DevOps CI/CD Project Report

Implementing DevSecOps with GitHub Actions and Kubernetes

Name:	Kushi Varadaraj
Roll Number:	10035
Course:	DevOps
GitHub:	https://github.com/kushivaradaraj/devops-project
Date:	January 2026

1. Problem Background & Motivation

Before learning about DevOps, I thought deployment was just copying files to a server. But in real projects, there are many steps: running tests, checking code quality, scanning for security issues, building containers, and finally deploying. Doing all this manually for every change is time-consuming and error-prone.

The concept of 'shift-left' really changed my perspective. Instead of finding problems late (in production), we find them early (during development). A bug that costs $10 to fix in development might cost $1000 in production. Security issues are even worse.

My objective for this project was to build a pipeline that automates everything from code push to deployment. I also wanted to include security scanning at multiple points, following the DevSecOps methodology. The final piece was deploying to Kubernetes, which is how most cloud applications run today.

2. Application Overview

For this project, I built a simple web service using Java. I used Spring Boot because it makes it easy to create REST APIs and it's very popular in companies. The application is intentionally simple so I could focus on the pipeline, not the application logic.

What the Application Does:

- Health Check (/health) - Returns 'OK'. Kubernetes uses this to check if the app is alive.
- Greeting (/hello) - Says hello to whoever calls it. Can pass a name as parameter.
- Version (/version) - Returns '1.0.0'. Useful for checking which version is deployed.
- Calculator Service - Does basic math. I use this to show how unit tests work.

Technology Choices:

What	Technology	Why I Chose It
Programming	Java 17	Assignment recommended Java
Framework	Spring Boot	Easy to build REST APIs
Dependencies	Maven	Standard for Java projects
Testing	JUnit 5	Comes with Spring Boot
Packaging	Docker	Run anywhere consistently
Automation	GitHub Actions	Free, works with GitHub
Deployment	Kubernetes	Standard for containers

3. CI/CD Workflow Diagram

The workflow has two main parts. CI (Continuous Integration) runs on every push and handles building, testing, and scanning. CD (Continuous Deployment) takes the verified container and deploys it to Kubernetes.

```
PIPELINE OVERVIEW ============================================================ TRIGGER: Push to main branch or
manual trigger CI PHASE (Building & Scanning): ------------------------------------------------------------
Step 1: Get code from GitHub Step 2: Set up Java 17 environment Step 3: Run Checkstyle (code formatting) Step 4:
Run JUnit tests (12 tests) Step 5: Build JAR with Maven Step 6: CodeQL security scan (source code) Step 7: OWASP
```

scan (dependencies) Step 8: Build Docker image Step 9: Trivy scan (container) Step 10: Test container with curl Step 11: Push to Docker Hub CD PHASE (Deploying): -- Step 12: Start Minikube cluster Step 13: Apply deployment.yaml (creates 2 pods) Step 14: Apply service.yaml (exposes on NodePort) Step 15: Verify pods are running and healthy ==

Detailed CI Stages:

Stage	What Happens	If It Fails...
Checkout	Downloads code	Nothing to build
Java Setup	Installs JDK 17	Can't compile
Checkstyle	Checks formatting	Code style issues
JUnit Tests	Runs 12 tests	Bug in code
Maven Build	Creates JAR file	Compilation error
CodeQL	Scans source code	Security vulnerability
OWASP	Scans libraries	Vulnerable dependency
Docker Build	Creates image	Dockerfile problem
Trivy	Scans container	OS vulnerability
Container Test	Runs and tests	App doesn't start
Push	Uploads to Hub	Auth problem

Detailed CD Stages:

Stage	What Happens	Result
Minikube Setup	Starts local cluster	K8s environment ready
Deployment	kubectl apply deployment.yaml	2 pods created
Service	kubectl apply service.yaml	NodePort 30080 open
Verification	Check pods + test endpoint	Confirm it works

4. Security & Quality Controls

I have security checks at three different levels. This way, if one misses something, another might catch it.

Level 1 - My Code (SAST with CodeQL):

CodeQL is from GitHub. It looks at my Java code and finds problems like SQL injection or places where I might be logging sensitive data. It doesn't run the code - it just analyzes the patterns. Results show up in my repo's Security tab.

Level 2 - My Dependencies (SCA with OWASP):

Spring Boot brings in dozens of libraries. Each one could have security bugs. OWASP Dependency Check downloads a database of known vulnerabilities and compares my libraries against it. It tells me if I'm using something with a known problem.

Level 3 - My Container (Trivy):

The Docker image has an operating system (Alpine Linux) with its own packages. Trivy scans all of that. So even if my code is perfect and my libraries are safe, I still check the underlying OS.

Other Security Practices:

- Docker image runs as 'appuser', not root (principle of least privilege)
- Passwords are in GitHub Secrets, not in my code or Dockerfile

- Multi-stage build means no build tools in production image
- K8s has health probes that restart unhealthy pods automatically

5. Results & Observations

Here is what I observed after running the pipeline several times:

Item	Result	My Notes
Pipeline time	About 10 min	CodeQL takes the longest
Tests	All 12 pass	Calculator + API tests
Checkstyle	0 violations	After fixing formatting
CodeQL	Clean	No security issues found
OWASP	Clean	No vulnerable libraries
Trivy	Clean	Alpine is pretty secure
Docker image	182 MB	Could be smaller with distroless
K8s pods	2 running	Both healthy

Things I Noticed:

- First pipeline run is slower because CodeQL builds its database from scratch.
- Maven dependency caching is very helpful - saves about 2 minutes per run.
- I was surprised Minikube works in GitHub Actions. Thought it would need more setup.
- The health endpoint is crucial - K8s probes use it constantly.

6. Limitations & Improvements

Current Limitations:

- Minikube is just for testing. Real apps need AWS/GCP/Azure Kubernetes.
- Only one environment - no staging for pre-production testing.
- If deployment fails, I have to fix it manually. No automatic rollback.
- Security scans only stop the build for critical issues, not medium ones.
- I have to check GitHub to see results. No Slack/email alerts.

If I Had More Time, I Would Add:

- Deployment to a real cloud K8s cluster (probably GKE because of free tier)
- DAST scanning - actually attack the running application to find issues
- Canary deployments - send 10% traffic to new version first
- Slack webhook to notify team when pipeline passes or fails
- Load testing to see how many requests the app can handle
- Prometheus + Grafana for monitoring the deployed application

7. Conclusion

Working on this project made me understand why companies invest so much in DevOps. The automation removes human error and gives confidence that code is tested and secure before it reaches users.

What surprised me most was learning that security isn't one check at the end. It's multiple checks at different levels - code, libraries, and container. Each catches different types of problems. This layered approach makes the final deployment much safer.

The Kubernetes deployment was the final piece that made this feel real. Code goes from my editor to running pods without me doing anything manual. That's the goal of CI/CD - push code, everything else happens automatically.

Resources I Used:

- GitHub Actions - docs.github.com/en/actions
- Kubernetes Basics - kubernetes.io/docs/tutorials
- OWASP Security - owasp.org/www-project-top-ten
- Docker Guide - docs.docker.com/get-started
- Spring Boot - spring.io/guides