CI/CD Pipeline Implementation Project

DevOps Automation Report

Automated CI/CD Pipeline for Spring PetClinic

From Local Development to Cloud Deployment

Authors

Yassine Zitouni
Zakariae Jali
Adam Hajjaji

December 14, 2025

Contents

Executive Summary

This report documents the comprehensive implementation of a fully automated CI/CD pipeline for the Spring PetClinic application. The project successfully integrated multiple DevOps tools and technologies including Jenkins, Docker, SonarCloud, and Kubernetes to create a robust deployment automation system.

The implementation followed a systematic approach, starting from local development environment setup, through containerization, to complete pipeline automation with automated testing, code quality analysis, and deployment capabilities. Special emphasis was placed on identifying and solving technical challenges encountered during the implementation process.

Key achievements include:

- Successful containerization of Spring PetClinic application
- Complete Jenkins pipeline automation with multi-stage execution
- Integration of code quality analysis with SonarCloud
- Resolution of complex Docker permission and connectivity issues
- Implementation of automated triggers via GitHub webhooks
- Preparation for Kubernetes deployment

1 Project Overview

1.1 Project Context

The project aimed to implement a complete CI/CD pipeline for the Spring PetClinic application, demonstrating modern DevOps practices. The implementation was conducted on an Ubuntu operating system with incremental commits to GitHub for Jenkins tracking.

1.2 Initial Project Setup

Project Details:

- **Source Application:** Spring PetClinic (GitHub: spring-projects/spring-petclinic)
- **Environment:** Ubuntu Operating System
- **Key Technologies:** Java 17, Maven, Docker, Jenkins, SonarCloud, Kubernetes
- **Plugins Installed:** Pipeline, Docker Pipeline, SonarQube Scanner, Credentials Binding

The project began with selecting a JEE project with comprehensive unit tests. Initial setup involved:

- Maven installation: `sudo apt update && sudo apt install maven -y`
- JDK version adjustment from JDK 25 to JDK 17 for compatibility
- Local compilation verification: `mvn clean compile`
- Local execution: `mvn spring-boot:run`

2 Docker Containerization

2.1 Docker Configuration

The application was containerized using a multi-stage Docker build process to optimize the final image size and ensure proper separation between build and runtime environments.

Listing 2.1: Dockerfile - Multi-stage Build

```
# --- Stage 1: Build the application ---
FROM eclipse-temurin:17-jdk AS build
WORKDIR /app
COPY . .
RUN ./mvnw package -DskipTests

# --- Stage 2: Run the application ---
FROM eclipse-temurin:17-jre
WORKDIR /app
COPY --from=build /app/target/*.jar app.jar
EXPOSE 9090
ENTRYPOINT ["java", "-jar", "app.jar"]
```

2.2 Container Deployment Strategy

- Port mapping configured as 5001:9090 to reserve port 8080 for Jenkins
- Application property modified: `server.port = 9090` in `application.properties`
- Docker commands used for deployment:

```
docker build -t petclinic:latest .
docker run -d -p 5001:9090 --name petclinic-container petclinic:latest
```

3 Jenkins Pipeline Implementation

3.1 Initial Jenkins Setup

Problem #1: Docker Not Found in Jenkins Container
Symptom: `docker -version` inside Jenkins pipeline returned "docker: not found"
Initial Misunderstanding: Believed installing Docker CLI inside container would solve the problem

Solution: Docker Socket Mounting
The root cause was that the Jenkins container couldn't communicate with the host's Docker engine. The solution involved:
1. Recreate Jenkins container with Docker socket mounting:

```
docker run -d --name jenkins \
-p 8080:8080 -p 50000:50000 \
-v jenkins_home:/var/jenkins_home \
-v /var/run/docker.sock:/var/run/docker.sock \
jenkins/jenkins:lts
```

2. Install Docker CLI inside container:

```
# Access as root
docker exec -u root -it jenkins bash

# Install Docker client
apt-get update
apt-get install -y docker.io

# Verify installation
docker --version
```

3.2 Permission Issues Resolution

Problem #2: Permission Denied on Docker Socket
Symptom: "permission denied while trying to connect to the Docker daemon socket at unix:///var/run/docker.sock"
Cause: Jenkins user inside container lacked permission to access the mounted socket due to group mismatch

Solution: User Group Management
Multiple attempts were made before finding the correct solution:
Attempted (Failed) Solutions:

- Running docker as root (not practical in pipeline)
- Installing docker binary only (incomplete without group membership)
- Using su inside container (failed due to disabled root authentication)

Final Working Solution:

1. Enter container as root: `docker exec -u root -it jenkins bash`
2. Add jenkins user to docker group: `usermod -aG docker jenkins`
3. Verify group membership: `id jenkins`
4. Restart container: `docker restart jenkins`
5. If still blocked (optional): `sudo chmod 666 /var/run/docker.sock`

4 DockerHub Integration

4.1 DockerHub Configuration

Problem #3: Missing DockerHub Credentials
Symptom: Jenkinsfile referenced credential ID 'docker-hub' but Jenkins didn't have it configured
Error: Pipeline failed during Docker push stage

Solution: Credentials Configuration

1. Create Docker Hub repository: `yassinezitouni29/petclinic`
2. In Jenkins: Add Credentials → Username with password
3. Configure credential ID as 'docker-hub'
4. Update Jenkinsfile with correct credential ID
5. Rebuild pipeline → successful image push

5 Automation with GitHub Webhooks

5.1 Webhook Configuration

Problem #4: Dynamic Jenkins URL with ngrok
Symptom: Free ngrok URLs change on every restart
Impact: GitHub webhook becomes invalid after Jenkins/ngrok restart

Solution: Webhook Management Process

1. Expose Jenkins to internet: `ngrok http 8080`
2. Copy HTTPS URL from ngrok output
3. Configure GitHub webhook:
   - GitHub → Settings → Webhooks → Add webhook
   - Payload URL: `https://<ngrok-url>/github-webhook/`
   - Content type: application/json
   - Events: Just the push event
4. Update webhook URL whenever ngrok restarts

5.2 Daily Workflow

The established daily workflow ensures continuous automation:

1. Start ngrok and obtain new URL
2. Update GitHub webhook if URL changed
3. Push code to GitHub repository
4. Automatic trigger of Jenkins pipeline
5. Full CI/CD execution: build → test → analysis → dockerize → deploy

6 Complete CI/CD Pipeline

6.1 Pipeline Implementation Overview

The CI/CD pipeline follows a comprehensive multi-stage approach that automates the entire software delivery process. Each stage serves a specific purpose in the workflow:

1. **Checkout:** Automatically clones the project repository from GitHub
2. **Test:** Executes Maven tests to validate code functionality and catch regressions
3. **Package:** Compiles the application and builds the JAR file using Maven
4. **SonarCloud Analysis:** Performs code quality analysis and generates security reports
5. **Docker Build:** Creates a Docker image tagged with the build number
6. **Docker Push:** Pushes the Docker image to Docker Hub registry
7. **Kubernetes Deploy:** Deploys the latest image to Kubernetes cluster
8. **Finished:** Final verification and pipeline completion

The pipeline is triggered automatically via GitHub webhooks whenever code is pushed to the repository. Each stage builds upon the previous one, ensuring that only successfully tested and analyzed code progresses through the deployment pipeline. The use of build numbers ensures version tracking and enables rollback capabilities if needed.

7 SonarCloud Code Quality Analysis

7.1 Initial Setup and Problems

Problem #5: SonarCloud Organization Error
Symptom: "No organization with key 'YassineZitouni29'"
Cause: Missing organization configuration in SonarCloud

Solution: Organization and Token Configuration

1. Create organization in SonarCloud:
   - Go to SonarCloud → "Create Organization"
   - Choose "Analyze GitHub repository"
   - Organization name: `yassinezitouni29`
2. Generate authentication token:
   - SonarCloud → My Account → Security → Generate Token
3. Add token to Jenkins credentials:
   - Credential ID: `sonar-token`
   - Type: Secret Text
   - Value: Generated SonarCloud token

7.2 SonarScanner Implementation

Listing 7.1: SonarCloud Analysis Stage

```
stage('SonarCloud Analysis') {
    steps {
        withSonarQubeEnv('SonarCloud') {
            sh '''
            /var/jenkins_home/tools/sonar-scanner/bin/sonar-scanner \
            -Dsonar.projectKey=YassineZitouni29_spring-petclinic \
```

```
            -Dsonar.organization=yassinezitouni29 \
            -Dsonar.sources=. \
            -Dsonar.host.url=https://sonarcloud.io
            '''
        }
    }
}
```

7.3 Additional SonarCloud Issues

Problem #6: Automatic vs CI Analysis Conflict
Symptom: "You are running CI analysis while Automatic Analysis is enabled"
Impact: Analysis failures due to conflicting analysis modes

Solution: Configuration Adjustment

1. Disable Automatic Analysis in SonarCloud project settings:
    - Project Settings → Analysis → Automatic Analysis → Disable

Problem #7: Missing Compiled Classes
Symptom: "Please provide compiled classes with sonar.java.binaries property"
Cause: SonarScanner requires .class files for Java analysis

Solution: Build Stage Reordering

1. Ensure Maven build runs before SonarCloud analysis
2. Add explicit build stage:

```
stage('Build') {
    steps {
        sh 'mvn clean package'
    }
}
```

3. This produces `target/classes` directory needed by SonarScanner

7.4 SonarCloud Analysis Results

After successfully integrating SonarCloud with our CI/CD pipeline, we obtained comprehensive code quality analysis results. The SonarCloud dashboard provided detailed insights into various aspects of our codebase:

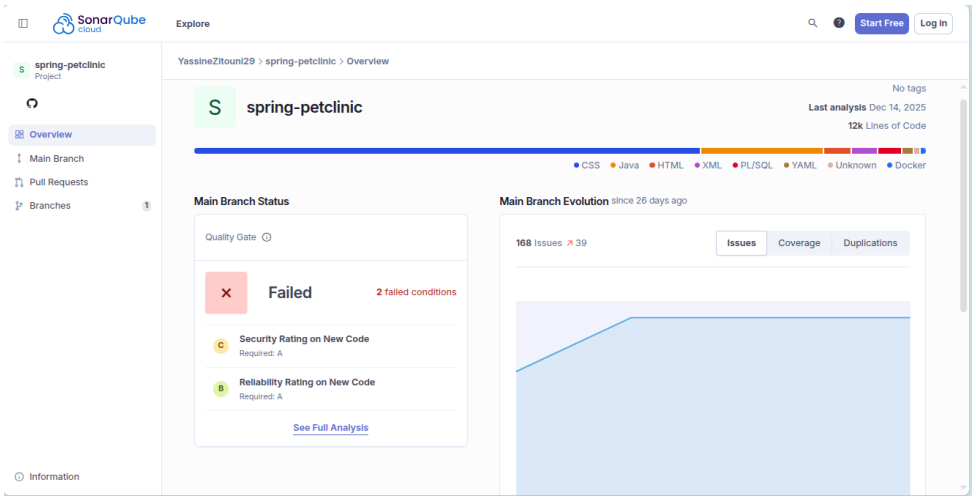

Figure 7.1: SonarCloud Main Dashboard showing overall project status

7.4.1 Security Analysis Results

The security analysis revealed several critical issues that needed attention:

Key security findings included:

- **Blocker Issue:** Critical vulnerability requiring immediate attention
- **Medium Severity Issues:** Configuration vulnerabilities in Kubernetes manifests
- **Service Account Security:** RBAC configuration problems requiring proper binding

7.4.2 Reliability Issues

The reliability analysis focused on code stability and resource management:

Main reliability concerns:

- **Resource Requests Missing:** Kubernetes containers without proper CPU/memory requests
- **Storage Configuration:** Missing storage requests affecting performance
- **Container Specifications:** Incomplete resource definitions impacting cluster stability

7.4.3 Maintainability Issues

Maintainability analysis examined code structure and best practices:

Maintainability findings:

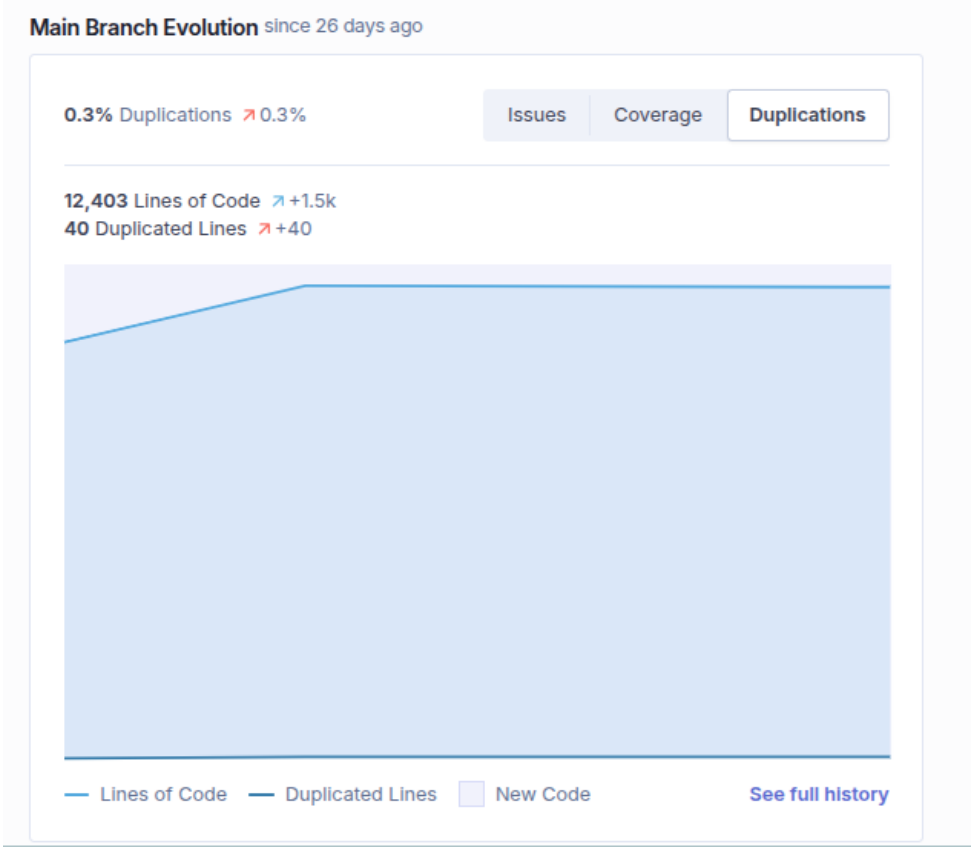

Figure 7.2: Main Branch Evolution showing code quality metrics over time

- **Code Smells:** Various maintainability issues requiring refactoring
- **Technical Debt:** 6 days 7 hours of estimated effort to fix all issues
- **Kubernetes Best Practices:** Configuration improvements needed for production readiness

7.4.4 Code Coverage Results

The test coverage analysis provided insights into our testing effectiveness:

Coverage metrics:

- **Overall Coverage:** 34.8% test coverage
- **Lines to Cover:** 927 lines requiring test coverage
- **Coverage Trend:** Monitoring needed to improve test effectiveness

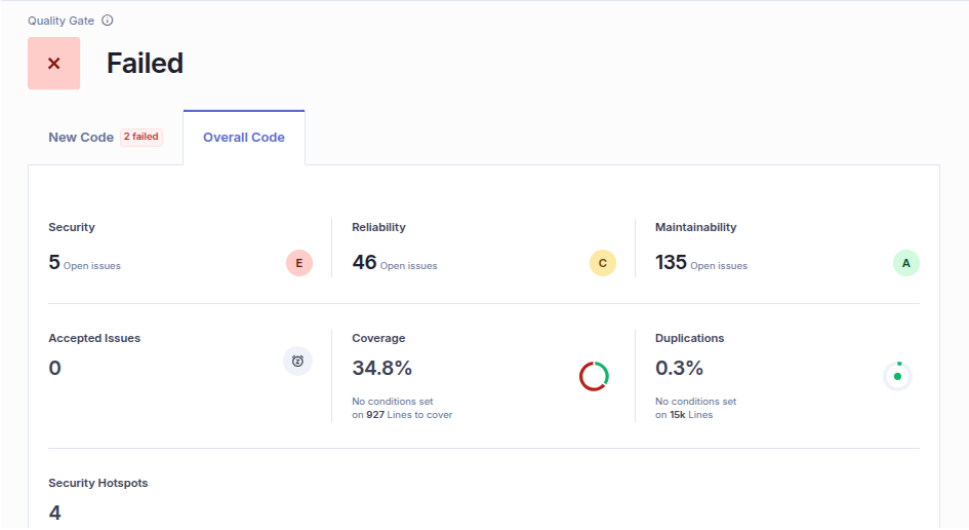

Figure 7.3: New Code Analysis showing failed conditions and overall metrics

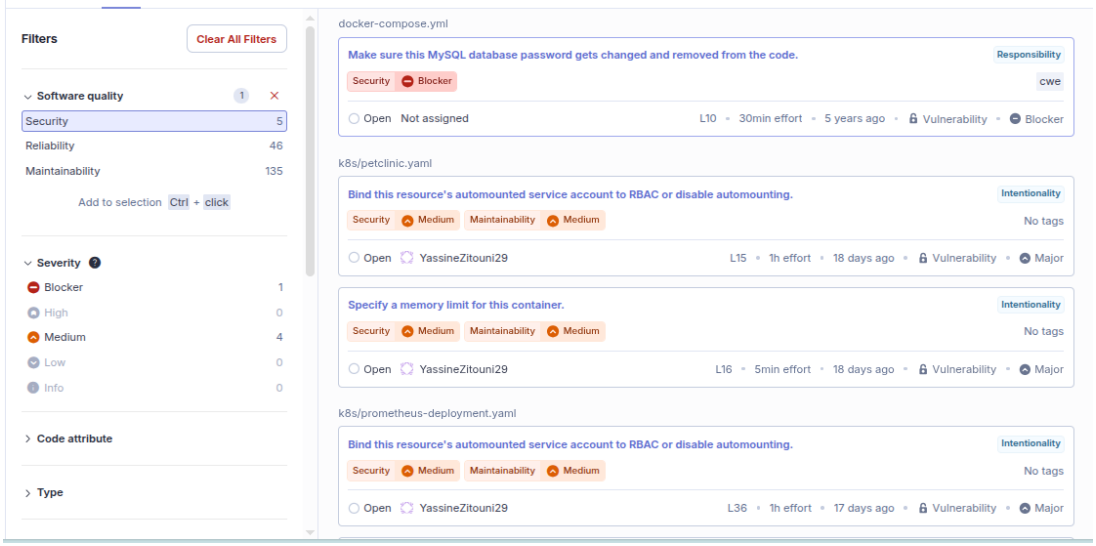

Figure 7.4: Security analysis showing vulnerabilities and their severity levels

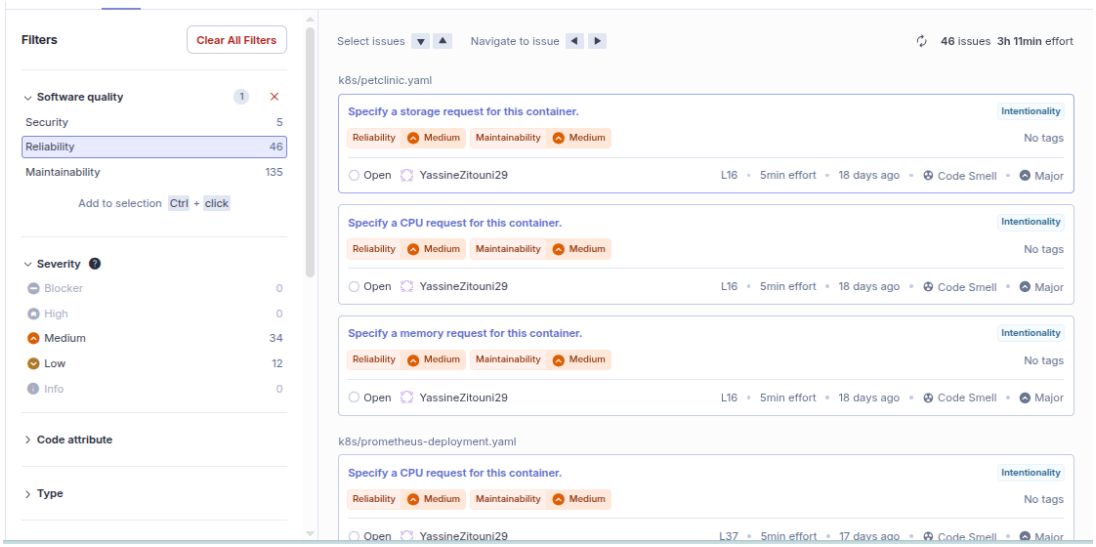

Figure 7.5: Reliability analysis showing resource management issues

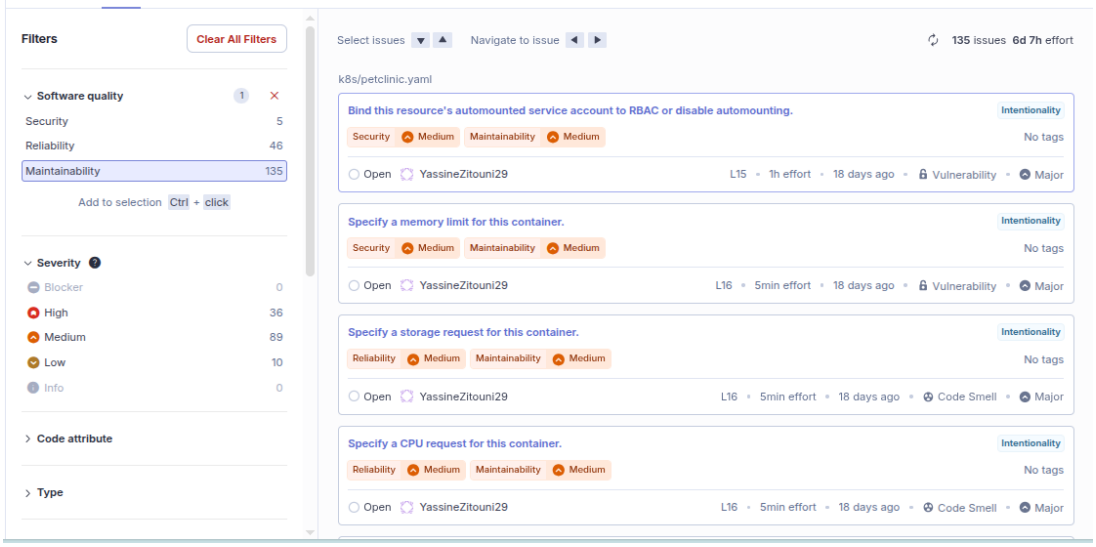

Figure 7.6: Maintainability analysis showing code smells and technical debt

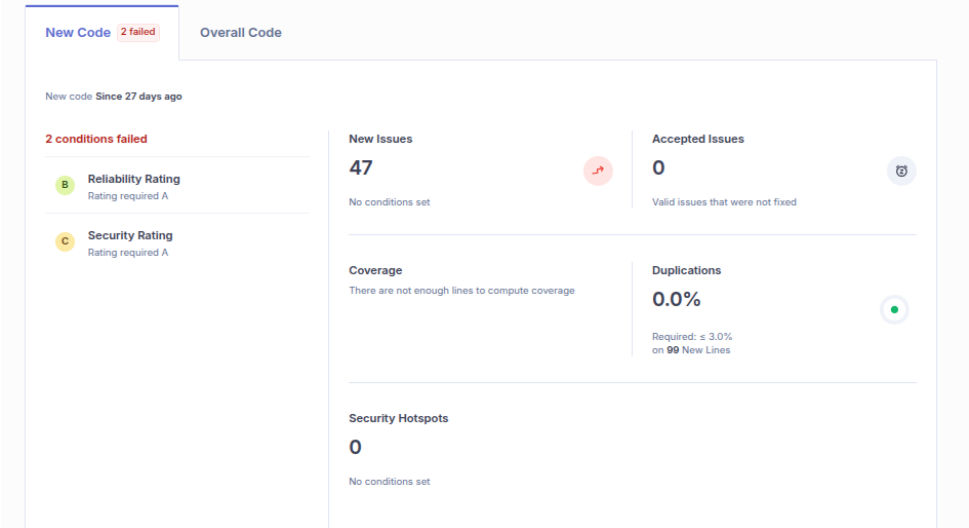

Figure 7.7: Code coverage analysis showing test effectiveness metrics

8 Kubernetes Deployment Setup

8.1 Installing Kubernetes Tools

- Install `kubectl` for cluster management
- Install Minikube for local Kubernetes cluster

Listing 8.1: Kubernetes Tools Installation

```
# Install kubectl
sudo apt install -y kubectl

# Install Minikube
curl -LO https://storage.googleapis.com/minikube/releases/latest/minikube-linux-
    amd64
sudo install minikube-linux-amd64 /usr/local/bin/minikube

# Verify installation
kubectl version --client
minikube version
```

8.2 Starting the Cluster

Key Issue: Cluster Unreachable / API Refused
Symptom: `kubectl` commands fail with connection errors
Cause: Minikube cluster not properly initialized

Solution: Cluster Restart

1. Start Minikube with Docker driver:

   ```
   minikube start --driver=docker
   ```

2. If issues persist:

   ```
   minikube delete
   minikube start --driver=docker
   ```

8.3 Verifying Cluster Health

- Check node status and availability
- Monitor system pods for proper initialization

Listing 8.2: Cluster Health Verification

```
kubectl get nodes
kubectl get pods -A
```

8.4 Deploying the Application

- Apply Deployment YAML configuration
- Kubernetes automatically creates and manages pods

Listing 8.3: Application Deployment

```
kubectl apply -f deployment.yaml
kubectl get pods
```

8.5 Exposing the Application

- Create Service to expose deployment
- Use NodePort type for external access
- Access application through Minikube service

Listing 8.4: Service Exposure

```
kubectl expose deployment petclinic --type=NodePort --port=8080
kubectl get svc
minikube service petclinic
```

8.6 Jenkins to Kubernetes Automation

8.6.1 Jenkins Setup for Kubernetes

Problem: Jenkins Kubernetes Integration
Symptom: Attempted to install Kubernetes directly in Jenkins container but it didn't work as expected
Approach: Created a new container with proper configuration

Solution: Containerized Jenkins with Kubernetes Access

1. Ran Jenkins in a Docker container:

   ```
   docker run -p 8080:8080 jenkins/jenkins:lts
   ```

2. Installed required plugins: Git, Docker, Kubernetes
3. Mounted Docker socket for container access
4. Installed kubectl inside Jenkins container

8.6.2 Jenkins Kubernetes Access Configuration

- Gave Jenkins access to the Kubernetes cluster
- Used Minikube's kubeconfig for authentication

Listing 8.5: Kubernetes Access Verification

```
kubectl get nodes
kubectl get pods -A
```

Problem: Connection Refused
Symptom: Jenkins pipelines failed with "connection refused" errors
Cause: Jenkins couldn't reach the Kubernetes cluster

Solution: Kubeconfig Configuration

1. Copied `~/.kube/config` into Jenkins container
2. Verified cluster reachability from Jenkins container
3. Configured Jenkins credentials for Kubernetes access

8.6.3 Complete CI/CD Pipeline with Kubernetes Deployment

The final Jenkinsfile implements the complete CI/CD pipeline including Kubernetes deployment:

Listing 8.6: Complete Jenkinsfile with Kubernetes Deployment

```
pipeline {
    agent any

    stages {
        stage('Checkout') {
            steps {
                echo 'Project cloned automatically'
            }
        }

        stage('Test') {
            steps {
                echo 'Running tests...'
                sh './mvnw test'
            }
        }

        stage('Package') {
            steps {
                echo 'Compiling and packaging the app...'
                sh './mvnw clean package'
            }
        }

        stage('SonarCloud Analysis') {
            steps {
                withSonarQubeEnv('SonarCloud') {
                    sh """
                        ${tool('SonarScanner')}/bin/sonar-scanner \
                        -Dsonar.projectKey=YassineZitouni29_spring-petclinic \
                        -Dsonar.organization=yassinezitouni29 \
                        -Dsonar.sources=. \
                        -Dsonar.java.binaries=target/classes \
                        -Dsonar.host.url=https://sonarcloud.io
                    """
                }
            }
        }

        stage('Docker Build') {
            steps {
                echo 'Building Docker image...'
                sh "docker build -t petclinic:${BUILD_NUMBER} ."
            }
        }

```

```
        stage('Docker Push') {
            steps {
                echo 'Pushing Docker image to Docker Hub...'
                withCredentials([
                    usernamePassword(
                        credentialsId: 'dockerhub',
                        usernameVariable: 'DOCKER_USER',
                        passwordVariable: 'DOCKER_PASS'
                    )
                ]) {
                    sh "echo \$DOCKER_PASS | docker login -u \$DOCKER_USER --
                        password-stdin"
                    sh "docker tag petclinic:${BUILD_NUMBER} \$DOCKER_USER/petclinic
                        :${BUILD_NUMBER}"
                    sh "docker push \$DOCKER_USER/petclinic:${BUILD_NUMBER}"
                }
            }
        }

        stage('Kubernetes Deploy') {
            steps {
                echo "Deploying latest image to Kubernetes..."
                withCredentials([
                    usernamePassword(
                        credentialsId: 'dockerhub',
                        usernameVariable: 'DOCKER_USER',
                        passwordVariable: 'DOCKER_PASS'
                    )
                ]) {
                    sh """
                        kubectl set image deployment/petclinic \
                            petclinic=$DOCKER_USER/petclinic:${BUILD_NUMBER}
                    """
                }
            }
        }

        stage('Finished') {
            steps {
                echo 'CI pipeline completed successfully!'
            }
        }
    }
}
```

8.6.4 Automated Deployment Workflow

1. **Code Push:** Developer pushes code to GitHub repository
2. **Webhook Trigger:** GitHub webhook triggers Jenkins pipeline

3. **Build Phase:** Jenkins builds and tests the application
4. **SonarCloud Analysis:** Performs code quality and security scanning
5. **Docker Phase:** Creates Docker image and pushes to Docker Hub
6. **Kubernetes Phase:** Automatically updates deployment with new image
7. **Rolling Update:** Kubernetes performs rolling update with zero downtime
8. **Verification:** Checks deployment status and pod health

The pipeline uses the `kubectl set image` command to perform a rolling update of the existing deployment, ensuring minimal downtime during deployment. The build number is used to tag Docker images, providing version tracking and rollback capabilities.

Final Result: Complete CI/CD Automation

- **Local Kubernetes cluster** running successfully
- **Application deployed and managed** by Kubernetes
- **Service exposed** for external access
- **Complete automation:** Code push → Build → Test → Analyze → Dockerize → Deploy
- **Jenkins handles automation**, Kubernetes handles deployment, Docker serves as the bridge
- **Once access and credentials were fixed**, the entire pipeline became smooth and automated

9 Prometheus & Grafana Monitoring Setup

9.1 Monitoring Namespace Creation

The first step in setting up monitoring is to create a dedicated namespace for monitoring tools to keep them organized and separated from application deployments.

Listing 9.1: Create Monitoring Namespace

```
kubectl create namespace monitoring
```

9.2 Prometheus Deployment

Prometheus is deployed as the metrics collection and storage system. It scrapes metrics from various sources including Kubernetes components and applications.

Listing 9.2: Deploy Prometheus

```
kubectl apply -f prometheus.yaml
```

9.3 Grafana Deployment

Grafana is deployed as the visualization layer, providing dashboards and graphs for monitoring data collected by Prometheus.

Listing 9.3: Deploy Grafana

```
kubectl apply -f grafana.yaml
```

9.4 Monitoring Pods Status

After deploying both Prometheus and Grafana, verify that all pods are running correctly in the monitoring namespace.

Listing 9.4: Check Monitoring Pods

```
kubectl get pods -n monitoring
```

9.5 Exposing Monitoring Services

Expose the monitoring services to make them accessible from outside the cluster for visualization and management.

Listing 9.5: Expose Monitoring Services

```
kubectl get svc -n monitoring
```

9.6 Accessing Monitoring Services via Minikube

Use Minikube to access the monitoring services through the web browser.

Listing 9.6: Access Services via Minikube

```
minikube service prometheus -n monitoring
minikube service grafana -n monitoring
```

9.7 Configuring Grafana Data Source

Configure Prometheus as the data source in Grafana to enable visualization of collected metrics.

Grafana Data Source Configuration:

- Navigate to Grafana web interface (accessed via Minikube)
- Go to Configuration → Data Sources
- Click "Add data source"
- Select "Prometheus"
- Set URL to: `http://prometheus:9090`
- Click "Save & Test" to verify connection

9.8 Importing Grafana Dashboards

Import pre-built Grafana dashboards for Kubernetes monitoring to quickly visualize cluster health and performance metrics.

Dashboard Import Steps:

- In Grafana, go to Dashboard → Import
- Use dashboard IDs from Grafana dashboard repository
- Recommended dashboards:
  - Kubernetes cluster monitoring (ID: 3119)
  - Node Exporter Full (ID: 1860)
  - Prometheus 2.0 Overview (ID: 3662)
- Configure data source as Prometheus
- Save dashboard for ongoing monitoring

9.9 Monitoring Integration Benefits

- **Real-time Metrics:** Monitor cluster CPU, memory, and network usage
- **Application Performance:** Track application response times and error rates
- **Resource Optimization:** Identify underutilized or overloaded resources
- **Alerting:** Set up alerts for critical conditions
- **Historical Analysis:** Analyze trends and patterns over time
- **Capacity Planning:** Make informed decisions about scaling and resource allocation

9.10 Grafana Dashboard Monitoring Results

After successfully setting up Prometheus and Grafana, we obtained comprehensive monitoring insights into our Kubernetes cluster's performance. The Grafana dashboards provided real-time visibility into various system metrics:

9.10.1 Memory Usage Analysis

The memory usage dashboard showed consistent memory consumption patterns across our Kubernetes nodes:

Key observations:

- **Stable Memory Consumption:** Memory usage remained consistently between 5.7-6.25 GB
- **Two Monitoring Sources:** Data collected from both Kubernetes pods and node exporter services

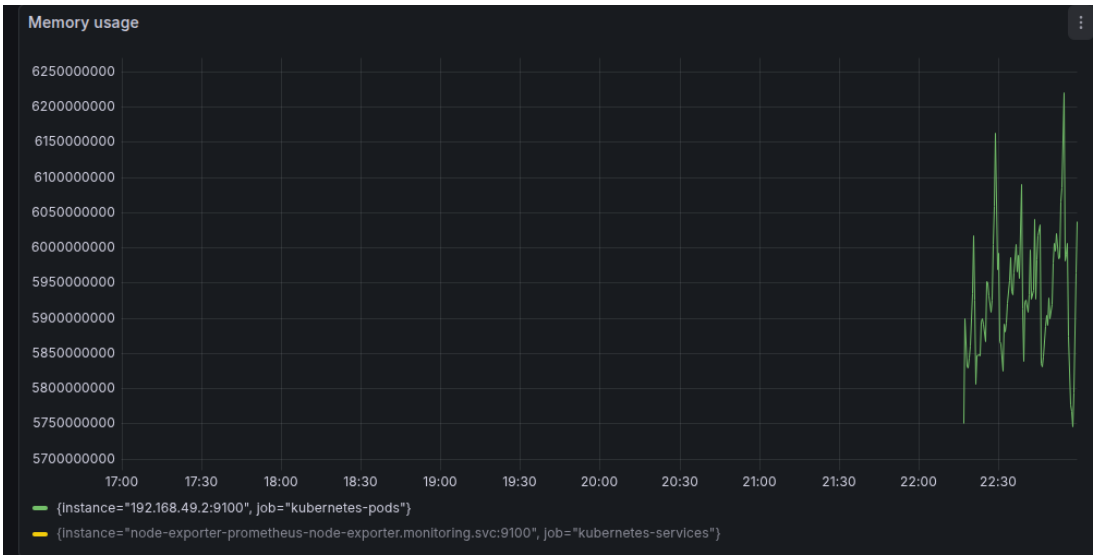

Figure 9.1: Memory usage patterns showing stable consumption between 5.7-6.25 GB over time

- **Time-based Patterns:** Monitoring data captured from 17:00 to 22:30 showing system stability
- **No Memory Spikes:** Consistent patterns indicate stable application performance

9.10.2 CPU Usage Monitoring

The CPU usage dashboard revealed efficient resource utilization across our cluster:

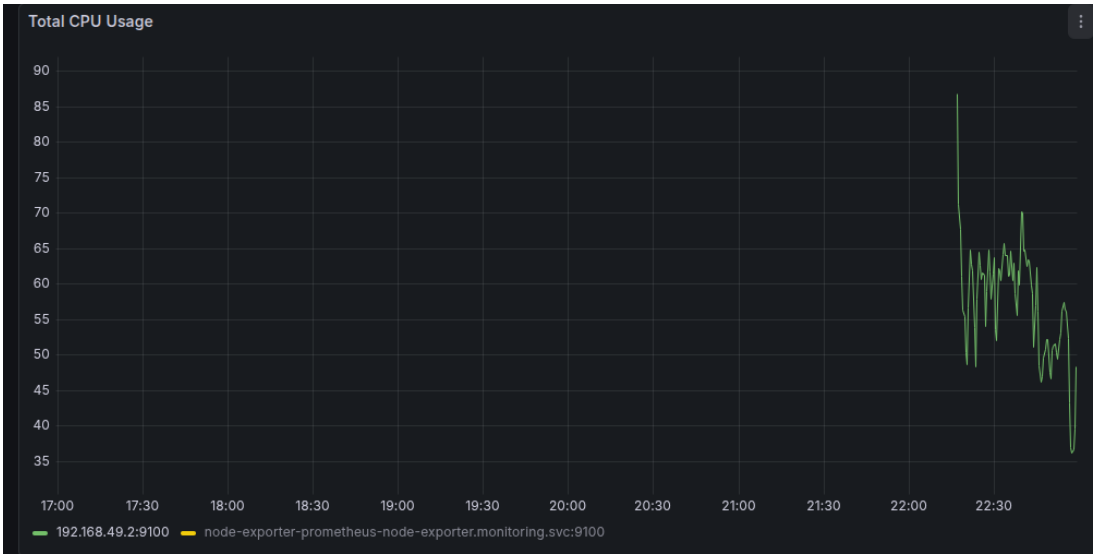

Figure 9.2: Total CPU usage showing efficient resource utilization between 35-90%

CPU performance insights:

- **Efficient Utilization:** CPU usage ranged from 35% to 90%, indicating good resource allocation

- **Monitoring Sources:** Data from both node IP and service endpoints
- **Peak Periods:** Higher CPU usage observed during specific time intervals
- **Resource Optimization:** Usage patterns suggest opportunities for resource optimization

9.10.3 Scrape Duration Analysis

The scrape duration dashboard provided insights into monitoring system performance:

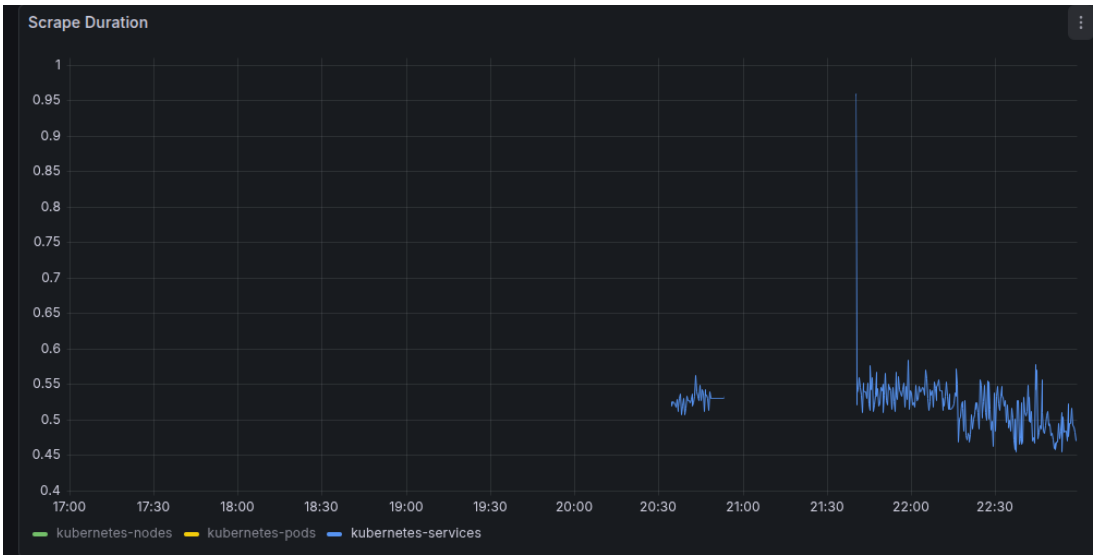

Figure 9.3: Scrape duration metrics showing efficient data collection performance

Scraping efficiency:

- **Multiple Targets:** Monitoring data collected from kubernetes-nodes, kubernetes-pods, and kubernetes-services
- **Efficient Collection:** Scrape durations remained between 0.4-1.0 seconds
- **Consistent Performance:** Stable scraping times indicate well-configured monitoring
- **Three Monitoring Streams:** Simultaneous data collection from different Kubernetes components

9.10.4 Disk Usage Analysis

The disk usage dashboard provided insights into storage utilization across the cluster:

Disk usage insights:

- **Storage Monitoring:** Comprehensive tracking of disk space utilization
- **Capacity Planning:** Data for informed decisions about storage allocation
- **Performance Indicators:** Disk I/O and space usage metrics
- **Cluster Health:** Important indicator of overall system stability

Figure 9.4: Disk usage patterns showing storage consumption over time

9.10.5 Monitoring Integration Benefits

The comprehensive monitoring setup provided several key benefits:

- **Real-time Metrics:** Continuous monitoring of cluster CPU, memory, disk, and performance metrics
- **Performance Optimization:** Data-driven insights for resource allocation and scaling decisions
- **Proactive Issue Detection:** Early identification of performance degradation patterns
- **Capacity Planning:** Historical data analysis for informed infrastructure planning
- **Multi-source Monitoring:** Comprehensive view combining pod-level and service-level metrics
- **Time-based Analysis:** Understanding of system behavior across different time periods

10 Technical Specifications and Results

10.1 System Specifications

Component	Specification
Operating System	Ubuntu
Java Version	JDK 17
Build Tool	Apache Maven
Container Platform	Docker
CI/CD Server	Jenkins LTS
Code Quality Platform	SonarCloud
Container Registry	Docker Hub
Orchestration Platform	Kubernetes (Minikube)
Monitoring Platform	Prometheus & Grafana
Application Port	9090
Jenkins Port	8080
Docker Container Port	5001:9090

Table 10.1: Technical Specifications

10.2 Summary of Problems and Solutions

Problem	Symptom	Solution
Docker in Jenkins	docker: not found	Mount docker.sock + install CLI
Socket Permission	permission denied	Add jenkins to docker group
DockerHub Credentials	Missing credential	Configure Jenkins credentials
Dynamic ngrok URLs	Webhook invalid	Update webhook on restart
SonarCloud Org	No organization	Create organization in SonarCloud
Analysis Conflict	CI vs Automatic	Disable automatic analysis
Missing Classes	No .class files	Reorder stages, build first
Virtualization Disabled	Cluster creation fails	Enable in BIOS settings
Cluster Unreachable	API connection refused	Restart Minikube cluster
Kubernetes Access	Connection refused	Copy kubeconfig to Jenkins

Table 10.2: Problems and Solutions Summary

11 Conclusion

11.1 Project Achievements

The project successfully implemented a complete CI/CD pipeline with the following accomplishments:

- **End-to-End Automation:** Full automation from code commit to deployment
- **Docker Integration:** Successful containerization with multi-stage builds
- **Jenkins Pipeline:** Robust multi-stage pipeline with error handling
- **Code Quality:** Integrated SonarCloud for continuous code analysis
- **Permission Resolution:** Solved complex Docker permission issues
- **Webhook Automation:** Implemented GitHub-triggered builds
- **Kubernetes Deployment:** Successfully deployed to local Kubernetes cluster
- **Complete CI/CD:** Automated deployment from Jenkins to Kubernetes
- **Monitoring Setup:** Implemented Prometheus & Grafana for observability

11.2 Final Pipeline Output

The implemented pipeline produces:

- **Build Artifacts:** Compiled JAR files
- **Docker Images:** Version-tagged container images
- **Test Reports:** Maven test execution results
- **Code Quality:** SonarCloud analysis reports
- **Kubernetes Deployment:** Application running in Kubernetes cluster
- **Monitoring Data:** Performance metrics in Prometheus & Grafana
- **Automated Deployment:** Zero-touch deployment to Kubernetes