ESPRIT SCHOOL OF ENGINEERING

www.esprit.tn - E-mail : contact@esprit.tn

Siége Social : 18 rue de l'Usine - Charguia II - 2035 - Tél. : +216 71 941 541 - Fax. : +216 71 941 889

Annexe : Z.I. Chotrana II - B.P. 160 - 2083 - Pôle Technologique - El Ghazala - Tél. : +216 70 685 685 - Fax. : +216 70 685 454

esprit

Se former autrement

2022 - 2023

GRADUATION PROJECT

NATIONAL ENGINEERING DEGREE

SPECIALTY : ARCTIC

Intelligent Cloud-Native DevSecOps Platform with AI Integration

By: *Hamza Kalech*

Academic supervisor: Mr. Mohamed Ridha Boulaâres

Corporate Internship Supervisor: Mr. Mhamed Laabidi,

Capgemini

uni Tuin

cdio

Délivré par: Cti

EUR-ACE

CONFÉRENCE DES GRANDES ÉCOLES

EXCELLENCE FOR A COMPLEX WORLD

Acknowledgements

I would like to begin by expressing my deepest gratitude to **Mr. Fathi Mnaja**, manager of the hosting company, for granting me this opportunity and for his continuous support and guidance throughout the duration of this project. His trust and involvement played a major role in the success of this internship.

I also extend my sincere thanks to **Mr. Mhamed Laabidi**, my company supervisor, for his technical assistance, valuable feedback, and day-to-day support during the implementation of the project.

My heartfelt appreciation goes to **Mr. Mohamed Ridha Boulaâres**, my academic supervisor at ESPRIT, for his valuable guidance, constructive remarks, and continuous encouragement throughout this internship.

I am also grateful to the entire **academic team at ESPRIT** for providing a solid foundation and an environment that fosters both learning and personal growth.

Lastly, I wish to thank my **family and friends** for their unwavering support, patience, and encouragement throughout this journey.

To all those who contributed, directly or indirectly, to the success of this project thank you.

Table of Contents

List of figures

1. GENERAL INTRODUCTION

In the context of my final engineering internship at ESPRIT, I undertook a six-month project focused on the implementation of an integrated **DevOps platform enhanced with AI capabilities**. The objective was to design and deploy a cloud-native solution that automates the software development lifecycle, enhances observability, and integrates intelligent analysis tools to support real time decision making.

With the increasing complexity of modern applications, companies are adopting DevOps practices to achieve agility, reliability, and faster time-to-market. However, integrating observability, security, and automation into a unified workflow remains a significant challenge. To address this, my project aimed to build an open-source DevOps platform capable of automating code integration, testing, deployment, and monitoring with AI-based insights added to further enhance performance and reliability.

The first phase of the project focused on setting up the virtualized environment, building the application to deploy, and developing the DevOps pipeline. I configured multiple Ubuntu virtual machines under a Windows 11 host using VMware and deployed tools such as **Jenkins**, **SonarQube**, **Nexus**, **Docker**, **Prometheus**, **Grafana**, and **Kubernetes on Azure using Terraform**. Security aspects such as **RBAC** and **SSL certificate management** were also addressed.

The second phase focused on integrating **artificial intelligence** into the platform. The second phase focused on integrating artificial intelligence into the platform. Two AI modules were developed:

- **An anomaly detection system**, trained on CI/CD pipeline data and implemented using the Isolation Forest algorithm, capable of detecting unusual build durations, test failures, or resource usage spikes.
- **A predictive autoscaling model**, built using XGBoost and trained on real workload metrics collected from Prometheus, which forecasts future resource consumption to dynamically scale the Kubernetes application with KEDA.

These components enhance the platform's intelligence by enabling proactive decision-making, improved reliability, and resource efficiency.

This report presents the entire project lifecycle from setup and implementation to testing and evaluation and highlights the technical and personal skills I developed throughout the experience. It also outlines the challenges encountered and the innovative solutions adopted to complete the project successfully.

2. GENERAL CONTEXT

2.1 Hosting Organization

2.1.1 Overview of Capgemini

Capgemini is a globally recognized leader in **digital transformation and technology consulting**. With a presence in over 50 countries and a workforce exceeding 340,000 professionals, the company supports organizations in achieving innovation-driven growth and operational efficiency. Its mission is to enable business and societal progress through **intelligent use of technology**, while maintaining a strong commitment to **sustainability and diversity**.

Founded in 1967 and headquartered in Paris, Capgemini collaborates with clients across industries to design and implement forward-looking IT strategies, cloud-native solutions, and smart enterprise systems.

2.1.2 Core Areas of Expertise

Capgemini's operations span a wide range of services, including:

- **Cloud Infrastructure & DevOps**: Modernizing infrastructure, automating deployments, and enabling scalable operations through Kubernetes and CI/CD.
- **Software Engineering**: Custom development of enterprise applications using cutting-edge technologies and agile methodologies.
- **Artificial Intelligence & Data**: Delivering AI-powered solutions for predictive analytics, process automation, and decision support.
- **Consulting & Digital Strategy**: Helping clients navigate digital transformation through innovation and business process reengineering.

2.1.3 Internship Environment

During my internship, I was integrated into a project team working on a cloud-native DevOps platform, contributing to the development and automation of a full CI/CD pipeline and AI-driven infrastructure. The project was supervised by **Mr. Mhamed Laabidi**, who provided both technical guidance and strategic oversight.

The collaborative and innovation-driven environment of Capgemini offered an excellent context to apply DevOps principles in a real-world setting, while aligning with the company's broader vision of intelligent and sustainable technology.

2.2 Problem Statement

In modern software development environments, the need for rapid, reliable, and secure deployment pipelines has become increasingly critical. Traditional methods of software

delivery often involve manual interventions, fragmented toolchains, limited visibility into system health, and poor scalability all of which can lead to delays, errors, and reduced productivity.

Furthermore, while DevOps practices have become more widely adopted, many organizations still struggle to integrate their tools into a unified, automated, and intelligent platform. In particular, there is a lack of solutions that combine **continuous integration/continuous deployment (CI/CD)** with **advanced observability**, **security features**, and **AI-powered insights** to support proactive decision-making.

The core challenge of this project is to **design and implement a DevOps platform** that:

- Automates the complete software lifecycle (from code integration to deployment)
- Ensures observability and monitoring of deployed services
- Supports secure and scalable deployments on a cloud infrastructure
- Integrates artificial intelligence to analyze logs and metrics for anomaly detection and reporting

The solution must rely exclusively on **open-source tools** and be deployed in a **cloud-native environment**, leveraging technologies such as Docker, Kubernetes, Jenkins, Prometheus, Grafana, SonarQube, Nexus, and Terraform.

The AI component must go beyond simple data visualization it should provide added value by enabling intelligent analysis of operational data, helping teams detect issues early, optimize performance, and reduce downtime.

This project therefore aims to address the following core questions:

- How can we build a secure, scalable, and automated DevOps pipeline using only open-source tools?
- How can observability be integrated into the DevOps lifecycle in a meaningful and actionable way?
- How can artificial intelligence be leveraged to improve operational monitoring and incident response?

2.3 State of the Art: Existing DevOps Platforms

Before initiating the design of this platform, a comprehensive analysis of existing DevOps solutions was conducted to evaluate their capabilities, limitations, and relevance to modern delivery needs particularly in the context of automation, scalability, and AI integration.

Azure DevOps

Azure DevOps provides a robust and integrated toolchain, combining version control, CI/CD, and project management in a single environment. However, it comes with several limitations:

- Licensing costs increase with the number of users and build agents.
- The platform is tightly coupled to the Azure ecosystem, reducing portability.
- It lacks built-in AI capabilities or advanced infrastructure observability.

GitHub Actions

GitHub Actions offers a seamless experience for CI workflows, especially for developers already using GitHub. Its advantages include easy syntax, reusable workflows, and integration with GitHub Copilot. However:

- It primarily focuses on CI, with no support for infrastructure provisioning.
- Runners are vendor-managed, limiting customization and scalability.
- Native integration with monitoring or AI-based analysis is minimal.

GitLab (Self-Hosted or Cloud)

GitLab stands out for its end-to-end DevOps lifecycle coverage and DevSecOps features. Nevertheless:

- Premium features are locked behind paid tiers.
- Self-hosted instances demand high system resources and operational overhead.
- Monitoring capabilities are basic, and predictive autoscaling support is absent.

Bitbucket Pipelines

Bitbucket Pipelines offers tight integration with Jira and a streamlined CI/CD interface. Despite this, it suffers from:

- A smaller plugin ecosystem compared to competitors.
- Limited support for custom DevOps workflows.
- No built-in infrastructure as code (IaC), monitoring, or security modules.

2.4 Identified Gap and Platform Positioning

While all of the above platforms bring unique strengths, none offer a fully open-source, extensible, and intelligent solution with:

- Full CI/CD and infrastructure automation,
- Built-in security and observability tooling,
- AI-based predictive scaling or anomaly detection,
- Cost efficiency without license or agent restrictions.

This gap highlights the need for a customizable, scalable, and AI-enhanced DevOps platform the very challenge addressed by the solution developed during this internship.

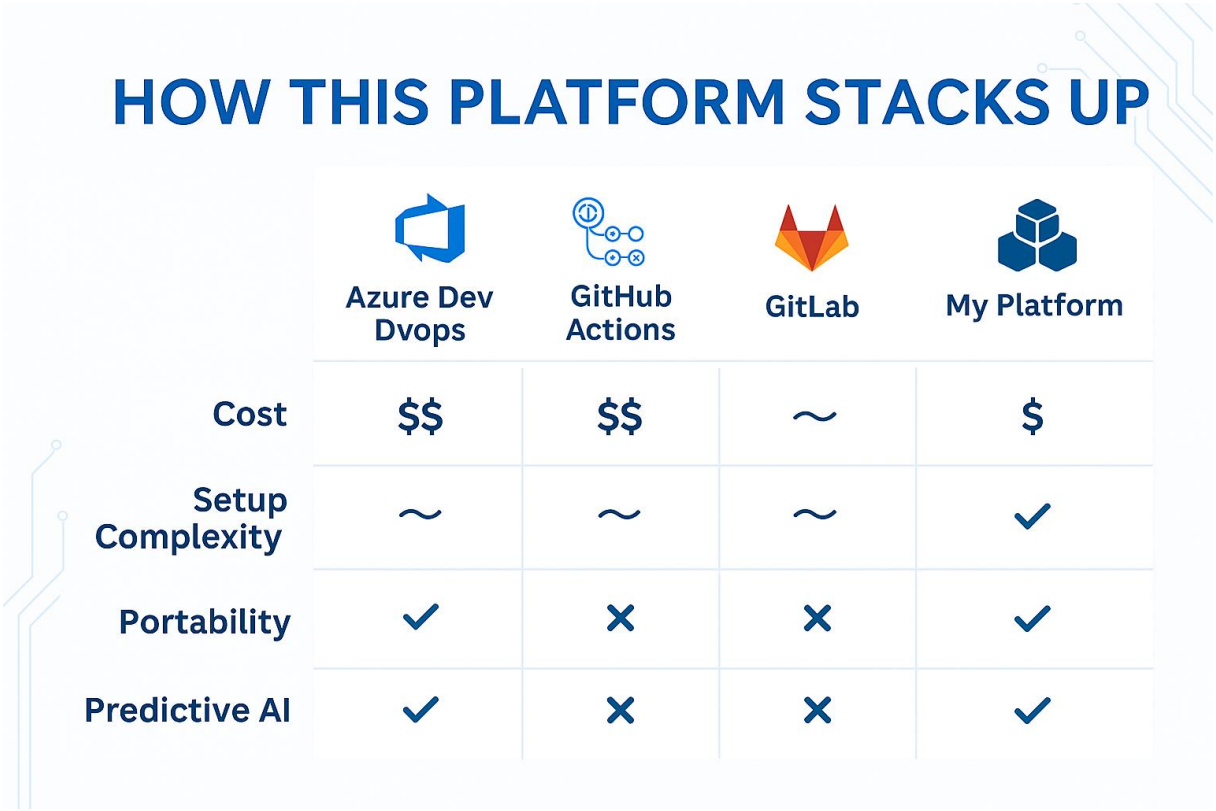


Figure 1:How this platform stacks up

2.5 Planning Strategy

To deliver a secure, intelligent, and fully automated DevOps platform, the project was structured into seven progressive phases, each addressing a critical aspect of the system's lifecycle:

- **Phase 1: Infrastructure Setup & Tool Configuration**
  Deployment of local virtual machines and initial tool installations (Jenkins, SonarQube, Nexus). Setup of Terraform, Helm, and Azure CLI on a dedicated INFRA VM to enable cloud provisioning.
- **Phase 2: CI/CD Pipelines & Security Configuration**
  Development of robust Jenkins pipelines handling build, test, analysis, artifact storage, Docker image management, and Kubernetes deployment. Integration of Trivy and SonarQube for vulnerability and code quality checks. SSL and RBAC were configured to enforce security.
- **Phase 3: Kubernetes Cluster Deployment & Access Control**
  Creation of a Kubernetes cluster on Azure using Terraform modules. Implementation of network security policies via Calico, role-based access control (RBAC), and secure ingress with cert-manager and Let's Encrypt.
- **Phase 4: AI Research & Proof of Concept**
  Investigation of AI use cases in DevOps. Two key directions were chosen: anomaly detection in pipeline metrics and predictive auto-scaling for Kubernetes workloads.
- **Phase 5: Full AI Integration & Enhancement**
  Development and deployment of both AI models:
  - Anomaly detection using Isolation Forest integrated with Jenkins logs.

    - Predictive autoscaling using XGBoost and KEDA with real-time Prometheus metrics.
- **Phase 6: Final Technical Testing**
  Comprehensive testing of the platform, including CI/CD flows, HPA behavior, AI model responses, security validation (Calico, SSL), and backup procedures with Velero.
- **Phase 7: Documentation & Reporting**
  Finalization of the technical report, architecture diagrams, and preparation for project defense.

This phased approach allowed iterative validation at each stage and ensured a production-grade DevSecOps platform backed by AI.

2.6 Project Planning

To effectively structure and execute the development of the DevSecOps platform, the work was divided into seven well-defined phases. Each phase represents a major milestone in the platform's lifecycle, from infrastructure setup to full AI integration and final documentation.

Phase	Main Tasks	Duration
Phase 1: Infrastructure Setup & Tool Configuration	- Install and configure Jenkins, SonarQube, Nexus on local machines/VMs.- Ensure basic connectivity and minimum security (admin accounts, open ports, etc.).	**2 weeks**
Phase 2: CI/CD Pipelines & Security Configuration	- Create CI/CD pipelines (build, test, Sonar analysis, artifact push to Nexus, etc.).- Install/activate necessary plugins (e.g., Credentials Binding, Docker, Trivy).- Implement security mechanisms (Jenkins RBAC, vulnerability scans, etc.).	**6 weeks**
Phase 3: Kubernetes Cluster Deployment + RBAC	- Create Kubernetes cluster using Terraform.- Configure Jenkins as a service inside the cluster.- Write and apply 4 RBAC YAML files (ServiceAccount, Role, RoleBinding, etc.).- Perform a test deployment and troubleshoot.	**4 weeks**
Phase 4: AI Research & Proof of Concept (PoC)	- Explore AI solutions for DevOps (e.g., anomaly detection, vulnerability classification, improvement suggestions).- Implement a basic PoC AI integration into the CI/CD flow (e.g., after SonarQube analysis or Trivy scan).- Use a lightweight model or existing AI API initially.	**2 weeks**

Phase	Main Tasks	Duration
Phase 5: Full AI Integration & Enhancement	- Finalize integration of AI module (custom or external).- Fine-tune AI behavior (e.g., severity classification, smart remediation tips).- Ensure compatibility with Jenkins pipelines and security constraints.	**7 weeks**
Phase 6: Final Technical Testing	- Run full integration and performance/stress tests for the CI/CD + AI system.- Troubleshoot and polish all configurations.- Prepare for production or live demo.	**2 weeks**
Phase 7: Documentation & Reporting	- Write detailed technical documentation (architecture, setup, best practices).- Prepare the final project report.- Provide operational documentation for future maintenance and upgrades.	**4 weeks**

Note: Durations are estimates and may vary depending on technical challenges or workload adjustments.

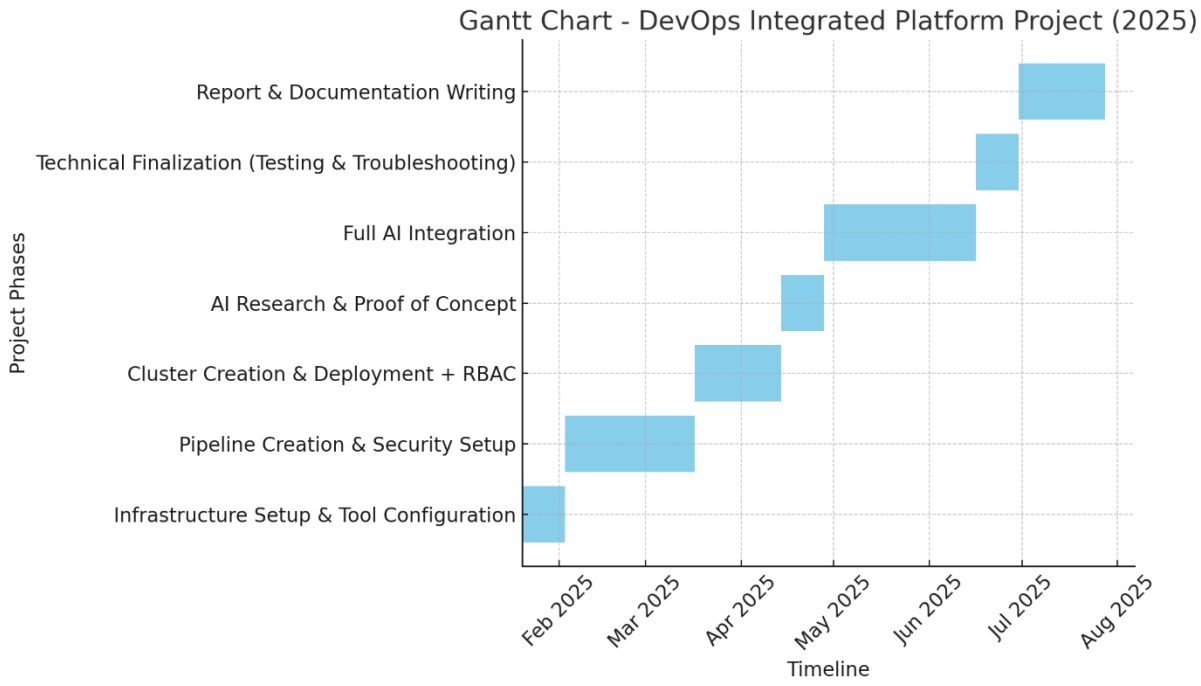


Figure 2: Gant Chart

2.7 Methodology

To ensure efficient project management and fast, incremental delivery, an **Agile methodology** was adopted drawing inspiration from the **SCRUM framework**. This approach proved highly effective for a DevOps-focused project, where adaptability, feedback-driven development, and continuous improvement were essential.

2.7.1 SCRUM-Inspired Workflow

While no formal SCRUM rituals (such as daily stand-ups or sprint reviews) were conducted, the project was structured to mirror core SCRUM principles:

- Each critical module CI/CD pipeline, infrastructure automation, and AI integration was approached as a focused iteration or "mini-sprint".
- Tasks were initially planned but regularly re-evaluated and adjusted based on technical progress, unforeseen challenges, and evolving priorities.
- Continuous integration, testing, and deployment were emphasized throughout to maintain stability and enable rapid validation of changes.

This iterative planning structure allowed clear tracking of progress and quick reaction to blockers, especially during complex phases like Kubernetes configuration and AI model deployment.

2.7.2 Benefits and Limitations

Advantages:

- **Modular Delivery:** Each project milestone (e.g., pipeline automation, infrastructure setup) was self-contained and independently testable.
- **Responsiveness to Change:** Agile planning made it easy to pivot quickly in response to technical issues, especially when troubleshooting cloud provisioning or refining the AI model.
- **Ongoing Refinement:** Regular testing and monitoring enabled continuous optimization of deployment scripts, model accuracy, and platform resilience.

Limitations:

- **Solo Execution:** Operating without a full development team limited collaboration opportunities and placed the entire planning, execution, and decision-making load on one person.
- **Compressed Timeline:** Balancing hands-on development, research, testing, and documentation within tight deadlines demanded high levels of focus and self-organization.

In conclusion, this **Agile-inspired, SCRUM-aligned methodology** enabled a modular, flexible, and production-grade implementation of the platform. It supported early validation of each component, smooth integration of AI functionality, and ultimately led to a robust final result tailored for real-world deployment.

3. DESCRIPTION OF THE PROPOSED WORK

The internship project proposes the design and implementation of an integrated **DevOps platform** using **only open-source solutions**. The objective is to automate and optimize the entire software development lifecycle including development, integration, testing, deployment, monitoring, and infrastructure management while also incorporating artificial intelligence to enhance operational decision-making and platform intelligence.

The project is inspired by the growing needs of organizations to achieve agility, reliability, and speed in software delivery, while also ensuring high availability, observability, and predictive insights. This platform should meet these needs by combining various technologies and services into a unified, scalable, and secure environment.

The main components and goals of the proposed platform include:

- **CI/CD Pipeline**: Automate the process of building, testing, analyzing, and deploying applications using tools such as **Jenkins**, **SonarQube**, **Nexus**, and **Docker**.
- **Container Orchestration**: Use **Kubernetes** to manage application deployments and ensure scalability and reliability across environments.
- **Infrastructure as Code**: Implement infrastructure provisioning and management using **Terraform**, following modern DevOps practices.
- **Monitoring & Observability**: Set up performance monitoring and alerting using **Prometheus** and **Grafana**, with dashboards providing visibility into application health and infrastructure metrics.
- **Security & Configuration Management**: Integrate security best practices including **SSL certificate management**, **role-based access control (RBAC)**.
- **AI Integration in DevOps**:
  One of the key goals of this project is to explore how **artificial intelligence can support DevOps practices**. The integration of AI is not limited to a specific use case but is instead focused on enhancing the platform's capabilities through automation, prediction, and intelligent insights.

  No specific solution chosen yet. Ideas are listed as inspiration only.
  Inspiration for this component includes:

    - Predictive analysis of performance and usage patterns

- AI-assisted troubleshooting using historical logs and metrics
- Automated test generation or prioritization
  These ideas serve as **guidelines**, but the main objective is to **introduce AI into the DevOps workflow** and demonstrate its potential for improving software quality, reducing downtime, and accelerating delivery.
  As AI continues to play a growing role in software engineering, this project aims to prepare the foundation for AI-augmented DevOps processes.

This work is intended to be developed and completed over a six-month period, by engineering student specializing in **DevOps** and **Web Development**. The expected outcome is a functional, modular, and intelligent platform that can be used by development teams to streamline and secure their software delivery pipelines.

4. ANALYSIS AND DESIGN

4.1 Functional Requirements

The functional requirements define what the DevOps platform must accomplish to meet the objectives of automation, reliability, and scalability. These requirements were derived from the project's goals and user needs.

The core functional requirements are as follows:

1. **Automate the Software Delivery Lifecycle**
   - Automate the build, test, and deployment phases using a CI/CD pipeline
   - Ensure every code push triggers a validation and deployment cycle.
2. **Secure Code and Artifact Flow**
   - Guarantee that all code, packages, and container images move securely between environments using tools like Nexus and signed images.
3. **Monitor System Health and Detect Anomalies**
   - Integrate monitoring tools such as Prometheus and Grafana to track performance metrics and alert on abnormal behavior.
4. **Integrate AI for Resource and Pipeline Anomaly Detection**
   - Deploy machine learning models that can forecast or detect failures or scaling issues across CI/CD workflows or Kubernetes pods.
5. **Enable Backup and Restore Mechanisms**

- Implement infrastructure and volume-level backup solutions (e.g., Velero) to ensure data protection and recovery in case of failure.

6. **Support Continuous Deployment with Rollback Capability**
    - Ensure zero-downtime deployments and provide the ability to roll back to a previous stable state in case of failures.
7. **Centralize Image and Package Management**
    - Use a centralized artifact repository (Nexus) to store and manage Docker images, Maven packages, and other build artifacts.

4.2 Non-Functional Requirements Analysis

Non-functional requirements define the qualities and constraints the system must meet to ensure reliability, security, performance, and future scalability. For this DevOps platform, the following non-functional requirements were essential:

- **Scalability:**

The platform must dynamically adjust to fluctuating workloads using **horizontal pod autoscaling**. This ensures optimal resource utilization during traffic peaks and cost efficiency during low activity.

- **Security**

Robust security is enforced through:

- **Role-Based Access Control (RBAC)** for Kubernetes and Jenkins
- **SSL/TLS encryption** for all exposed services
- **Container image scanning** using tools like Trivy to detect vulnerabilities

- **Availability**

The system must guarantee **high uptime**, ensured through resilient Kubernetes deployments, regular health probes, and **centralized monitoring** with Prometheus and Grafana.

- **Observability**

Key metrics (CPU, memory, replica count) must be observable in real-time through dashboards, alerts, and logs. This enables quick detection of bottlenecks or failures.

- **Maintainability**

The platform is designed to be **modular, version-controlled, and well-documented**, making it easier to update components and integrate changes through CI/CD workflows.

- **Automation**

Infrastructure provisioning and application deployment are fully automated using **Infrastructure as Code (IaC)** via **Terraform** and **Helm**, ensuring reproducibility and minimizing human error.

- **Extensibility**

The architecture is built to support future extensions, such as integrating **additional AI models**, **new monitoring tools**, or **alternative CI/CD engines**.

4.3 Project Architecture

The overall architecture of the DevOps platform was divided into three logical layers, each corresponding to a distinct phase of the software delivery process: **infrastructure setup**, **CI/CD pipelines**, and **application runtime deployment**. This separation facilitates modularity, scalability, and clear maintenance responsibilities.

4.3.1 Infrastructure Setup Architecture

This phase focuses on the provisioning and configuration of all supporting tools and services necessary to enable DevOps workflows.

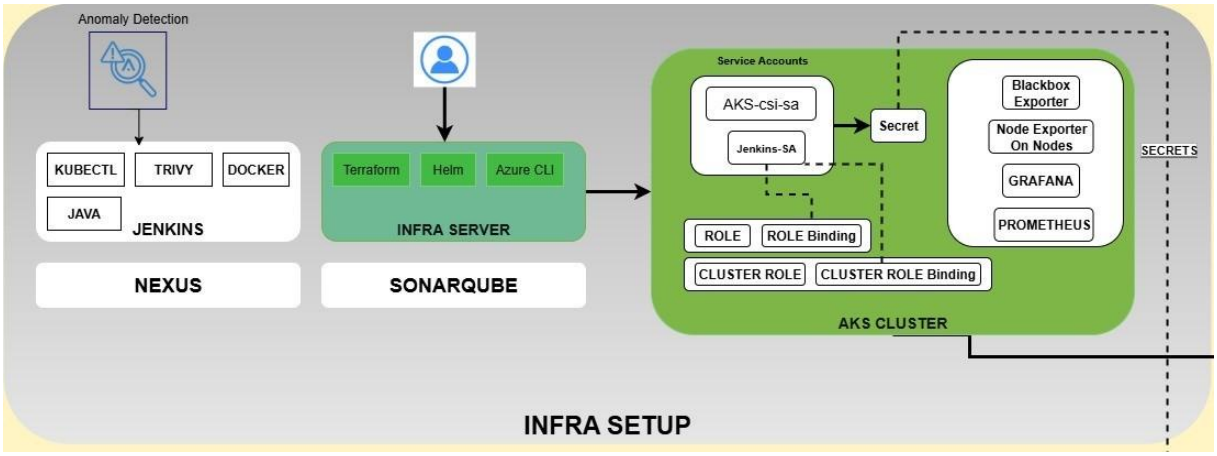


Figure 3:Infra Setup

Key components:

- **Jenkins VM**: Installed with required tools (kubectl, Docker, Trivy, Java) for CI/CD execution and anomaly detection.
- **Nexus**: Hosts Maven artifacts and Docker images.
- **SonarQube**: Performs static code analysis and enforces quality gates.
- **Infra Server**: Acts as a control plane for infrastructure automation, with **Terraform**, **Helm**, and **Azure CLI** installed.

- **AKS Cluster (Azure Kubernetes Service)**: Provisioned via Terraform and managed using Helm.
    - Contains pre-defined **RBAC roles** and **service accounts** (e.g., Jenkins-SA, AKS-csi-sa).
    - Integrates with **Prometheus**, **Grafana**, and **Node Exporters** for system monitoring.

4.3.2 CI/CD Pipeline Architecture

This diagram illustrates the automated continuous integration and delivery pipeline, triggered via GitHub pushes.

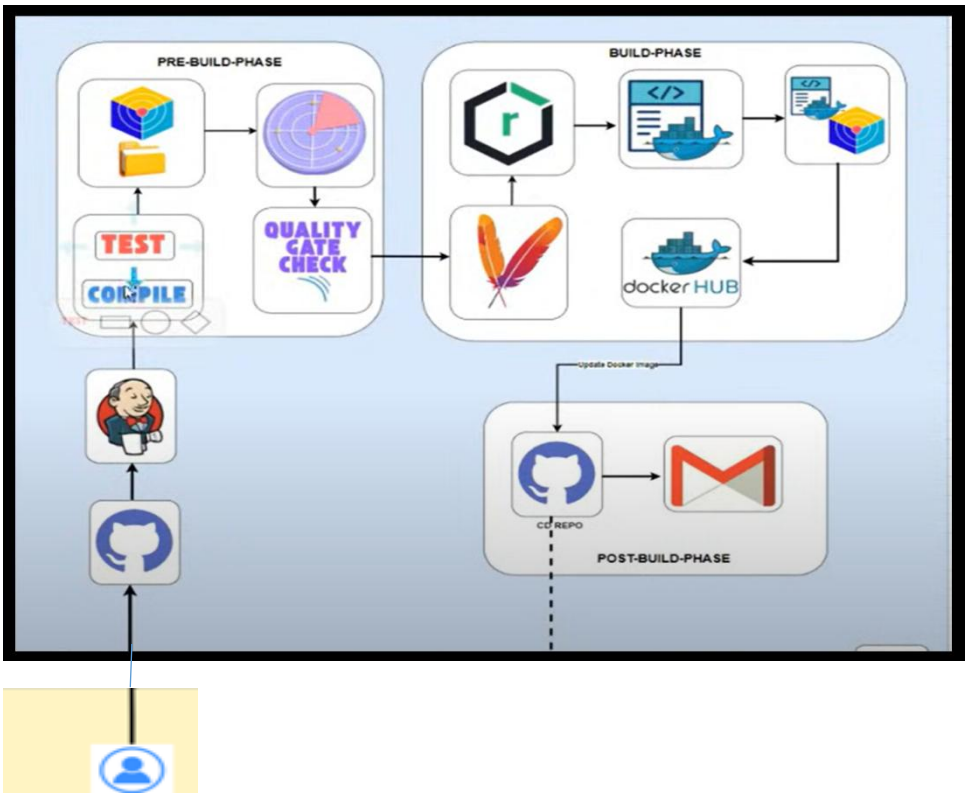


Figure 4:Pipeline Architechture

Pipeline breakdown:

- **Pre-Build Phase**:
    - Jenkins pulls code from GitHub.
    - Code is compiled, tested, and analyzed by SonarQube.
    - The **Quality Gate** decision determines pipeline continuation.
- **Build Phase**:
    - Artifacts are uploaded to **Nexus**.
    - Docker images are built and pushed to **Docker Hub**.

- Kubernetes manifests are updated with new image tags.

- **Post-Build Phase**:
  - Docker image tag is pushed back to the Git repository.
  - A webhook triggers a deployment or sends an email notification.

This pipeline ensures automation of the entire software lifecycle with traceability and rollback capability.

4.3.3 Application Runtime & AI Integration Architecture

This layer represents how the deployed application runs in Kubernetes and integrates with the AI module for predictive autoscaling.

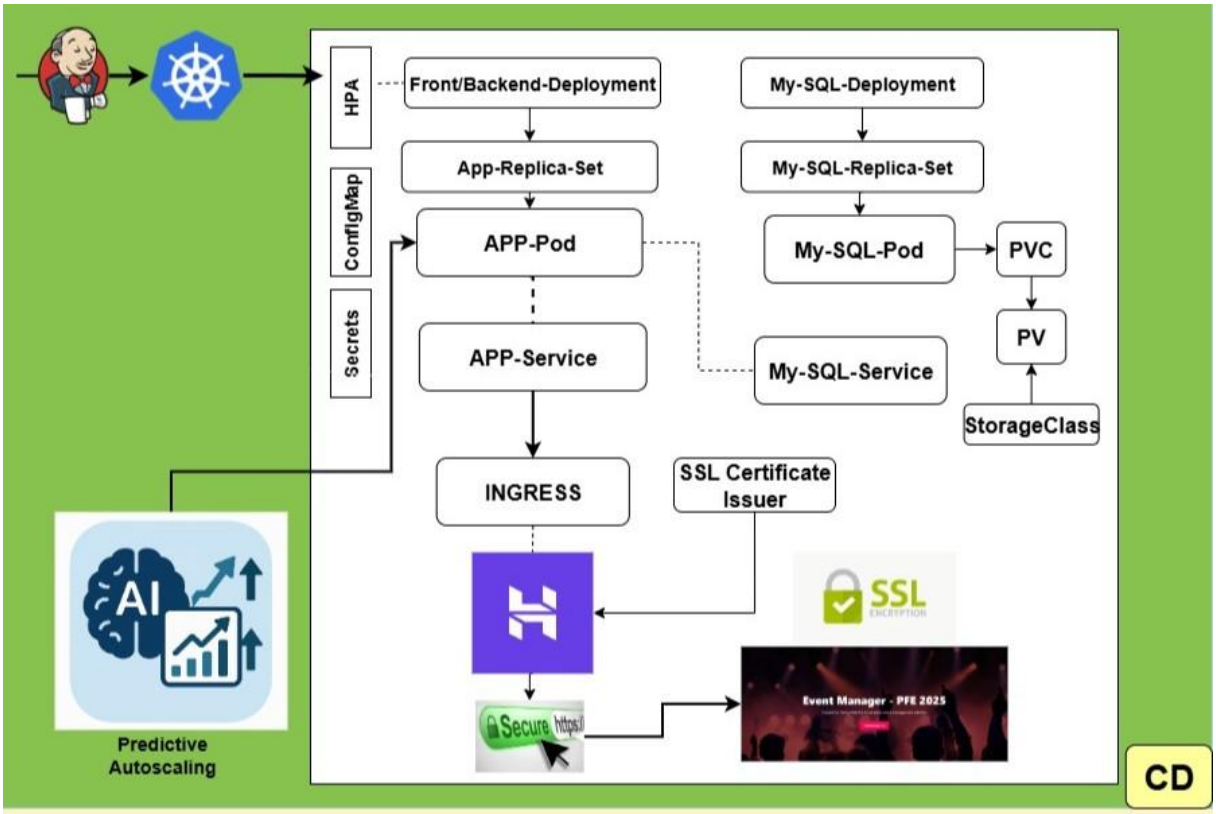


Figure 5:Deployment Architecture

Key components:

- **Frontend and Backend Deployments**: Managed via Kubernetes Deployments and exposed through a unified **Ingress Controller**.
- **MySQL Deployment**: Uses Persistent Volume Claims (PVCs) for data durability via Azure Disks.
- **ConfigMaps & Secrets**: Injected into pods securely for dynamic configuration.
- **SSL Certificate Issuer**: Enables HTTPS using **cert-manager** and Let's Encrypt.

- **Predictive Autoscaling**:
    - A custom AI model monitors **Prometheus metrics**.
    - Based on predictions, it adjusts Horizontal Pod Autoscalers (HPA) dynamically.

This architecture ensures scalability, security, and intelligent resource allocation in real-time.

5. DESCRIPTION OF THE WORK DONE

5.1 Environment Setup

The initial phase of the project involved setting up the development and integration environment using local virtualization. The goal was to create a modular, cost-effective, and fully isolated environment that mirrors professional DevOps architectures without relying on external infrastructure during the early stages.

5.1.1 Virtual Machine Configuration

Three separate **Ubuntu 22.04 Server** virtual machines were created using **VMware Workstation** on a host running **Windows 11**. Each VM was dedicated to a core DevOps component:

VM	Role	Services	Specs
VM1	CI/CD	Jenkins, Trivy, Maven	4 vCPUs, 10 GB RAM
VM2	Code Quality	SonarQube (Docker)	2 vCPUs, 4 GB RAM
VM3	Artifact Storage	Nexus (Docker)	2 vCPUs, 4 GB RAM
VM4	Infrastructure Management	Terraform, Helm, Azure CLI	2 vCPUs, 4 GB RAM

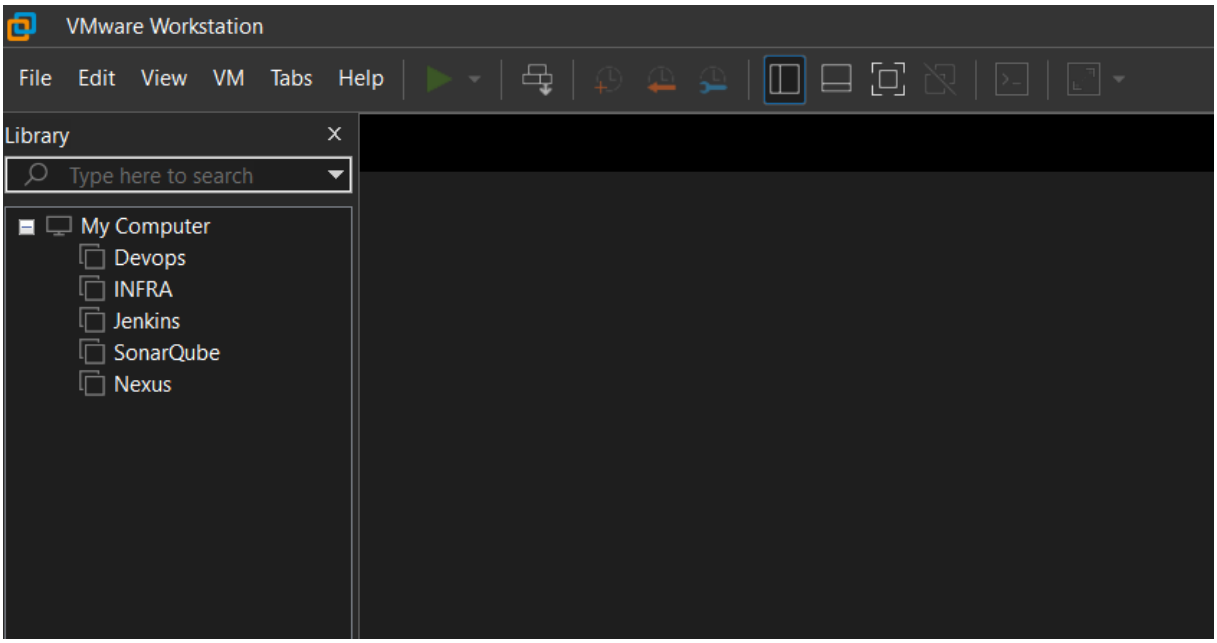


Figure 6: Infrastructure

All VMs were configured with:

- **60 GB disk space**
- **NAT networking** for inter-VM communication
- **OpenSSH server** for remote access
- Accessed using **PuTTY** from the Windows host

In addition, the following tools were installed:

- **Docker** on all three VMs
- **Maven** and **Trivy (security scanner)** on the Jenkins VM

5.1.2 Tool Installation

- **Jenkins** was installed directly on VM1 as a system service, ensuring it launches automatically at boot.
- **SonarQube** was deployed on VM2 using the official Docker image. It is started manually when needed.
- **Nexus** was deployed on VM3 as a container using Docker. Like SonarQube, it is started manually.
- **Terraform**, **Helm**, and the **Azure CLI** were installed on the dedicated **INFRA VM (VM4)**, which was used to manage and provision cloud infrastructure including the Azure Kubernetes Service (AKS) cluster.

Although SonarQube and Nexus were not configured to launch on system boot, Docker containers can be set to restart automatically in future iterations using Docker's --restart policy if needed.

5.1.3 Testing and Validation

After installation, basic validation steps were performed to ensure proper functionality:

- **Jenkins**: Web dashboard accessed, plugins installed, pipeline tested.
- **SonarQube**: UI loaded successfully after container startup, initial configuration applied.
- **Nexus**: Repository manager tested for Docker and Maven support.
- The INFRA VM successfully connected to Azure and was used to create and manage cloud resources using Terraform and Helm.

This virtualized environment provided a solid foundation for development, automation, and later deployment to the cloud.

5.2 Application Development

The application developed as part of this project is called **EventManagement**. It is a full-stack event management platform that allows users to create, update, view, filter, and delete events. The goal of the application is to serve as a testbed for building and validating the entire DevOps pipeline, including CI/CD, monitoring, deployment, and AI integration.

5.2.1 Backend – Spring Boot Microservice

The backend is developed using **Spring Boot (version 3.2.4)** and follows a RESTful architecture.

Main features:

- CRUD operations for event entities (create, read, update, delete)
- Event filtering by date
- REST endpoints exposed under /api
- Integration with a **MySQL** database using JPA and Hibernate
- Prometheus metrics exposed via Actuator

- Unit testing with JUnit and Mockito

Spring Configuration:

```
spring.datasource.url=jdbc:mysql://mysql-service:3306/eventdb?useSSL=false&allowPublicKeyRetrieval=true
spring.datasource.username=root
spring.datasource.password=P@$sW0rd123
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQLDialect
spring.jpa.hibernate.ddl-auto=update
server.port=8083
management.endpoint.health.probes.enabled=true
management.endpoints.web.exposure.include=*
management.endpoint.health.show-details=always
management.endpoint.prometheus.enabled=true
server.servlet.context-path=/api
```

Figure 7: Spring Configuration

Key dependencies:

Dependency	**Purpose**
spring-boot-starter-data-jpa	ORM and DB operations via Hibernate
spring-boot-starter-web	RESTful API development
mysql-connector-j	MySQL database connectivity
lombok	Reduce boilerplate (getters/setters)
spring-boot-starter-test	Testing with JUnit and Mockito
spring-boot-starter-actuator	Monitoring endpoints
micrometer-registry-prometheus	Prometheus metrics export

Testing:
Unit tests were written for all service methods using **Mockito**, ensuring that:

- Event creation, retrieval, update, and deletion behave correctly
- Filtering works based on dates
- The service layer is correctly isolated from the database

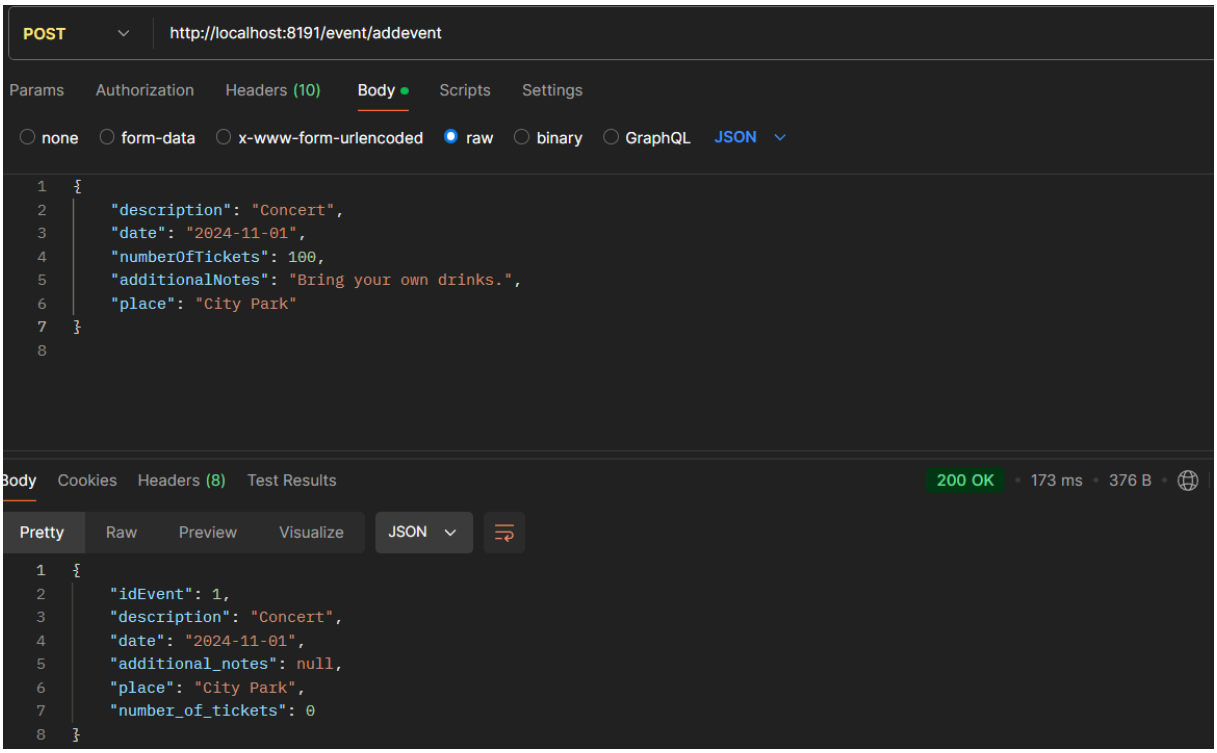


Figure 8:Backend API endpoints tested in Postman

5.2.2 Frontend – Angular Application

The frontend is developed using **Angular 15**. It interacts with the backend through REST APIs and provides a responsive UI for managing events.

Main features:

- Welcome section (Hero + animated background)
- Event list display using styled Bootstrap cards
- Event creation and update forms
- Search and filtering by event name or date
- Responsive layout for desktop and mobile

Technologies used:

- Angular CLI
- Bootstrap 5
- Custom CSS animations
- Angular services for HTTP communication

5.2.3 Database – MySQL

A **MySQL** database was used to store event data. The backend connects using Spring Data JPA. The database runs in a Docker container in the final deployment setup.

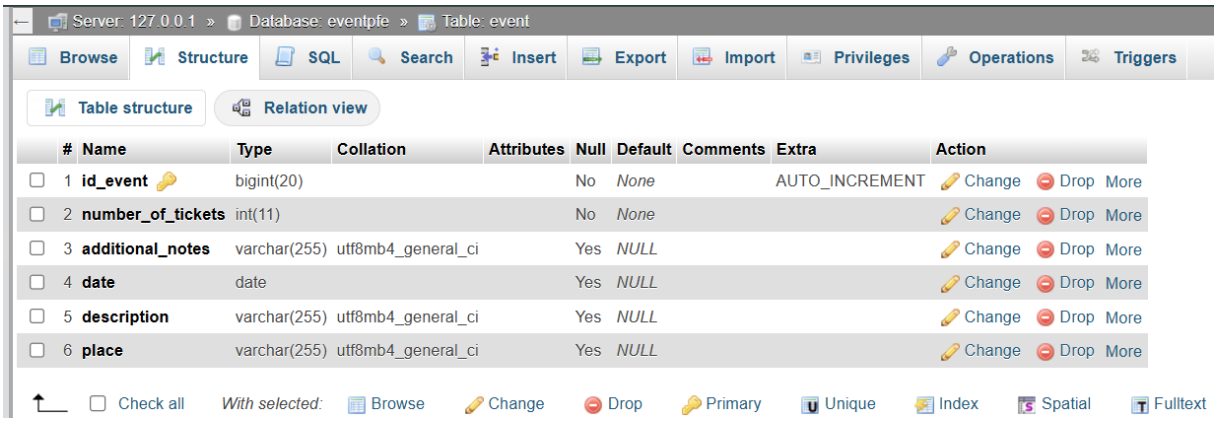


Figure 9: MySQL Event Table Schema

5.2.4 Dockerization

Both frontend and backend were containerized for deployment.

Backend Dockerfile:

```
# Use Eclipse Temurin JDK (Recommended for Java 17 applications)
FROM eclipse-temurin:17-jdk-jammy

# Set working directory
WORKDIR /app

# Copy the application JAR file
COPY target/EventManagement-0.0.1.jar app.jar

# Expose the application port
EXPOSE 8083

# Run the application
ENTRYPOINT ["java", "-jar", "/app/app.jar"]
```

Figure 10: Backend Dockerfile

Frontend Dockerfile (Multi-stage with NGINX):

```
FROM node:14 as build
WORKDIR /app

# Install dependencies
COPY package*.json ./
RUN npm install
# Copy project files into the Docker image
COPY . .

# Build the Angular application
RUN npm install -g @angular/cli@15
RUN ng build --configuration production

# Stage 2: Serve the application with Nginx
FROM nginx:alpine
COPY --from=build /app/dist/angular-hamza /usr/share/nginx/html
COPY nginx.conf /etc/nginx/nginx.conf
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

Figure 11: Frontend Dockerfile

5.3 CI/CD Pipeline Implementation

5.3.1 Tool Selection and Justification

Although many CI/CD tools are available, Jenkins was chosen for its unmatched flexibility, extensive plugin ecosystem, and self-hosted support. Unlike GitHub Actions or GitLab CI, Jenkins does not lock the pipeline to a specific source platform and integrates seamlessly with third-party tools such as SonarQube, Docker, and Nexus.

Why Jenkins *specifically*?

Criteria	Why Jenkins Wins in This Project
Open-source & self-hosted	Completely open-source, no limitations, and can run locally (perfect for student/local testing)
Plugin ecosystem	1800+ plugins for everything: Git, Docker, Nexus, SonarQube, Prometheus...
Custom pipelines	Can write fully customized workflows using Jenkinsfile
Tool compatibility	Works natively with **SonarQube, Docker, Maven, Nexus, Prometheus**, and more

Criteria	Why Jenkins Wins in This Project
Independence from Git hosting	GitHub Actions only works inside GitHub; Jenkins works with GitHub, GitLab, Bitbucket, etc.
Community & Documentation	Huge support community with endless tutorials, fixes, and examples
Proven in production	Used by companies like **Netflix, LinkedIn, Intel** enterprise-tested and production-ready

Jenkins supports declarative and scripted pipelines, making it suitable for both simple and complex workflows. Its mature plugin library and compatibility with the technologies used in this project including Maven, Prometheus, and infrastructure tools made it an ideal choice for building a robust, end-to-end DevOps pipeline. Furthermore, its ability to run on a local VM without requiring external services aligned perfectly with the project's resource constraints.

When Others Might Be Better

GitHub Actions : If your entire repo is in GitHub and you want simplicity with low setup.

GitLab CI/CD : If you're using GitLab for code and want tight integration.

CircleCI : If you want fast, cloud-hosted pipelines (paid).

Drone CI : If you prefer container-based pipelines with YAML simplicity.

But they often:

- Limit what you can do on free plans
- Require external integrations for Sonar/Nexus/Docker hosting
- Are **harder to host locally** (which is critical for your setup)

5.3.2 Jenkins Setup

To automate the build, test, and deployment processes, **Jenkins** was installed on **VM1**, which serves as the dedicated CI/CD server in the project's infrastructure. The installation followed a structured approach:

Installation Process

Jenkins was installed using the official Debian package repository on **Ubuntu 22.04**, following these steps:

- **Install Java**
- **Install Maven**
- **Install Trivy**
- **Install latest Docker version**
- **Install Ngrok**
- **Install Jenkins**
- **Start Jenkins**

Jenkins was then made accessible via the web interface on:

http://192.168.163.9:8080

Initial setup was completed by unlocking Jenkins using the generated password located at:

/var/lib/jenkins/secrets/initialAdminPassword

5.3.3 Plugin Configuration

The following plugins were installed to support the pipeline and tool integrations:

After selection of " Install suggested plugins" in the beginning we add :

- **GitHub Integration Plugin**
- **Role-based Authorization Strategy**
- **Maven integration**
- **Sonargraph integration**
- **Config File Provider**
- **Docker Pipeline**
- **SonarQube Scanner**
- **Pipeline : Stage View**
- **Kubernetes**
- **Kubernetes CLI**
- **Kubernetes Credentials**
- **Kubernetes Client API**

5.3.4 System Configuration

Jenkins was configured with:

- **Credentials**:

T	P	Store ↓	Domain	ID	Name
		System	(global)	Git-credential	hamzakalech/****** (Git-credential)
		System	(global)	Sonar-token	Sonar-token
		System	(global)	docker-cred	hamzakalech/****** (docker-cred)
		System	(global)	mail	kalechhamza238@gmail.com/****** (mail)
		System	(global)	k8s	k8s
		System	(global)	Git-token	Git-token
		System	(global)	JENKINS_API_TOKEN	JENKINS_API_TOKEN

Figure 12: Jenkins Credentials

- **Docker host access** (Jenkins user added to the docker group)

sudo usermod -aG docker Jenkins

newgrp docker

- **Role-based Authorization Strategy**

Manage Roles

Global roles

	Role	Overall		Credentials					Agent							Job								Run			View				SCM	Metrics		
		Administer	Read	Create	Delete	ManageDomains	Update	View	Build	Configure	Connect	Create	Delete	Disconnect	Provision	Build	Cancel	Configure	Create	Delete	Discover	Move	Read	Workspace	Delete	Replay	Update	Configure	Create	Delete	Read	Tag	HealthCheck	ThreadDump	View
🗑	admin	☑	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐
🗑	developer	☐	☑	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☑	☑	☐	☐	☐	☐	☐	☑	☐	☐	☐	☐	☐	☐	☑	☐	☐	☐	☐

Figure 13: Rbac Manage Roles

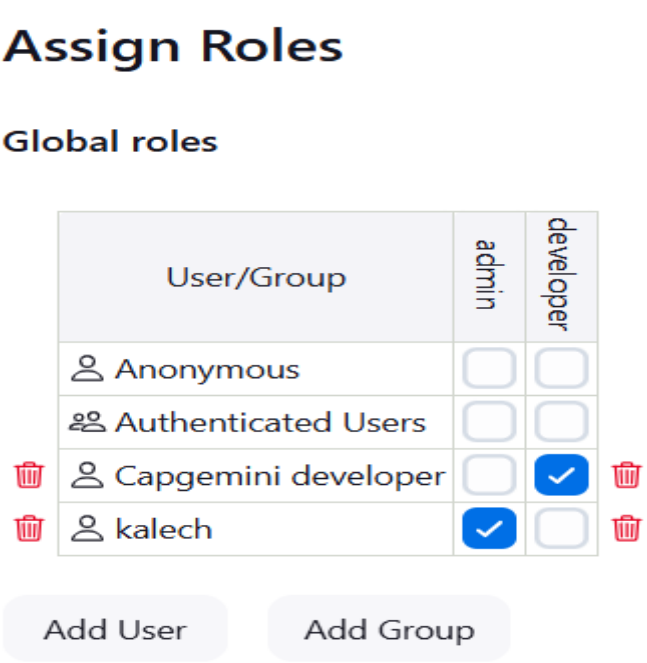


Figure 14: Rbac Assign Role

- **SonarQube server settings** under Global Tools Configuration

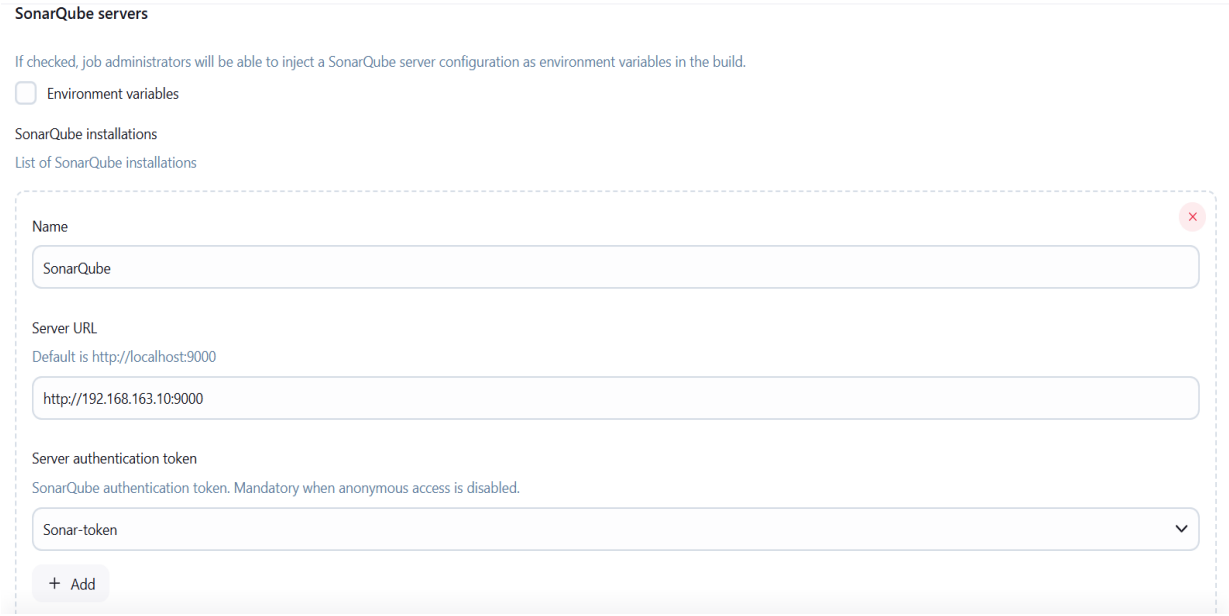


Figure 15: SonarQube Server Settings

- **Maven and JDK paths** for project builds

Figure 16: Maven ans JDK Paths

- **E-mail And Extended E-mail Notification**

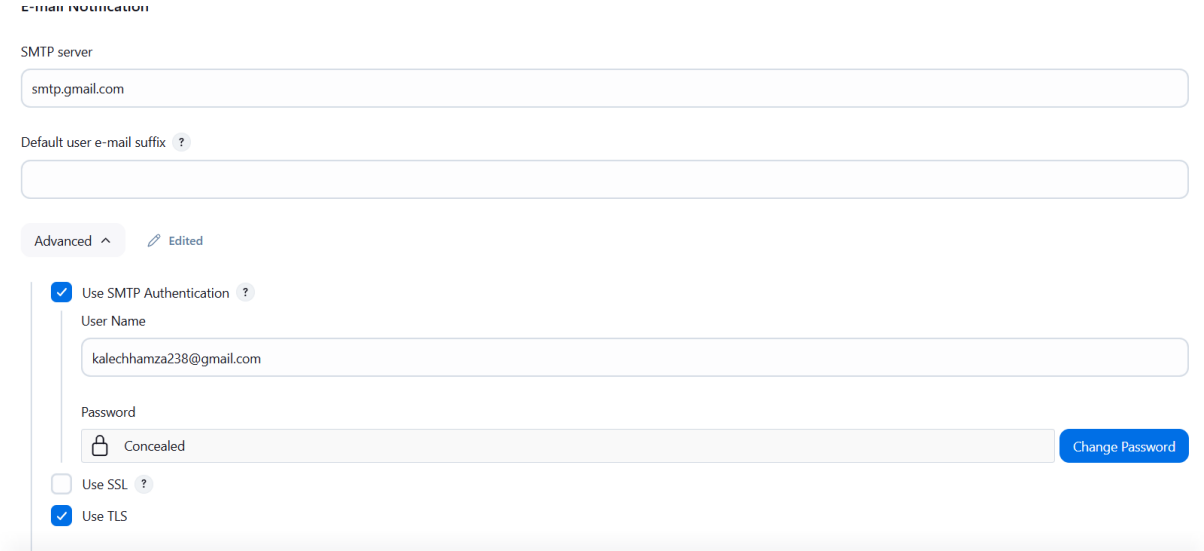


Figure 17: E-mail Notification

This setup allowed Jenkins to serve as the central automation tool in the platform, orchestrating the CI/CD pipeline and integrating seamlessly with the other components.

5.3.5 Pipeline Design

To automate the entire DevOps workflow from source code integration to deployment in a cloud-native environment a complete **CI/CD pipeline** was implemented using Jenkins. The pipeline was designed as two separate jobs:

- **CI Job**: Handles code compilation, testing, analysis, packaging, Docker image build, scanning, and artifact push.
- **CD Job**: Responsible for deploying the updated version of the application to **Azure Kubernetes Service (AKS)**.

Both pipelines are defined using **declarative syntax** in Jenkins and executed via scripted stages.

CI Pipeline Overview

The Continuous Integration pipeline performs the following steps:

Stage	Description
Git Checkout	Clones the source code from GitHub (private repo and branch)
Maven Build	Cleans, compiles, and runs unit tests using Maven
Unit Testing	Executes JUnit and Mockito test suite
Trivy FS Scan	Performs a filesystem vulnerability scan on the project source code
SonarQube Analysis	Analyzes code quality and security using SonarQube
Quality Gate Check	Validates the result against SonarQube Quality Gate policy
Artifact Deployment	Uploads the JAR to Nexus Maven repository
Docker Image Build	Builds a Docker image and tags it dynamically using the Jenkins build number (e.g., event-management:v12)
Trivy Image Scan	Performs a security scan of the Docker image
Docker Publish	Pushes the Docker image to Docker Hub (secured via Jenkins credentials)
Kubernetes Manifest Update	Updates the image tag in the Kubernetes manifest repo and pushes to GitHub
Trigger CD Pipeline	Triggers a separate Jenkins job for deploying the new version to AKS

Each Docker image is tagged with the **Jenkins build number** using the variable ${BUILD_NUMBER}. This ensures a unique version for every CI execution (e.g., event-management:v17), enabling reliable tracking and rollback of image versions in both development and production environments.

A notification email is sent upon completion, displaying the pipeline status with color-coded formatting.

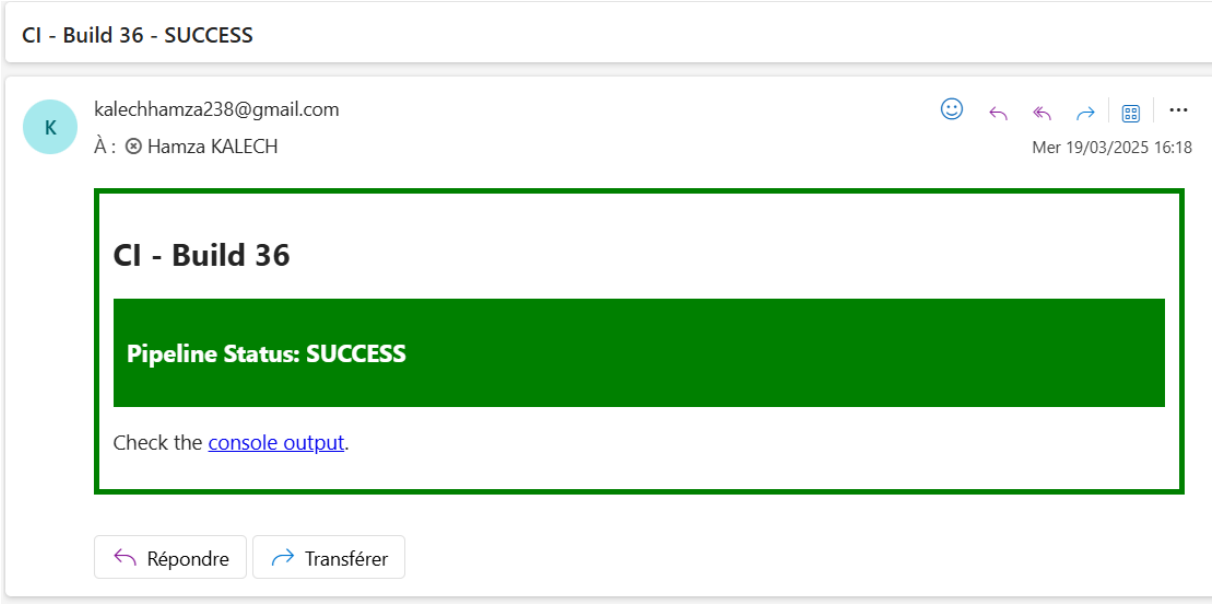


Figure 18: Success Build E-mail

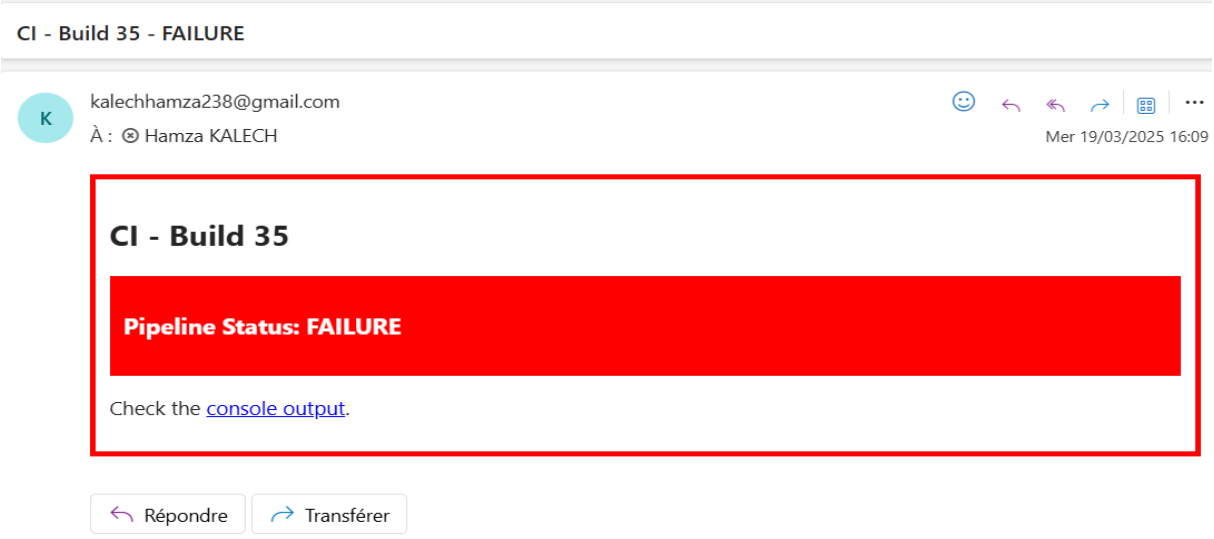


Figure 19: Failure Build E-mail

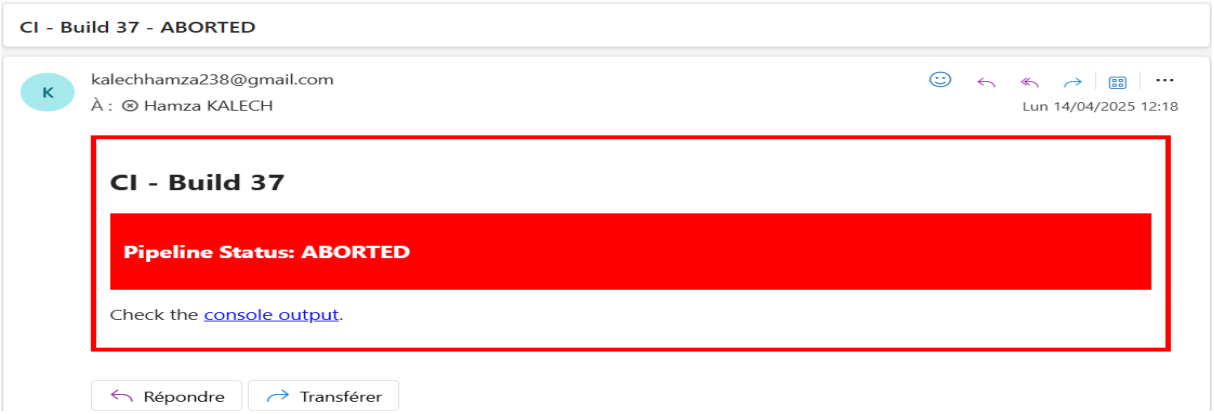


Figure 20: Aborted Job

With a link to the console output of concerned build

CD Pipeline Overview

The Continuous Deployment pipeline is triggered once the new Docker image tag is committed and pushed to the **Devops_CD** GitHub repository and CI pipeline completed successfully. It performs the following tasks:

Stage	Description
Git Checkout	Clones the updated deployment manifests
Kubernetes Deployment	Applies updated Kubernetes manifests including Deployment, HPA, and optionally Ingress
TLS Certificate Setup	Verifies existence of ClusterIssuer for Let's Encrypt and applies ci.yaml if missing
Deployment Verification	Displays deployment details: pods, services, ingress, and volumes

These steps are performed using kubectl commands, authenticated through the **Azure AKS kubeconfig** stored in Jenkins credentials.

```
stage('Deployment') {
    steps {
        echo 'Deploying App to AKS'
        withKubeConfig(caCertificate: '', clusterName: 'hamzaDevOps', contextName: '', credentialsId: 'k8s', namespace: 'hamzadevops',
                sh " kubectl apply -f Manifest/manifest.yaml -n hamzadevops"
                sh " kubectl apply -f Manifest/hpa.yaml"
                sleep 30
               // Conditionally apply Ingress
                script {
                    def ingressExists = sh(
                        script: "kubectl get ingress pievent-ingress -n hamzadevops --ignore-not-found | grep pievent-ingress",
                        returnStatus: true
                    )
                    if (ingressExists != 0) {
                        echo "Ingress not found. Applying ingress.yaml"
                        sh "kubectl apply -f Manifest/ingress.yaml -n hamzadevops"
                    } else {
                        echo "Ingress already exists. Skipping apply."
                    }
                }
                sleep 30

                script {
                    def ciExists = sh(
                        script: "kubectl get clusterissuer letsencrypt-prod --ignore-not-found | grep letsencrypt-prod",
                        returnStatus: true
                        )
                    if (ciExists != 0) {
                        echo "ClusterIssuer not found. Applying ci.yaml"
                        sh "kubectl apply -f Manifest/ci.yaml"
                    } else {
                        echo "ClusterIssuer already exists. Skipping apply."
```

Figure 21: CD Pipeline

This two-pipeline setup ensures **automated and secured delivery** of the application from code commit to cloud deployment, with added security scans, static code analysis, and versioned releases.

5.3.6 GitHub Integration

GitHub played a central role in the CI/CD pipeline, serving as the version control platform for both the **application source code** and the **deployment configuration**.

Two repositories were integrated into Jenkins:

Repository	Purpose
Hamza_Devops	Hosts the application source code (Spring Boot project)
Devops_CD	Hosts the Kubernetes manifest files for deployment

Authentication & Credential Management

To securely connect Jenkins with GitHub, credentials were configured using Jenkins' **Credentials Manager**:

- A **GitHub Personal Access Token** was stored with ID: Git-token
- A **Git username/password** pair was stored with ID: Git-credential

These credentials were referenced within the pipeline using:

credentialsId: 'Git-credential'

and

withCredentials(credentialsId: 'Git-token', variable: 'GITHUB_TOKEN')

This setup enabled Jenkins to both **pull code** from GitHub and **push updates** (like manifest changes) securely.

Webhook Triggering with Ngrok

To enable **automated CI triggers** on code push events despite Jenkins being hosted locally on a VM a secure tunnel was created using **Ngrok**. This provided a temporary but public HTTPS URL that GitHub could use to reach the local Jenkins instance.

This approach allows webhook-based automation without hosting Jenkins in a public cloud, combining local development with real-world CI/CD behavior.

Steps:

1. Exposed local Jenkins server on port 8080 via:

ngrok http 8080

2. Used the generated Ngrok HTTPS URL (e.g., https://random-id.ngrok.io) as the **GitHub Webhook URL**
3. Configured the webhook under the repository settings to send **push events** to Jenkins

As a result, every new push to the Hamza-branch automatically triggered the pipeline in Jenkins without manual polling.

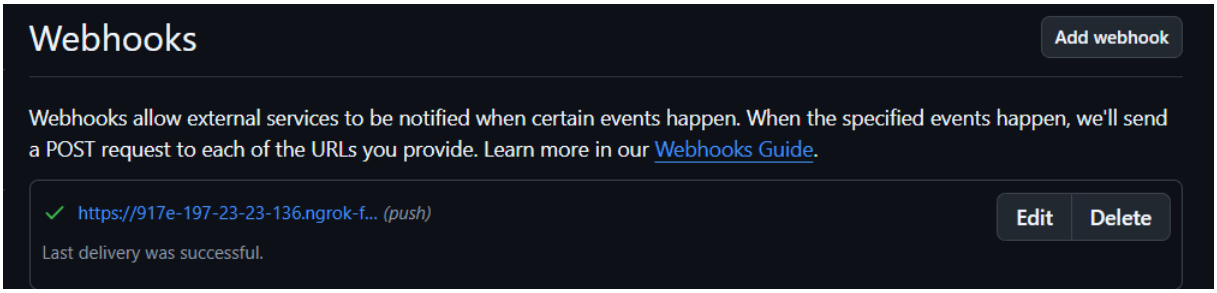


Figure 22: Webhooks active on Github

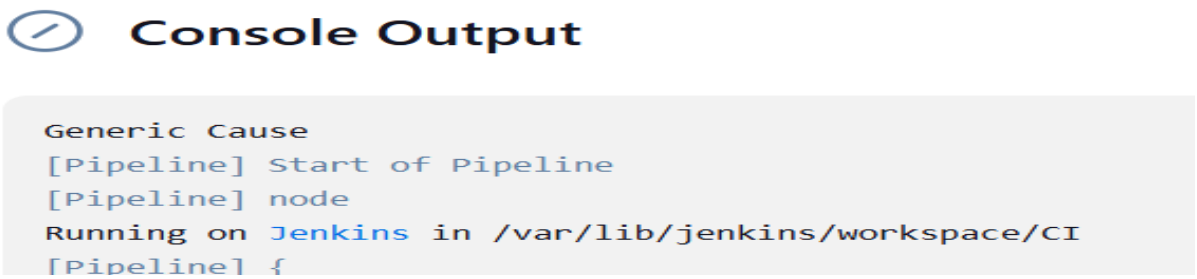


Figure 23: Launched Jod with Generic Cause

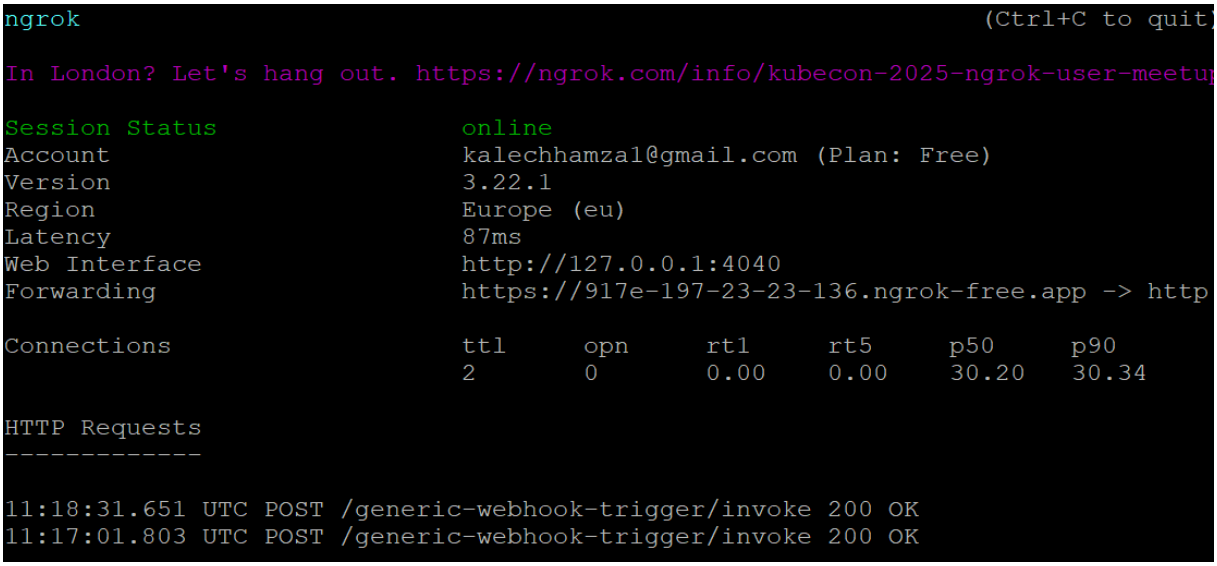


Figure 24: Ngrok View

This allows Jenkins to detect when new commits are pushed to the Hamza-branch, and trigger builds accordingly.

Manifest Repository Automation

During each CI run, the pipeline updates the Kubernetes manifest inside the Devops_CD repository to reflect the newly built Docker image version:

- Jenkins clones the Devops_CD repo
- It modifies the image tag in the YAML file using sed
- Then commits and pushes the changes back to GitHub using a Git identity set in the script:

git config user.name "Jenkins"

git commit -m "Updated image tag to ${IMAGE_TAG}"

git push origin main

This automatic version tracking ensures that the deployment always uses the latest image produced by the CI process.

Figure 25: Git Repo History

5.3.7 SonarQube Integration

To ensure code quality, maintainability, and security, **SonarQube** was integrated into the CI pipeline as a static code analysis tool. SonarQube provides real-time feedback on code issues, complexity, duplication, and potential vulnerabilities all aligned with industry standards like OWASP and CWE.

SonarQube Setup

SonarQube was deployed as a **Docker container** on **VM2** (the dedicated Code Quality server) and accessed via:

http://192.168.163.10:9000

After initial configuration, a dedicated **SonarQube token** was created and stored in Jenkins' Credentials Manager (Sonar-token) to authenticate and run analysis jobs from jenkins.

Integration into Jenkins

The following steps were performed to enable SonarQube integration:

1. **Installed the SonarQube Scanner Plugin** in Jenkins
2. Added the token (Sonar-token) to Jenkins credentials
3. Configured **SonarQube server** in Jenkins under:

Manage Jenkins > Configure System > SonarQube Servers

In the pipeline, the following stages were added:

stage('SonarQube Analysis')

stage('Quality Gate Check')

The first stage performs the analysis. The second stage **checks the quality gate status**, and **aborts the build** if the gate fails enforcing code quality as a policy, not just a recommendation.

Webhook Configuration

To improve integration and automate downstream processes, a SonarQube Webhook was configured. This webhook sends a JSON payload to a specific endpoint when an analysis is completed allowing Jenkins to be notified immediately and perform the Quality Gate check.

The webhook was configured under:
`Administration > Configuration > Webhooks`
with the following target:
`http://192.168.163.9:8080/sonarqube-webhook`

This local Jenkins endpoint is responsible for listening to analysis results and synchronizing them with the pipeline. While Jenkins also waits for the Quality Gate result via the `waitForQualityGate` step, the webhook ensures real-time delivery and faster build decision-making.

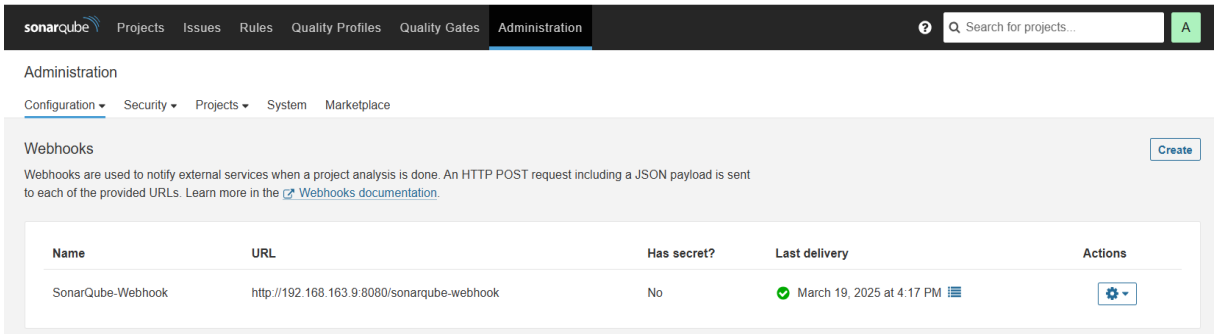


Figure 26: Sonar-Webhook

Dashboard & Quality Metrics

SonarQube provides rich dashboards where metrics such as:

- **Code coverage**
- **Code smells**
- **Bugs and vulnerabilities**
- **Duplications**
- **Security hotspots**

...are displayed per commit and per branch.

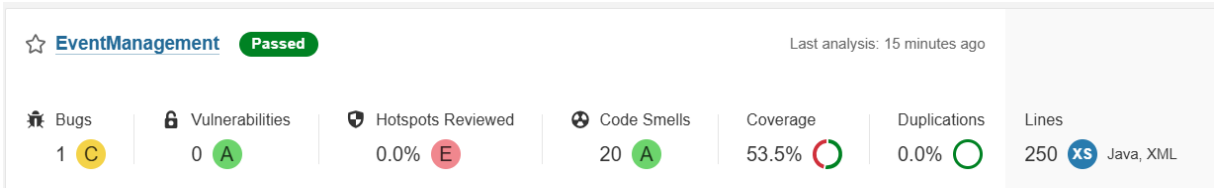


Figure 27: Sonar_Quality_gate

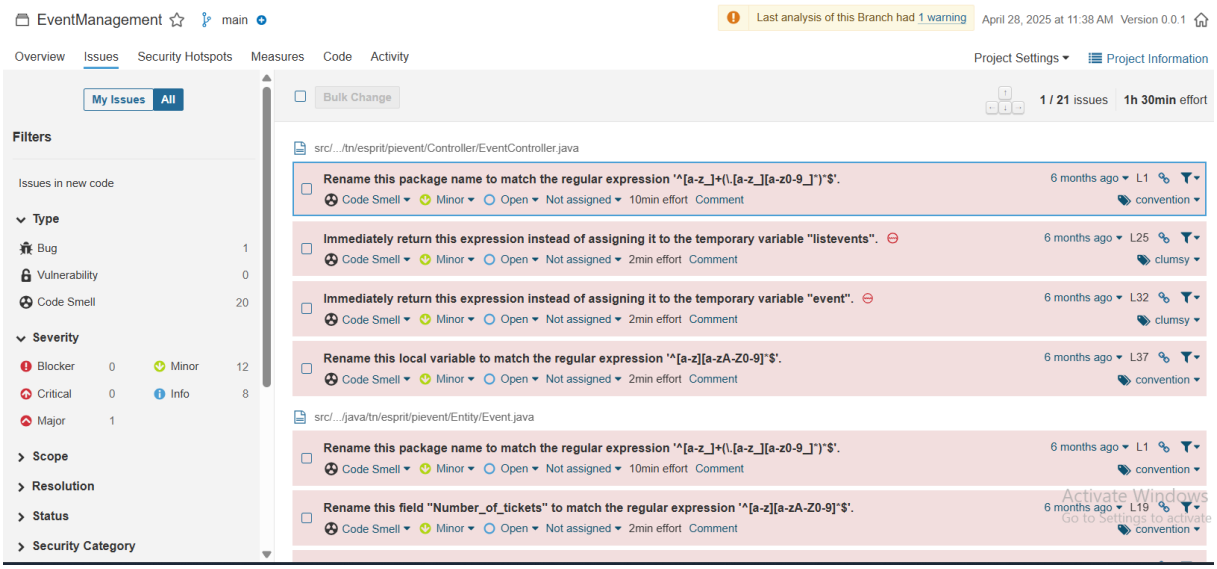


Figure 28: Sonar_Issues

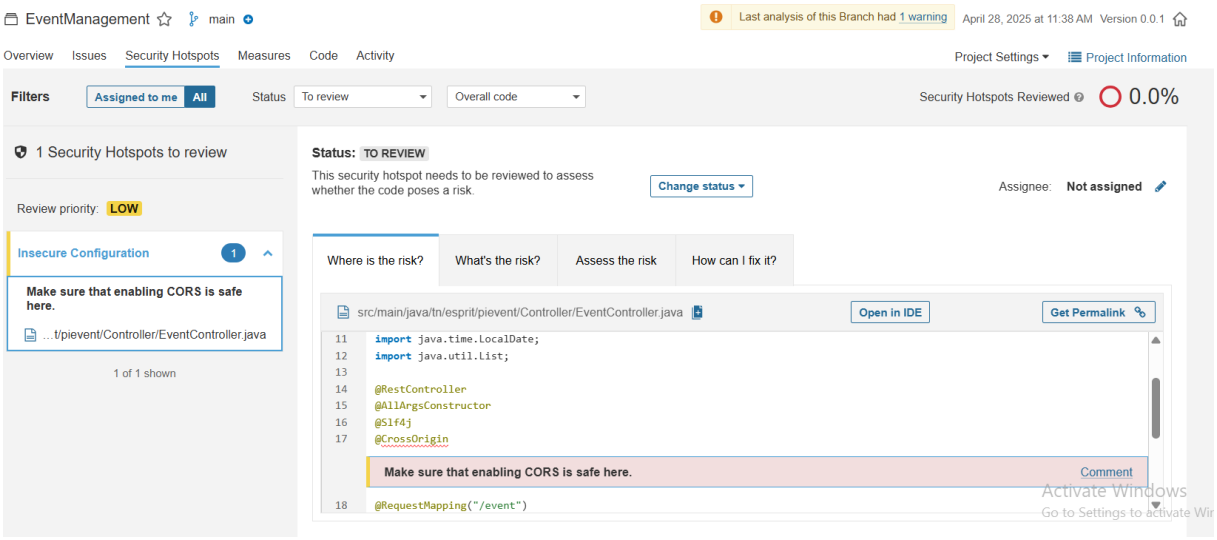


Figure 29: Sonar_Security_Hotspots

This integration helps **catch issues early in the pipeline**, reducing technical debt and promoting clean code practices across the development lifecycle.

5.3.8 Nexus Artifact Repository Integration

To manage application build artifacts and ensure reliable version control for both JAR files, **Nexus Repository Manager** was integrated into the CI/CD pipeline. This allows the DevOps platform to store, organize, and reuse builds supporting traceability and rollback capabilities.

Nexus Setup

Nexus was installed on **VM3** as a Docker container and made accessible at:

http://192.168.163.11:8081

Two repositories were configured:

Repository Type	Name	Purpose
Maven (hosted)	maven-releases	Stores versioned Maven artifacts (JARs)
Maven (hosted)	maven-snapshots	Stores snapshot builds during development

Maven Integration with Jenkins

The Maven pom.xml was configured with the distributionManagement section pointing to the Nexus URLs:

```
<!-- Deploy to Nexus -->
<distributionManagement>
        <repository>
          <id>maven-releases</id>
          <url>http://192.168.163.11:8081/repository/maven-releases/</url>
        </repository>
        <snapshotRepository>
          <id>maven-snapshots</id>
          <url>http://192.168.163.11:8081/repository/maven-snapshots/</url>
        </snapshotRepository>
</distributionManagement>
```

Figure 30: Maven Integration

Credentials for Nexus were securely added to Jenkins and used during the mvn deploy stage.

This ensured that each successful build published its .jar file to the appropriate Maven repository on Nexus depending on whether it was a snapshot or a release version.

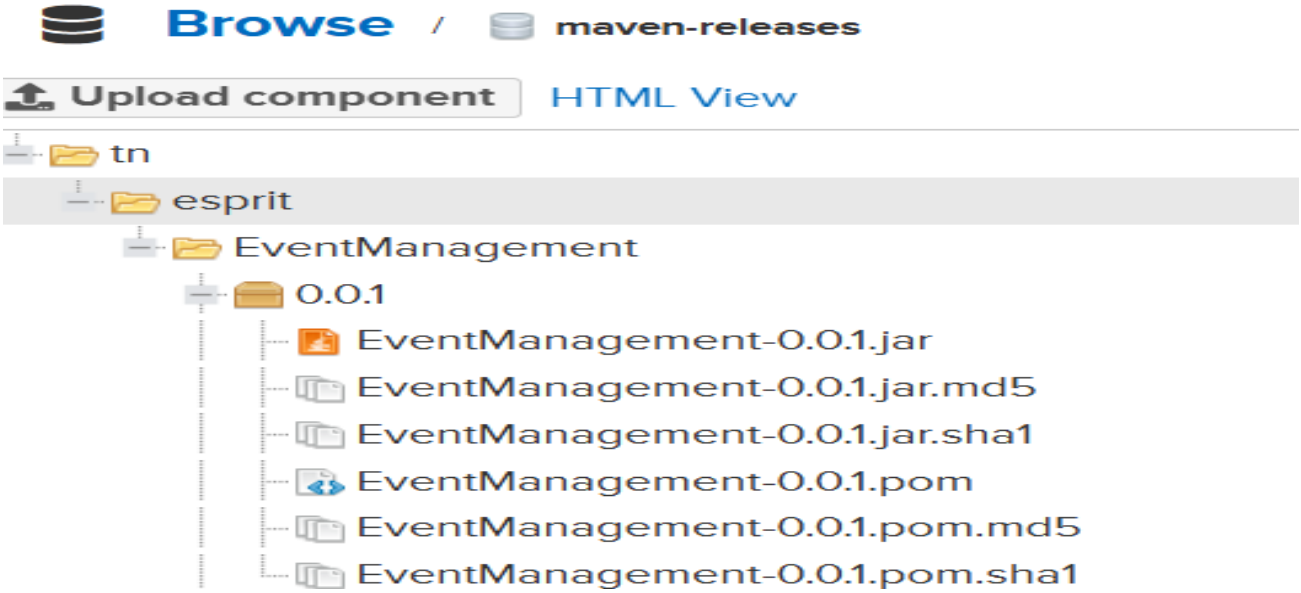


Figure 31: Maven-releases

This setup provides a reliable and secure way to manage and distribute software artifacts across environments essential for a professional DevOps pipeline.

5.4 Infrastructure as Code (Terraform)

To provision and manage cloud resources in a scalable and repeatable manner, the project adopted **Terraform** for implementing Infrastructure as Code (IaC). Terraform allows complete automation of infrastructure provisioning, enforces best practices, and guarantees consistency across different environments.

Why Terraform?

- Open-source and widely adopted across industries
- Declarative configuration language (HCL)
- Multi-cloud support (Azure, AWS, GCP)
- Enables modularization and code reusability
- Strong ecosystem for integrating Kubernetes and Helm deployments

Terraform fits naturally into a DevOps pipeline by ensuring that infrastructure creation, updates, and deletions are version-controlled, automated, and safe.

Terraform Setup

Terraform was installed on a dedicated **INFRA VM** alongside:

- **Azure CLI**: for authenticating and managing Azure resources
- **Helm**: for deploying Kubernetes Helm charts

Installed via standard Ubuntu APT package manager.

Modular Terraform Project Structure

To follow best practices and ensure scalability, the Terraform project was fully modularized. The folder structure is organized as follows:

```
terraform/
│
├── main.tf          # Aggregates and calls all modules
├── outputs.tf       # Defines exported outputs
├── variables.tf     # Defines reusable global variables
│
└── modules/
   ├── resource_group/    # Creates Azure Resource Group
```

```
│   ├── main.tf
│   ├── outputs.tf
│   └── variables.tf
│
├── network/          # Virtual Network and Subnets
│   ├── main.tf
│   ├── outputs.tf
│   └── variables.tf
│
├── aks/              # Azure Kubernetes Service Cluster
│   ├── main.tf
│   ├── outputs.tf
│   └── variables.tf
│
├── bootstrap/        # Kubernetes RBAC (Jenkins ServiceAccount, Roles, etc.)
│   ├── jenkins-clusterrole.yaml
│   ├── jenkins-role.yaml
│   ├── namespace.yaml
│   └── secret.yaml
│
├── monitoring/       # Prometheus and Grafana deployment via Helm
│   ├── main.tf
│   └── values.yaml
│
├── argocd/           # ArgoCD deployment via Helm
│   ├── main.tf
│   ├── outputs.tf
│   ├── variables.tf
│   └── values.yaml
```

```
│
└── velero/          # Velero backup solution deployment
    ├── main.tf
    ├── variables.tf
    └── velero-values.yaml
```

Each module is **self-contained**, with its own set of:

- main.tf for resource definition
- variables.tf for input parameters
- outputs.tf for exporting resource outputs

Key Infrastructure Components

Module	Description
Resource Group	Logical grouping of Azure resources
Network	Virtual network and subnet for AKS
AKS Cluster	Kubernetes cluster with Azure CNI networking
Bootstrap	Deploys Kubernetes RBAC (Roles, RoleBindings, ServiceAccounts) for Jenkins
Monitoring	Installs Prometheus and Grafana into Kubernetes using Helm
ArgoCD	GitOps continuous delivery tool, installed via Helm
Velero	Cloud-native backup and disaster recovery tool for Kubernetes workloads

RBAC Bootstrap Configuration

A **Bootstrap module** was created to apply Kubernetes YAML files using Terraform to secure the Jenkins environment:

- **Service Account** for Jenkins
- **ClusterRole** with limited permissions
- **RoleBinding** and **ClusterRoleBinding**
- **Namespace creation** and **Secrets management**

This ensures **fine-grained access control** inside Kubernetes right from the initial deployment.

jenkins-role.yaml :

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: jenkins-role
  namespace: hamzadevops
rules:
  # Permissions for core API resources
  - apiGroups: [""]
    resources:
      - secrets
      - configmaps
      - persistentvolumeclaims
      - services
      - pods
    verbs: ["get", "list", "watch", "create", "update", "delete", "patch"]

  # Permissions for apps API group
  - apiGroups: ["apps"]
    resources:
      - deployments
      - replicasets
    verbs: ["get", "list", "watch", "create", "update", "delete", "patch"]

  # Permissions for networking API group
  - apiGroups: ["networking.k8s.io"]
    resources:
      - ingresses
    verbs: ["get", "list", "watch", "create", "update", "delete", "patch"]

  # Permissions for autoscaling API group
  - apiGroups: ["autoscaling"]
    resources:
```

Figure 32:Jenkins-role yaml file

Velero Backup and Disaster Recovery Integration

A dedicated **velero module** was created to automate the deployment of **Velero** for backup and disaster recovery purposes:

- **Azure Blob Storage Account** is provisioned dynamically using Terraform:

  resource "azurerm_storage_account" "velero" { ... }

  resource "azurerm_storage_container" "velero" { ... }

- **Primary Access Key** for Azure Storage was extracted and injected securely into the Velero Helm Chart values.
- **Velero Helm Release** deployed the Velero service into Kubernetes, configured for:

- Backup of Kubernetes persistent volumes
- Snapshotting resources and cluster states

- **Azure Storage Account** `velerobackupkalech` is created dynamically via Terraform.
- A **private Azure Blob Storage container** `velero` is created inside it.
- **Access keys** are injected into Kubernetes as Secrets securely.
- Velero is deployed via a **Helm release**, configured to:
  - Store backups in Azure Blob
  - Enable volume snapshots
  - Attach Azure plugins dynamically at runtime

Highlights:

- **Use of Access Key Authentication** (no Azure AD dependency) : azure student account can't use Azure AD dependency .
- **Custom velero-values.yaml** configured for Azure integration

```yaml
credentials:
  existingSecret: cloud-credentials
  useSecret: true
  name: cloud-credentials
  key: cloud

configuration:
  features: EnableCSI
  backupStorageLocation:
    - name: default
      provider: azure
      bucket: velero
      config:
        resourceGroup: hamza-resources
        storageAccount: velerobackupkalech
        storageAccountKeyEnvVar: AZURE_STORAGE_ACCOUNT_ACCESS_KEY

  volumeSnapshotLocation:
    - name: default
      provider: azure
      config:
        resourceGroup: MC_hamza-resources_hamzaDevOps_eastus
        subscriptionId: 6126c082-d56e-4f42-8033-834a1072516c
        apiTimeout: 5m
        incremental: true

snapshotsEnabled: true

initContainers:
  - name: velero-plugin-for-microsoft-azure
    image: velero/velero-plugin-for-microsoft-azure:v1.9.0
    volumeMounts:
```

Figure 33: velero-values.yaml

- **Snapshots enabled** for volume protection

This ensures the platform supports **automated backup** and **disaster recovery** out-of-the-box.

Deployment Flow

The overall Infrastructure Deployment sequence:

1. Authenticate to Azure (az login)
2. Run terraform init
3. Run terraform plan
4. Run terraform apply
5. Resources and Kubernetes applications are deployed automatically
6. RBAC policies, Monitoring, GitOps, and Backup are provisioned without manual intervention

Terraform automatically handled:

- Creation of the Azure Resource Group
- Setup of Virtual Network and Subnet
- Provisioning of a fully configured AKS Cluster
- Installation of ArgoCD, Prometheus/Grafana, and Velero through Helm Charts
- Application of Bootstrap YAMLs for Jenkins RBAC setup inside Kubernetes

Security & Best Practices

- **RBAC security** was enforced using manually written Kubernetes YAML files applied via Terraform (for Jenkins Service Accounts, Roles, and Bindings).
- **Terraform modules** were created for logical separation of concerns.

This modular, secure, and cloud-native Infrastructure as Code approach ensures that the DevOps platform can be deployed, scaled, and maintained easily in any cloud environment with minimal human intervention.

5.5 Kubernetes Deployment

Following infrastructure provisioning through Terraform, the platform was deployed to the **Azure Kubernetes Service (AKS)** cluster using a suite of declarative Kubernetes YAML manifests stored in a structured Manifest/ directory.

The platform consists of :

- EventManagement app
- Full manifest.yaml logic
- Calico network policies
- HPA
- Ingress + TLS
- cert-manager
- SSL Labs result
- Proper namespace structure

all orchestrated by Kubernetes for high availability, scalability, and operational automation.

5.5.1 Deployment Overview

Layer	Technology	Purpose
Backend	Spring Boot (EventManagement)	Business logic and REST API
Frontend	Angular	User interface, served via NGINX
Database	MySQL	Persistent event data storage
Ingress	NGINX + cert-manager	HTTPS routing and certificate management
Monitoring	Prometheus/Grafana	Metrics and dashboarding (handled in 4.6)
Backup	Velero	Scheduled backup of namespace and volumes
Security	Calico	Network policies and zero-trust rules

5.5.2 Namespace Isolation

All application workloads were deployed inside a dedicated Kubernetes namespace:

namespace: hamzadevops

This ensures resource scoping, isolation, and simplified backup/restoration using Velero.

5.5.3 Application Components

The EventManagement platform was deployed using the file manifest.yaml, which includes:

- **Secrets** and **ConfigMaps** for MySQL configuration
- **PersistentVolumeClaim** using Azure Disk CSI + StorageClass
- **Spring Boot Deployment** with:

- Probes (/actuator/health/liveness and readiness)
- Environment variable injection via secrets/configs

- **Angular Frontend Deployment**
- **MySQL Stateful Deployment**
- **ClusterIP Services** for all three components

Each component defines memory and CPU requests and limits for fair scheduling and resource enforcement.

```
kalech@infra:~/Devops_CD$ kubectl get all -n hamzadevops
NAME                                  READY   STATUS    RESTARTS   AGE
pod/angular-app-86ffc97dc-zt49c       1/1     Running   0          2m17s
pod/cm-acme-http-solver-nhlcn         1/1     Running   0          80s
pod/eventmanagement-8d567df4-nmwjl    1/1     Running   0          2m18s
pod/mysql-57bcdfbb95-p2wqd            1/1     Running   0          2m20s

NAME                                TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)          AGE
service/angular-service             ClusterIP   10.0.2.226   <none>        80/TCP           2m16s
service/cm-acme-http-solver-fbz5n   NodePort    10.0.2.125   <none>        8089:30536/TCP   80s
service/eventmanagement-service     ClusterIP   10.0.2.219   <none>        8083/TCP         2m17s
service/mysql-service               ClusterIP   10.0.2.212   <none>        3306/TCP         2m19s

NAME                                READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/angular-app         1/1     1            1           2m18s
deployment.apps/eventmanagement     1/1     1            1           2m19s
deployment.apps/mysql               1/1     1            1           2m21s

NAME                                           DESIRED   CURRENT   READY   AGE
replicaset.apps/angular-app-86ffc97dc          1         1         1       2m18s
replicaset.apps/eventmanagement-8d567df4       1         1         1       2m19s
replicaset.apps/mysql-57bcdfbb95               1         1         1       2m21s

NAME                                                    REFERENCE                    TARGETS              MINPOD
MAXPODS   REPLICAS   AGE
horizontalpodautoscaler.autoscaling/angular-hpa         Deployment/angular-app       cpu: <unknown>/65%   1
150       1          2m
horizontalpodautoscaler.autoscaling/eventmanagement-hpa Deployment/eventmanagement   cpu: <unknown>/65%   1
150       1          2m
```

Figure 34:Application Components in hamzadevops Namespace

5.5.4 Ingress and TLS Configuration

External access was configured using:

- **NGINX Ingress Controller** deployed via Helm
- ingress-app.yaml manifest to route:
  - /api to EventManagement backend
  - / to Angular frontend
- **Cert-Manager** issued a valid TLS certificate via ClusterIssuer from Let's Encrypt (ci.yaml)

- All routes are HTTPS by default using strict redirect annotations

5.5.5 Domain used:

https://hamzakalech.com

5.5.6 SSL Labs Security Rating

The deployment was tested using SSL Labs and received an **A+ rating**, indicating:

- Strong TLS version and cipher suite selection
- Proper use of HTTP Strict Transport Security (HSTS)
- DNS Certification Authority Authorization (CAA) configuration

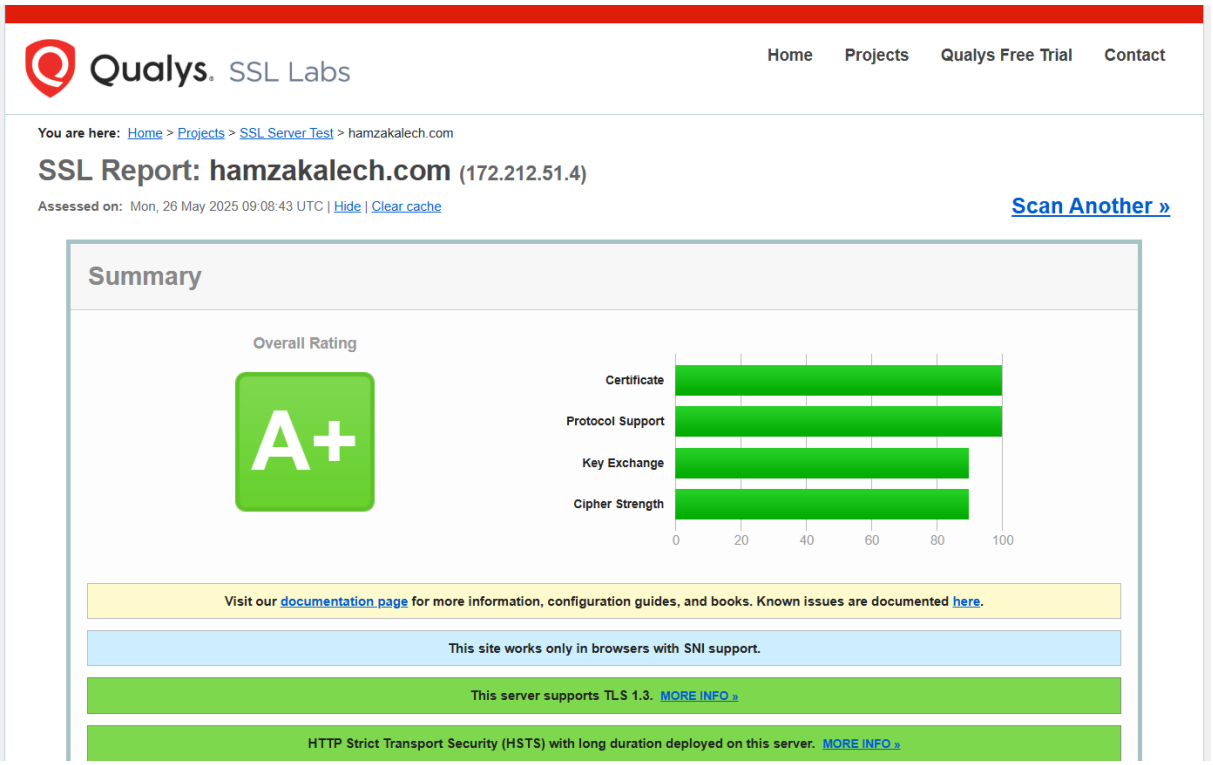


Figure 35:SSL Labs A+ Security Rating for hamzakalech.com

5.5.7 Auto-Scaling with HPA

Using hpa.yaml, two Horizontal Pod Autoscalers were configured:

- **EventManagement**: auto-scales from 1 to 3 pods based on CPU > 80%
- **Angular Frontend**: auto-scales from 1 to 3 pods if CPU > 70%

This ensures elasticity based on workload fluctuations.

5.5.8 Network Policies (Zero Trust)

Using Calico CRDs defined in network-policies.yaml, strict ingress policies were enforced:

- **Default Deny All** rule at the namespace level
- Only NGINX can access Angular & EventManagement pods
- Only EventManagement can access MySQL (port 3306)
- Only Prometheus can scrape EventManagement metrics
- Temporary allowance for cert-manager's ACME challenge

```
kalech@infra:~/Devops_CD$ kubectl get networkpolicy -n hamzadevops
NAME                POD-SELECTOR                                AGE
allow-acme-solver   acme.cert-manager.io/http01-solver=true   14m
default-deny-all    <none>                                      14m
```

Figure 36:Enforced Network Policies with Calico

5.5.9 Deployment Verification

Post-deployment validation was done via:

kubectl get pods -n hamzadevops

kubectl get svc -n hamzadevops

kubectl get ingress -n hamzadevops

kubectl describe certificate myapp-tls -n hamzadevops

All probes were passing, TLS certificates were active, and ingress routing was verified.

This deployment setup demonstrates a **production-grade**, **secure**, and **scalable** Kubernetes environment fully automated and observable.

5.6 Backup & Recovery / Security

Ensuring the reliability, safety, and resilience of a DevOps platform requires more than just deployment automation, it demands secure runtime behavior and continuous backup of critical data. In this project, two major practices were implemented to achieve this:

- **Velero** for backup and disaster recovery
- **Calico** for fine-grained Kubernetes network security (Zero Trust)

5.6.1 Velero – Backup and Recovery

To protect workloads and persistent volumes in Kubernetes, **Velero** was deployed using a dedicated Terraform module and configured to back up the entire hamzadevops namespace.

Deployment via Terraform + Helm

- A dedicated **Azure Storage Account** and **Blob Container** were provisioned using Terraform
- Access credentials were injected securely into Velero using Kubernetes Secrets
- Velero was installed via helm_release with a custom velero-values.yaml file

Key configurations included:

- Backup volume snapshots enabled
- Azure plugin injected dynamically via initContainers
- A VolumeSnapshotClass was also defined to work with Azure Disk CSI

Scheduled Backups

A scheduled backup was created via the backup-schedule.yaml manifest:

```yaml
apiVersion: velero.io/v1
kind: Schedule
metadata:
  name: daily-backup
  namespace: velero
spec:
  schedule: "0 2 * * *"     # Run every day at 2 AM UTC
  template:
    ttl: 168h               # Retain backups for 7 days
    includedNamespaces:
      - hamzadevops
    snapshotVolumes: true
```

Figure 37: Velero Daily Backup Schedule

This setup ensures daily protection of:

- All Kubernetes resources (Deployments, Services, PVCs, Secrets, etc.)
- Volume data used by MySQL and EventManagement

```
kalech@infra:~/Devops_CD$ kubectl get schedule -n velero
NAME           STATUS    SCHEDULE    LASTBACKUP   AGE   PAUSED
daily-backup   Enabled   0 2 * * *                23s
kalech@infra:~/Devops_CD$
```

Figure 38:Velero Daily Backup Schedule

5.6.2 Calico Network Policies – Zero Trust Security

To enforce strict communication control and adhere to a **Zero Trust architecture**, a set of **Calico network policies** were defined and applied via network-policies.yaml.

Policy Highlights

Policy Name	Purpose
default-deny-all	Denies all ingress traffic by default in the namespace
allow-ingress-to-frontend-backend	Allows NGINX to access frontend and backend pods
allow-eventmanagement-to-mysql	Allows EventManagement backend to access MySQL
allow-prometheus-to-eventmanagement	Allows Prometheus to scrape /actuator/prometheus metrics
allow-acme-solver	Temporarily allows cert-manager to complete ACME challenges

All policies were written using **Calico CRD syntax** (crd.projectcalico.org/v1) instead of standard Kubernetes networking.k8s.io, enabling:

- Match by label
- Use of selectors
- Port-level restrictions
- Namespaced and cross-namespaced rules

This security posture prevents:

- Unauthorized lateral traffic between pods
- Accidental service exposure
- Misuse of backend or database ports

```
kalech@infra:~/Devops_CD$ kubectl describe networkpolicy -n hamzadevops
Name:         allow-acme-solver
Namespace:    hamzadevops
Created on:   2025-05-26 08:55:46 +0000 UTC
Labels:       <none>
Annotations:  <none>
Spec:
  PodSelector:     acme.cert-manager.io/http01-solver=true
  Allowing ingress traffic:
    To Port: <any> (traffic allowed to all ports)
    From: <any> (traffic not restricted by source)
  Not affecting egress traffic
  Policy Types: Ingress


Name:         default-deny-all
Namespace:    hamzadevops
Created on:   2025-05-26 08:55:43 +0000 UTC
Labels:       <none>
Annotations:  <none>
Spec:
  PodSelector:     <none> (Allowing the specific traffic to all pods in this namespace)
  Allowing ingress traffic:
    <none> (Selected pods are isolated for ingress connectivity)
  Not affecting egress traffic
  Policy Types: Ingress
```

Figure 39:Active Calico Network Policies for Zero Trust Enforcement

5.6.3 Benefits Achieved

- Automated disaster recovery for entire namespace + PVC data
- TLS certificates issued securely with Let's Encrypt
- Ingress traffic strictly controlled via Calico
- Reduced attack surface and strong isolation

Together, these backup and security strategies ensure that the **EventManagement platform** is **resilient**, **restorable**, and **defended** against internal and external threats.

5.7 Monitoring and Observability

To gain visibility into application health, infrastructure usage, and system behavior, a complete monitoring stack was deployed using the official **kube-prometheus-stack**. This stack includes **Prometheus** for metrics collection and **Grafana** for data visualization.

Deployment was performed via **Terraform + Helm**, ensuring full automation, configuration-as-code, and repeatability.

5.7.1 Deployment via Terraform and Helm

Monitoring resources were deployed using a dedicated monitoring Terraform module. The Helm chart used is kube-prometheus-stack from the Prometheus Community repository:

```
resource "helm_release" "monitoring" {
  name             = "monitoring"
  namespace        = "monitoring"
  repository       = "https://prometheus-community.github.io/helm-charts"
  chart            = "kube-prometheus-stack"
  version          = "45.7.1"  # specify chart version if needed
  create_namespace = true

  values = [file("${path.module}/values.yaml")]
}
```

Figure 40:Monitoring installation

5.7.2 Custom Configuration (values.yaml)

Key configurations in the Helm values file:

- **Grafana**:
    - Enabled with internal ClusterIP service
    - Admin credentials: kalech / grafana
    - Root URL: /grafana/ for subpath ingress routing
- **Prometheus**:
    - Internal ClusterIP service
    - External URL configured as /prometheus
    - Persistent volume (1Gi) using Azure Disk via azure-disk-standard
- **Kube State Metrics** and **Node Exporter**: Enabled with ClusterIP services

5.7.3 Metrics Collection

Once deployed, Prometheus automatically scraped metrics from:

- Kubernetes nodes, pods, and system components
- cAdvisor and kubelet metrics

- Kube-state-metrics (HPA, deployments, PVCs)
- Spring Boot /actuator/prometheus endpoint of the **EventManagement** app (via ServiceMonitor)

No additional exporters were required due to native support from the kube-prometheus-stack.

5.7.4 Grafana Dashboards

Grafana was auto-configured with a rich set of default dashboards. These allowed:

- Filtering by namespace, deployment, container, or pod
- Tracking memory/CPU usage, request/limit ratios, HPA behavior
- Viewing metrics from the hamzadevops namespace where the application runs

Figure 41:Grafana Dashboard for EventManagement Namespace

5.7.5 TLS-Secured Access via Ingress

Although Grafana and Prometheus services were ClusterIP by default, they were later exposed securely via Ingress using ingress-monitoring.yaml. Two subdomains were configured:

- https://grafana.hamzakalech.com
- https://prometheus.hamzakalech.com

TLS certificates were automatically issued using **cert-manager** with a ClusterIssuer referencing Let's Encrypt. Ingress rules included strict HTTPS enforcement.

```
kalech@infra:~/Devops_CD$ kubectl get ingress -n monitoring
NAME                 CLASS   HOSTS                                                ADDRESS        PORTS     AGE
monitoring-ingress   nginx   grafana.hamzakalech.com,prometheus.hamzakalech.com   172.212.51.4   80, 443   27m
kalech@infra:~/Devops_CD$
```

Figure 42:Secure Monitoring Access via TLS Ingress

5.7.6 Observability Outcomes

Feature	Benefit
Namespace-aware filtering	Drill-down into specific apps or environments
Pre-built dashboards	Production-grade visualization without manual setup
TLS-secured external access	Protects sensitive metrics and dashboard login
Persistent Prometheus storage	Metrics survive pod restarts or re-deployments

This monitoring stack provides real-time observability into all Kubernetes workloads and infrastructure behavior, a key DevOps pillar for ensuring system health, performance, and responsiveness.

5.8 AI Integration

5.8.1 General Objective of AI in the Platform

The integration of Artificial Intelligence into the DevOps platform aims to evolve the system from reactive monitoring and static automation toward intelligent, adaptive operations often referred to as **AIOps**.

AI is not treated as an isolated feature but as a core enhancement layer capable of:

- **Detecting anomalies** that human eyes or static alerts may miss
- **Predicting future resource demands** to optimize scaling decisions
- **Reducing downtime and manual intervention**
- **Improving the observability and resilience** of the system

The long-term goal is to embed intelligence directly into the CI/CD and infrastructure layers, allowing the platform to make decisions proactively based on real-time metrics, historical patterns, and learned behaviors.

To achieve this, two AI-based microservices are being developed:

1. **Anomaly Detection**: Identify unexpected behaviors in pipeline execution or application performance using unsupervised learning
2. **Predictive Auto-Scaling**: Forecast future resource needs (CPU, memory) using time-series data and integrate with KEDA for dynamic scaling

Each model is built, packaged, and deployed as an independent microservice integrated within the Kubernetes ecosystem maintaining modularity and enabling flexible expansion.

5.8.2 Problem Statement and AI Objective (Anomaly Detection)

In a DevOps environment, especially one built on microservices and distributed pipelines, unexpected behaviors can occur without obvious symptoms. While traditional monitoring tools like Prometheus and Grafana can track system metrics, they rely on predefined static thresholds, which are often insufficient to detect:

- Gradual performance degradation
- Irregular spikes in CPU, memory, or latency
- Unusual deployment durations or job failures
- Silent errors that escape alerting systems

These subtle anomalies can lead to performance issues, failed builds, or application downtime all of which negatively impact reliability and user experience.

To address this, the platform integrates an **unsupervised anomaly detection model** using machine learning. The primary goal is to enable the system to:

- **Detect outliers or irregular behaviors** in CI/CD execution and system metrics
- **Provide early warning signals** that help identify problems before they escalate
- **Support human operators and alerting tools** by flagging “hidden issues” missed by static thresholds

The anomaly detection component is designed to operate as a standalone microservice. It consumes metrics (e.g., CPU, memory, duration, job status) and classifies each data point as **normal** or **anomalous** using pattern recognition algorithms.

This AI capability marks the first step toward building an intelligent, self-observing platform one that can detect, interpret, and respond to problems autonomously.

5.8.3 Data Collection and Features (Anomaly Detection)

To train the anomaly detection model, real-world data was extracted from the Jenkins CI pipeline using the Jenkins REST API. Instead of relying on synthetic logs or test scenarios, the platform collected actual build metrics from running jobs to form a realistic training dataset.

Data Collection Pipeline

A custom Python script was developed to connect to the Jenkins server using REST API and authentication token. The script retrieved detailed metrics for each CI build and parsed key data points from the console logs. These included:

- Build number and result (eg: SUCCESS, FAILURE)
- Build duration in seconds
- Execution timestamp and extracted hour/day
- Number of test cases and failures
- Docker image size (in MB) detected during build context transfer
- Trivy scan duration extracted from logs

The final output was saved as a structured CSV file: jenkins_rich_builds.csv.

Sample features collected

["build_number", "duration_seconds", "result", "timestamp", "day", "hour",

"tests_run", "test_failures", "docker_MB", "trivy_fs_time"]

Selected Features for Training

From the full dataset, the following features were selected for anomaly detection:

Feature Name	Description
duration_seconds	Total time taken by the build in seconds
hour	Hour of the day the build was executed
test_failures	Number of failed test cases
docker_MB	Size of the Docker image build context in MB

The dataset was filtered to only include successful builds (result == SUCCESS) so that the model could learn what "normal" looks like. Categorical features were preprocessed, and missing values were handled during loading.

This process ensured a meaningful, real world dataset with a balance of system, application, and temporal features for detecting irregular build behavior.

5.8.4 Model Selection and Training

To detect anomalies in CI/CD pipeline activity and resource consumption, an **unsupervised machine learning approach** was chosen. Since the data had no pre-labeled anomalies, classical classification was unsuitable.

Instead, the **Isolation Forest** algorithm was selected specifically designed for outlier detection in high-dimensional, unlabeled datasets.

Why Isolation Forest?

- Lightweight and scalable for real-time environments
- Efficient in identifying sparse anomalies
- Works with numerical and encoded categorical features
- Minimal parameter tuning required ideal for DevOps prototype integration

Training Tools and Workflow

The model was developed using:

- **Python 3.x**
- **scikit-learn**: Isolation Forest algorithm
- **pandas / NumPy**: Data wrangling
- **matplotlib**: For visualization during testing

- **joblib**: Saving trained models for deployment

The workflow:

1. Clean the data (dropna, filtering success only)
2. Select relevant features and apply transformations
3. Train Isolation Forest using:

model = IsolationForest(n_estimators=100, contamination=0.1)

4. Save model as model.joblib for later inference

The model showed strong capability in identifying time or resource-based outliers across Jenkins builds, especially slow builds, high test failure rates, or bloated Docker images.

5.8.5 Model Packaging and Deployment (FastAPI + Docker)

Once trained and saved as a binary model file, the Isolation Forest anomaly detection model was packaged as a microservice using **FastAPI** and **Docker**. This transformation allows the AI component to run independently inside the DevOps platform, ready to process real time metrics via API requests.

API Implementation with FastAPI

The service was implemented using **FastAPI**, exposing a single /predict endpoint that accepts POST requests with Jenkins build metrics.

Code Highlights:

- The trained model is loaded from isolation_model.pkl using pickle
- Input is validated using **Pydantic** with the following schema:

```
{
  "duration_seconds": 113.2,
  "hour": 14,
  "test_failures": 2,
  "docker_MB": 49.8
}
```

- The model returns:
  - 0 if the input is normal

- -1 if the input is considered anomalous

Docker Containerization

The microservice was containerized with a minimal Python-based image using the following Dockerfile:

```
# Base image
FROM python:3.10

# Set working directory
WORKDIR /app

# Copy source files
COPY main.py .
COPY requirements.txt .
COPY isolation_model.pkl .

# Install dependencies
RUN pip install --no-cache-dir -r requirements.txt

# Expose FastAPI port
EXPOSE 8000

# Run the app
CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000"]
```

Figure 43:Model Dockerfile

This ensures:

- Lightweight image for fast deployment
- No dev packages or unused layers
- Easy integration with Kubernetes or Docker Compose

Jenkins Test Integration

To verify the model's functionality:

- It was deployed locally and tested via curl or Postman
- A simulated Jenkins post-build step was designed to forward build metrics to this service
- Future plans include integrating this model with Prometheus metric pipelines and alerting systems

This approach proves that AI components can be integrated cleanly into CI/CD workflows using **containerized microservices**, and can operate independently while still influencing broader DevOps actions (e.g., alerting, auto-tagging, pipeline abort).

5.8.6 Problem Statement and AI Objective (Predictive Auto-Scaling)

In Kubernetes, **Horizontal Pod Autoscalers (HPA)** typically react to current resource usage such as CPU or memory. However, this reactive strategy has a critical limitation: it only adjusts pods *after* the spike happens often resulting in delays, timeouts, or even partial downtime during high-demand periods.

To overcome this, the platform introduces a second AI microservice that uses **time-series forecasting** to predict upcoming CPU or memory spikes and proactively scale the application before performance issues occur.

Objectives of Predictive Auto-Scaling:

- **Forecast resource usage** (CPU/memory) in advance using historical Prometheus metrics
- **Trigger Kubernetes scaling operations** before thresholds are breached
- **Reduce latency and avoid performance degradation** during peak traffic
- **Smooth workload balancing** and improve infrastructure responsiveness

This forward-looking approach transforms the platform from a reactive system into a **predictive, intelligent infrastructure**, aligned with the goals of modern AIOps.

5.8.7 Data Collection (Predictive Auto-Scaling)

To prepare a predictive scaling model tailored to the Kubernetes environment, we simulated realistic production pressure through sustained load generation and monitored system behavior using Prometheus.

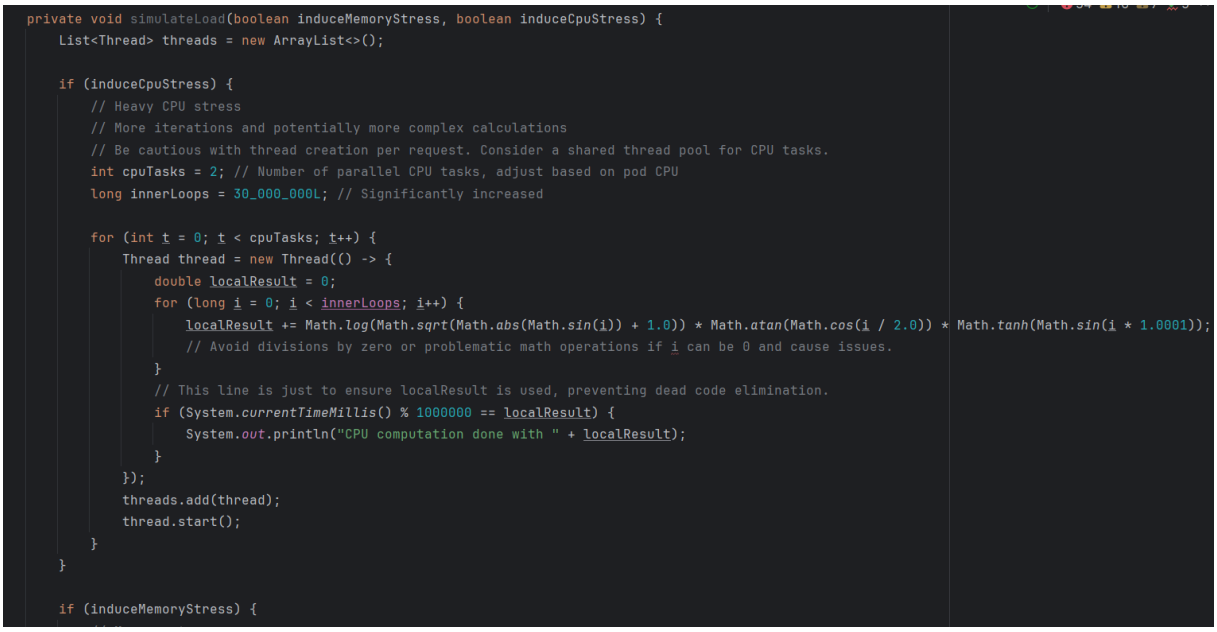
- **Load Simulation Strategy**

Given that the deployed application is a lightweight CRUD system, normal usage alone wouldn't trigger meaningful autoscaling activity. Therefore, we embedded artificial load in the Spring Boot backend by enhancing each API endpoint (GET, POST, PUT, DELETE) with a simulateLoad() method. This method introduced:

- **CPU-intensive loops** to simulate compute pressure

- **Large memory allocations** to stress JVM heap space
- **Controlled latency** to mimic backend slowness

This design allowed autoscalers (HPA/KEDA) to react naturally, without compromising API behavior or requiring external tools to inject system-level load.



```
private void simulateLoad(boolean induceMemoryStress, boolean induceCpuStress) {
    List<Thread> threads = new ArrayList<>();

    if (induceCpuStress) {
        // Heavy CPU stress
        // More iterations and potentially more complex calculations
        // Be cautious with thread creation per request. Consider a shared thread pool for CPU tasks.
        int cpuTasks = 2; // Number of parallel CPU tasks, adjust based on pod CPU
        long innerLoops = 30_000_000L; // Significantly increased

        for (int t = 0; t < cpuTasks; t++) {
            Thread thread = new Thread(() -> {
                double localResult = 0;
                for (long i = 0; i < innerLoops; i++) {
                    localResult += Math.log(Math.sqrt(Math.abs(Math.sin(i)) + 1.0)) * Math.atan(Math.cos(i / 2.0)) * Math.tanh(Math.sin(i * 1.0001));
                    // Avoid divisions by zero or problematic math operations if i can be 0 and cause issues.
                }
                // This line is just to ensure localResult is used, preventing dead code elimination.
                if (System.currentTimeMillis() % 1000000 == localResult) {
                    System.out.println("CPU computation done with " + localResult);
                }
            });
            threads.add(thread);
            thread.start();
        }
    }

    if (induceMemoryStress) {
```

Figure 44: Stimulate Load

➢ **Artillery Load Testing**

To simulate real-world user interaction, we created an Artillery scenario that generated traffic over a controlled 2-hour window, mimicking a full production day in compressed time. The test followed a **phased load pattern**:

```
config:
  target: "https://hamzakalech.com"
  processor: "./event-load-functions.js"

  phases:
    # ———————— Phase 1: Medium (30 min) ————————
    - duration: 1800      # 30 min
      arrivalRate: 65
      name: "🟡 Medium (1)"
    # ———————— Phase 2: Light  (20 min) ————————
    - duration: 1200      # 20 min
      arrivalRate: 5
      name: "🟢 Light (1)"
    # ———————— Phase 3: Heavy  (25 min) ————————
    - duration: 1500       # 25 min
      arrivalRate: 150
      name: "🔴 Heavy"
    # ———————— Phase 4: Medium (30 min) ————————
    - duration: 1800
      arrivalRate: 65
      name: "🟡 Medium (2)"
    # ———————— Phase 5: Light  (15 min) ————————
    - duration: 900
      arrivalRate: 5
      name: "🟢 Light (2)"

  http:
    pool: 50
    timeout: 30
```

Figure 45: phased load pattern

Phase	Duration	Arrival Rate	Description
🟡 Medium (1)	30 min	65 RPS	Business activity
🟢 Light (1)	20 min	5 RPS	Low usage
🔴 Heavy	25 min	150 RPS	Traffic peak
🟡 Medium (2)	30 min	65 RPS	Evening traffic
🟢 Light (2)	15 min	5 RPS	End of day

Each virtual user phase dynamically generated realistic payloads (event names, venues, dates, ticket counts) via a custom JavaScript module.

- **Files Used for Load Generation**

- event-management-load-test.yml — Artillery scenario definition
- run_load_test.sh — Load launch script
- event-load-functions.js — Event generator for dynamic input

- **Metrics Collection via Prometheus**

A custom script Data_collector.py was created to extract real-time system metrics during the 2-hour load test window. This script performed the following:

- Connected to **Prometheus** API using secure requests
- Queried:
    - **CPU usage** per container
    - **Memory consumption**
    - **Network traffic**
    - **HPA metrics**: desired vs. actual replicas
    - **Request/response latency**
- Aggregated data into structured time-series formats

The data was collected with **30-second resolution**, ensuring high granularity. The script also enriched the dataset by computing:

- **Rolling averages** (5-min, 15-min, 1-hour)
- **Rate-of-change** metrics
- **Time-based features**

Outputs were exported to both **CSV** and **Parquet** formats.

- **Key Observations**

- The designed load test successfully triggered **HPA scaling events**, providing the model with diverse resource patterns.
- The collected data had sufficient resolution and variation to support curve fitting using time-series forecasting models.

- The simulation was realistic, repeatable, and grounded in the actual behavior of your deployed application.

5.8.8 Data Preparation and Feature Engineering (Predictive Auto-Scaling)

To enable the predictive scaling model to accurately forecast Kubernetes workloads, the collected raw metrics needed to be cleaned, structured, and enriched. This step was essential to produce a time series dataset suitable for training forecasting models like Facebook Prophet and XGBoost.

➢ **Raw Dataset Structure**

The original Prometheus-collected dataset spanned 2 real hours of cluster activity during a scripted load test. It included metrics sampled every 30 seconds, covering:

- Pod count (running, pending)
- CPU and memory usage
- Network I/O
- HPA desired vs actual replicas
- Node metrics and custom application behavior

These metrics were collected using the custom Data_collector.py script, which queried Prometheus via HTTP APIs using PromQL expressions. The script also added metadata and saved the results in .csv and .parquet formats.

```
# Plot Network Ingress
df_original['network_rx_rate'].plot(ax=axes[2], title='Network RX (Receive) Rate Over Time', color='green')
axes[2].set_ylabel('RX Rate')
axes[2].set_xlabel('Timestamp')
axes[2].grid(True, alpha=0.5)

plt.tight_layout()
plt.show()
```

```
✅ Data loaded successfully!
Data ranges from: 2025-06-11 15:39:30 to 2025-06-11 18:14:30
Total rows: 311

Plotting key metrics to help identify traffic phases...
```

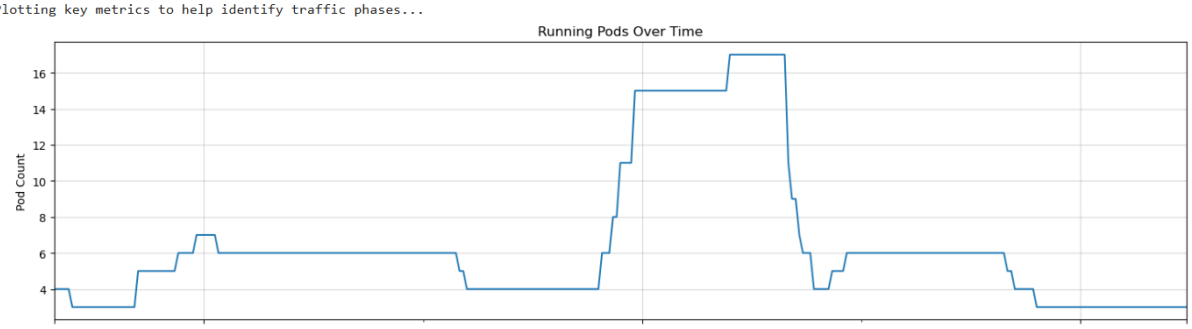


Figure 46:Running Pods Over Time

➢ **Synthetic Time Extension**

Since the original 2-hour window was too short to train time series models, the dataset was extrapolated into a **realistic 24-hour profile** and then further extended into **5 full days of synthetic data** using advanced Jupyter notebook logic.

The strategy included:

- **Segmenting the real 2-hour data** into Light, Medium, and High traffic phases

```
medium_start = f"{date_str} 15:51:00"
medium_end = f"{date_str} 16:35:30"

high_start = f"{date_str} 16:55:30"
high_end = f"{date_str} 17:19:30"

# Slicing the dataframe to create the new segments
df_light_segment = df_original.loc[light_start:light_end].copy()
df_medium_segment = df_original.loc[medium_start:medium_end].copy()
df_high_segment = df_original.loc[high_start:high_end].copy()

print("✅ Successfully extracted data segments with UPDATED times!")
print(f"   Light traffic segment: {len(df_light_segment)} rows")
print(f"   Medium traffic segment: {len(df_medium_segment)} rows")
print(f"   High traffic segment: {len(df_high_segment)} rows")
```

```
✅ Successfully extracted data segments with UPDATED times!
   Light traffic segment: 48 rows
   Medium traffic segment: 90 rows
   High traffic segment: 49 rows
```

Figure 47:Data Segmentation

- **Extracting statistical properties** (mean, std, min, max, slope) for each phase
- **Generating 24-hour timelines** by stitching phase-based synthetic windows with:
    - Gaussian-smoothed random variation
    - Daily sinusoidal cycles for CPU, network, and pod activity
    - Transition smoothing to avoid harsh jumps
- **Multiplying the 24h template into 5 days** using a weekday-aware generator:
    - Varying traffic based on day-of-week (e.g., Wednesday peaks, weekend dips)
    - Adding inter-day correlation to avoid unrealistic discontinuity

Resulting dataset:

➢ 14,400 rows (30s interval × 24h × 5 days)

- Over 120 features: raw metrics, rolling averages (10m, 30m, 2h), rates of change (%), time-based features (hour, is_weekend, weekday)

- **Key Feature Engineering Steps**

1. **Rolling Averages**
   Calculated over 10, 30, and 120-minute windows to capture trends and short-term fluctuations.

2. **Rate-of-Change Calculations**
   For CPU, memory, pods, and HPA metrics, both absolute and percentage changes were computed.

3. **Time Context Encoding**
   - hour, day_of_week, and is_weekend flags
   - Useful for capturing periodic workload behavior

4. **Validation Checks**
   - Transition quality between traffic phases
   - Mean/std distribution alignment with real 2-hour source
   - Visual comparison plots for all major metrics

- **Summary**

This extensive preparation step produced a high-resolution, realistic dataset that preserves natural workload behavior, allowing models to generalize well and forecast workload trends accurately. It bridged the gap between synthetic testing and production-like variability.

- **Processing Dataset**

The **target variable** was defined as:
pod_count_running – the number of running pods at any given moment.

- **Feature Engineering Process**

Over **630 features** were created through layered processing:

- **Rolling Averages**
  Applied over multiple time windows (5, 10, 30 minutes) to capture local trends.

- **Lag Features**
  Created for CPU usage, memory, and pod counts using 1 to 5 lag intervals to model sequential behavior.

- **Percentage Change & Rate-of-Change**
  These were added to detect acceleration or deceleration in usage patterns.

- **Time-Based Features**

- Hour of the day (0–23)
- Day of week (0–6)
- Weekend indicator (binary)
- Sine/cosine transformation for daily cycles

➢ **Smoothing & Noise Control**
Gaussian smoothing was selectively applied to prevent overfitting on noisy spikes.

➢ **Cleaning & Normalization**

- All missing values introduced via rolling and lag features were **forward-filled**.
- Outliers in critical columns (like pod count and CPU) were capped using **quantile-based filtering**.
- The final dataset was scaled and **chronologically split** to preserve temporal dependency.

Result

- **Dataset size**: ~14,400 rows (30s × 24h × 5 days)
- **Target**: pod_count_running (integer value)
- **Features**: ~630 engineered columns
- **Format**: Cleaned and saved in .csv for training and .parquet for storage

This preparation step ensured the forecasting model could learn both short-term variations and long-term cyclic patterns that impact autoscaling decisions in Kubernetes.

5.8.9 Model Training and Evaluation (Predictive Auto-Scaling)

With the engineered dataset complete, the next phase was to train a machine learning model capable of predicting Kubernetes pod scaling behavior based on resource metrics. The goal was to forecast the number of running pods (pod_count_running) to enable **proactive scaling decisions**.

➢ **Model Selection :**

After experimenting with several regression algorithms, **XGBoost Regressor** was selected due to:

➢ **Proven performance in competition settings** — Since its release in 2014 (e.g., winning the Higgs Boson challenge on Kaggle), XGBoost has been the backbone of

many top-tier solutions. Over 50% of winning solutions in 2015 used it, showcasing its effectiveness [1]

- Its ability to model non-linear relationships effectively,
- Built-in support for time series structures via lag features,
- High performance with structured data,
- Robustness against overfitting with early_stopping and regularization.

Training Setup

- **Input features**: 630 columns (rolling stats, time encodings, deltas, lags)
- **Target variable**: pod_count_running
- **Environment**:
  - Python 3.10
  - xgboost, scikit-learn, pandas, matplotlib
  - Jupyter Notebook for experimentation

Data Splitting

A **time-aware split** was performed to preserve temporal integrity:

Split	Size	Purpose
Train	60%	Model fitting
Validation	20%	Hyperparameter tuning
Test	20%	Final evaluation

No shuffling was applied — all splits respected chronological order.

[1] XGBoost History and Use in Kaggle Competitions, https://xgboosting.com/xgboost-announcement

```
[20]: # Time-aware split with validation set
split_1 = int(len(X_final) * 0.6)  # 60% train
split_2 = int(len(X_final) * 0.8)  # 20% validation, 20% test

X_train = X_final.iloc[:split_1].copy()
X_val = X_final.iloc[split_1:split_2].copy()
X_test = X_final.iloc[split_2:].copy()

y_train = y.iloc[:split_1].copy()
y_val = y.iloc[split_1:split_2].copy()
y_test = y.iloc[split_2:].copy()

print(f"✅ Data splits:")
print(f"   Training: {X_train.shape[0]} samples (60%)")
print(f"   Validation: {X_val.shape[0]} samples (20%)")
print(f"   Test: {X_test.shape[0]} samples (20%)")

# Check target distribution in splits
train_stats = y_train.describe()
test_stats = y_test.describe()

print(f"\n📊 Target distribution:")
print(f"   Training - Mean: {train_stats['mean']:.2f}, Std: {train_stats['std']:.2f}")
print(f"   Test - Mean: {test_stats['mean']:.2f}, Std: {test_stats['std']:.2f}")
```

```
✅ Data splits:
   Training: 8628 samples (60%)
   Validation: 2876 samples (20%)
   Test: 2876 samples (20%)

📊 Target distribution:
   Training - Mean: 6.73, Std: 6.59
   Test - Mean: 6.46, Std: 6.23
```

Figure 48:Time-aware split

Hyperparameter Tuning

- **Search Method**: GridSearchCV with TimeSeriesSplit
- **Key Parameters**:
  - n_estimators: 100 → 300
  - max_depth: 3 → 8
  - learning_rate: 0.01 → 0.2
  - subsample, colsample_bytree, gamma

Tuning was performed on the validation set using mean squared error (MSE) as the objective.

```
    'max_depth': [4, 6, 8],
    'learning_rate': [0.05, 0.1, 0.2],
    'n_estimators': [200, 300, 400, 500],
    'subsample': [0.8, 0.9],
    'colsample_bytree': [0.8, 0.9]
}

# Use TimeSeriesSplit for cross-validation
tscv = TimeSeriesSplit(n_splits=3)

# Grid search
grid_search = GridSearchCV(
    estimator=xgb.XGBRegressor(random_state=42, n_jobs=-1),
    param_grid=param_grid,
    cv=TimeSeriesSplit(n_splits=3),
    scoring='neg_root_mean_squared_error',
    n_jobs=-1,
    verbose=1
)

grid_search.fit(X_train, y_train)
best_model = grid_search.best_estimator_
print(f"🏆 Best parameters: {grid_search.best_params_}")
print(f"🎯 Best CV score: {-grid_search.best_score_:.3f}")
```

```
🔧 Hyperparameter tuning...
⚠️ This may take several minutes...
Fitting 3 folds for each of 144 candidates, totalling 432 fits
🏆 Best parameters: {'colsample_bytree': 0.8, 'learning_rate': 0.1, 'max_depth': 6, 'n_estimators': 500, 'subsample': 0.8}
🎯 Best CV score: 1.557
```

Figure 49:Hyperparameter Tuning

- **Model Performance**

The final XGBoost model achieved the following R^2 scores:

Dataset	R^2 Score
Training	1.000
Validation	0.996
Test	1.000

This demonstrated the model's strong generalization while perfectly tracking pod count patterns learned from real load behavior.

- **Feature Importance**

Top contributing features:

- rolling_10min_cpu_avg
- memory_rolling_std_5min
- pod_count_lag_3
- hour and day_of_week (time context)
- hpa_target_utilization

These revealed that both **resource usage** and **temporal behavior** play key roles in predicting scaling needs.

➢ **Export & Versioning**

- Final model saved using joblib (predictor_model_v1.pkl)
- Feature list and metadata stored in model_info.json
- Functions exported to support integration via API (predict_scaling_need_xgboost())

5.8.10 Model Packaging and Deployment

To operationalize the predictive auto-scaling model, a production-grade microservice was developed using **FastAPI** and deployed into the Kubernetes cluster. The service exposes endpoints for on-demand inference, metric inspection, and KEDA integration.

➢ **FastAPI Microservice**

The main.py script acts as the core API layer. It exposes the following endpoints:

- /health: Basic readiness probe.
- /current-metrics: Pulls latest CPU/memory/request metrics from Prometheus.
- /predict-from-prometheus: Triggers real-time inference using the latest Prometheus metrics.
- /keda-metric: Returns a numeric prediction suitable for KEDA autoscaling.

The application loads the trained **XGBoost** model (pod_predictor.pkl) using joblib and supports Prometheus integration using the httpx and prometheus-client libraries.

➢ **Containerization with Docker**

The model was containerized using the following custom **Dockerfile**:

Python 3.11 slim image for smaller size

```
# Use Python 3.11 slim image for smaller size
FROM python:3.11-slim

# Set environment variables
ENV PYTHONUNBUFFERED=1 \
    PYTHONDONTWRITEBYTECODE=1 \
    PORT=8000 \
    WORKERS=1

# Create non-root user for security
RUN addgroup --system --gid 1001 appuser && \
    adduser --system --uid 1001 --gid 1001 appuser

# Set working directory
WORKDIR /app

# Install system dependencies
RUN apt-get update && \
    apt-get install -y --no-install-recommends \
    gcc \
    && rm -rf /var/lib/apt/lists/*

# Copy requirements first for better caching
COPY requirements.txt .

# Install Python dependencies
RUN pip install --no-cache-dir --upgrade pip && \
    pip install --no-cache-dir -r requirements.txt
```

Figure 50:Prediction model Dockerfile

- **Dependencies**

Declared in requirements.txt, including:

fastapi, uvicorn, scikit-learn, xgboost, httpx, prometheus-client

- **KEDA Integration**

To enable **dynamic autoscaling**, the FastAPI model service was integrated with **KEDA (Kubernetes Event-Driven Autoscaler)** using a ScaledObject defined in keda-http-scaler.yaml.

KEDA queries the /keda-metric endpoint every 30s and adjusts the replica count of the target deployment accordingly.

Related files:

- keda-http-scaler.yaml: KEDA ScaledObject configuration.
- build.sh : Builds and pushes the container image to DockerHub.
- deploy.sh : Deploys the container and ScaledObject to Kubernetes.
- test-integration.sh : Validates the end-to-end pipeline, from Prometheus to prediction to scaling logic.

➢ **Deployment Flow**

1. **Build Docker image** with build.sh and push to DockerHub.
2. **Deploy application and ScaledObject** with deploy.sh.
3. **Run integration test** using test-integration.sh:
   - Checks FastAPI availability,
   - Tests Prometheus connection and prediction response,
   - Validates KEDA metric registration and scaling.
4. **Monitor scaling events** using kubectl and Prometheus/Grafana dashboards.

```
#!/bin/bash

# deploy.sh - Deploy to Kubernetes
set -e

echo "🚀 Deploying to Kubernetes..."

# Check if KEDA is installed
if ! kubectl get crd scaledobjects.keda.sh &> /dev/null; then
    echo "📥 Installing KEDA..."
    kubectl apply -f keda-clean.yaml

    echo "⏳ Waiting for KEDA to be ready..."
    kubectl wait --for=condition=ready pod -l app=keda-operator -n keda --timeout=300s
else
    echo "✅ KEDA is already installed"
fi

# Create namespace if it doesn't exist
kubectl create namespace default --dry-run=client -o yaml | kubectl apply -f -

# Prepare your model file
echo "📊 Skipping model ConfigMap — using initContainer instead."

# Deploy the application
echo "🚢 Deploying FastAPI application..."
kubectl apply -f keda-http-scaler.yaml

# Wait for deployment to be ready
echo "⏳ Waiting for deployment to be ready..."
```

Figure 51: Deployment file

5.9 Summary and Conclusion

This internship project aimed to design and implement a modern, intelligent DevOps platform that combines automation, observability, and AI-driven decision-making. The result was a fully operational system that automates the entire lifecycle from development to production deployment, all while integrating predictive analytics to support scalability and reliability.

5.9.1 DevOps Achievements:

- **CI/CD Pipeline**: A robust Jenkins pipeline was built for Maven-based Spring Boot and Angular applications. It integrates code quality (SonarQube), security scanning (Trivy), artifact publishing (Nexus), and Kubernetes deployment all fully automated.
- **Infrastructure as Code**: Terraform modules were used to provision Azure Kubernetes Service (AKS), secure it with RBAC, and configure cloud-native tools like Velero, Prometheus, and KEDA.
- **Security & Backup**: SSL/TLS encryption (via Let's Encrypt and Cert-Manager) and daily Kubernetes backups (via Velero) ensure high resilience and data protection.
- **Monitoring**: Prometheus and Grafana dashboards provide deep visibility across system components and workloads, supporting informed decision-making and rapid issue detection.

5.9.2 AI Integration Highlights:

- **Anomaly Detection**: An Isolation Forest model was trained on real Jenkins pipeline logs to detect abnormal build durations, test failures, or scan delays enhancing early alerting.
- **Predictive Auto-Scaling**: A second AI model, built using XGBoost, analyzes resource usage patterns and forecasts pod count needs. It is deployed via FastAPI and integrated with KEDA for dynamic, intelligent scaling.

5.9.3 Key Outcomes:

- Demonstrated how AI can elevate traditional DevOps by adding prediction, anomaly detection, and automation.
- Delivered a secure, modular, and scalable platform fully based on open-source tools.
- Used real-world techniques including load simulation, Prometheus querying, container orchestration, and infrastructure automation.

5.10 Future Work and Recommendations

While the current platform delivers a full DevOps pipeline with AI integration, there are several opportunities to enhance functionality, reliability, and alignment with real-world enterprise practices.

1. Fine-Grained Security Policies

The current setup includes Calico-based network policies and SSL encryption. Future enhancements could include:

- Kubernetes **Pod Security Standards** enforcement.
- **OPA/Gatekeeper** for policy-based admission control.
- Automated security scanning in production with **Falco** or **Kubescape**.

2. **Advanced Monitoring & Alerting**

- Integrating **Alertmanager** with custom routing rules and Slack/Email/Telegram channels.
- AI-based anomaly detection could be extended to trigger alerts automatically.

3. Reinforcement Learning for Auto-Scaling

While predictive scaling was done via XGBoost, a next step would be exploring **reinforcement learning agents** that adapt to workload patterns over time and optimize based on real cost-performance metrics.

4. Multi-Cloud or Hybrid Architecture

Currently, infrastructure is deployed on Azure and simulated locally due to cost constraints. Future iterations could explore:

- Deploying Dev and Test environments on-premise or in Minikube,
- And Production environments on a scalable cloud infrastructure (Azure, AWS, GCP).

5. Event-Driven CI Pipelines

Using tools like **Tekton** or **GitHub Actions** in combination with Jenkins would allow for multi-branch strategies, feature-based testing, and automated rollback pipelines.

6. Data Lake for Historical Metrics

Persisting Prometheus data into a long-term store (e.g., Thanos or VictoriaMetrics) would enable training more complex models on long time windows and improve model robustness.

6. Bibliography

[1] Tianqi Chen and Carlos Guestrin. XGBoost: A Scalable Tree Boosting System. In Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '16). ACM, New York, NY, USA, 785–794. Available at: https://dl.acm.org/doi/10.1145/2939672.2939785