DevOps CI/CD Pipeline Project
Automated Deployment to AWS Kubernetes

Implementing DevSecOps Practices with GitHub Actions and Cloud Infrastructure

Name:	Kushi Varadaraj
Student ID:	10035
Course:	DevOps
GitHub:	github.com/kushivaradaraj/devops-project
Docker Hub:	Image published successfully
Deployment:	AWS EC2 (t3.small) running k3s
Status:	Deployment verified, EC2 terminated to save costs

1. Problem Background & Motivation

Before this project, my understanding of deployment was quite basic - I thought it was just about getting code onto a server somehow. But real-world software delivery involves much more: ensuring code quality, running tests, checking for security vulnerabilities, building containers, and deploying reliably. Doing all of this manually for every code change is not sustainable.

The DevOps approach automates this entire workflow. When a developer pushes code, everything happens automatically - tests run, security scans execute, containers get built, and the application deploys. This concept of "shift-left" means finding problems early in the process when they are easier and cheaper to fix, rather than discovering them in production.

My objective was to implement a complete CI/CD pipeline that covers building, testing, security scanning, containerization, and deployment to a real Kubernetes cluster. More importantly, I wanted to understand the reasoning behind each stage - not just how to configure tools, but why they matter.

2. Application Overview

I developed a simple Java Spring Boot REST API for this project. The application itself is straightforward because the main focus was on the CI/CD pipeline rather than complex application logic. The endpoints include:

- /health - Returns 'OK' status, used by Kubernetes probes to check if the app is running
- /hello - A greeting endpoint that accepts a name parameter and returns a personalized message
- /version - Returns the current application version number
- Calculator Service - Provides basic arithmetic operations, used to demonstrate unit testing

Technologies Used:

What	Technology	Why I Chose It
Programming	Java 17	Project requirement
Framework	Spring Boot 3.2	Simplifies REST API creation
Build	Maven	Standard Java build tool
Testing	JUnit 5	Works well with Spring
Containerization	Docker	Ensures consistent environments
Automation	GitHub Actions	Integrated with GitHub, free tier
Orchestration	Kubernetes (k3s)	Production-standard container orchestration
Infrastructure	AWS EC2 + Terraform	Real cloud with infrastructure as code

3. CI/CD Pipeline Design

I designed the pipeline as two separate workflows. The CI Pipeline focuses on code quality and security verification. The CD Pipeline handles the actual deployment to AWS infrastructure.

CI Pipeline Sequence:

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

CD Pipeline Sequence:

```
Triggered by CI → Terraform Init → Provision EC2 → Wait for Setup → Install k3s → Deploy to Kubernetes → Expose Service → Verify Health
```

CI Stages Explained:

Stage	Tool Used	What It Does	Why It Matters
Checkout	actions/checkout	Downloads code	Need code to build
Java Setup	actions/setup-java	Installs JDK	Runtime environment
Lint	Checkstyle	Checks code style	Maintainability
Tests	JUnit 5	Runs unit tests	Catches bugs early
Build	Maven	Creates JAR	Deployable artifact
SAST	CodeQL	Scans source code	Finds security flaws
SCA	OWASP DC	Scans dependencies	Supply chain security
Docker	Docker	Builds image	Container packaging
Image Scan	Trivy	Scans container	OS vulnerabilities
Test Run	curl	Tests container	Validates it works
Push	Docker Hub	Publishes image	Makes it deployable

4. Security Integration (DevSecOps)

Security scanning happens at three different levels in my pipeline. The idea is that each tool catches different types of problems, creating multiple layers of protection.

Level 1 - Source Code Analysis (CodeQL):

CodeQL from GitHub analyzes my Java source code without running it. It looks for dangerous patterns like SQL injection, cross-site scripting, path traversal attacks, and hardcoded secrets. Any findings appear in the GitHub Security tab. This catches problems in my own code.

Level 2 - Dependency Analysis (OWASP Dependency Check):

My application uses Spring Boot, which brings in many third-party libraries. Any of these could have known vulnerabilities. OWASP Dependency Check compares every dependency against the National Vulnerability Database. This would catch dangerous issues like Log4Shell before deployment.

Level 3 - Container Analysis (Trivy):

The Docker image includes an operating system (I used a slim base image) with its own packages. Trivy scans everything in the container - OS packages, application libraries, configuration issues. This ensures even the runtime environment is secure.

Additional Security Practices:

- The Docker container runs as a non-root user to limit potential damage if compromised
- Credentials are stored in GitHub Secrets - never committed to the repository
- Multi-stage Docker build keeps build tools out of the final image
- Kubernetes probes automatically restart pods that become unhealthy

5. Continuous Deployment to AWS

First Attempt - Minikube in Pipeline:

My initial approach was using Minikube directly in the GitHub Actions runner. This actually worked - it spun up a local Kubernetes cluster, deployed my application, and verified it was running. The problem was that the deployment was temporary. Once the pipeline finished, everything was gone. There was no way for anyone to actually access the running application.

Second Attempt - Real AWS Deployment:

To make the deployment persistent and publicly accessible, I moved to AWS EC2 with k3s (a lightweight Kubernetes distribution). The CD pipeline uses Terraform for infrastructure as code, which means the entire AWS setup can be created and destroyed automatically.

Problems I Encountered:

Getting the AWS deployment working was not straightforward. Here are the issues I ran into and how I solved them:

1. Instance Type Restrictions: My AWS account was limited to free tier instances. When I tried t2.medium, it was rejected.
2. Not Enough Memory: I switched to t3.micro (1GB RAM), but k3s kept timing out during startup. Kubernetes needs more memory.
3. Found a Working Size: Discovered t3.small (2GB RAM) was available in my free tier. This had enough memory for k3s.
4. Connection Drops: SSH sessions kept disconnecting while k3s was installing. Added keep-alive settings to fix this.
5. Leftover Resources: Failed pipeline runs left AWS resources behind (security groups, key pairs) causing conflicts. Had to clean these up manually.
6. Workflow File Issues: Had emoji characters in my YAML file that caused parsing errors. Cleaned those out.
7. Package Manager Conflicts: The EC2 was still running initial setup when k3s tried to install, causing apt lock errors. Added wait logic.

Working Setup:

Setting	Value
Instance Type	t3.small (2 vCPU, 2GB Memory)
OS	Ubuntu 22.04 LTS
Kubernetes	k3s (traefik and metrics-server disabled to save memory)
App Replicas	2 pods running
Service	NodePort type
Access	Port 30821 on EC2 public IP

After all the troubleshooting, the deployment worked successfully. I was able to access /health and /hello endpoints from a browser using the EC2 public IP and NodePort. The EC2 has been terminated to avoid accumulating costs. Screenshots of the working state are included with this submission.

6. Results & What I Observed

What	Outcome	My Observations
CI Time	About 6 minutes	CodeQL is the slowest part
CD Time	About 5 minutes	Most time is waiting for k3s
Tests	12 passed	Calculator and REST tests
Code Style	No issues	Had to fix some initially
CodeQL	Clean	No vulnerabilities in my code
OWASP	Clean	No CVEs in dependencies
Trivy	Clean	Container image is safe
Image Size	About 180 MB	Multi-stage build helped
Pods	2 running	Both showed as healthy
Port	30821	NodePort assigned by K8s

Things I Learned:

- The first CodeQL scan takes longer because it builds a database. Later runs are faster.
- Caching Maven dependencies in GitHub Actions makes a noticeable difference in build time.
- k3s is surprisingly capable on limited resources if you disable components you do not need.
- Terraform is powerful for infrastructure - I could recreate everything from scratch if needed.
- The /health endpoint is not just nice to have - Kubernetes depends on it to manage pods.

7. Limitations & Future Work

What My Project Does Not Have:

- Only one EC2 instance - if it goes down, the application is unavailable
- HTTP only - no SSL/TLS encryption for traffic
- DAST is just a placeholder - not actually scanning the running application
- Fixed number of replicas - no automatic scaling based on load
- No monitoring or alerts - I would not know if something went wrong unless I checked
- Have to remember to terminate EC2 to avoid charges

What I Would Add With More Time:

- Use managed Kubernetes like AWS EKS for better reliability
- Add HTTPS with Let's Encrypt certificates
- Implement real DAST scanning with OWASP ZAP
- Set up blue-green deployments for zero-downtime updates
- Add Prometheus and Grafana for monitoring
- Configure Slack notifications for pipeline results

8. Conclusion

This project taught me that DevOps is much more than just running commands. It is about creating automated, reliable, and secure software delivery. The most valuable part was understanding why each pipeline stage exists, not just how to configure it.

The security aspect was particularly interesting. Having three different types of scans (code, dependencies, container) makes sense because each catches different problems. A vulnerability in my code is different from a CVE in a library I am using, which is different from an issue in the base operating system image.

The AWS deployment challenges were frustrating at the time but taught me a lot. Real cloud environments have constraints - you cannot always use whatever instance type you want. Debugging SSH timeouts, resource conflicts, and YAML errors is apparently a normal part of DevOps work.

In the end, I have a pipeline that takes code from a commit all the way to a publicly accessible Kubernetes deployment without any manual steps. Push code, wait a few minutes, and it is live. That is what CI/CD promises, and it actually works.

Attached Screenshots:

- CI Pipeline - all stages completing successfully
- CD Pipeline - Terraform and deployment stages
- AWS Console - EC2 instance details
- Browser - /health and /hello endpoint responses
- Terminal - kubectl showing nodes, pods, and services
- Docker Hub - published image

Repository: github.com/kushivaradaraj/devops-project