DevOps CI/CD Project Report

Building a Complete CI/CD Pipeline with Kubernetes

Name:	Jiya Singhal
Roll Number:	10043
Course:	DevOps
GitHub:	https://github.com/jiya-singhal/devops-ci-cd
Date:	January 2026

1. Problem Background & Motivation

When I started learning about software development, I noticed that many teams struggle with repetitive manual tasks. Every time code changes, someone has to manually run tests, check for issues, build the application, and deploy it. This process is slow and mistakes happen frequently.

I read that bugs found in production can cost 100 times more to fix than bugs caught during development. This made me realize why companies invest in automation. CI/CD pipelines automate all these steps so that every code change is automatically validated.

For this project, I wanted to implement not just CI/CD but also security scanning. This approach is called DevSecOps. Instead of checking security at the end, we check it at every stage. My goal was to create a pipeline that builds, tests, scans, and deploys to Kubernetes automatically.

2. Application Overview

I developed a REST API using Java and Spring Boot. I chose Spring Boot because it's one of the most popular frameworks in the industry, and the assignment recommended Java. The application is simple but has all the components needed to demonstrate a real pipeline.

Application Endpoints:

- /health - Health check endpoint returning 'OK' (critical for K8s probes)
- /hello - Greeting endpoint that accepts a name parameter
- /version - Returns current application version (1.0.0)

I also created a Calculator service with add, subtract, multiply, and divide operations. This service has unit tests that verify each operation works correctly. Having testable business logic was important to demonstrate the testing stage of the pipeline.

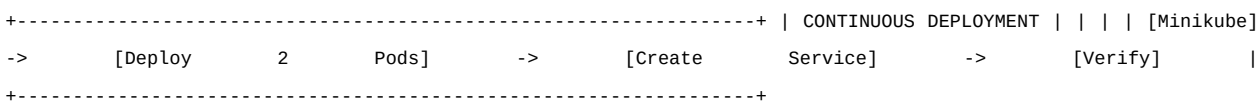
Tech Stack:

Layer	Choice	Reason
Language	Java 17	LTS version, recommended by assignment
Framework	Spring Boot 3.2	Industry standard for Java APIs
Build	Maven	Handles dependencies and builds
Tests	JUnit 5	Modern testing framework
Container	Docker	Consistent deployment
CI/CD	GitHub Actions	Free and integrated with GitHub
Orchestration	Kubernetes	Industry standard for containers

3. CI/CD Workflow Diagram

My pipeline is split into CI (Continuous Integration) and CD (Continuous Deployment). The CI part handles code quality and security. The CD part handles deployment to Kubernetes.

```
+-------------------------------------------------------------------+ | CONTINUOUS INTEGRATION | | | | [Push] ->
[Checkout] -> [Java 17] -> [Lint] -> [Test] -> [Build]| | | | | +--------------------+----------------+ | | | |
| | v v | | [CodeQL SAST] [OWASP SCA] | | (code security) (library security) | | | | | | +--------+-----------+ |
|   |   |   |   v   |   |   [Docker   Build]   ->   [Trivy   Scan]   ->   [Run   &   Test]   ->   [Push   Image]   |
+-------------------------------------------------------------------+                    |                    v
```


+---+ | CONTINUOUS DEPLOYMENT | | | | [Minikube]
-> [Deploy 2 Pods] -> [Create Service] -> [Verify] |
+---+

CI Stages Explained:

#	Stage	Tool	Why It's Needed
1	Checkout	actions/checkout	Get code from repository
2	Setup Java	actions/setup-java	Install JDK with caching
3	Linting	Checkstyle	Enforce consistent code style
4	Unit Tests	JUnit 5	Catch bugs before deployment
5	Build	Maven	Create executable JAR
6	SAST	CodeQL	Find code vulnerabilities
7	SCA	OWASP	Find library vulnerabilities
8	Docker	Buildx	Create container image
9	Scan	Trivy	Find container vulnerabilities
10	Test	curl	Verify container runs
11	Push	Docker Hub	Store verified image

CD Stages Explained:

#	Stage	Command	Result
1	Setup	minikube start	Local K8s cluster running
2	Deploy	kubectl apply -f deployment.yaml	2 pods created
3	Service	kubectl apply -f service.yaml	App exposed on port 30080
4	Verify	kubectl get pods + curl	Confirm healthy deployment

4. Security & Quality Controls

Security is not a single stage but integrated throughout the pipeline. This is the 'shift-left' approach - checking security early instead of at the end.

SAST - Scanning My Code:

CodeQL builds a database of my source code and queries it for security patterns. It finds issues like SQL injection, XSS attacks, and hardcoded passwords. I can see the results in GitHub's Security tab.

SCA - Scanning Libraries:

My app uses Spring Boot which pulls in many libraries. OWASP Dependency Check scans all of them against the National Vulnerability Database. If any library has a known CVE, the report will show it.

Container Security:

Trivy scans the final Docker image. Even if my code and libraries are secure, the base Alpine Linux image might have vulnerable packages. Trivy catches these.

Additional Security Measures:

- Container runs as non-root user 'appuser' (limits damage if compromised)
- Docker Hub credentials stored in GitHub Secrets (never in code)
- Multi-stage Dockerfile (build tools not in final image)

• K8s liveness/readiness probes (auto-restart unhealthy pods)

5. Results & Observations

After running the pipeline multiple times, here are my observations:

Measurement	Value	Notes
Full pipeline	~9 minutes	First run slower due to caching
Build + Test	~2 minutes	Faster with Maven cache
Security scans	~5 minutes	CodeQL is the slowest
K8s deployment	~2 minutes	Including verification
Unit tests	12 passing	Calculator + Controller tests
Code coverage	Good	Key paths covered
Final image	178 MB	Multi-stage build helped

What I Learned From Running the Pipeline:

• The first CodeQL run took almost 6 minutes because it builds a database. After that, it's faster because it reuses the database.

• Maven caching made a big difference. Without it, downloading dependencies took 2+ minutes every time.

• The multi-stage Docker build cut the image size significantly. The build stage image was over 400MB.

• Minikube works surprisingly well in GitHub Actions. I expected more issues.

6. Limitations & Improvements

What My Pipeline Doesn't Do (Yet):

- Uses Minikube which is a local cluster. A real project would use cloud K8s like EKS or GKE.
- No separate environments. Real pipelines have dev, staging, and production.
- No rollback mechanism. If deployment fails, I have to manually fix it.
- Security scans only block on critical issues. Medium/low issues just generate warnings.
- No notifications. I have to check GitHub to see if the pipeline passed.

What I Would Add in a Real Project:

- Deployment to a managed Kubernetes service (AWS EKS, Google GKE, or Azure AKS)
- DAST (Dynamic Application Security Testing) to test the running application
- Blue-green or canary deployments for zero-downtime updates
- Slack notifications when pipeline succeeds or fails
- Performance testing with a tool like JMeter or k6
- Helm charts for more complex K8s deployments

7. Conclusion

This project taught me that CI/CD is more than just automation - it's about building confidence in your code. Every push goes through the same checks, so you know exactly what state the code is in.

The most valuable lesson was understanding why each stage exists. It's easy to copy a pipeline configuration, but knowing what problems each stage solves makes debugging much easier. For example, SAST finds bugs in MY code, SCA finds bugs in LIBRARIES I use, and Trivy finds bugs in the CONTAINER OS.

Adding Kubernetes deployment (CD) made the pipeline complete. Now code goes from my laptop to a running cluster automatically. This is how software is delivered in modern companies.

References:

- GitHub Actions Documentation - docs.github.com/en/actions
- Kubernetes Official Docs - kubernetes.io/docs
- OWASP Top 10 Web Vulnerabilities - owasp.org
- Docker Best Practices - docs.docker.com
- Spring Boot Reference - spring.io/projects/spring-boot