implications for vulnerability scanning. When a system processes special category data, the resulting vulnerability reports may themselves contain personal data. They are not merely technical artifacts but become part of the overall protection measures, requiring enhanced confidentiality and integrity. For example, technical identifiers such as IP addresses, usernames, or specific API endpoints can qualify as personal data if they allow, directly or indirectly, the identification of a natural person.

In vulnerability scanning reports, several data types may fall within the GDPR definition of personal data:

Directly identifiable data:

- IP addresses linked to specific users or corporate devices
- Usernames or email addresses stored in configuration files
- Hostnames containing personal identifiers

Indirectly identifiable data:

- Log timestamps that, when combined with other information, reveal user activity
- Usage patterns disclosing individual behaviors
- Aggregated metadata that could lead to re-identification

This classification creates a compliance challenge. Vulnerability data is essential for cybersecurity, but its prolonged retention may conflict with GDPR principles such as data minimization (Article 5(1)(c)) and storage limitation (Article 5(1)(e)). Furthermore, accountability (Article 5(2)) requires organizations to demonstrate the lawfulness and proportionality of their retention practices.

This regulatory context is especially important today for organizations. Technical teams need to design specific architectures to automate compliance information retrieval and validate the data extracted.

Possible compliance strategies include:

- **Pseudonymization:** Masking of personal identifiers while preserving data utility for analysis
- **Data aggregation:** Converting detailed records into statistical summaries that prevent re-identification
- **Differentiated retention:** Shorter retention for personal data, with longer retention for anonymized metrics
- **Access and encryption controls:** Use of role-based access control (RBAC) and encryption at rest/in transit to protect sensitive reports

In this context, the APIsec platform integrated into the DevSecOps pipeline automatically flags endpoints that process personal or sensitive information. Security reports generated from these flagged endpoints require special handling, as they inherit the sensitivity of the underlying data. Consequently, stricter measures such as pseudonymization or differentiated retention should be applied to ensure GDPR compliance and reduce the risk of unauthorized disclosure.

5.7 NIS2 Directive

The NIS2 Directive (Directive (EU) 2022/2555) establishes a stronger and more harmonized framework for cybersecurity across the European Union. It applies to both essential and important entities in *critical sectors such as energy, transport, banking, healthcare, digital infrastructure, and public administration* [37].

One of the main objectives of the directive is to raise the overall level of cybersecurity resilience within the Union by setting clearer rules and common requirements. Organizations covered by NIS2

must adopt technical, operational, and organizational measures that are considered both appropriate and proportionate to the risks they face. These measures are expected to address areas such as access control, encryption, secure communication, and the use of cryptographic protections that ensure the **confidentiality** and **integrity** of information.

NIS2 also places a strong emphasis on incident response. Entities must be able to detect, manage, and report significant cybersecurity incidents within strict timeframes, providing early warning and follow-up reports to competent authorities. This creates a more coordinated approach to threat monitoring and crisis handling across Member States.

Another important aspect of the directive is its focus on accountability and governance. Senior management is given direct responsibility for overseeing cybersecurity risk management and ensuring compliance with the directive. Failure to meet these obligations can lead to supervisory actions and financial penalties, as NIS2 introduces stronger enforcement mechanisms and harmonized sanctioning regimes across the EU.

5.8 Architectural Flexibility for Multi-Standard Compliance

The foundational security controls of this architecture provide a strong baseline for aligning with diverse cybersecurity standards. However, these standards often present conflicting requirements, particularly regarding data retention.

This regulatory tension is resolved through the architecture's design that allows the organization to balance security, audit, and privacy requirements by setting appropriate retention periods based on specific legal obligations and internal risk assessments. Rather than adopting a rigid approach, the system provides the flexibility needed to operate across different compliance domains.

The benefits of this flexible and robust approach are significant:

- **Adaptive Compliance:** The system can be configured to meet diverse regulatory mandates, from the one-year requirement of PCI DSS to the six-year term of HIPAA and further, without architectural changes.
- **Risk-Proportionate Retention:** Organizations can implement retention strategies based on risk assessments, retaining critical security artifacts longer than routine operational data.
- **Privacy by Design:** The architecture supports GDPR principles like storage limitation. In addition, granular access control restricts access to sensitive logs and reports, ensuring that only authorized users can handle personal data.

Chapter 6

Conclusions and Future Works

6.1 Conclusions

This thesis has presented the design and implementation of a platform that integrates APIOps and DevSecOps practices to automate the release and security scanning of APIs in enterprise cloud environments. The solution demonstrates how modern DevOps processes can be made both more secure and more efficient by shifting security checks to the left, embedding them earlier in the development and deployment workflow. This approach is not only important for improving organizational workflows, but also for reducing the long-term cost of security maintenance, since vulnerabilities are detected and remediated before they reach production.

The proposed platform shows that automation is a key enabler for scaling API management in large enterprises. By orchestrating pipelines for extraction, publishing, security scanning, rollback, and monitoring, the platform reduces manual effort and standardizes how APIs are released across environments. In doing so, it provides not only speed and consistency, but also detailed audit trails that strengthen compliance with security regulations and industry standards.

Another important contribution of this research is the flexibility of the platform. The pipelines and workflows are highly customizable and can be adapted to different organizational contexts. The current implementation can be considered as a skeleton or a reference architecture, which can be extended with other tools and frameworks as technology evolves. For example, more advanced static or dynamic analysis tools can be integrated, or new logging and monitoring systems can be plugged in without changing the overall architecture. This modularity makes the platform both future-proof and adaptable to enterprise needs.

An additional dimension of this research relates to its potential contribution to digital forensics investigations. The comprehensive logging and audit trail capabilities of the platform create a detailed record of API lifecycle events, including deployment actions, security scan results, configuration changes, and access patterns. This forensic-ready documentation can be invaluable during incident response and digital forensics investigations, as it provides investigators with timestamped, tamper-evident records of system activities. The immutable storage implementation using WORM storage ensures that forensic evidence is preserved without risk of alteration or deletion, while the cryptographic signing of artifacts with cosign provides verifiable authenticity. This combination of immutable storage and cryptographic integrity creates a robust foundation for digital forensics, enabling investigators to reconstruct events with confidence in the evidence's reliability and supporting both incident response

and compliance auditing requirements.

6.2 Future Works

Although the platform already covers the full lifecycle of API release and security testing, several directions for future improvement have been identified during this research.

A first area of extension concerns usability. Currently, the parameters for the pipelines are set manually through the Azure DevOps interface when the master pipeline is triggered. In the future, a dedicated user interface could be developed to allow easier configuration and management of pipeline parameters, making the system more user friendly for non-technical users and lowering the entry barrier for adoption across larger teams.

A second improvement relates to observability and compliance. Enterprises are facing increasingly strict regulations, and advanced logging and monitoring features could be integrated to provide richer audit trails and real-time alerts. These enhancements would make the platform even more suitable for environments where compliance and traceability are critical.

The platform could also be improved by adding another environment in addition to the current test and production stages. A pre-production environment would give organizations the chance to test APIs in conditions that are very close to production, but still in a safe and controlled space. This would make it easier to catch problems earlier and run advanced security checks before releasing to users. Microsoft also underlines in its Cloud Adoption Framework that using multiple environments such as development, test, staging, and production reduces risks, keeps systems more reliable, and supports smoother transitions between stages [38].

From an infrastructure perspective, the current implementation was deployed on a single virtual machine, which hosted the agents, the security tools, and the scheduled jobs for extracting and storing logs. While this approach was effective for prototyping, it does not reflect the scalability requirements of enterprise deployments. Future work could include splitting the architecture across multiple specialized virtual machines or containers, each dedicated to a specific function such as pipeline orchestration, security scanning, logging, or compliance auditing. This specialization would not only improve scalability and reliability, but also enhance security by isolating sensitive components.

A further direction for future development is the extension of the platform to multi-cloud environments. Many organizations are adopting hybrid or multi-cloud strategies to improve resilience, avoid vendor lock-in, and optimize costs. Extending the current solution beyond Azure would require adapting the pipelines and security integrations to heterogeneous cloud providers, while still maintaining a unified governance and compliance model. This would allow the platform to be applied in more diverse enterprise contexts and strengthen its flexibility as a general-purpose APIOps and DevSecOps framework.

Another key direction for future work is to make the platform work not only with Azure API Management but also with other enterprise API management solutions, such as Kong, AWS API Gateway, Layer7 API Gateway, and Tyk. This would require creating abstraction layers and standard interfaces to handle the different APIs, authentication methods, and deployment models of each platform. Making the platform compatible with multiple vendors would expand its use across different enterprise environments and make it a truly vendor-independent solution for APIOps and DevSecOps.

From a digital forensics perspective, future developments for the DevSecOps platform could build upon the existing immutable storage in Azure Blob WORM and the cryptographic integrity provided by Cosign signatures to further enhance forensic capabilities. The platform could export logs in Common Event Format (CEF) or according to the NIST Cybersecurity Framework logging guidelines, enabling seamless ingestion into Microsoft Sentinel for real-time correlation with other enterprise security events. This standardization would facilitate cross-platform forensic analysis and improve integration with existing Security Information and Event Management (SIEM) systems.

A dedicated forensic analysis dashboard could be built on top of the current *tamper-proof audit*

trails, providing investigators with a centralized and immutable view of incidents. This dashboard could include advanced search and filtering capabilities, timeline reconstruction of events, and integration with threat intelligence feeds to enrich the forensic context. By leveraging the existing cryptographic signatures, investigators could verify the authenticity of logs and artifacts directly from the dashboard, streamlining the evidence validation process.

Future iterations could also address business continuity and disaster recovery concerns. This could be achieved by implementing automated failover mechanisms for evidence collection pipelines, ensuring continuous forensic readiness even during infrastructure outages.In the event of security breaches, the platform could be extended with automated incident response capabilities. These capabilities could trigger forensic evidence preservation workflows when security thresholds are breached and integrate with threat intelligence feeds to automatically correlate detected vulnerabilities with known attack patterns.

Finally, the platform could be extended with the integration of more advanced or emerging security tools. As new technologies appear, such as more powerful runtime protection systems, they could be connected to the existing architecture without major changes. In this sense, the solution proposed here provides a solid and flexible backbone that can continue to evolve with the state of the art in DevSecOps and API security.

In conclusion, this research has shown how a combined APIOps and DevSecOps approach can provide a secure, automated, and flexible framework for API management in enterprise cloud environments. The platform not only addresses today's challenges of automation and compliance, but also lays the foundation for continuous improvement and integration of future technologies. It is a practical contribution to the growing need for scalable and secure API management, while also opening several paths for further academic and industrial exploration.

Appendix A

User Manual for the Test Environment

A.1 Introduction

This user manual describes the steps required to reproduce the test environment developed during this thesis project. The environment has been designed entirely on Microsoft Azure and integrates multiple services to simulate both development and production scenarios. Its purpose is to provide a realistic setup where **APIOps** and **DevSecOps** practices can be applied consistently, ensuring security, automation, and traceability across the entire API lifecycle.

The initial phase of the setup focuses on the Azure infrastructure, starting with the deployment of **Azure API Management (APIM)** instances, which act as the central points for API publishing and traffic management. In this project, two APIM instances were created: one simulating the production environment and another for development and testing. Following this, secure storage and key management mechanisms are established through **Azure Key Vault** and **Azure Blob Storage** configured in **WORM (Write-Once-Read-Many)** mode. These services ensure that secrets, cryptographic keys, and audit evidence are managed with integrity and immutability guarantees.

The environment also relies on a **Linux-based virtual machine** of size `Standard D2as v5` (2 vCPUs, 8 GiB RAM). This VM hosts several critical components, including the Azure DevOps self-hosted agents, the APIsec agent, and a containerized instance of SonarQube. It also runs the vulnerable application **crAPI** which was used as the target for security tests. Additionally, a scheduled cronjob automates the collection of reports and their upload into the WORM storage for secure retention. Azure **Managed Identity** is employed to simplify and secure authentication between services, eliminating the need for static credentials and enhancing compliance with cloud-native security practices.

Once the Azure infrastructure is in place, the manual proceeds to describe the **Azure DevOps configuration**, including project setup, service connections, and the design of modular pipelines that integrate static and dynamic analysis, evidence management, and secure release processes.

A.2 Guide Topics

The manual is structured into the following main topics:

- **Prerequisites** This covers the initial requirements for the Azure subscription, the Azure

DevOps organization, and the local workstation tools needed to begin the setup.

- **Azure Infrastructure Setup.** This part details the provisioning of all core cloud resources, including the creation of the Resource Group, the development and production API Management instances, the configuration of Custom Domains with self-signed certificates, and the setup of secure storage with Azure Key Vault and WORM-enabled Blob Storage.
- **Virtual Machine Configuration.** This covers the setup of the Linux VM, from provisioning and initial connection to installing all necessary software (Docker, CLI tools), configuring internal DNS, deploying containerized applications (SonarQube, crAPI), and installing the APIsec and Azure DevOps agents.
- **Azure DevOps and Pipeline Configuration.** The final part explains how to set up the Azure DevOps project, import the source code, configure all prerequisites (Service Connections, Variable Groups, Environments), and finally create and run the DevSecOps pipelines.

A.3 Prerequisites

Before starting the setup, several prerequisites must be satisfied to ensure that the environment can be created and managed correctly.

First, an active **Azure subscription** with sufficient permissions is required. The user should have at least `Contributor` rights on the resource group where the services will be deployed, and `Owner` rights are recommended for full control during the initial configuration. It is also important to ensure that the subscription allows the creation of resources such as API Management instances, Virtual Machines, Key Vault, and Storage Accounts.

Furthermore, access to an **Azure DevOps organization** is required, where projects, repositories, service connections, and pipelines will be configured. A Personal Access Token (PAT) with adequate scopes (such as `Build`, `Code`, and `Service Connections`) will be necessary to enable integration between Azure DevOps and the Azure services.

A.3.1 Installing the Azure CLI

The Azure Command-Line Interface (CLI) is a cross-platform tool used to manage Azure resources directly from the terminal. This manual relies on the CLI for provisioning all the necessary infrastructure components. Before proceeding, you must install it on your local workstation.

The installation process varies depending on your operating system. For the most up-to-date instructions, always refer to the official Microsoft documentation: Install Azure CLI.

- **Windows:** The recommended method is to download and run the MSI installer from the official documentation page. This provides a simple, guided installation that handles all necessary path configurations.
- **macOS:** The easiest way to install the Azure CLI is by using the Homebrew package manager. Open a terminal and run the following command:

```
1 brew update && brew install azure-cli
2
```

- **Linux:** The installation method depends on your distribution's package manager. Microsoft provides a one-liner script for most common distributions. For example, on Debian or Ubuntu, you can use:

```
curl -sL https://aka.ms/InstallAzureCLIDeb | sudo bash

```

It is highly recommended to consult the official documentation to find the correct command for your specific Linux distribution.

Verifying the Installation

After the installation is complete, open a **new** terminal session and run the following command. This will confirm that the CLI was installed correctly and display its version.

```
az --version
```

A.1: Command to verify the Azure CLI installation

A successful installation will output the installed version of the Azure CLI and its components. Once verified, you can proceed with logging into your Azure account.

A.4 Azure Resource Group and APIM Setup

Before creating any services, it is a best practice to organize all resources within a dedicated resource group. A resource group acts as a logical container for all related Azure resources for this project, simplifying management, billing, permissions, and the overall lifecycle. In this guide, it is recommended to create a single resource group to house the API Management instances, storage accounts, virtual machine, and other supporting services.

A.4.1 Creating the Resource Group

A resource group can be created via the Azure Portal, PowerShell, or the Azure CLI. To ensure maximum reproducibility, the following steps describe the procedure using the Azure CLI.

1. Open a terminal or a PowerShell session where the Azure CLI is installed and configured.
2. Log in to your Azure account using the following command:

```
az login

```

A.2: Azure Login Command

   Follow the on-screen instructions to complete the authentication process in your browser.
3. Once authenticated, create the resource group with the `az group create` command. You must specify a unique name for the resource group and the *location* (Azure region) where it will be created.

```
az group create --name <resource-group-name> --location <location
    >

```

A.3: Command to create a Resource Group

Note: Replace `<resource-group-name>` with a name of your choice (e.g., `APIOps-DevSecOps-RG`) and `<location>` with an Azure region (e.g., `westeurope` or `northeurope`).

4. **Practical Example:**

```
az group create --name APIOps-DevSecOps-RG --location westeurope

```

A.4: Practical example for Resource Group creation

If the operation is successful, the terminal will return a JSON output containing the details of the newly created resource group, with `"provisioningState": "Succeeded"`.

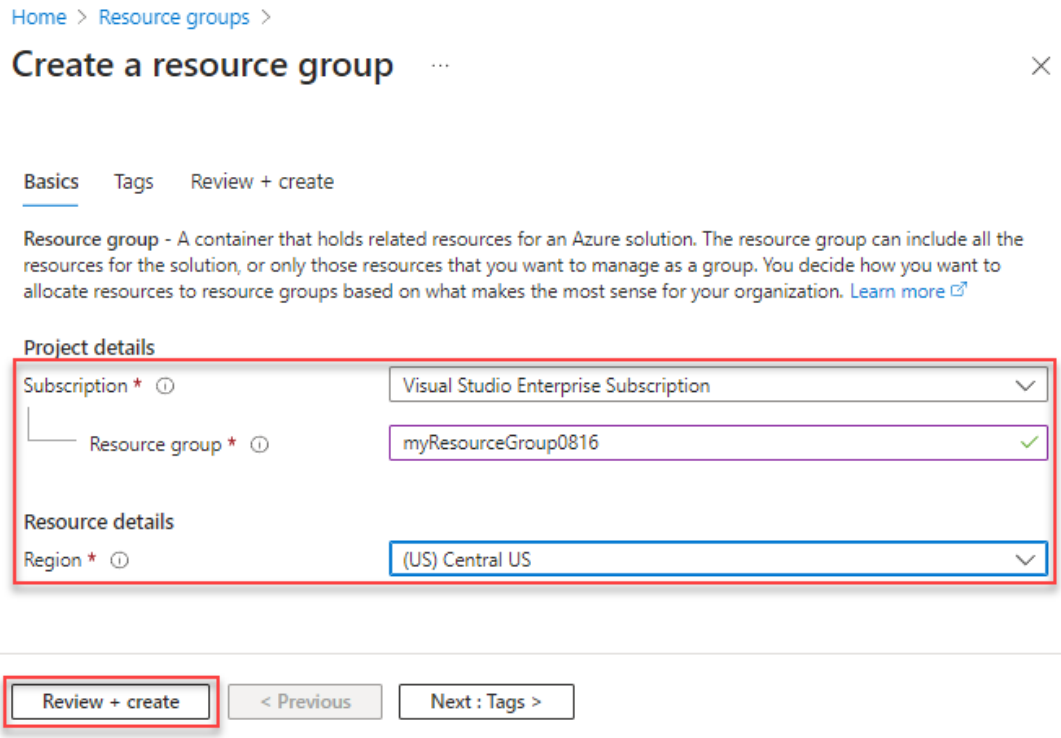


Figure A.1: The resource group creation in the Azure Portal.

For further details on managing resource groups, you can consult the official Microsoft documentation: Manage Azure Resource Groups using Azure CLI.

A.4.2 Creating the API Management Instances

Once the resource group is in place, the next step is to set up the *Azure API Management (APIM)* services. As described in the introduction, two separate instances were created for this project to simulate a realistic API lifecycle:

- A **development instance**, used for deployment validation and security testing.
- A **production instance**, which simulates the live environment with stricter security policies.

The Azure CLI will be used for this procedure as well. The steps must be repeated for both instances.

1. **Create the Development Instance:** Execute the `az apim create` command, providing a unique name, the resource group, organization details, and specifying the `Developer` SKU.

```
az apim create --name <apim-dev-name> \
    --resource-group <resource-group-name> \
    --publisher-name "<your-organization-name>" \
    --publisher-email <your-email@example.com> \
    --sku-name Developer \
    --location <location>

```

A.5: Command to create the Development APIM instance

2. **Practical Example for the Development instance:**

```
az apim create --name apim-dev \
    --resource-group <your-resource-group> \
    --publisher-name "Thesis Project" \
    --publisher-email "admin@thesis.com" \
    --sku-name Developer \
    --location westeurope

```

A.6: Practical example for the Development APIM instance

3. **Create the Production Instance:** Repeat the command with a different name for the production environment.

```
az apim create --name apim-prod \
    --resource-group <your-resource-group> \
    --publisher-name "Thesis Project" \
    --publisher-email "admin@thesis.com" \
    --sku-name Developer \
    --location westeurope

```

A.7: Practical example for the Production APIM instance

4. Once the provisioning process is complete, both instances will be visible in the specified resource group within the Azure Portal.

For a comprehensive guide on all configuration options, refer to the official documentation: Create an API Management service instance using the Azure CLI.

A.4.3 Developer Tier Considerations

In this project, the `Developer` tier was adopted. This SKU is designed for non-production workloads such as evaluation, development, and testing. It provides full access to all APIM features (such as policies, products, users, and developer portal) but does not offer SLA guarantees. This makes it a cost-effective option for prototyping and research purposes, while maintaining feature parity with higher tiers such as `Premium`. Using the Developer tier is considered a best practice in academic or proof-of-concept scenarios, as it reduces cost while enabling a realistic configuration workflow.

A.4.4 Custom Domains

To clearly separate environments and simplify integration with client applications, two custom domains were configured for the API Gateway endpoints of the API Management services. This configuration provides clear isolation, avoids ambiguity when consuming APIs, and presents a professional, branded URL to consumers.

The configured domains are:

- `api-dev.example.com`, pointing to the development APIM instance.
- `api-prod.example.com`, pointing to the production APIM instance.

The setup process is divided into three main steps: generating a self-signed certificate for testing, configuring the custom domain in Azure by uploading the certificate, and finally updating the DNS records. The following sections detail this procedure for the development environment.

Generating a Self-Signed Certificate for Testing

For development and testing environments, a self-signed SSL/TLS certificate can be used to enable HTTPS.

Important Note: Self-signed certificates are not trusted by default by browsers or client applications. They are suitable **only for testing purposes**. For a production environment, a certificate issued by a trusted public CA is mandatory. **Before running,** modify the `$dnsName` variable to match your custom domain.

1. **Open PowerShell as an Administrator.**
2. **Run the generation command.** The command below creates a self-signed certificate correctly configured for a specific domain name and stores it in the current user's personal certificate store.

```
# Set the DNS name for the certificate
$dnsName = "<api-dev.example.com>"

# Generate the certificate
New-SelfSignedCertificate -Subject "CN=$dnsName" `
    -DnsName $dnsName `
    -CertStoreLocation "cert:\CurrentUser\My" `
    -KeyExportPolicy Exportable `
    -KeySpec Signature `
    -KeyLength 2048 `
    -KeyAlgorithm RSA `
    -HashAlgorithm SHA256 `
    -NotAfter (Get-Date).AddYears(2)

```

A.8: PowerShell command to generate a self-signed certificate

   To create the certificate for the production environment, change the variable to `"<api-prod.example.com>"` and re-run the command.

3. **Export the certificate to a `.pfx` file.** Once created, the certificate must be exported to a password-protected `.pfx` file. In the same PowerShell session, run the following script to retrieve and export the certificate.

```
# --- Variables to customize ---
$dnsName = "api-dev.example.com"
```

```
3  $password = "<YourSecurePassword>" # Choose a strong password
4  $pfxPath = "C:\certs\api-dev-certificate.pfx"
5
6  # Get the certificate from the store
7  $cert = Get-ChildItem "cert:\CurrentUser\My" | Where-Object { $_.
       Subject -match "CN=$dnsName" } | Select-Object -First 1
8
9  # Convert the password to a SecureString
10 $securePassword = ConvertTo-SecureString -String $password -Force
       -AsPlainText
11
12 # Export the certificate
13 Export-PfxCertificate -Cert $cert -FilePath $pfxPath -Password
       $securePassword
14
15
```

A.9: PowerShell script to export the certificate

After running the script, the `.pfx` file is ready for the next step.

Configuring the Custom Domain and Uploading the Certificate

The custom domain is configured directly in the API Management instance, a process during which the generated `.pfx` file is uploaded.

1. In the Azure Portal, navigate to your development API Management instance.
2. In the left-hand menu, under the **Gateway** section, select **Custom domains**.
3. Click the + **Add** button.
4. In the "Add custom domain" blade, provide the following information:
   - For **Host name**, enter your custom domain, for example `api-dev.example.com`.
   - For the **Certificate** dropdown menu, select the `Custom` option. This will reveal the fields for file upload.
   - For **Certificate file (.pfx)**, click the folder icon to browse and upload the `.pfx` file created in the previous step.
   - For **Password**, enter the password you chose for the certificate file.
5. Click the **Add** button at the bottom to save the configuration and upload the certificate.

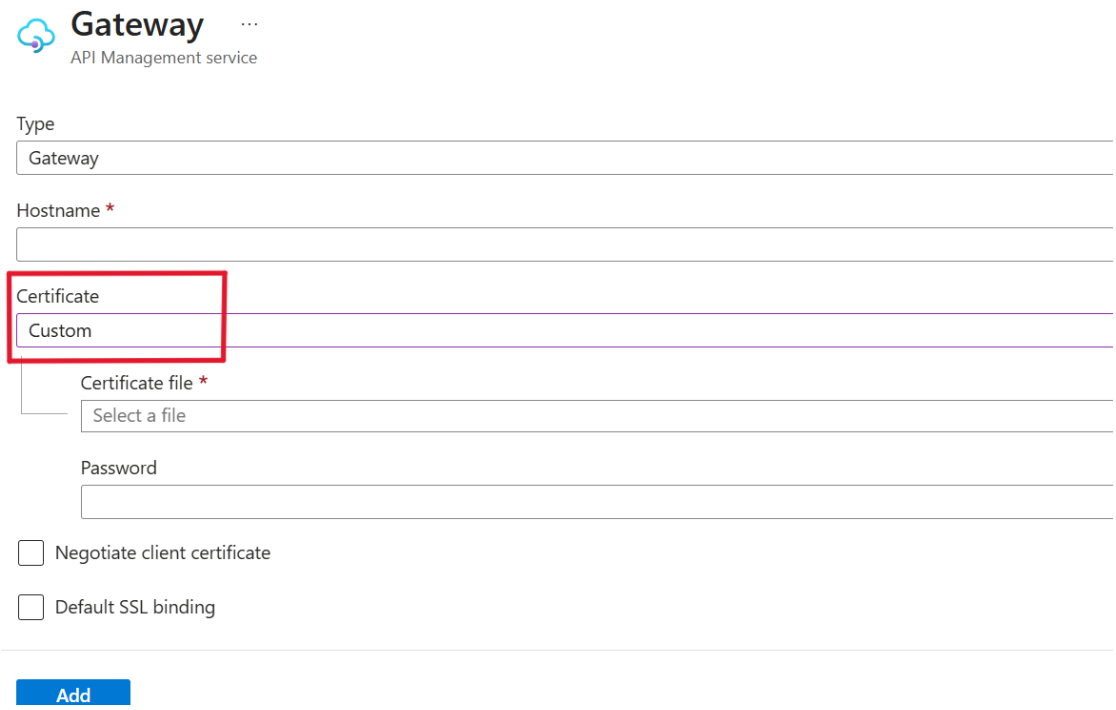


Figure A.2: Uploading the certificate directly within the Custom Domain configuration.

A.5 Key Vault and Blob Storage Setup

Once the API Management instances are in place, the next critical step is to configure the secure storage services that will be used for secrets and for immutable logging. This is a foundational element of the DevSecOps toolchain, ensuring that sensitive data is protected and that compliance artifacts are preserved with integrity. Two Azure services are required for this purpose, *Azure Key Vault* and *Azure Blob Storage* with WORM capabilities.

A.5.1 Azure Key Vault

Azure Key Vault acts as the central, secure repository for secrets, certificates, and cryptographic keys. In this project, it is used to store sensitive items such as API subscription keys, Cosign signing keys, and credentials required by the pipelines. The following steps detail its creation and initial configuration.

1. **Create the Azure Key Vault.** The Key Vault is created using the Azure CLI. The name must be globally unique across all of Azure. For maximum security, both soft-delete and purge protection are enabled. Soft-delete ensures that a deleted vault or secret remains recoverable for a retention period, while purge protection makes that deletion irreversible by anyone, including Microsoft, until the retention period ends. We also enable RBAC for authorization, the modern standard for granting permissions via Managed Identities.

```
# Key Vault name must be globally unique
az keyvault create --name "<your-unique-keyvault-name>" \
```

```
    --resource-group "<your-resource-group>" \
    --location "<location>" \
    --enable-soft-delete true \
    --enable-purge-protection true \
    --enable-rbac-authorization true

```

A.10: Command to create an Azure Key Vault

2. **Assign Permissions for Initial Setup.** By default, no one has permissions to access the secrets in a new RBAC-enabled Key Vault. As an initial step, we assign the "Key Vault Administrator" role to the current user. This allows us to add the necessary secrets for the project. Permissions for Azure services, like the Linux VM, will be granted later using their Managed Identities.

```
# Get the Object ID of the currently signed-in user
USER_ID=$(az ad signed-in-user show --query id -o tsv)

# Assign the "Key Vault Administrator" role to the current user
az role assignment create --role "Key Vault Administrator" \
    --assignee $USER_ID \
    --scope "/subscriptions/$(az account show --query id -o tsv)/
    resourceGroups/<your-resource-group>/providers/Microsoft.
    KeyVault/vaults/<your-key-vault-name>"

```

A.11: Command to assign permissions to the current user

Note, it may take a minute or two for the role assignment to become effective.

3. **Add a Secret to the Vault.** As a practical example, a placeholder secret for a pipeline credential can be added.

```
az keyvault secret set --vault-name <your-key-vault-name> \
    --name "CosignSigningKeyPassword" \
    --value "<your-password>"

```

A.12: Command to add a secret to the Key Vault

For a full reference on creating a Key Vault with the Azure CLI, refer to the official Microsoft quickstart guide: Quickstart: Create a key vault using the Azure CLI.

A.5.2 Azure Blob Storage with WORM Mode

For audit trails and security reports, the system requires an immutable storage solution. Azure Blob Storage provides this via *immutability policies* (Write Once, Read Many, or WORM).

1. **Create a Storage Account.** A general-purpose v2 storage account is required. Storage account names must be globally unique and contain only lowercase letters and numbers.

```
# Storage account name must be globally unique and lowercase
STORAGE_ACCOUNT_NAME="evidence$(openssl rand -hex 4)"

```

```
az storage account create --name $STORAGE_ACCOUNT_NAME \
    --resource-group <your-resource-group> \
    --location westeurope \
    --sku Standard_LRS \
    --kind StorageV2
```

A.13: Command to create a new storage account

2. **Enable Blob Versioning.** Time-based retention policies require that versioning is enabled on the storage account. This is a critical prerequisite to ensure that any modification to a blob results in a new version instead of overwriting the original, preserving the full history.

```
az storage account blob-service-properties update \
    --account-name $STORAGE_ACCOUNT_NAME \
    --resource-group <your-resource-group> \
    --enable-versioning true
```

A.14: Command to enable versioning on the storage account

3. **Create a Storage Container.** A container is created within the storage account to hold the evidence files.

```
az storage container create --name "<your-container-name>" \
    --account-name $STORAGE_ACCOUNT_NAME \
    --auth-mode login
```

A.15: Command to create a blob container

4. **Apply a Time-based Retention Policy.** Finally, we apply the immutability policy to the container. For this project, a time-based retention policy is used, ensuring reports remain tamper-proof for a specified compliance period. The policy is created in an **Unlocked** state for flexibility during testing. Once the system is validated, this policy can be locked, making it irreversible for its duration.

```
az storage container immutability-policy create \
    --account-name $STORAGE_ACCOUNT_NAME \
    --container-name "<your-container-name>" \
    --period 30 \
    --policy-mode Unlocked
```

A.16: Command to set a 30-day WORM policy

With the secure storage components configured, the next step is to deploy the Linux Virtual Machine which will serve as the primary compute host for the automation agents.

For further details on container-level immutability policies, the official Microsoft documentation can be consulted here: Configure immutability policies for container-level WORM.

A.6 Virtual Machine Setup

The platform requires a dedicated compute environment to host the various agents and applications for CI/CD, security scanning, and testing. For this purpose, a Linux-based Azure VM was created in the same resource group as the other services. The selected size was **Standard_D2as_v5** (2 vCPUs, 8 GiB RAM), which provides a balance of performance and cost for the intended workload. This section details the creation and configuration of this virtual machine.

A.6.1 Network Infrastructure Setup

A dedicated virtual network (VNet) is required to provide a secure and isolated environment for the virtual machine and other services. This VNet will host the subnet for the self-hosted agent.

1. **Create the Virtual Network (VNet).** This command creates a VNet with a user-defined address space. Replace the placeholders with your desired values.

```
az network vnet create \
    --name <your-vnet-name> \
    --resource-group <your-resource-group> \
    --address-prefix <vnet-address-prefix>

```

A.17: Command to create the VNet

2. **Create the Subnet.** Within the VNet, create a specific subnet where the VM will be placed. Ensure the subnet prefix is within the VNet's address space.

```
az network vnet subnet create \
    --name <your-subnet-name> \
    --resource-group <your-resource-group> \
    --vnet-name <your-vnet-name> \
    --address-prefixes <subnet-address-prefix>

```

A.18: Command to create the agent's subnet

With the network infrastructure in place, you can now proceed to provision the virtual machine within this secure environment. For additional details, refer to the official Microsoft documentation: Quickstart: Create an Azure Virtual Network.

A.6.2 Provisioning the Linux Virtual Machine

The VM is provisioned using the Azure CLI. A key step in this process is the enablement of a **system-assigned managed identity**. This creates an identity for the VM in Microsoft Entra ID that can be used to authenticate to other Azure services, like Key Vault and Blob Storage, without needing to store any credentials on the VM itself.

1. **Create the Virtual Machine.** The command below creates an Ubuntu LTS VM with the specified size, enables the system-assigned managed identity, and generates SSH keys for access.

```
# Replace <vnet-name> and <subnet-name> with your network details
az vm create --name apiops-agent-vm \
    --resource-group <your-resource-group> \
```

```
    --size Standard_D2as_v5 \
    --image UbuntuLTS \
    --vnet-name <vnet-name> \
    --subnet <subnet-name> \
    --admin-username <vm-user-name> \
    --generate-ssh-keys \
    --assign-identity '[system]'

```

A.19: Command to create the Linux VM with a Managed Identity

2. **Note the Public IP Address.** After creation, the command will output a JSON block containing the public IP address of the VM. Note this IP, as it will be used to connect to the machine .

A.6.3 Connecting and Installing Prerequisites

Once the VM is running, the next step is to connect to it and install the necessary software prerequisites.

1. **Connect to the VM .** Use the noted public IP address and the generated SSH key to connect.

```
ssh <vm-user-name>@<public-ip-address>
```

A.20: Connecting to the VM using SSH

2. **Update the System.** It is a best practice to update the package lists and upgrade all installed packages.

```
sudo apt update && sudo apt upgrade -y
```

A.21: Updating the Linux system

3. **Install Docker and Docker Compose.** These are required to run the containerized instances of SonarQube and crAPI.

```
# Install Docker Engine
sudo apt install docker.io -y
sudo systemctl start docker
sudo systemctl enable docker
sudo usermod -aG docker $USER

# Install Docker Compose
sudo apt install docker-compose -y

```

A.22: Installing Docker and Docker Compose

Note, you may need to log out and log back in for the user group changes to take effect.

4. **Install other required tools,** such as Git, Azure CLI (for the cronjob), and Cosign (check for the latest version at this link: Cosign releases) .

```
# Install Git
sudo apt install git -y

# Install Azure CLI
curl -sL https://aka.ms/InstallAzureCLIDeb | sudo bash

# Install Cosign
wget "https://github.com/sigstore/cosign/releases/download/<
    latest.version>/cosign-linux-amd64"
sudo mv cosign-linux-amd64 /usr/local/bin/cosign
sudo chmod +x /usr/local/bin/cosign

```

A.23: Installing Git, Azure CLI, and Cosign

A.6.4 Configuring DNS Resolution on the Linux Agent VM

As the API Management instances in this project are deployed within an Azure Virtual Network (VNet) and are not publicly exposed, name resolution for the custom domains must be handled internally.

This configuration is critical on the VM that hosts the DevOps self-hosted agents, the APIsec agent, and the SonarQube container. These components need to communicate with the APIM instances using their custom domain names (e.g., `api-dev.example.com`) during pipeline executions for tasks like API deployments and security scanning. Without proper name resolution, these operations would fail.

The most direct method to achieve this within the VM is by modifying the **'/etc/hosts'** file. This file acts as a local DNS resolver, allowing you to manually map IP addresses to hostnames.

The procedure is as follows:

1. **Find the Private IP Addresses of the APIM Gateways.** Before editing the file, you need the internal IP addresses for **both** the development and production APIM instances. You can find these in the Azure Portal by navigating to each APIM instance and looking for the "Private IP address" in the **Overview** blade.
2. **Connect to the Linux VM and Edit the 'hosts' file.** Connect to your Linux VM. Then, open the '/etc/hosts' file with a text editor that has root privileges, such as 'nano' or 'vim'.

```
sudo nano /etc/hosts
```

A.24: Command to open the hosts file for editing

3. **Add the New Entries.** At the end of the file, add new lines to map the private IPs of your APIM gateways to the respective custom domain names. Replace the placeholders with the actual IP addresses you found in the first step.

```
# Azure APIM Custom Domains for Thesis Project
<dev-apim-private-ip>    api-dev.example.com
<prod-apim-private-ip>   api-prod.example.com
```

A.25: Example entries in the VM's /etc/hosts file

4. **Save and Close the File.** If using ‘nano‘, press ‘Ctrl+X‘, then ‘Y‘ to confirm, and ‘Enter‘ to save. Once saved, the VM will immediately be able to resolve the custom domain names to the correct internal IP addresses.

With local DNS configured, the VM can now correctly resolve the custom domain names to the private APIM endpoints.

For a full reference on the topic, the official Microsoft guide is available here: Configure a custom domain name for your Azure API Management instance.

A.6.5 Deploying Containerized Applications (SonarQube and crAPI)

With Docker and its prerequisites installed, the test application and static analysis tool can be deployed.

Deploying SonarQube with a PostgreSQL Database.

For a robust and persistent setup, SonarQube is deployed using two linked containers: one for the SonarQube application itself and another for its PostgreSQL database. A dedicated Docker network is created to allow the containers to communicate.

1. **Create a Docker network.**

```
docker network create sonarnet
```

A.26: Creating a dedicated Docker network

2. **Run the PostgreSQL database container.** This container stores all SonarQube data, with its data persisted to a Docker volume named `pg_data`.

```
1 docker run -d ‘
2     --name sonarqube-db ‘
3     --network sonarnet ‘
4     -e POSTGRES_USER=<sonaruser> ‘
5     -e POSTGRES_PASSWORD=<sonarpassword> ‘
6     -e POSTGRES_DB=sonarqube ‘
7     -v pg_data:/var/lib/postgresql/data ‘
8     postgres:15
9
```

A.27: Command to run the PostgreSQL container

3. **Run the SonarQube application container.** This container connects to the database via the `sonarnet` network. Its data, extensions, and logs are persisted to Docker volumes and a local host path for easy access.

```
1 docker run -d ‘
2     --name sonarqube ‘
3     --network sonarnet ‘
4     -p 9000:9000 ‘
5     -e SONAR_JDBC_URL=jdbc:postgresql://sonarqube-db:5432/
    sonarqube ‘
```

```
    -e SONAR_JDBC_USERNAME=<sonaruser> `
    -e SONAR_JDBC_PASSWORD=<sonarpassword> `
    -e SONAR_WEB_ACCESSLOGS_ENABLE=true `
    -e SONAR_WEB_ACCESSLOGS_PATTERN='%h [%t] "%r" %s "%i{Referer}" "%i{User-Agent}" "%reqAttribute{ID}" %D "%reqAttribute{LOGIN}"' `
    -v sonarqube_data:/opt/sonarqube/data `
    -v sonarqube_extensions:/opt/sonarqube/extensions `
    -v path/to/sonarqube_logs:/opt/sonarqube/logs `
    sonarqube:community

```

A.28: Command to run the SonarQube container

After running these commands, the containers can be started or stopped as needed using `docker start sonarqube` and `docker start sonarqube-db`.

A.6.6 Installing the APIsec Agent

The APIsec agent is installed to perform dynamic security testing. The standard installation command is provided by the vendor's platform, but it must be customized to work in this project's internal network environment, which uses private SSL certificates.

The process involves obtaining the base command from the APIsec dashboard and then adapting it to use a custom-built Docker image that trusts the internal APIM endpoints.

1. **Obtain the Base Command from the APIsec Dashboard.** First, log in to the APIsec platform at link: apisec.ai. Navigate to the agent management dashboard to create a new agent. The platform will provide a complete `docker run` command personalized with the necessary authentication variables (like `BROKER_ID` and `BROKER_TOKEN`). This command is the starting point for our customization. Copy this entire command and save it into a new shell script file on the VM, for example, `run_apisec_agent.sh`. This file will be edited in a later step.
2. **Define Configuration Variables.** On the Linux VM, prepare a script with the following variables to define the target hostnames and the name for the new custom Docker image.

```
HOSTNAME_DEV="<api-dev.example.com>"
HOSTNAME_PROD="<api-prod.example.com>"
CERT_FILE_DEV="<apim_dev_ca.crt>"
CERT_FILE_PROD="<apim_prod_ca.crt>"
CUSTOM_IMAGE_NAME="apisec-agent-custom:latest"

```

A.29: Configuration variables for the script

3. **Download APIM Public Certificates.** The `openssl` command connects to each APIM gateway and saves its public SSL certificate, which is necessary for the custom agent to trust these endpoints.

```
openssl s_client -connect $HOSTNAME_DEV:443 -showcerts </dev/null | openssl x509 -outform PEM > "$CERT_FILE_DEV"
openssl s_client -connect $HOSTNAME_PROD:443 -showcerts </dev/null | openssl x509 -outform PEM > "$CERT_FILE_PROD"

```

A.30: Downloading the public certificates

4. **Create a Custom Dockerfile.** A `Dockerfile` is generated on the fly. It uses the official APIsec agent as a base and adds the downloaded certificates to the Java truststore using `keytool`.

```
cat > Dockerfile <<EOF
# Use the official agent image as a base
FROM apisec/hostedagent:2025-06-27-40

# Copy certificates into the image
COPY $CERT_FILE_DEV /tmp/$CERT_FILE_DEV
COPY $CERT_FILE_PROD /tmp/$CERT_FILE_PROD

# Add certificates to the Java truststore
RUN keytool -importcert -trustcacerts -file /tmp/$CERT_FILE_DEV -
    alias apim-dev-ca -cacerts -storepass changeit -noprompt
RUN keytool -importcert -trustcacerts -file /tmp/$CERT_FILE_PROD
    -alias apim-prod-ca -cacerts -storepass changeit -noprompt
EOF

```

A.31: Dynamically creating the Dockerfile

5. **Build the Custom Docker Image.** The `Dockerfile` is used to build the new, trusted agent image.

```
DOCKER_BUILDKIT=1 docker build -t "$CUSTOM_IMAGE_NAME" .

```

A.32: Building the custom Docker image

6. **Adapt and Run the Agent Container.** Finally, take the original `docker run` command obtained from the APIsec dashboard and modify it. The key changes are:

   - Replace the official image name (e.g., `apisec/hostedagent:...`) with the custom image name (`$CUSTOM_IMAGE_NAME`).
   - Add the `-add-host` flags for container-level DNS resolution.
   - Add the `JAVA_TOOL_OPTIONS` environment variable to point to the correct truststore.

```
# Adapt the command from the APIsec dashboard with the variables
    below
docker run -d --name apisechostedagent --cpus=1 --memory=2g \
    --add-host="$HOSTNAME_DEV:<dev-apim-private-ip>" \
    --add-host="$HOSTNAME_PROD:<prod-apim-private-ip>" \
    -e MANAGER_HOST_URL="<your-manager-host-url>" \
    -e BROKER_ID=<your-broker-id> \
    -e BROKER_TOKEN="<your-broker-access-token>" \
    -e JAVA_TOOL_OPTIONS="-Djavax.net.ssl.trustStore=$JAVA_HOME/
    lib/security/cacerts -Djavax.net.ssl.trustStorePassword=
    changeit" \
```

```
    "$CUSTOM_IMAGE_NAME"

```

A.33: Example of the final, adapted docker run command

Save the file, make it executable, and run it to start the agent:

```
chmod +x run_apisec_agent.sh
./run_apisec_agent.sh
```

A.34: Running the customized APIsec agent

A.6.7 Deploying the crAPI Test Application

The "Completely Ridiculous API" (crAPI) is downloaded and deployed from its official repository to serve as a vulnerable test target.

1. **Download and unzip the application repository.**

```
curl -o /tmp/crapi.zip https://github.com/OWASP/crAPI/archive/
    refs/heads/main.zip
unzip /tmp/crapi.zip
cd crAPI-main/deploy/docker/

```

A.35: Downloading and extracting crAPI

2. **Pull and run the containers using Docker Compose.** The `LISTEN_IP` variable is set to ensure the services are accessible from outside the container.

```
docker compose pull
LISTEN_IP="0.0.0.0" docker compose -f docker-compose.yml --
    compatibility up -d

```

A.36: Running crAPI with Docker Compose

A.6.8 Installing the Azure DevOps Self-Hosted Agent

To execute pipelines within the project's private network, a self-hosted agent must be configured on the Linux VM. This agent will connect to Azure DevOps to listen for and run pipeline jobs. The setup process is initiated from the Azure DevOps portal, which provides the specific commands needed for the installation.

1. **Navigate to the Agent Pools Section.** Log into your Azure DevOps organization and navigate to the correct project. From the bottom-left corner, select **Project settings**. In the settings menu, under the **Pipelines** category, click on **Agent pools**.
2. **Select the Target Pool and Create a New Agent.** Select the agent pool where you want to add the new agent (e.g., the default `Default` pool or a custom one). Inside the pool, click the **New agent** button on the top-right.
3. **Follow the On-Screen Instructions for Linux.** A pop-up window will appear with tabs for different operating systems. Select the **Linux** tab. This screen provides the exact, up-to-date commands required to download and configure the agent on your VM. The steps are as follows.

4. **Prepare the VM.** Connect to your Linux VM. The first step is to create a directory for the agent and download the agent package using the provided `wget` command.

```
# Create a directory for the agent
mkdir myagent && cd myagent

# Download the agent package using the URL from the DevOps UI
wget <URL provided in the Azuregem DevOps agent setup window>

```

A.37: Downloading the agent package

5. **Unpack and Configure the Agent.** Next, unpack the downloaded archive. The crucial step is running the interactive configuration script, `./config.sh`.

```
# Run the interactive configuration script
./config.sh
```

A.38: Configuring the agent

During execution, the script will prompt for server URL, agent pool name, and other details. For authentication, when prompted for the authentication type, select **PAT (Personal Access Token)**.

This token acts as a password for the agent. To generate one, navigate to **User settings** in the top-right corner of the Azure DevOps portal and then to **Personal Access Tokens**, as shown in Figure A.3. Create a new token, ensuring it has the **Agent Pools (Read & manage)** scope. Copy the generated token immediately, as it will not be displayed again, and paste it into the script's prompt when requested.

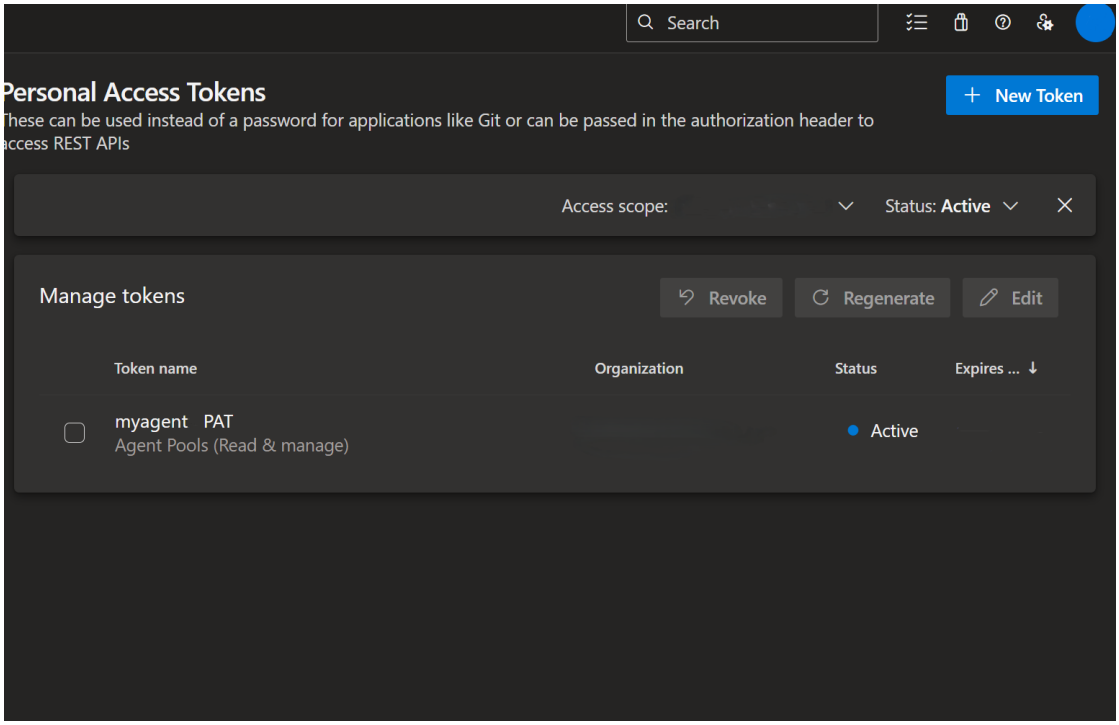


Figure A.3: Generating a Personal Access Token (PAT) in Azure DevOps.

After completing these steps, the new self-hosted agent will appear as "Online" in the Azure DevOps agent pool, ready to run pipelines. For further details, the official Microsoft documentation can be referenced here: Deploy an Azure Pipelines agent on Linux.

A.6.9 Automated Evidence Collection Setup

A cronjob is configured to run a script that collects, signs, and uploads evidence files to the immutable storage container.

1. **Create the upload script.** Create a shell script on the VM, for example at `path/to/upload_evidence.sh`.

```
#!/bin/bash
set -euo pipefail

echo "[INFO] Cron script started at $(date +"%Y-%m-%dT%H:%M:%S%:z")"

# Parameters
RESOURCE_GROUP="<your-resource-group>"
STORAGE_ACCOUNT_NAME="<your-storage-account-name>"
CONTAINER_NAME="<your-container-name>"
VM_USER="<vm-user-name>"

BASE_DIR="/home/$VM_USER"
```

```
13 EXPORT_DIR="$BASE_DIR/export_artifacts"
14 SCRIPT_DIR="$BASE_DIR/scripts"
15
16 TIMESTAMP=$(date +%Y-%m-%d_%H-%M)
17 ZIP_FILE="$EXPORT_DIR/devsecops-artifacts-$TIMESTAMP.zip"
18
19 # 1. Run data-generation scripts
20 mkdir -p "$EXPORT_DIR"
21 "$SCRIPT_DIR/access_log_to_csv.sh"      || { echo "[ERROR]
       access_log_to_csv.sh failed"; exit 1; }
22 "$SCRIPT_DIR/export_sonarqube_data.sh" || { echo "[ERROR]
       export_sonarqube_data.sh failed"; exit 1; }
23 "$SCRIPT_DIR/export_other_data.sh" || { echo "[ERROR]
       export_other_data.sh failed"; exit 1; }
24 # 2. Build ZIP with all generated files
25 echo "[INFO] Creating ZIP archive: $ZIP_FILE"
26 FILES_TO_ZIP=$(find "$EXPORT_DIR" -maxdepth 1 -type f)
27
28 if [[ -z "$FILES_TO_ZIP" ]]; then
29   echo "[WARNING] No artifacts found in $EXPORT_DIR. Exiting."
30   exit 0
31 fi
32 zip -j "$ZIP_FILE" $FILES_TO_ZIP
33
34 # 3. Upload ZIP to Blob Storage
35 echo "[INFO] Uploading ZIP to Azure Blob Storage..."
36 # Using --auth-mode login to authenticate with the VM's Managed
       Identity
37 az storage blob upload \
38     --account-name   "$STORAGE_ACCOUNT_NAME" \
39     --container-name "$CONTAINER_NAME" \
40     --name           "$(basename "$ZIP_FILE")" \
41     --file           "$ZIP_FILE" \
42     --auth-mode      login \
43     --only-show-errors
44
45 echo "[INFO] Cron script finished at $(date +"%Y-%m-%dT%H:%M:%S%:
       z")"
46
```

A.39: Content of the upload_evidence.sh script

Make the script executable with `chmod +x /path/to/upload_evidence.sh`.

2. **Create the cronjob.** Open the crontab editor with ***crontab -e*** and add a line to schedule the script to run daily (at 2:00 AM in this example).

```
0 2 * * * path/to/upload_evidence.sh
```

A.40: Example cronjob entry

A.6.10 Core Evidence Processing Scripts

The main orchestration script (`upload_evidence.sh`) does not perform the data processing tasks directly. Instead, it delegates these responsibilities to a set of modular helper scripts, such as `access_log_to_csv.sh`, `export_sonarqube_data.sh`, etc... These scripts implement the core logic for transforming raw data into forensically sound evidence.

The following subsections detail the three key stages of this process: securely retrieving cryptographic keys, standardizing unstructured data, and applying digital signatures.

Securely Retrieving Secrets from Azure Key Vault

A foundational security principle of this process is that no cryptographic private keys are ever stored on the VM's disk. All keys required for signing are retrieved directly from Azure Key Vault at runtime using the VM's Managed Identity.

```
VAULT_NAME="<your-key-vault-name>"
COSIGN_SECRET_NAME="<your-cosign-private-key-secret-name>"
COSIGN_PUB_NAME="<your-cosign-public-key-secret-name>"
COSIGN_PASSWORD_NAME="<your-cosign-password-secret-name>"

COSIGN_PRIVATE_KEY=$(az keyvault secret show \
    --vault-name "$VAULT_NAME" \
    --name "$COSIGN_SECRET_NAME" \
    --query "value" -o tsv | base64 -d)

COSIGN_PUBLIC_KEY=$(az keyvault secret show \
    --vault-name "$VAULT_NAME" \
    --name "$COSIGN_PUB_NAME" \
    --query "value" -o tsv | base64 -d)

COSIGN_PASSWORD=$(az keyvault secret show \
    --vault-name "$VAULT_NAME" \
    --name "$COSIGN_PASSWORD_NAME" \
    --query "value" -o tsv)
```

A.41: Retrieving secrets from Azure Key Vault

Data Formatting and Standardization

Raw log files are often unstructured and difficult to analyze. A key step is to parse these logs and transform them into a standardized, structured format like CSV. The `awk` utility is used for this powerful text processing. The script first defines a CSV header, then uses an `awk` program to parse the source log and format the output.

```
# Define the CSV header
echo "ip;date;time;method;url;http_version;status;referrer;user_agent
    ;request_id;latency;user_name" > "$CSV_FILE"

# Use awk to parse the source log and append to the CSV
awk '{
    # ... awk parsing logic to extract variables like ip, date, etc.
    ...
```

```
7
8       # Print the extracted variables in the correct CSV format
9       printf "\"%s\";\"%s\";\"%s\";\"%s\";\"%s\";\"%s\";\"%s\";\"%s\";\"%s\";\"%s\";\"%s\";\"%s\";\"%s\";\"%s\";\"%s\"\n", ip, date, time, method, url, http_ver, status, referrer, user_agent, request_id, latency, user_name
10
11  }' "$SOURCE_LOG" >> "$CSV_FILE"
```

A.42: Example of log formatting logic

Digital Signing with Cosign

To guarantee the integrity and non-repudiation of each artifact, a digital signature is created using Cosign. The private key, fetched securely from Key Vault, is exported as an environment variable. Cosign is then instructed to use this key to sign the artifact file and produce a separate signature file. An optional verification step is included to ensure the signature is valid immediately after creation.

```
1  # Sign the artifact, creating a .sig file
2  cosign sign-blob --yes \
3      --key env://COSIGN_PRIVATE_KEY \
4      --output-signature "$CSV_FILE.sig" "$CSV_FILE"
5
6  # Verify the signature with the public key
7  cosign verify-blob \
8      --key env://COSIGN_PUBLIC_KEY \
9      --signature "$CSV_FILE.sig" "$CSV_FILE"
```

A.43: Signing an artifact with Cosign

This process is repeated for every artifact. The resulting files (e.g., `audit.csv` and `audit.csv.sig`) are then packaged together by the main script and uploaded to the WORM storage, providing a complete and verifiable audit trail.

A.6.11 Configuring Managed Identity Permissions

A core security principle of this architecture is the elimination of static credentials. Instead of using passwords or access keys, Azure services authenticate to each other using **Managed Identities**.

The Linux VM was provisioned with a **system-assigned managed identity**, which is an identity tied directly to the lifecycle of the VM itself. Now that the VM and the other resources (Key Vault, Blob Storage) exist, we must grant the VM's identity the necessary permissions to access them. This is done by assigning specific roles to the identity's service principal.

1. **Get the VM's Managed Identity Principal ID.** First, we need to retrieve the unique ID (known as the Principal ID or Object ID) of the VM's managed identity.

```
1  VM_PRINCIPAL_ID=$(az vm identity show -g <your-resource-group> -n
       apiops-agent-vm --query principalId -o tsv)
2
```

A.44: Command to get the VM's Principal ID

2. **Grant Key Vault Permissions.** To allow the pipelines running on the VM to fetch secrets (like API keys or passwords), we assign the "Key Vault Secrets User" role. This grants read-only access to secrets, adhering to the principle of least privilege.

```
1 az role assignment create --assignee $VM_PRINCIPAL_ID \
2     --role "Key Vault Secrets User" \
3     --scope "/subscriptions/$(az account show --query id -o tsv)/
    resourceGroups/<your-resource-group>/providers/Microsoft.
    KeyVault/vaults/<your-key-vault-name>"
4
```

A.45: Command to grant Key Vault permissions

3. **Grant Blob Storage Permissions.** To allow the cronjob on the VM to upload evidence reports to the WORM container, we assign the "Storage Blob Data Contributor" role. This grants write permissions to the blob container.

```
1 az role assignment create --assignee $VM_PRINCIPAL_ID \
2     --role "Storage Blob Data Contributor" \
3     --scope "/subscriptions/$(az account show --query id -o tsv)/
    resourceGroups/<your-resource-group>/providers/Microsoft.
    Storage/storageAccounts/$STORAGE_ACCOUNT_NAME"
4
```

A.46: Command to grant Blob Storage permissions

With these role assignments in place, any script or application on the VM that uses the Azure CLI or Azure SDKs can automatically authenticate to Key Vault and Blob Storage using the VM's identity, without needing any stored secrets.

A.7 Azure DevOps Setup and Pipeline Configuration

This section describes the necessary steps to configure an Azure DevOps project to run the CI/CD and DevSecOps workflows defined for this thesis.

The source code for all YAML pipelines discussed in this manual is publicly available in the following GitHub repository:

`https://github.com/alereply/azure-devsecops-pipelines-thesis.git`

The following steps will guide you through creating an Azure DevOps project, importing this source code, and configuring the necessary prerequisites to make the pipelines operational.

A.7.1 Create a New Azure DevOps Project

The first step is to create a dedicated project to house the repositories, pipelines, and other components.

1. Log into your Azure DevOps organization at `dev.azure.com`.
2. Click on the **New project** button.
3. Provide a **Project name** (e.g., `SecureAPILifecycle`).
4. Choose the project visibility (e.g., `Private`).

5. Under Advanced settings, ensure that the version control is set to **Git**.
6. Click **Create**. This will provision a new project with services like Azure Repos and Azure Pipelines enabled.

A.7.2 Import the Git Repository

Next, you need to populate Azure Repos with the pipeline code from the public GitHub repository.

1. Within your newly created project, navigate to the **Repos** section.
2. Since the repository is new, you will see several options to get started. Choose the **Import a repository** option.
3. In the "Import a Git repository" dialog paste the URL of the repository.
4. Click **Import**. Azure DevOps will clone the entire repository, including all YAML files and scripts, into your project's Azure Repos.

A.7.3 Configure Project Prerequisites

Before the pipelines can run, you must configure the necessary connections, variables, and resources within the project settings.

Agent Pool

The pipelines are designed to run on a self-hosted agent. Navigate to **Project settings** > **Agent pools** (under the Pipelines section). Ensure that the agent pool you created during the Virtual Machine setup (e.g., `<your-agent-pool-name>`) is present and that the agent is shown as "Online".

Service Connections

The pipelines need to authenticate with external services. Navigate to **Project settings** > **Service connections**. As shown in the project configuration, three connections are required:

- **Azure Resource Manager** : These connections (e.g., `service-connection-apim`, `service-connection-keyvault`) link Azure DevOps to your Azure subscription, allowing pipelines to manage resources. When creating them, it is recommended to use the **Service Principal (automatic)** authentication method for enhanced security.
- **SonarQube:** This connection allows the static analysis pipeline to communicate with your SonarQube server, enabling it to publish scan results and check Quality Gates.

The names you choose for these connections must match the values specified in the pipeline YAML files or in the variable groups.

Variable Groups (Library)

Sensitive information and parameters are managed in the Azure DevOps Library. Navigate to **Pipelines** > **Library** and create the following four variable groups. Inside each group, create the specified variables, providing your own custom values where indicated.

apim-automation-group: Contains general configuration for the APIM instances. Create the following variables:

- `APIM_DEV_NAME`: The name of the development APIM service.
- `APIM_PROD_NAME`: The name of the production APIM service.
- `APIM_RG_NAME`: The name of the resource group where the APIM instances are located.
- `APIM_DEV_HOSTNAME`: The custom domain for the development gateway (e.g., `api-dev.example.com`).
- `APIM_PROD_HOSTNAME`: The custom domain for the production gateway (e.g., `api-prod.example.com`).

APIsec-group: Holds the credentials for the APIsec platform.

- `APISEC_TOKEN`: The access token for the APIsec API. Remember to mark this variable as a secret by clicking the lock icon.

blob-worm-storage-group: Contains details for the immutable storage account.

- `BLOB_RG_NAME`: The resource group of the storage account.
- `STORAGE_ACCOUNT_NAME`: The name of the WORM storage account.
- `CONTAINER_NAME`: The name of the blob container for evidence.

cosign-group: This group manages the cryptographic keys by linking directly to Azure Key Vault.

- After creating the group, toggle on **Link secrets from an Azure key vault as variables**.
- Choose your Azure subscription and the Key Vault service you created earlier.
- Authorize the connection if prompted.
- Add references to the secrets within the vault: `cosign-password`, `cosign-private-key`, and `cosign-public-key`.

Environments and Manual Approvals

To enforce a "human-in-the-loop" governance model for critical operations, such as deploying to production or applying a Git tag, Azure DevOps uses **Environments**. An environment is a logical boundary that can be protected with manual approval checks. Before a pipeline stage targeting a protected environment can run, one or more designated users must give their explicit approval.

The following steps describe how to create an environment and configure a manual approval check:

1. In your Azure DevOps project, navigate to **Pipelines** > **Environments**.
2. Click the **New environment** button.
3. In the creation panel:
    - Enter a **Name** for the environment that matches the one used in your pipeline YAML (e.g., `prod` or `tagging-gate`).
    - For **Resource**, select **None**. This type of environment is used as a logical gate for deployment jobs, not for targeting specific virtual machines.
    - Click **Create**.
4. Once the environment is created, click on the three vertical dots (⋮) on the environment's card and select **Approvals and checks**.

5. Click on the **Approvals** check.
6. In the configuration panel, add the required **Users and groups** who must approve any deployment targeting this environment. You can also configure settings like timeouts.
7. Click **Create**.

From now on, any pipeline `deployment` job that references this environment name will automatically pause and wait for a manual approval before executing its steps, as shown in the example below.

```
- stage: Deploy_PROD
  jobs:
  - deployment: Deploy_APIs_PROD
    displayName: 'Publishing to APIM PROD'
    environment: 'prod' # This name triggers the approval check
    strategy:
      runOnce:
        deploy:
          steps:
          # ... deployment steps ...
```

A.47: Example of a pipeline job targeting a protected environment

A.7.4 Create the Pipelines in Azure DevOps

The final step is to instruct Azure DevOps to recognize and run the YAML files from your repository. Before instructing Azure DevOps to recognize the YAML files, it is crucial to verify that all prerequisite resources have been created with the *exact names* expected by the pipelines. The YAML definitions reference Service Connections, Variable Groups, and Environments by name, and any mismatch will cause the pipeline to fail.

1. Navigate to the **Pipelines** section in your project.
2. Click the **New pipeline** button.
3. For the location of your code, select **Azure Repos Git**.
4. Select the repository you imported in earlier.
5. On the "Configure your pipeline" screen, select **Existing Azure Pipelines YAML file**.
6. Set the **Path** to one of the main pipeline files in your repository, for example, `/extractor-pipeline.yml`.
7. Click **Continue**. You will see a review of the YAML file. Click **Save** to create the pipeline.
8. **Repeat this process** for each of the main pipeline files in the repository (`static-security-analysis.yml`, `publisher-pipeline.yml`, etc.) to make them all available to run from the Azure DevOps interface.

A.7.5 Start the Self-Hosted Agent

Before any pipeline can run, you must ensure the self-hosted agent on the Linux VM is active and listening for jobs. There are two methods to run the agent: interactively for temporary testing, or as a system service for persistent operation.

Method 1: Interactive Mode (For Testing).

This method is useful for initial setup and debugging, as it runs the agent in the foreground and shows live log output. The agent will stop when you close the SSH session.

1. Connect to your Linux VM and navigate to the agent's directory.
2. Execute the `run.sh` script.

```
cd /path/to/your/agent/directory
./run.sh
```

A.48: Starting the agent in interactive mode

The terminal will show "Listening for jobs..." when the agent is ready.

Method 2: As a Service (Recommended for Normal Use).

This is the standard and most robust method. It installs the agent as a `systemd` service that runs persistently in the background and starts automatically when the VM boots.

1. Connect to your Linux VM and navigate to the agent's directory.
2. Run the following commands to install and start the service using root privileges.

```
# Install the service
sudo ./svc.sh install

# Start the service
sudo ./svc.sh start

# Check the service status to ensure it's running
sudo ./svc.sh status
```

A.49: Installing and starting the agent as a service

Once started as a service, the agent will remain online and available to run jobs without requiring an active terminal session.

Run the Pipeline from Azure DevOps

Once the agent is online, you can trigger any of the configured pipelines from the web interface.

1. In your Azure DevOps project, navigate to the **Pipelines** section.
2. Select the pipeline you wish to run from the list (e.g., `Extractor`).
3. Click the **Run pipeline** button, typically located in the top-right corner.
4. A side panel will appear, displaying all the runtime **parameters** that were defined in the YAML file. This is the main control menu for the pipeline.
5. Configure the parameters as needed for your specific run. For example, in the `Extractor` pipeline, you would select the *Target Environment* ('dev' or 'prod') to extract from.
6. Once the parameters are set, click the **Run** button at the bottom of the panel to queue the pipeline for execution.

The job will be assigned to your self-hosted agent, and you can monitor its progress in real-time from the Azure DevOps interface.

Bibliography

[1] Postman. *The State of the API Report 2024*. Feb. 2024. URL: https://www.postman.com/state-of-api/2024/api-global-growth/ (cit. on p. 1).

[2] Erik Wilde. *The 5 API styles: Understanding REST, OpenAPI, HTTP, gRPC, GraphQL, and Kafka*. Aug. 2020. URL: https://blog.axway.com/learning-center/apis/api-management/api-styles-rest-openapi-http-grpc-graphql-and-kafka (cit. on p. 4).

[3] Wikipedia. *Capability Maturity Model*. URL: https://en.wikipedia.org/wiki/Capability_Maturity_Model (cit. on p. 6).

[4] Jordi Fernandez Moledo. *API Platform Engineering Maturity Model*. Mar. 2024. URL: https://konghq.com/blog/enterprise/api-platform-engineering-maturity-model (cit. on pp. 7–9, 16).

[5] Tewelle Welemariam. *What are DevOps, GitOps, and APIOps?* Jan. 2024. URL: https://medium.com/@tewelle.welemariam/what-are-devops-gitops-and-apiops-6caa1ea98e6f (cit. on p. 10).

[6] Jenkins Project. *Jenkins User Documentation*. URL: https://www.jenkins.io/doc/ (cit. on p. 11).

[7] Luke Seelenbinder. *Stadia Maps and the API as a Product Mindset*. https://conversations.apievangelist.com/store/2024-10-24-luke-seelenbinder-stadia-maps/. Interview by Kin Lane on API Evangelist Conversations. Oct. 2024 (cit. on p. 16).

[8] Straits Research. *Open API Market Size, Share, Growth & Trends Chart by 2033*. Tech. rep. Straits Research, 2024. URL: https://straitsresearch.com/report/open-api-market (cit. on p. 18).

[9] Kin Lane. *We Need More APIOps Cycles, Arazzo, Overlays, and We Get MCP and A2A*. https://apievangelist.com/2025/04/24/we-need-more-apiops-cycles-arazzo-overlays-and-we-get-mcp-and-a2a/. Published on API Evangelist. 2025 (cit. on p. 18).

[10] Cloud Native Computing Foundation. *Cloud Native 2024: Approaching a Decade of Code, Cloud, and Change*. Tech. rep. CNCF, 2024. URL: https://www.cncf.io/reports/cncf-annual-survey-2024/ (cit. on p. 18).

[11] Microsoft. *Automate API deployments with APIOps*. URL: https://learn.microsoft.com/en-us/azure/architecture/example-scenario/devops/automated-api-deployments-apiops (cit. on p. 20).

[12] Kong Inc. *Kong Gateway Documentation*. URL: https://docs.konghq.com/gateway/latest/ (cit. on p. 21).

[13] Kong Inc. *Services and Routes - Kong Gateway Documentation*. URL: https://docs.konghq.com/gateway/3.10.x/get-started/services-and-routes/ (cit. on p. 22).

[14] Kong Inc. *Kong Konnect Documentation*. URL: https://docs.konghq.com/konnect/ (cit. on p. 23).

[15] Luís Prates and Rúben Pereira. «DevSecOps practices and tools». In: *International Journal of Information Security* 24 (2025). Springer Nature. URL: https://doi.org/10.1007/s10207-024-00914-z (cit. on pp. 27, 28).

[16] Department of Defense. *DoD Enterprise DevSecOps Fundamentals v2.5*. Oct. 2024. URL: https://dodcio.defense.gov/Portals/0/Documents/Library/DoD%20Enterprise%20DevSecOps%20Fundamentals%20v2.5.pdf (cit. on pp. 29–31).

[17] Walker James. *21 Best DevSecOps Tools and Platforms for 2025*. Spacelift. URL: https://spacelift.io/blog/devsecops-tools (cit. on p. 33).

[18] Stoplight. *Spectral – Flexible JSON/YAML Linter*. URL: https://stoplight.io/open-source/spectral (cit. on p. 33).

[19] SonarSource. *Security Hotspots*. URL: https://docs.sonarsource.com/sonarqube-server/latest/ (cit. on pp. 34, 53).

[20] Semgrep Inc. *Rule Syntax*. URL: https://semgrep.dev/docs (cit. on p. 34).

[21] Truffle Security. *TruffleHog – Find, Verify, and Analyse Leaked Credentials*. URL: https://github.com/trufflesecurity/trufflehog (cit. on p. 34).

[22] Chainguard Academy. *Keyless Sign and Verify Images With Cosign*. URL: https://edu.chainguard.dev/open-source/sigstore/cosign/an-introduction-to-cosign/ (cit. on p. 34).

[23] OWASP Foundation. *Zed Attack Proxy (ZAP) Documentation*. https://www.zaproxy.org/docs/ (cit. on p. 35).

[24] APIsec docs. *Welcome to the APIsec*. https://docs.apisecapps.com/docs/intro (cit. on p. 35).

[25] Microsoft. *APIops github*. https://github.com/Azure/apiops (cit. on p. 39).

[26] OWASP Foundation. *OWASP crAPI: Completely Ridiculous API*. URL: https://github.com/OWASP/crAPI (cit. on p. 49).

[27] OWASP Foundation. *OWASP Top 10*. Web Page. 2021. URL: https://owasp.org/www-project-top-ten/ (cit. on p. 50).

[28] OWASP Foundation. *OWASP: Open Worldwide Application Security Project*. https://owasp.org (cit. on p. 50).

[29] SonarSource. *Quality Gates*. URL: https://docs.sonarsource.com/sonarqube/latest/user-guide/quality-gates/ (cit. on p. 52).

[30] SonarSource. *Clean Code definition*. URL: https://docs.sonarsource.com/sonarqube-server/10.4/user-guide/clean-as-you-code/ (cit. on p. 54).

[31] Ariadi Nugroho, Joost Visser, and Tobias Kuipers. *An Empirical Model of Technical Debt and Interest*. URL: https://www.researchgate.net/publication/228684782_An_Empirical_Model_of_Technical_Debt_and_Interest (cit. on p. 54).

[32] Microsoft. *Store business-critical blob data with immutable storage in a write once, read many (WORM) state*. https://learn.microsoft.com/en-us/azure/storage/blobs/immutable-storage-overview (cit. on p. 67).

[33] PCI Security Standards Council. *Payment Card Industry Data Security Standard Requirements and Testing Procedures*. URL: https://docs-prv.pcisecuritystandards.org/PCI%20DSS/Standard/PCI-DSS-v4_0_1.pdf (cit. on p. 70).

[34] Karen Kent and Murugiah Souppaya. *Guide to Computer Security Log Management*. DOI: 10.6028/NIST.SP.800-92. URL: https://csrc.nist.gov/pubs/sp/800/92/final (cit. on p. 70).

[35] U.S. Department of Health and Human Services. *Health Insurance Portability and Accountability Act of 1996 (HIPAA): Administrative Requirements*. 45 CFR 164.530(j) - Documentation and retention requirements. 1996. URL: https://www.hhs.gov/hipaa/for-professionals/privacy/laws-regulations/index.html (cit. on p. 71).

[36] European Parliament and Council of the European Union. *Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation)*. Article 4(1) - Definitions. URL: https://eur-lex.europa.eu/eli/reg/2016/679/oj (cit. on p. 71).

[37] European Parliament and Council of the European Union. *Directive (EU) 2022/2555 of the European Parliament and of the Council of 14 December 2022 on measures for a high common level of cybersecurity across the Union, amending Regulation (EU) No 910/2014 and Directive (EU) 2018/1972, and repealing Directive (EU) 2016/1148 (NIS 2 Directive)*. URL: https://eur-lex.europa.eu/eli/dir/2022/2555/oj (cit. on p. 72).

[38] Microsoft. *Environment considerations in the Cloud Adoption Framework*. 2024. URL: https://learn.microsoft.com/en-us/azure/cloud-adoption-framework/ready/considerations/environments (cit. on p. 75).

Acknowledgements

Desidero ringraziare il Prof. Atzeni per la preziosa guida e il supporto durante lo sviluppo di questa tesi.

Un sentito ringraziamento va all'azienda Cloud9 Reply, al mio tutor aziendale Leonardo Frioli e a Simone Mauri per il supporto, i suggerimenti e gli spunti che hanno arricchito il lavoro, rendendolo più completo e significativo.

Grazie a tutti per aver reso possibile questo percorso e per il contributo alla riuscita di questo progetto.