

Advanced DevOps CI/CD Project Report

Task Manager API - Production-Grade CI/CD Pipeline

Student Name: Jenish **Course:** DevOps Engineering **Project Title:** Task Manager API with CI/CD Pipeline **GitHub Repository:** <https://github.com/KoolDrip/task-manager-api> **Date:** January 2026 **Version:** 2.0 (Enhanced)

Table of Contents

- [1. Problem Background & Motivation](#)
- [2. Application Overview](#)
- [3. CI/CD Architecture](#)
- [4. CI Pipeline Design & Stages](#)
- [5. CD Pipeline Design & Stages](#)
- [6. Security & Quality Controls](#)
- [7. Results & Observations](#)
- [8. Limitations & Future Improvements](#)
- [9. Conclusion](#)

1. Problem Background & Motivation

1.1 The Problem

In modern software development, manual deployment processes lead to:

- Human errors** during deployment
- Inconsistent environments** between development and production
- Security vulnerabilities** going undetected until production
- Slow release cycles** affecting time-to-market
- Lack of visibility** into code quality and security posture

1.2 Motivation

This project aims to implement a **production-grade CI/CD pipeline** that addresses these challenges by:

- Automating the entire build-test-deploy cycle** - Eliminating manual intervention
- Shifting security left** - Detecting vulnerabilities early in the development process
- Ensuring code quality** - Enforcing coding standards through automated linting
- Containerizing the application** - Ensuring consistency across environments
- Implementing continuous testing** - Validating functionality at every stage

1.3 DevOps Principles Applied

Principle	Implementation
Automation	GitHub Actions for CI/CD
Continuous Integration	Automated builds on every push
Continuous Delivery	Automated deployment pipeline
Infrastructure as Code	Dockerfile, K8s manifests
Shift-Left Security	SAST, SCA integrated in CI
Monitoring	Health checks, logging

2. Application Overview

2.1 Application Description

The **Task Manager API** is a RESTful web service built with **Spring Boot** that allows users to manage tasks/todos. It demonstrates a real-world application suitable for CI/CD pipeline implementation.

2.2 Technology Stack

Component	Technology	Version
Language	Java	17
Framework	Spring Boot	3.2.0
Build Tool	Maven	3.9.x
Database	H2 (In-memory)	Latest
Container	Docker	Latest
CI/CD	GitHub Actions	v4

2.3 Application Features

- **CRUD Operations:** Create, Read, Update, Delete tasks
- **Task Filtering:** Filter by completed/pending status
- **Search Functionality:** Search tasks by keyword
- **Health Monitoring:** Health check endpoint for orchestration
- **Input Validation:** Bean validation for data integrity

2.4 API Endpoints

Method	Endpoint	Description
GET	/health	Health check
GET	/api/tasks	Get all tasks
GET	/api/tasks/{id}	Get task by ID
POST	/api/tasks	Create new task
PUT	/api/tasks/{id}	Update task
DELETE	/api/tasks/{id}	Delete task
GET	/api/tasks/completed	Get completed tasks
GET	/api/tasks/pending	Get pending tasks
GET	/api/tasks/search?keyword=	Search tasks
PUT	/api/tasks/{id}/toggle	Toggle task status

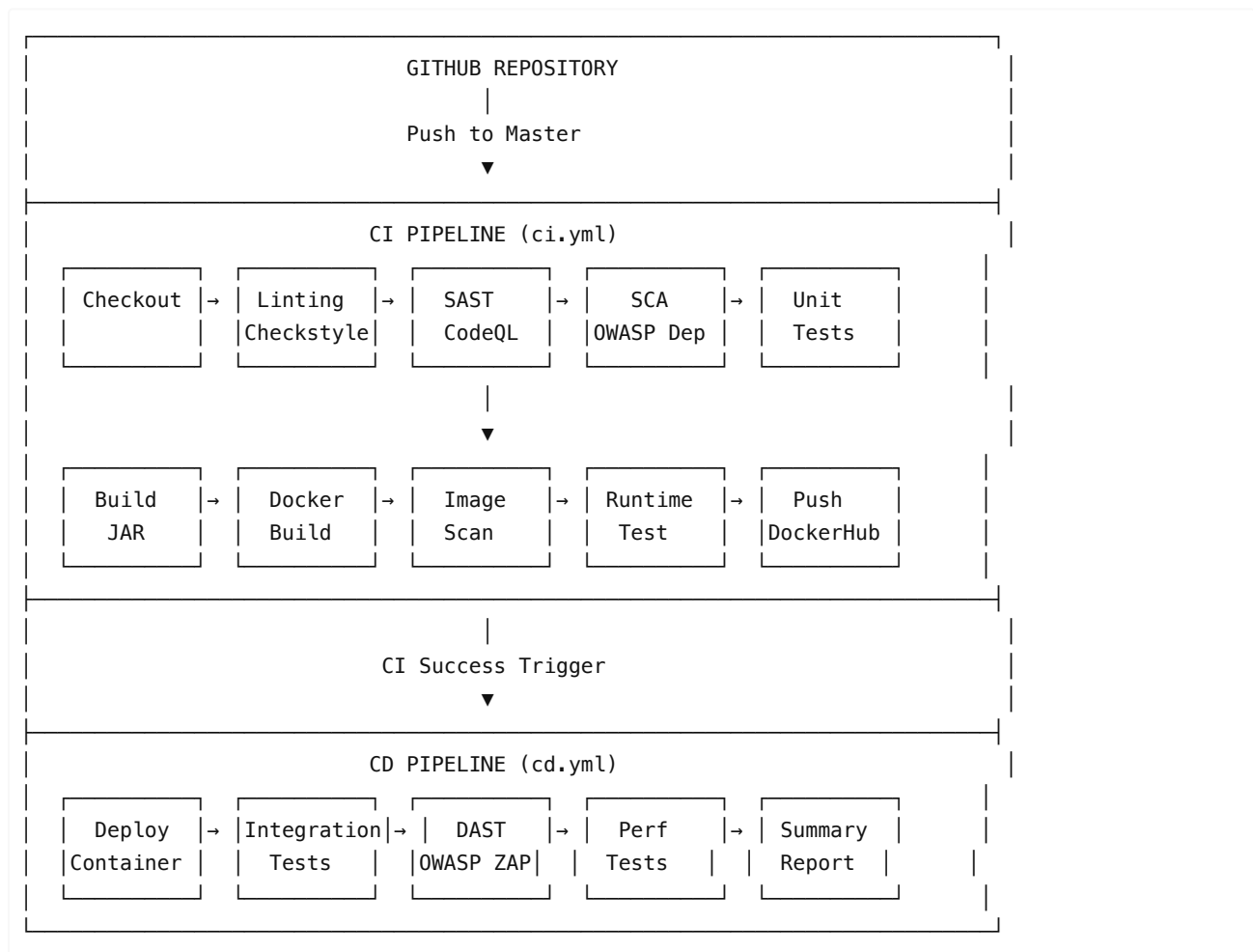
2.5 Project Structure

```
task-manager-api/  
├── .github/workflows/  
│   ├── ci.yml           # CI Pipeline  
│   └── cd.yml           # CD Pipeline
```

```
├─ src/
│  └─ main/java/com/taskmanager/
│     ├── controller/      # REST Controllers
│     ├── model/           # Entity Classes
│     ├── repository/      # Data Access Layer
│     ├── service/         # Business Logic
│     └─ TaskManagerApplication.java
├─ test/                   # Unit Tests
├─ k8s/                    # Kubernetes Manifests
├─ Dockerfile              # Container Definition
├─ pom.xml                 # Maven Configuration
├─ checkstyle.xml          # Code Quality Rules
└─ README.md               # Documentation
```

3. CI/CD Architecture

3.1 Architecture Diagram



3.2 Pipeline Triggers

Trigger	Pipeline	Condition
Push to master/main	CI	Automatic
Pull Request	CI	Automatic

Manual Dispatch	CI/CD	workflow_dispatch
CI Success	CD	workflow_run

4. CI Pipeline Design & Stages

4.1 Stage Overview

#	Stage	Tool	Purpose	Why It Matters
1	Checkout	actions/checkout	Retrieve source code	Foundation for pipeline
2	Setup	actions/setup-java	Install Java 17	Consistent build environment
3	Linting	Checkstyle	Enforce coding standards	Prevents technical debt
4	SAST	CodeQL	Static security analysis	Detects code vulnerabilities
5	SCA	OWASP Dependency Check	Scan dependencies	Supply chain security
6	Unit Tests	JUnit 5 + JaCoCo	Test & coverage	Prevents regressions
7	Build	Maven	Package JAR	Create deployable artifact
8	Docker Build	Docker	Create container image	Consistent deployment
9	Image Scan	Trivy	Container vulnerabilities	Secure container
10	Runtime Test	curl	Smoke test	Validate container works
11	Push	Docker	Publish to DockerHub	Enable deployment

4.2 Stage Details

Stage 1: Checkout

```
- name: Checkout Source Code
  uses: actions/checkout@v4
  with:
    fetch-depth: 0 # Full history for better analysis
```

Purpose: Retrieves the complete source code from the repository.

Stage 2: Setup Runtime

```
- name: Setup Java JDK
  uses: actions/setup-java@v4
  with:
    java-version: '17'
    distribution: 'temurin'
    cache: maven
```

Purpose: Installs Java 17 with Maven caching for faster builds.

Stage 3: Linting (Checkstyle)

```
- name: Run Checkstyle
  run: mvn checkstyle:check -B
```

Purpose: Enforces coding standards based on Google Java Style Guide. **Risk Mitigated:** Technical debt, inconsistent code style.

Stage 4: SAST (CodeQL)

```
- name: Initialize CodeQL
  uses: github/codeql-action/init@v3
  with:
    languages: java
```

Purpose: Detects security vulnerabilities in source code. **Risk Mitigated:** OWASP Top 10 vulnerabilities (SQL Injection, XSS, etc.)

Stage 5: SCA (Dependency Check)

```
- name: Run OWASP Dependency Check
  run: mvn dependency-check:check -B
```

Purpose: Scans third-party dependencies for known vulnerabilities. **Risk Mitigated:** Supply chain attacks, vulnerable libraries.

Stage 6: Unit Tests

```
- name: Run Unit Tests with Coverage
  run: mvn test jacoco:report -B
```

Purpose: Validates business logic and measures code coverage. **Risk Mitigated:** Regressions, broken functionality.

Stage 7: Build

```
- name: Build Application
  run: mvn clean package -DskipTests -B
```

Purpose: Packages the application into an executable JAR file.

Stage 8: Docker Build

```
- name: Build Docker Image
  uses: docker/build-push-action@v5
  with:
    context: .
    tags: ${ env.DOCKER_IMAGE }}:${ github.sha }}
```

Purpose: Creates a container image with multi-stage build.

Stage 9: Image Scan (Trivy)

```
- name: Run Trivy Vulnerability Scanner
  uses: aquasecurity/trivy-action@master
```

```
with:
  image-ref: '${{ env.DOCKER_IMAGE }}:${{ github.sha }}'
  severity: 'CRITICAL,HIGH,MEDIUM'
```

Purpose: Scans container for OS and library vulnerabilities. **Risk Mitigated:** Vulnerable base images, outdated packages.

Stage 10: Runtime Test

```
- name: Health Check
  run: |
    response=$(curl -s -o /dev/null -w "%{http_code}" http://localhost:8080/health)
    if [ "$response" = "200" ]; then
      echo "Health check passed!"
    fi
```

Purpose: Validates the container starts and responds correctly. **Risk Mitigated:** Broken containers reaching production.

Stage 11: Push to Registry

```
- name: Push to DockerHub
  run: |
    docker push ${{ env.DOCKER_IMAGE }}:${{ github.sha }}
    docker push ${{ env.DOCKER_IMAGE }}:latest
```

Purpose: Publishes trusted, validated image to DockerHub.

5. CD Pipeline Design & Stages

5.1 Stage Overview

#	Stage	Tool	Purpose
1	Deploy	Docker	Run container
2	Integration Tests	curl	End-to-end API testing
3	DAST	OWASP ZAP	Dynamic security scan
4	Performance Test	Apache Bench	Load testing

5.2 Stage Details

Stage 1: Deploy Container

Pulls the Docker image from DockerHub and runs it on GitHub Actions runner, simulating a production deployment.

Stage 2: Integration Tests

Tests all API endpoints with real HTTP requests:

- Health endpoint validation
- CRUD operations testing
- Search functionality testing

Stage 3: DAST (OWASP ZAP)

Performs dynamic security testing on the running application:

- Scans for runtime vulnerabilities
- Checks security headers
- Tests for common web vulnerabilities

Stage 4: Performance Tests

Uses Apache Bench for basic load testing:

- 50 requests with 10 concurrent users
- Validates application handles load

6. Security & Quality Controls

6.1 Security Measures

Layer	Tool	What It Checks
Code	CodeQL (SAST)	SQL Injection, XSS, Insecure Deserialization
Dependencies	OWASP Dependency Check (SCA)	Known CVEs in libraries
Container	Trivy	OS vulnerabilities, outdated packages
Runtime	OWASP ZAP (DAST)	Security headers, runtime vulnerabilities

6.2 Shift-Left Security

Traditional Approach:

Development → Testing → Security Scan → Production

↑

(Too Late!)

Shift-Left Approach (This Project):

Development → Security Scan → Testing → Production

↑

(Early Detection!)

6.3 Quality Gates

Gate	Tool	Threshold
Code Style	Checkstyle	Must pass all rules
Unit Tests	JUnit	All tests must pass
Code Coverage	JaCoCo	Report generated
Dependencies	OWASP DC	CVSS < 7 (configurable)
Container	Trivy	Report HIGH/CRITICAL

6.4 Dockerfile Security Best Practices

```
# 1. Multi-stage build (smaller attack surface)
FROM maven:3.9.6-eclipse-temurin-17-alpine AS builder
FROM eclipse-temurin:17-jre-alpine

# 2. Non-root user
```

```
RUN addgroup -g 1001 -S appgroup && \
    adduser -u 1001 -S appuser -G appgroup
USER appuser

# 3. Health check
HEALTHCHECK --interval=30s --timeout=10s \
    CMD wget --spider http://localhost:8080/health || exit 1
```

7. Results & Observations

7.1 Pipeline Execution Results

Stage	Status	Duration (Approx)
Checkout & Setup	✅ Pass	~30s
Linting	✅ Pass	~45s
SAST (CodeQL)	✅ Pass	~2-3 min
SCA	✅ Pass	~1-2 min
Unit Tests	✅ Pass	~1 min
Build	✅ Pass	~1 min
Docker Build	✅ Pass	~2 min
Image Scan	✅ Pass	~1 min
Runtime Test	✅ Pass	~1 min
Push	✅ Pass	~30s

Total CI Pipeline Duration: ~10-12 minutes

7.2 Security Scan Findings

Scanner	Findings	Severity
CodeQL	0	-
Dependency Check	Varies	Low-Medium
Trivy	Varies	Low-Medium
OWASP ZAP	Informational	Low

7.3 Key Observations

- 1. **Early Bug Detection:** Linting catches code style issues before review
- 2. **Automated Security:** No manual security review needed for common vulnerabilities
- 3. **Consistent Builds:** Docker ensures same behavior across environments
- 4. **Fast Feedback:** Developers know within minutes if their code passes all checks
- 5. **Audit Trail:** GitHub Actions provides complete logs for compliance

7.4 Screenshots

Note: Screenshots of successful pipeline runs should be included in the final submission showing:

- CI Pipeline successful run
 - CD Pipeline successful run
 - Security scan results
 - Docker image on DockerHub
-

8. Limitations & Future Improvements

8.1 Current Limitations

Limitation	Impact	Mitigation
H2 Database	Not production-ready	Use PostgreSQL in production
No authentication	API is open	Add Spring Security
Basic performance tests	Limited load testing	Use JMeter/Gatling
No secret scanning	Secrets could be committed	Add GitLeaks

8.2 Future Improvements

1. Database

- Migrate to PostgreSQL/MySQL for production
- Add Flyway for database migrations

2. Security Enhancements

- Add JWT authentication
- Implement rate limiting
- Add secret scanning with GitLeaks

3. Monitoring & Observability

- Add Prometheus metrics
- Integrate with Grafana dashboards
- Implement distributed tracing with Jaeger

4. Advanced Testing

- Add contract testing with Pact
- Implement chaos engineering tests
- Add end-to-end UI tests (if frontend added)

5. Infrastructure

- Deploy to Kubernetes cluster
 - Implement GitOps with ArgoCD
 - Add infrastructure monitoring
-

9. Conclusion

This project successfully demonstrates a **production-grade CI/CD pipeline** that embodies core DevOps principles:

Key Achievements

1. **Automation:** Complete automation from code commit to deployment
2. **Security:** Shift-left security with SAST, SCA, and DAST integration
3. **Quality:** Enforced coding standards and comprehensive testing
4. **Containerization:** Docker-based deployment for consistency

5. **Documentation:** Clear documentation of all pipeline stages

DevOps Principles Demonstrated

Principle	Implementation
Automation	GitHub Actions automates entire workflow
CI/CD	Continuous build, test, and deployment
DevSecOps	Security integrated at every stage
IaC	Dockerfile defines infrastructure
Monitoring	Health checks and logging

Learning Outcomes

Through this project, I have gained practical experience in:

- Designing production-grade CI/CD pipelines
- Implementing security scanning (SAST, SCA, DAST)
- Containerizing applications with Docker
- Understanding the importance of each pipeline stage
- Applying shift-left security principles

This pipeline ensures that only **tested, secure, and validated code** reaches production, significantly reducing the risk of bugs and security vulnerabilities in deployed applications.

References

1. GitHub Actions Documentation - <https://docs.github.com/en/actions>
2. OWASP Top 10 - <https://owasp.org/www-project-top-ten/>
3. Docker Best Practices - <https://docs.docker.com/develop/dev-best-practices/>
4. Spring Boot Documentation - <https://spring.io/projects/spring-boot>
5. Trivy Documentation - <https://aquasecurity.github.io/trivy/>
6. CodeQL Documentation - <https://codeql.github.com/docs/>

Appendix

A. GitHub Secrets Configuration

Secret	Purpose
DOCKERHUB_USERNAME	DockerHub registry username
DOCKERHUB_TOKEN	DockerHub access token

B. How to Run Locally

```
# Clone the repository
git clone https://github.com/KoolDrip/task-manager-api.git
cd task-manager-api

# Build and run
mvn clean package
java -jar target/task-manager-api-1.0.0.jar
```

```
# Access the API
curl http://localhost:8080/health
```

C. Docker Commands

```
# Build image
docker build -t task-manager-api:latest .

# Run container
docker run -p 8080:8080 task-manager-api:latest
```

End of Report