DevOps CI/CD Pipeline Project with AWS Kubernetes Deployment

Building a Production-Grade CI/CD Pipeline with Security Integration

Name:	Jiya Singhal
Student ID:	10043
Course:	DevOps
GitHub:	github.com/jiya-singhal/devops-ci-cd
Docker Hub:	Docker image pushed successfully
Deployment:	AWS EC2 (t3.small) with k3s Kubernetes
Status:	Verified working, terminated to avoid costs

1. Problem Background & Motivation

In traditional software development, deployment is often a manual and error-prone process. Developers push code without proper testing, security vulnerabilities in code or third-party libraries go unnoticed, and the classic "works on my machine" problem causes failures in production environments.

CI/CD pipelines solve these problems by automating the entire process from code commit to deployment. Every push triggers automatic building, testing, and security scanning. This catches issues early when they are cheaper to fix. Adding security scans at multiple stages makes it DevSecOps - where security is built into the pipeline rather than being an afterthought.

The goal of this project was to build a complete CI/CD pipeline that not only builds and tests the application but also performs comprehensive security scanning and deploys to a real Kubernetes cluster on AWS. I wanted to understand not just how to set up these tools, but why each stage exists and what problems it solves.

2. Application Overview

For this project, I built a Java Spring Boot REST API. The application is intentionally simple because the focus was on the pipeline, not the application complexity. It includes:

- /health - Returns 'OK', used by Kubernetes for liveness and readiness probes
- /hello - Returns a greeting message, accepts a name parameter
- /version - Returns the application version
- Calculator Service - Basic math operations used for unit testing

Technology Stack:

Component	Technology	Reason
Language	Java 17	Recommended for the project
Framework	Spring Boot 3.2	Easy REST API development
Build Tool	Maven	Standard for Java projects
Testing	JUnit 5	Integrated with Spring Boot
Container	Docker	Consistent deployment
CI/CD	GitHub Actions	Free, integrates with GitHub
Orchestration	Kubernetes (k3s)	Industry standard
Cloud	AWS EC2	Real cloud deployment

3. CI/CD Pipeline Architecture

The pipeline is split into two separate workflows: CI Pipeline handles building, testing, and security scanning; CD Pipeline handles infrastructure provisioning and deployment.

CI Pipeline Flow:

```
Code Push → Checkout → Setup Java → Checkstyle (Lint) → Unit Tests → Build JAR → CodeQL (SAST) → OWASP (SCA) → Docker Build → Trivy Scan → Container Test → Push to Docker Hub
```

CD Pipeline Flow:

```
CI Success → Terraform Init → Terraform Apply → Create EC2 → Install k3s → Deploy App → Create Service → Health Check → Verify Deployment
```

CI Pipeline Stages:

Stage	Tool	Purpose	Fail Condition
Checkout	actions/checkout	Get source code	Repo access issue
Setup Java	actions/setup-java	Install JDK 17	Version mismatch
Linting	Checkstyle	Enforce code style	Style violations
Unit Tests	JUnit 5	Test business logic	Test failures
Build	Maven	Create JAR file	Compilation error
SAST	CodeQL	Scan code for vulnerabilities	Critical findings
SCA	OWASP DC	Scan dependencies	Critical CVEs
Docker Build	Docker	Create container image	Build failure
Image Scan	Trivy	Scan container	Critical vulnerabilities
Container Test	curl	Test running container	Health check fail
Push	Docker Hub	Publish image	Auth failure

4. Security Integration (DevSecOps)

Security is integrated at multiple stages following the shift-left principle - catching vulnerabilities as early as possible in the development lifecycle.

SAST - Static Application Security Testing (CodeQL):

CodeQL analyzes the source code without executing it. It looks for patterns that indicate security vulnerabilities like SQL injection, cross-site scripting (XSS), path traversal, and hardcoded credentials. Results appear in GitHub's Security tab. This catches issues in my own code before it even runs.

SCA - Software Composition Analysis (OWASP Dependency Check):

Spring Boot pulls in dozens of transitive dependencies. Each library could have known vulnerabilities. OWASP Dependency Check compares all dependencies against the National Vulnerability Database (NVD) to find known CVEs. This is critical for supply chain security - it would catch issues like Log4Shell (CVE-2021-44228) before deployment.

Container Scanning (Trivy):

Even if my code and libraries are secure, the Docker image contains an operating system with its own packages. Trivy scans the entire container image for vulnerabilities in OS packages and application libraries. This ensures the runtime environment is also secure.

Other Security Measures:

- Docker container runs as non-root user 'appuser' (principle of least privilege)
- All credentials stored in GitHub Secrets, never in code
- Multi-stage Docker build excludes build tools from production image
- Kubernetes health probes automatically restart unhealthy pods

5. CD Pipeline & Kubernetes Deployment

Initial Approach - Minikube:

Initially, I implemented the CD pipeline using Minikube running inside GitHub Actions. This worked - it created a local Kubernetes cluster, deployed the application, and verified it was running. However, the deployment was temporary and not publicly accessible. The cluster existed only during the pipeline run.

Final Approach - AWS EC2 with k3s:

To achieve a real, persistent, publicly accessible deployment, I upgraded to AWS EC2 with k3s (lightweight Kubernetes). The CD pipeline uses Terraform to provision infrastructure as code.

Challenges Faced During AWS Setup:

The AWS deployment was not straightforward and I encountered several issues that required troubleshooting:

1. Free Tier Restriction: My AWS account only allowed certain instance types. Initially tried t2.medium but got 'not eligible for Free Tier' error.
2. Memory Issues with t3.micro: Tried t3.micro (1GB RAM) but k3s kept timing out - not enough memory for Kubernetes.
3. Solution - t3.small: Discovered my account allowed t3.small (2GB RAM) as free tier. This was enough for k3s with optimizations.
4. SSH Timeout: The k3s installation took too long and SSH connection dropped. Fixed by adding ServerAliveInterval settings.
5. Resource Conflicts: Failed runs left behind Security Groups and Key Pairs. Had to manually clean up AWS resources between attempts.
6. YAML Syntax Errors: Emoji characters in the workflow file caused parsing errors. Removed all emojis to fix.
7. apt-get Lock: Cloud-init was still running when k3s tried to install. Added wait logic for apt locks to release.

Final Working Configuration:

Component	Configuration
EC2 Instance	t3.small (2 vCPU, 2GB RAM)
Operating System	Ubuntu 22.04 LTS
Kubernetes	k3s with --disable traefik --disable metrics-server
Replicas	2 pods
Service Type	NodePort
Port	8080 (container) → 30821 (NodePort)
Public Access	http://<EC2-IP>:30821

The deployment was successfully verified with both /health and /hello endpoints responding correctly. The EC2 instance has been terminated to avoid ongoing costs. Screenshots of the working deployment are attached.

6. Results & Observations

Metric	Result	Notes
CI Pipeline Duration	~5-6 minutes	SAST (CodeQL) takes the longest
CD Pipeline Duration	~4-5 minutes	Includes Terraform + k3s setup
Unit Tests	12 tests, 100% pass	Calculator + REST endpoint tests
Checkstyle	0 violations	After fixing initial style issues
CodeQL (SAST)	No critical issues	Clean code scan
OWASP (SCA)	No critical CVEs	Dependencies are secure
Trivy	No critical vulnerabilities	Container image is clean
Docker Image Size	~180 MB	Multi-stage build optimization
Kubernetes Pods	2 replicas running	Both healthy
NodePort	30821	Publicly accessible

Key Learnings:

- CodeQL's first run is slow because it builds a database of the codebase. Subsequent runs are faster.
- Maven dependency caching in GitHub Actions saves significant build time.
- k3s can run on 2GB RAM if you disable unnecessary components like traefik and metrics-server.
- Terraform makes infrastructure reproducible - I could destroy and recreate the entire setup easily.
- The health endpoint is critical - Kubernetes probes use it constantly to determine pod health.

7. Limitations & Future Improvements

Current Limitations:

- Single EC2 instance - no high availability across availability zones
- No HTTPS/SSL - application runs on HTTP only
- DAST is a placeholder - not actually running OWASP ZAP scans
- No auto-scaling - fixed at 2 replicas regardless of load
- No monitoring or alerting system in place
- EC2 must be terminated manually to avoid costs

Future Improvements:

- Deploy to managed Kubernetes (AWS EKS) for production-grade setup
- Add HTTPS using Let's Encrypt certificates
- Implement actual DAST scanning with OWASP ZAP
- Add blue-green or canary deployment strategy
- Integrate Prometheus and Grafana for monitoring
- Add Slack notifications for pipeline status

8. Conclusion

This project gave me hands-on experience with building a production-style CI/CD pipeline. The most valuable learning was not just how to use the tools, but understanding why each stage exists and what problems it solves.

The shift-left security approach was eye-opening. Instead of one security check at the end, having SAST, SCA, and container scanning at different stages creates defense in depth. Each tool catches different types of vulnerabilities - CodeQL finds issues in my code, OWASP finds vulnerable dependencies, and Trivy finds OS-level issues.

The journey from Minikube to AWS was challenging but rewarding. Dealing with real cloud constraints like instance type restrictions and memory limitations taught me more than a smooth deployment would have. The troubleshooting process - from free tier errors to SSH timeouts to YAML syntax issues - is a realistic representation of DevOps work.

The final pipeline successfully automates the entire path from code push to a publicly accessible Kubernetes deployment. Every commit triggers building, testing, security scanning, containerization, and deployment without any manual intervention. This is the core promise of CI/CD - push code, everything else happens automatically.

Attachments:

- Screenshots of CI Pipeline execution
- Screenshots of CD Pipeline execution
- AWS EC2 console screenshots
- Application endpoints (/health, /hello) screenshots
- kubectl get nodes, pods, svc output
- Docker Hub repository screenshot

Repository: github.com/jiya-singhal/devops-ci-cd