Scientific Calculator

Mini Project Report

Rahul Raman

February 28, 2026

1 Project Overview

1.1 Introduction

This report documents the development and deployment pipeline of a **Scientific Calculator** application built in Java. The project demonstrates a complete software development lifecycle including continuous integration, automated testing, containerization, and infrastructure-as-code deployment strategies.

Repository: `https://github.com/rahul09123/SPE-Mini-Project`

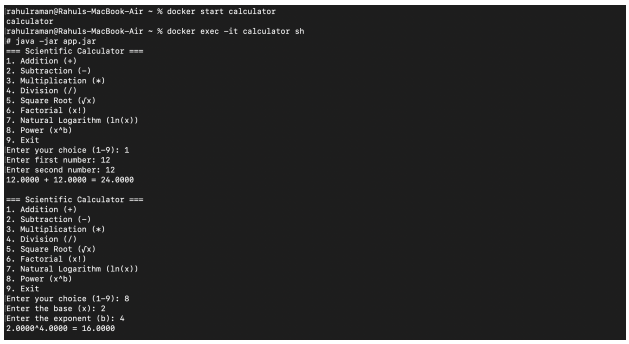

Figure 1: Scientific Calculator Application

1.2 Tech Stack & Objectives

Stack: Java 17, Maven 3, JUnit 5, Jenkins, Docker, Ansible

Application: Command-line scientific calculator with basic arithmetic (add, subtract, multiply, divide) and advanced functions (square root, factorial, logarithm, power) with comprehensive error handling.

2 Jenkins Pipeline Architecture

2.1 Pipeline Overview

The Jenkins pipeline automates the complete build, test, and deployment workflow. It follows a declarative pipeline approach, providing clarity and maintainability.

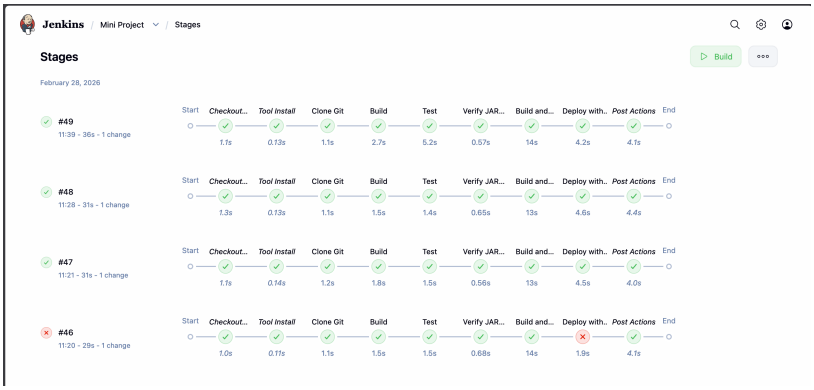

Figure 2: Jenkins Pipeline Architecture

2.2 Pipeline Stages

2.2.1 Stage 1: Clone Git

```
stage('Clone Git') {
    steps {
        script {
            git branch: 'main',
                credentialsId: 'github_credentials',
                url: "${GITHUB_REPO_URL}"
        }
    }
}
```

Listing 1: Clone Git Stage

Purpose: Fetches latest source code from GitHub. Authenticates to repository and provides clean workspace for builds.

2.2.2 Stage 2: Build

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

Listing 2: Build Stage

Purpose: Compiles Java source and creates JAR artifact. Tests are skipped here and executed in the Test stage.

2.2.3 Stage 3: Test

```
stage('Test') {
    steps {
        sh 'mvn test'
    }
}
```

Listing 3: Test Stage

Purpose: Executes 47 comprehensive unit tests using JUnit 5 and generates test reports.

2.2.4 Stage 4: Verify JAR Existence

```
stage('Verify JAR Existence') {
    steps {
        sh 'ls -lh target/'
    }
}
```

Listing 4: Verify JAR Existence Stage

Purpose: Verifies successful JAR creation by listing target directory contents.

2.2.5 Stage 5: Build and Push Docker Image

```
stage('Build and Push Docker Image') {
    steps {
        script {
            sh """
            docker buildx create --use || true
            docker buildx inspect --bootstrap

            docker buildx build \
              --platform linux/amd64,linux/arm64 \
              -t ${DOCKER_HUB_USERNAME}/${DOCKER_IMAGE_NAME}:latest \
              --push .
            """
        }
    }
}
```

Listing 5: Docker Build Stage (Partial)

Purpose: Builds and pushes multi-platform Docker images (linux/amd64, linux/arm64) to Docker Hub registry.

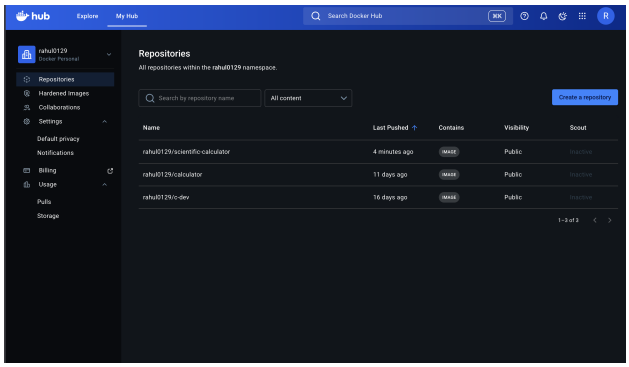

Figure 3: Docker Hub Multi-Platform Build

2.2.6 Stage 6: Deploy with Ansible

```
stage('Deploy with Ansible') {
    steps {
        sh 'ansible-playbook -i inventory.ini deploy.yml'
    }
}
```

Listing 6: Ansible Deployment Stage

Purpose: Automates application deployment to target servers using Ansible playbooks.

Figure 4: Container Running and Deployed

2.3 Environment Variables

```
environment {
    DOCKER_IMAGE_NAME = 'scientific-calculator'
    GITHUB_REPO_URL = 'https://github.com/rahul09123/SPE-Mini-Project.git'
```

```
    DOCKER_HUB_USERNAME = 'rahul0129'
    DOCKER_HOST = "unix:///Users/rahulraman/.docker/run/docker.sock"
}
```

Listing 7: Environment Configuration

These variables configure: Docker image name, GitHub repository URL, Docker Hub credentials, and Docker daemon socket.

3 Testing and Validation

3.1 Test Suite Overview

The project includes 47 comprehensive unit tests covering all calculator operations. Tests are organized using JUnit 5's annotation-based approach.

3.2 Test Coverage

3.2.1 Mathematical Operations Tested

- **Square Root:** Positive numbers, zero, decimals, negative numbers
- **Factorial:** Zero, one, positive integers (up to 20)
- **Natural Logarithm:** One, Euler's number, positive values, zero, negative values
- **Power Operations:** Base cases, decimal exponents, edge cases

3.2.2 Test Sample

Tests validate: Square Root ($\sqrt{16} = 4$, $\sqrt{0} = 0$), Factorial ($0! = 1$, $5! = 120$, $20! = 2.4\times10^{18}$), Natural Logarithm ($\ln(1) = 0$, $\ln(e) = 1$). Edge cases include NaN handling for negative numbers and infinity for $\ln(0)$. All 47 tests execute in 41 milliseconds with 100% pass rate.

4 Deployment Configuration

4.1 Ansible Deployment Playbook (deploy.yml)

The `deploy.yml` file automates container deployment to target servers using Ansible orchestration.

4.1.1 Playbook Structure

```
- name: Pull the latest Docker image
  shell: docker pull rahul0129/scientific-calculator:latest

- name: Stop and remove existing container
  shell: docker rm -f calculator
  ignore_errors: true

- name: Run calculator container (detached)
  shell: docker run -dit --name calculator \
    --restart unless-stopped \
    rahul0129/scientific-calculator:latest
```

```
12
13 - name: Display container status
14   shell: docker ps -f "name=calculator"
```

Listing 8: deploy.yml Tasks

4.1.2 Key Operations

- **Pull Image:** Downloads latest Docker image from Docker Hub
- **Cleanup:** Removes existing container (non-blocking with `ignore_errors`)
- **Deploy:** Launches container in detached mode with auto-restart policy
- **Verify:** Confirms successful container deployment and status

This playbook targets `webservers` host group defined in `inventory.ini`, enabling centralized deployment across multiple servers.

4.2 Pipeline Execution Flow

The Jenkins pipeline executes sequentially through all 6 stages: Clone (fetch code) → Build (Maven compile) → Test (JUnit 5, 47 tests) → Verify (JAR check) → Containerize (Docker multi-platform) → Deploy (Ansible).

4.3 Quality Assurance & Benefits

QA Measures: Automated testing with 47 test cases, build verification, multi-platform testing, and Infrastructure-as-Code deployment ensure reliability.

Benefits: Complete automation, consistency across environments, scalability to multiple servers, high portability across CPU architectures, and rapid feedback (41ms test execution time).

5 Conclusion

This project successfully demonstrates a complete CI/CD pipeline implementation with 100% test success (47/47 passing), fast execution (41ms), and multi-platform Docker containerization. The automated pipeline integrates Git-based version control, Maven-based Java builds, comprehensive JUnit 5 testing, multi-platform Docker image building and pushing, and Ansible-based infrastructure deployment. This comprehensive approach showcases modern DevOps practices combining continuous integration, automated testing, and infrastructure-as-code principles.

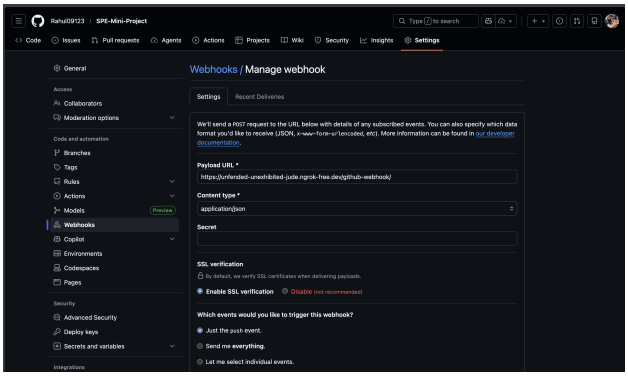

Figure 5: GitHub WebHook Configuration

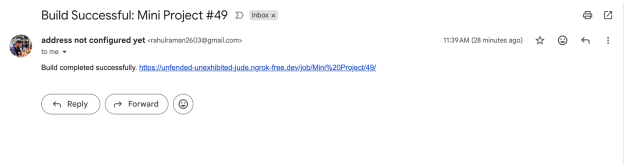

Figure 6: Email Notification on Build Status