Notes API – DevOps & Cloud-Native Project Report

1. Introduction

1.1 Project Context

This project demonstrates a complete DevOps and Cloud-Native workflow applied to a simple backend application called Notes API. The objective is not the business logic itself, but the automation, security, observability, and Kubernetes deployment around the application. The project follows industry best practices including CI/CD pipelines, GitOps deployment, monitoring, logging, security scanning, and automated testing with coverage.

1.2 Project Goals

The main goals of this project are to build a REST API with Node.js and Express, containerize the application using Docker, deploy the application on Kubernetes using Helm, automate deployments using Argo CD (GitOps), implement full observability with metrics and logs, secure the pipeline with SAST and DAST scans, implement automated testing with coverage reports, and provide a production-like DevOps workflow.

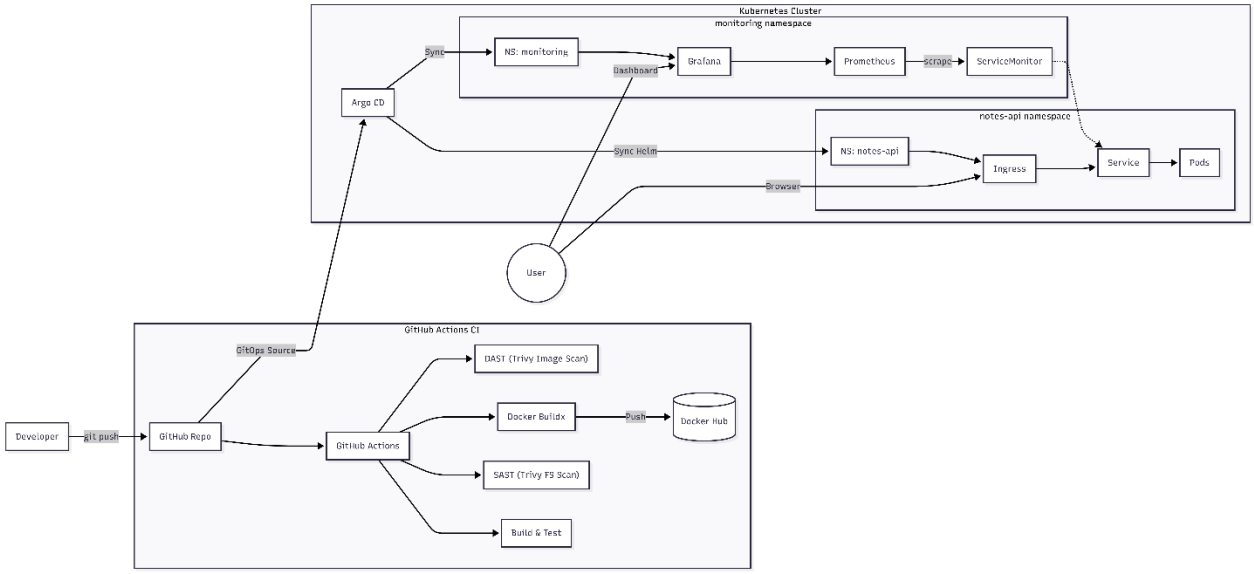
1.3 Technologies Used

The project utilizes Node.js and Express for the backend, Docker for containerization, Kubernetes with Minikube for orchestration, Helm for packaging, Argo CD for GitOps, GitHub Actions for CI/CD, Prometheus for monitoring, Grafana for visualization, Trivy for security scanning, Jest and Supertest for testing, prom-client for metrics, and structured JSON logs for logging.

2. Architecture Overview

2.1 High-Level Architecture

The system follows a cloud-native microservice architecture. Developers push code to GitHub, which triggers GitHub Actions CI/CD pipelines. The application is tested, scanned, built, and pushed as a Docker image. Argo CD automatically syncs the Kubernetes cluster with the desired state from Git. Kubernetes exposes the API through Ingress, while Prometheus scrapes metrics and Grafana visualizes them in dashboards.



2.2 Components Description

The Notes API provides RESTful CRUD operations on notes. The Docker image serves as an immutable artifact deployed across environments. The Kubernetes Deployment ensures pod lifecycle management, while Service and Ingress expose the API internally and externally. Prometheus collects application and system metrics, and Grafana displays dashboards for observability. Argo CD handles GitOps-based continuous deployment.

3. Kubernetes Setup

3.1 Cluster Creation

The Kubernetes cluster is created using Minikube, simulating a production-like environment locally.

	Name	Container ID	Image	Port(s)	CPU (%)	Last st	Actions
☐	minikube	28b03f137f89	k8s-minikul		0%	2 days	
☐	cool_blackburn	30937729a267	notes-api:v	3000:3000	0%	2 mont	

☐	notes-api	v1	0d1772bb0a2c	2 months ago	206.31 MB	
☐	nouromry/notes-api	v1	0d1772bb0a2c	2 months ago	206.31 MB	

3.2 Deployments

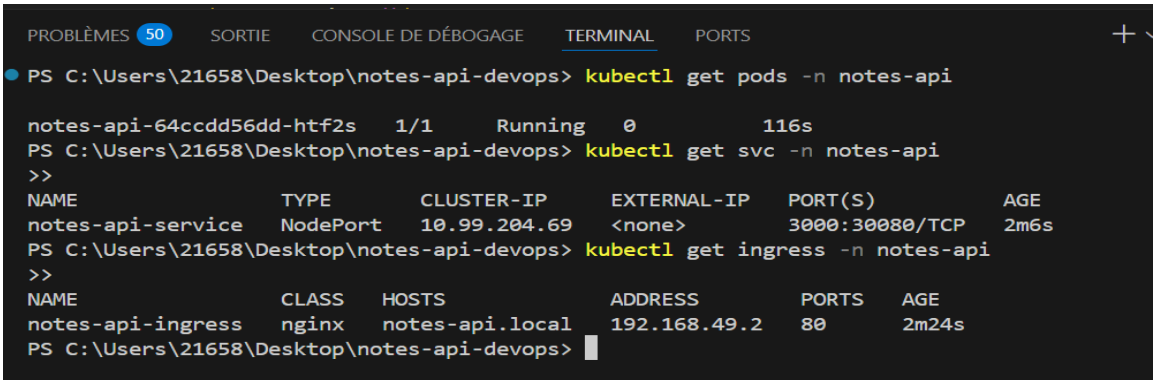
The application is deployed using a Kubernetes Deployment managed by Helm. The deployment defines replica counts, uses resource requests and limits, includes readiness and liveness probes, and runs as a non-root container for security.

```
PROBLÈMES 50   SORTIE   CONSOLE DE DÉBOGAGE   TERMINAL   PORTS
PS C:\Users\21658\Desktop\notes-api-devops> kubectl get pods -n notes-api

notes-api-64ccdd56dd-htf2s   1/1     Running   0          116s
```

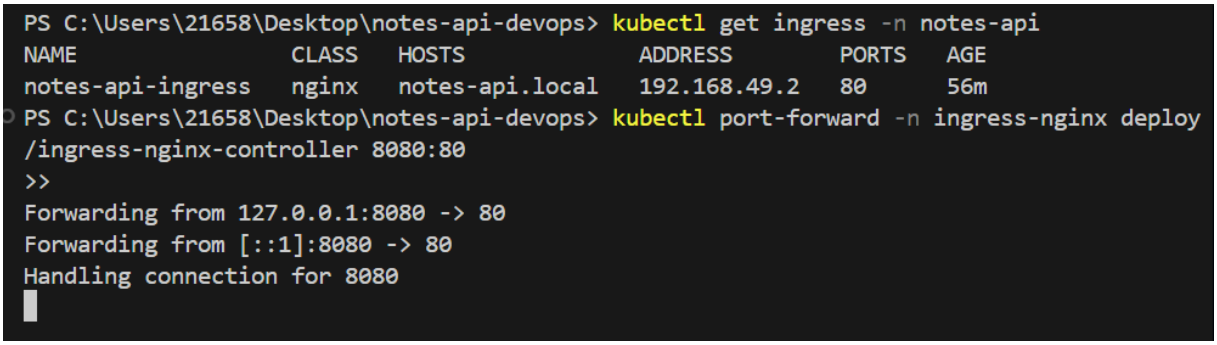
3.3 Services

Two services are used: a NodePort Service for external access and a ClusterIP Service for internal communication and monitoring.



3.4 Ingress

An NGINX Ingress Controller exposes the API via a defined hostname. The Ingress routes external traffic to the appropriate Kubernetes Service.



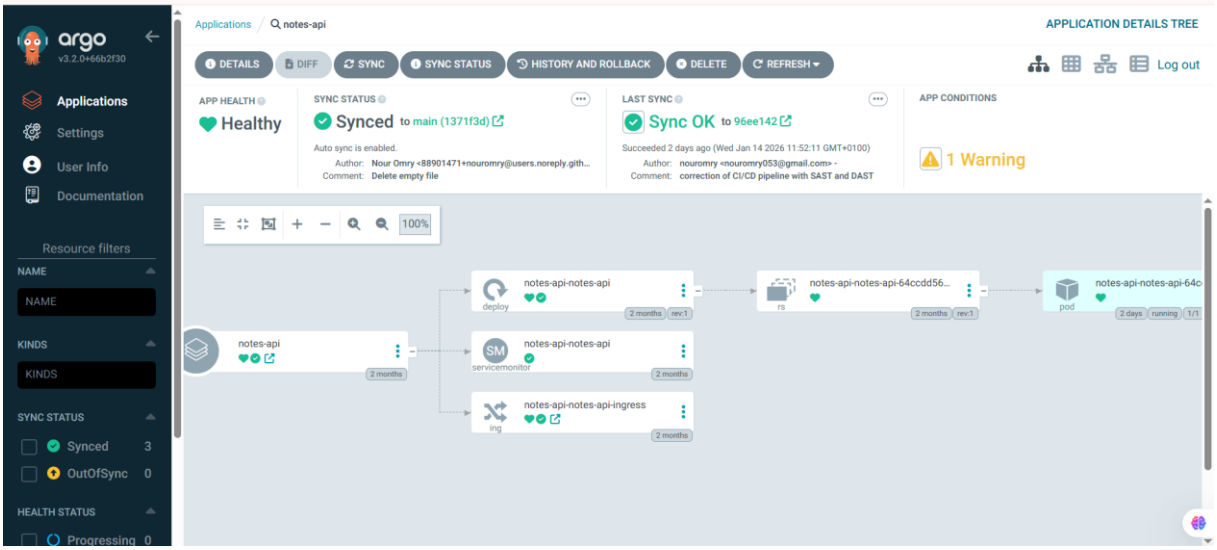
3.5 Helm Chart Explanation

The Helm chart contains deployment, service, and ingress configuration files, along with a values.yaml for environment configuration and helper templates for reusable Helm patterns. Helm allows for easy configuration, versioning, and reuse of deployment packages.

```
PS C:\Users\21658\Desktop\notes-api-devops> helm upgrade --install notes-
api helm/notes-api -n notes-api
>>
NAME: notes-api
LAST DEPLOYED: Fri Nov 21 22:08:05 2025
NAMESPACE: notes-api
STATUS: deployed
REVISION: 2
TEST SUITE: None
PS C:\Users\21658\Desktop\notes-api-devops> kubectl get pods -n notes-api
>> kubectl get svc -n notes-api
>> kubectl get ingress -n notes-api
NAME                          READY   STATUS              RESTARTS   AGE
notes-api-64ccdd56dd-htf2s    0/1     ContainerCreating   0          23s
NAME                 TYPE       CLUSTER-IP     EXTERNAL-IP   PORT(S)          AGE
notes-api-service    NodePort   10.99.204.69   <none>        3000:30080/TCP   23s
NAME                 CLASS   HOSTS              ADDRESS   PORTS   AGE
notes-api-ingress    nginx   notes-api.local              80      23s
PS C:\Users\21658\Desktop\notes-api-devops> minikube ip
>>
192.168.49.2
```

3.6 GitOps with Argo CD

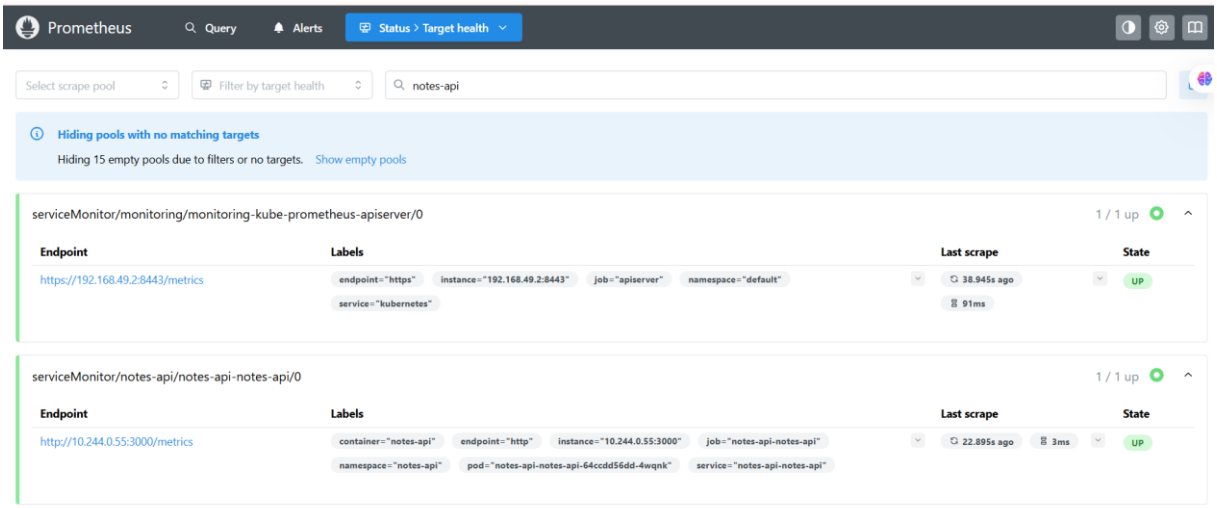
Argo CD continuously monitors the GitHub repository and ensures that the Kubernetes cluster state matches the Git state. This provides benefits including declarative deployments, automatic drift correction, clear deployment history, and UI-based visibility.



4. Observability Stack

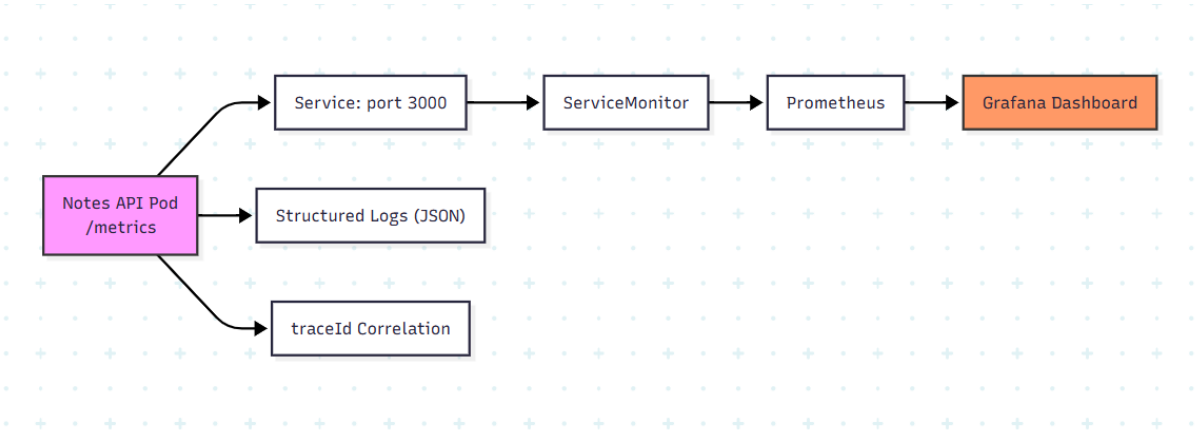
4.1 Prometheus Monitoring

The application exposes metrics via a dedicated endpoint using the prom-client library. Collected metrics include HTTP request counts, request rates, status codes, memory usage, CPU usage, and notes count.



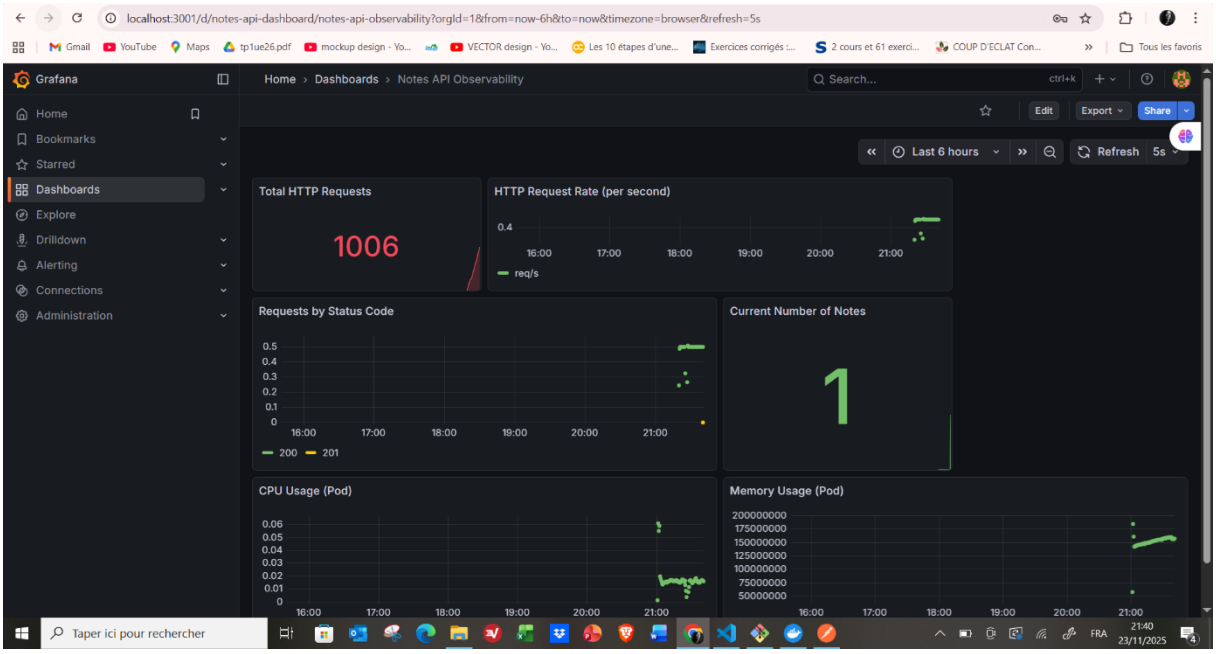
4.2 ServiceMonitor

A ServiceMonitor resource is deployed, allowing Prometheus to automatically discover and scrape the Notes API metrics endpoint. This integrates seamlessly with the Prometheus operator stack.



4.3 Grafana Dashboard

A custom Grafana dashboard visualizes key metrics including total HTTP requests, requests per second, requests by status code, number of notes, and CPU and memory usage per pod. The dashboard is configured to refresh at regular intervals.



4.4 Logs, Metrics & Tracing

The application implements structured JSON logs with timestamps and trace IDs for correlation. Metrics are exposed for Prometheus collection, and lightweight request tracing is implemented using unique identifiers. This combination provides comprehensive observability.

```
PROBLÈMES 50   SORTIE   CONSOLE DE DÉBOGAGE   TERMINAL   PORTS

PS C:\Users\21658\Desktop\notes-api-devops> kubectl logs -n notes-api dep
loy/notes-api
>>
6-bb487021f5b4","message":"Incoming GET /health"}
{"timestamp":"2025-11-21T21:16:42.929Z","traceId":"f58f3461-edd9-48c2-8c6
6-bb487021f5b4","message":"Completed 200 in 0ms"}
{"timestamp":"2025-11-21T21:16:45.492Z","traceId":"09251344-bdbd-49ea-98d
f-be7df19820b2","message":"Incoming GET /health"}
{"timestamp":"2025-11-21T21:16:45.493Z","traceId":"09251344-bdbd-49ea-98d
f-be7df19820b2","message":"Completed 200 in 1ms"}
f-c049331db0d3","message":"Incoming GET /health"}
{"timestamp":"2025-11-21T21:16:47.926Z","traceId":"49cd88a5-7578-483e-9ce
f-c049331db0d3","message":"Completed 200 in 0ms"}
{"timestamp":"2025-11-21T21:16:50.492Z","traceId":"350ff061-4744-411a-994
5-41f2f61d3831","message":"Incoming GET /health"}
{"timestamp":"2025-11-21T21:16:50.493Z","traceId":"350ff061-4744-411a-994
5-41f2f61d3831","message":"Completed 200 in 1ms"}
PS C:\Users\21658\Desktop\notes-api-devops> kubectl exec -n notes-api -it
 deploy/notes-api -- wget -qO- http://localhost:3000/health
>>
{"status":"healthy"}
PS C:\Users\21658\Desktop\notes-api-devops>
```

5. Security

5.1 SAST – Static Application Security Testing

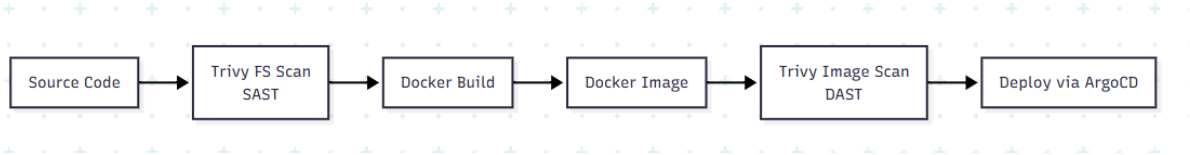
Trivy filesystem scans are executed during the CI pipeline to detect vulnerable dependencies, misconfigurations, and known CVEs in the source code before the build stage

5.2 DAST – Container Image Scanning

After building the Docker image, Trivy scans the final image to detect runtime vulnerabilities in the container layers and installed packages.

5.3 Why Security Matters

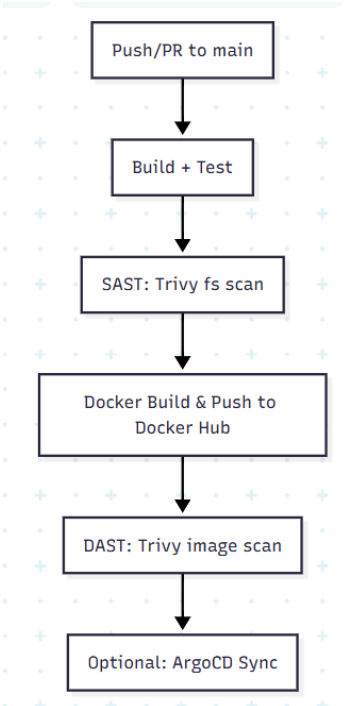
Security scanning ensures early detection of vulnerabilities, promotes secure container images, enforces compliance with DevSecOps practices, and reduces risks in production deployments.



6. CI/CD Pipelines

6.1 GitHub Actions Workflow

The pipeline includes multiple automated jobs: installing dependencies, running Jest tests, generating a coverage report, running Trivy filesystem scans, building the Docker image, pushing the image to a registry, and scanning the built image with Trivy.



6.2 Testing Flow

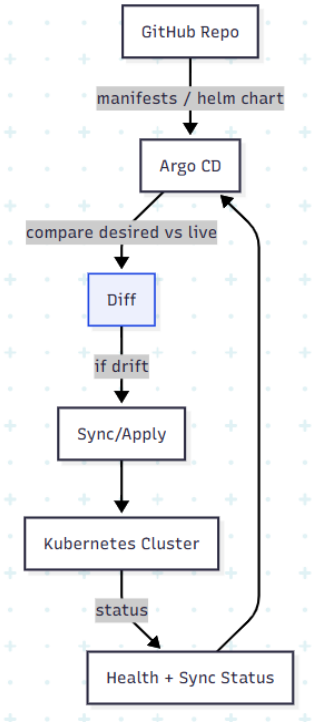
Automated tests cover the health endpoint, all CRUD operations, and metrics endpoints. The test coverage results achieved high percentages across statements, functions, and lines of code.

6.3 Build & Deployment Flow

The Docker image is built once and the same immutable image is promoted across environments. Argo CD automatically deploys updates to Kubernetes when a new image is available.

6.4 Argo CD Automation

Argo CD continuously syncs the Kubernetes cluster state with the configuration defined in the GitHub repository, enabling true GitOps deployment with automatic reconciliation.



7. Results

7.1 What Works

The application runs successfully on Kubernetes. The CI/CD pipeline is fully automated from code commit to deployment. The metrics and dashboards are operational, providing live observability. Security scans are integrated into the pipeline, and high test coverage has been achieved.

7.2 Deployment Evidence

Running pods are visible in the Kubernetes cluster. Argo CD shows the application as synced and healthy. Grafana dashboards display live metrics, and Prometheus is correctly scraping metrics from the application.

7.3 Security Results

No critical vulnerabilities are blocking deployment. Images are scanned for vulnerabilities before deployment, and secure runtime configuration is applied within the Kubernetes cluster.

8. Lessons Learned

8.1 Real DevOps Workflow

This project demonstrates key DevOps principles including Infrastructure as Code, GitOps practices, DevSecOps integration, and observability-driven operations in a practical implementation.

8.2 Common Mistakes Encountered

During development, challenges included Ingress configuration conflicts, Service label mismatches, Prometheus scrape configuration issues, and Kubernetes context connectivity problems.

8.3 Improvements Made

Issues were resolved by fixing Helm templating problems, improving security context settings, enhancing test coverage, and implementing a more structured logging format.

8.4 Future Improvements

Potential future enhancements include adding distributed tracing, implementing persistent storage, adding rate limiting and authentication features, and establishing multi-environment deployment strategies for development, staging, and production.

9. Conclusion

This project successfully demonstrates a complete DevOps lifecycle applied to a cloud-native application. It combines automation, security, observability, and Kubernetes best practices into a coherent, production-ready system. The project is suitable for academic submission, portfolio demonstration, DevOps interviews, and serves as a reference architecture for real-world implementations.