

Mini Project Report: Scientific Calculator with a Full DevOps CI/CD Pipeline

Hemang Seth

Roll Number: *IMT2022098*

Github Link: [Scientific-Calculator](#)

Docker Link: [scientific-calculator](#)

October 9, 2025

Contents

1	What and Why of DevOps?	3
1.1	What is DevOps?	3
1.2	Why is DevOps Important?	3
2	The DevOps Toolchain	4
3	Project Implementation: The CI/CD Pipeline	5
3.1	Source Control (Git & GitHub)	5
3.2	Building and Testing (Maven)	5
3.3	Containerization (Docker)	6
3.4	Continuous Integration (Jenkins)	6
3.4.1	GitHub Webhook & ngrok Setup	6
3.5	Deployment (Ansible)	6
3.6	Email Notifications	7
4	Troubleshooting and Challenges Faced	7
4.1	Challenge 1: 'command not found' in Jenkins	7
4.2	Challenge 2: Container Stops After Successful Deployment	8
4.3	Challenge 3: Email Notification Setup	8
5	Stages of the Jenkins Pipeline	9
5.1	Stage 1: Test Java App	9
5.2	Stage 2: Build Java App	9
5.3	Stage 3: Build Docker Image	10
5.4	Stage 4: Push Docker Image to Docker Hub	10
5.5	Stage 5: Deploy via Ansible	10
5.6	Post-Build Actions: Email Notifications	11
6	Steps to Be Followed For Installation	11
6.1	Phase 1: Local Environment Setup and Installation	12
6.2	Phase 2: Project Scaffolding and Code Development	12
6.3	Phase 3: Repository and Service Configuration	12
6.4	Phase 4: Jenkins Configuration	13
6.5	Phase 5: Enabling Automation with Webhooks	14
6.6	Phase 6: Final Pipeline Execution and Verification	14
7	Final Results and Observations	15
8	Conclusion	16
8.1	Manual Execution Steps	17

1 What and Why of DevOps?

1.1 What is DevOps?

DevOps is a modern software development philosophy and set of practices that combines software development (Dev) and IT operations (Ops). The primary goal of DevOps is to shorten the systems development life cycle and provide continuous delivery with high software quality. It emphasizes automation, collaboration, and communication between software developers and other information-technology (IT) professionals while automating the process of software delivery and infrastructure changes.

The core of DevOps is a culture of shared responsibility, transparency, and faster feedback loops. Instead of siloing development and operations teams, DevOps encourages them to work together across the entire application lifecycle, from development and testing through deployment and operations.

1.2 Why is DevOps Important?

In today's fast-paced digital world, businesses need to be able to deliver new features and services to customers quickly and reliably. Traditional software development models often struggle to keep up with these demands, leading to long release cycles, buggy software, and a disconnect between development and operations. DevOps addresses these challenges by offering several key benefits:

- **Speed:** DevOps enables faster delivery of new features and updates by automating the build, test, and deployment processes. This allows organizations to respond more quickly to customer feedback and market changes.
- **Reliability:** By automating testing and deployment, DevOps reduces the risk of human error and ensures that software is delivered with higher quality. Continuous integration and continuous delivery (CI/CD) pipelines ensure that every code change is automatically tested.
- **Improved Collaboration:** DevOps breaks down the barriers between development and operations teams, fostering a culture of collaboration and shared ownership. This leads to better communication and a more efficient workflow.
- **Security:** By integrating security practices into the development pipeline (a practice known as DevSecOps), organizations can identify and address security vulnerabilities early in the development process, rather than waiting until the end.
- **Scalability:** DevOps practices, particularly when combined with cloud computing and containerization, make it easier to manage and scale infrastructure to meet changing demands.

This project implements a complete DevOps pipeline to demonstrate these principles in a practical, hands-on manner.

2 The DevOps Toolchain

For this project, a specific set of industry-standard tools was chosen to create a seamless CI/CD pipeline. Each tool serves a distinct purpose in the automation process.

1. **Source Control Management (Git & GitHub):** Git is a distributed version control system used to track changes in source code. GitHub is a cloud-based hosting service for Git repositories. In this project, GitHub served as the central repository for all code and configuration files, enabling version control and triggering the CI/CD pipeline.
2. **Build Tool (Apache Maven):** Maven is a powerful build automation tool used primarily for Java projects. It manages project dependencies, compiles the source code, runs tests, and packages the application into an executable format (a JAR file).
3. **Continuous Integration (Jenkins):** Jenkins is an open-source automation server that acts as the backbone of our CI/CD pipeline. It was configured to monitor the GitHub repository for changes, and upon detecting a push, it automatically executes a series of predefined stages: building, testing, containerizing, and deploying the application.
4. **Containerization (Docker & Docker Hub):** Docker is a platform for developing, shipping, and running applications in containers. A container packages an application and all its dependencies into a single, standardized unit. Docker Hub is a cloud-based registry for storing and sharing Docker container images. Our Java application was packaged into a Docker image and pushed to Docker Hub for easy distribution.
5. **Configuration Management & Deployment (Ansible):** Ansible is an open-source automation tool for configuration management, application deployment, and task automation. It uses simple, human-readable YAML files (called "playbooks") to define automation jobs. In this project, Ansible was used to automate the final deployment step: pulling the latest Docker image from Docker Hub and running it on the local machine.

3 Project Implementation: The CI/CD Pipeline

This section provides a detailed, step-by-step walkthrough of how the scientific calculator application was developed and integrated into the automated DevOps pipeline. You can refer the Figure 1 for seeing how the flow is done .

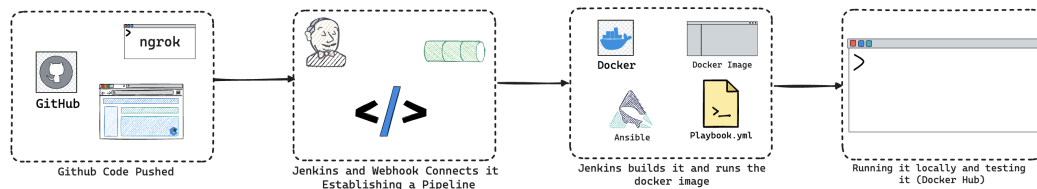


Figure 1: This here contains the entire flow of the project that was asked

3.1 Source Control (Git & GitHub)

All project files, including the Java source code, Maven POM, Dockerfile, Jenkinsfile, and Ansible playbook, were committed to a Git repository and pushed to a public GitHub repository. This acted as the "single source of truth" for the entire project.

Hemang-2004	Done the mail	e0692b0 · 21 minutes ago	41 Commits
idea	Adding Dockerfile and the contents	2 days ago	
ScientificCalculator/idea	Adding Dockerfile and the contents	2 days ago	
src	Changes Final	2 hours ago	
target	Added testing	1 hour ago	
DS_Store	Adding Dockerfile and the contents	2 days ago	
.dockerignore	ansible errors	15 hours ago	
Dockerfile	Done final	1 hour ago	
Jenkinsfile	Done the mail	21 minutes ago	
README.md	Update README.md	13 hours ago	
inventory.ini	Done the env changes	30 minutes ago	
playbook.yml	Done the env changes	30 minutes ago	
pom.xml	Adding Dockerfile and the contents	2 days ago	

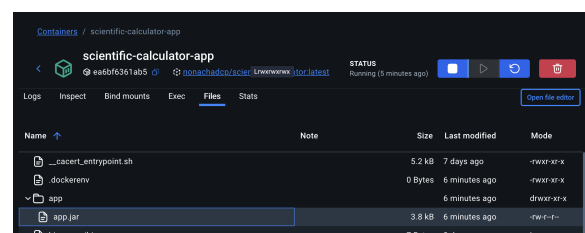
Figure 2: The public GitHub repository containing all project files.

3.2 Building and Testing (Maven)

The project was configured as a Maven project. The `pom.xml` file defines project dependencies (like JUnit) and configures the build process. The command `mvn clean install & mvn test` as shown in Figure 3a was used locally and in the pipeline to compile the code, run the unit tests, and package the application into an executable JAR file.

```
[INFO] -----
[INFO] T E S T S
[INFO] -----
[INFO] Running com.spe.calculator.AppTest
[INFO] Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.018 s -- in com.spe.calculator.AppTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 4, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
```

(a) Testing has been done



(b) Docker container shown in the system

3.3 Containerization (Docker)

A **Dockerfile** was created to define the steps for packaging the Java application into a lightweight, portable Docker image. A multi-stage build was used for efficiency: the first stage builds the JAR using a full Maven image, and the second stage copies only the final JAR into a minimal OpenJDK image. The resulting image was then pushed to a public repository on Docker Hub as shown in Figure 3b

3.4 Continuous Integration (Jenkins)

A Jenkins pipeline was created to automate the entire workflow. A **Jenkinsfile** was added to the root of the repository, defining the CI/CD pipeline as code. The successful execution of this pipeline, showing all stages passing, is shown below.

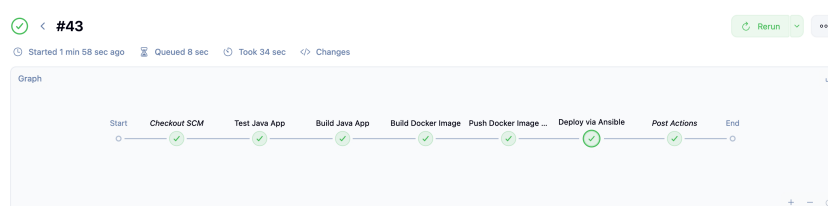
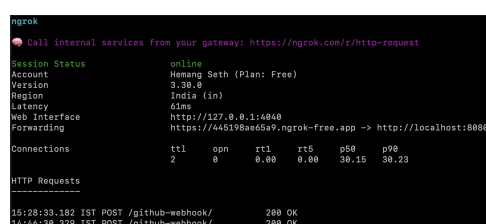


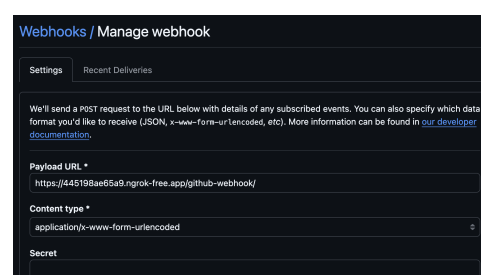
Figure 4: The successful pipeline execution in Jenkins Stage View.

3.4.1 GitHub Webhook & ngrok Setup

To enable GitHub to automatically trigger the Jenkins pipeline, a webhook was configured. Since Jenkins was running on a local machine, it was not accessible from the public internet. The tool **ngrok** was used to create a secure tunnel from a public URL to the local Jenkins server on port 8080. This public ‘ngrok’ URL was then used as the webhook endpoint in GitHub’s settings, allowing GitHub to send POST requests to the local Jenkins instance upon every ‘git push’. as shown in Figure 5a & Figure 5b



(a) The ngrok terminal showing incoming webhook requests from GitHub.

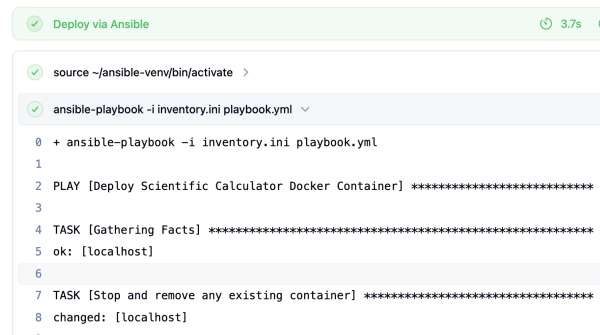


(b) The GitHub webhook configuration used in the setup.

Figure 5: Demonstration of webhook setup using ngrok and GitHub.

3.5 Deployment (Ansible)

An Ansible playbook named **playbook.yml** & **inventory.ini** (was used for marking the environment) automates the deployment process on the local machine. The playbook performs three tasks : (Refer figure 6.)



```

✓ Deploy via Ansible 3.7s
✓ source ~/ansible-venv/bin/activate
✓ ansible-playbook -i inventory.ini playbook.yml
0 + ansible-playbook -i inventory.ini playbook.yml
1
2 PLAY [Deploy Scientific Calculator Docker Container] *****
3
4 TASK [Gathering Facts] *****
5 ok: [localhost]
6
7 TASK [Stop and remove any existing container] *****
8 changed: [localhost]
~

```

Figure 6: This the ansible file for running it

1. Stops and removes any old version of the container.
2. Pulls the ‘latest’ image from Docker Hub to ensure it has the newest version.
3. Starts a new container from the pulled image in interactive mode.

3.6 Email Notifications

The pipeline was configured to send an email notification upon the success or failure of a build. This provides immediate feedback on the status of a code change. The Jenkins system was configured to use Gmail’s SMTP server with a secure App Password for authentication.

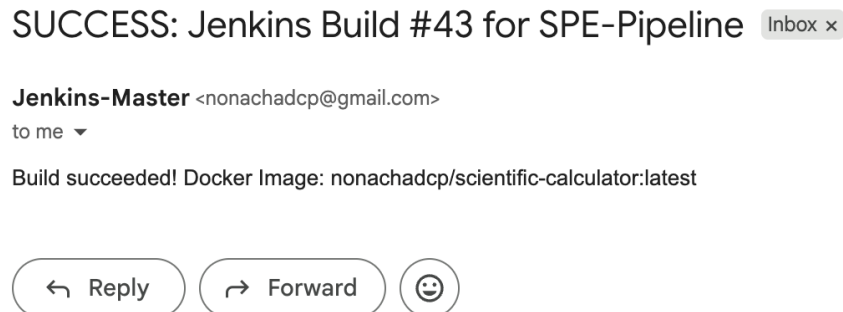


Figure 7: The success notification email received from Jenkins.

4 Troubleshooting and Challenges Faced

Building a CI/CD pipeline from scratch often involves overcoming several common configuration hurdles. This project was no exception. Here are some of the key challenges encountered and their solutions.

4.1 Challenge 1: ‘command not found’ in Jenkins

- **Problem:** The initial pipeline builds failed at the first stage with an error indicating that commands like `mvn` and `docker` were not found, even though they were installed and working in the local terminal.

- **Reason:** Jenkins runs in an isolated environment and does not automatically inherit the system's PATH environment variables. It did not know where to find the executables installed by Homebrew.
- **Solution:** The issue was resolved by configuring the Jenkins "Built-In Node." I navigated to *ManageJenkins* > *Nodes* > *Built – InNode* > *Configure* and added a new environment variable.
 - **Name:** PATH+EXTRA
 - **Value:** /opt/homebrew/bin:/usr/local/bin

After restarting the Jenkins service, Jenkins was able to find all necessary command-line tools.

4.2 Challenge 2: Container Stops After Successful Deployment

- **Problem:** Even after a successful pipeline run, the deployed Docker container would immediately stop. The 'docker ps' command showed an empty list.
- **Reason:** This is a standard Jenkins security feature. When a build step finishes, Jenkins automatically "cleans up" and terminates any background processes spawned by that step to prevent orphaned processes.
- **Solution:** The fix was to prepend a special environment variable to the Ansible command in the 'Jenkinsfile'. This command tells Jenkins to make an exception for this step and leave the spawned processes (our Docker container) running.

4.3 Challenge 3: Email Notification Setup

- **Problem:** Initially, Jenkins was unable to send emails, and the build would fail during the 'post' action, often with authentication errors.
- **Reason:** Sending emails through Gmail's SMTP server requires specific security configurations. Using a regular account password directly is blocked by Google for less secure apps.
- **Solution:** A multi-step process was required:
 1. **Google App Password:** A 16-character App Password was generated from the Google Account's security settings. This special password allows applications like Jenkins to access the account securely.
 2. **Jenkins Plugin:** The "Email Extension Plugin" was installed in Jenkins.
 3. **Jenkins System Configuration:** In *ManageJenkins* > *System*, the "Extended E-mail Notification" section was configured with Gmail's SMTP details ('smtp.gmail.com', port '465', with SSL enabled).
 4. **Jenkins Credentials:** A new "Username with password" credential was created in Jenkins, using the Gmail address as the username and the 16-character App Password as the password. This credential was then selected in the email configuration.

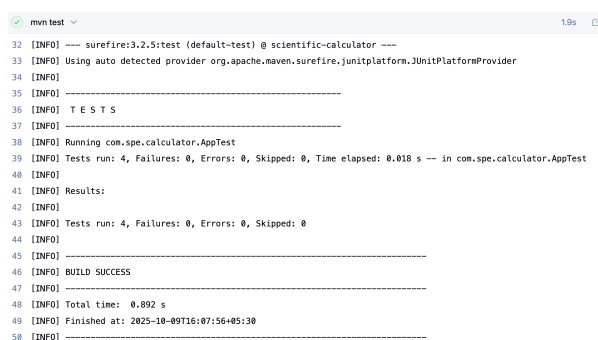
5 Stages of the Jenkins Pipeline

The core of the automation is defined in the `Jenkinsfile`, which outlines a series of stages that are executed sequentially. Each stage represents a logical unit in the CI/CD workflow, from testing to deployment.

5.1 Stage 1: Test Java App

This initial stage is a crucial quality gate. It runs only the unit tests to provide a rapid check on the latest code changes.

- **Purpose:** To quickly validate the correctness of the calculator's functions without performing a full build.
- **Command:** `mvn test` is executed, which triggers the Surefire plugin to run all JUnit tests in the project.
- **Benefit:** If tests fail, the pipeline stops immediately, providing fast feedback to the developer and saving system resources.



```

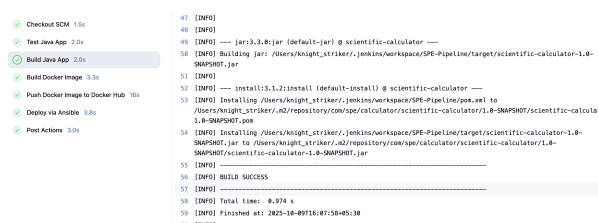
32 [INFO] --- surefire:3.2.5:test (default-test) @ scientific-calculator ---
33 [INFO] Using auto detected provider org.apache.maven.surefire.junitplatform.JUnitPlatformProvider
34 [INFO]
35 [INFO] -----
36 [INFO] T E S T S
37 [INFO] -----
38 [INFO] Running com.spe.calculator.AppTest
39 [INFO] Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.018 s -- in com.spe.calculator.AppTest
40 [INFO]
41 [INFO] Results:
42 [INFO]
43 [INFO] Tests run: 4, Failures: 0, Errors: 0, Skipped: 0
44 [INFO]
45 [INFO] -----
46 [INFO] BUILD SUCCESS
47 [INFO] -----
48 [INFO] Total time: 0.892 s
49 [INFO] Finished at: 2025-10-09T16:07:56+05:30
50 [INFO] -----
  
```

Figure 8: Console output showing successful execution of unit tests.

5.2 Stage 2: Build Java App

Once the tests pass, this stage compiles the entire application and packages it into an executable JAR file.

- **Purpose:** To create the final, distributable artifact of the Java application.
- **Command:** `mvn clean install` cleans previous build artifacts, compiles the source code, and packages it into a JAR file located in the `target/` directory.



```

47 [INFO]
48 [INFO]
49 [INFO] --- jar:3.3.0:jar (default-jar) @ scientific-calculator ---
50 [INFO] Building jar: /Users/knight_striker/.jenkins/workspace/SPE-Pipeline/target/scientific-calculator-1.8-SNAPSHOT.jar
51 [INFO]
52 [INFO] --- install:3.1.2:install (default-install) @ scientific-calculator ---
53 [INFO] Installing /Users/knight_striker/.jenkins/workspace/SPE-Pipeline/pom.xml to /Users/knight_striker/.m2/repository/com/spe/calculator/scientific-calculator/1.8-SNAPSHOT/pom.xml
54 [INFO] Installing /Users/knight_striker/.jenkins/workspace/SPE-Pipeline/target/scientific-calculator-1.8-SNAPSHOT.jar to /Users/knight_striker/.m2/repository/com/spe/calculator/scientific-calculator/1.8-SNAPSHOT/scientific-calculator-1.8-SNAPSHOT.jar
55 [INFO]
56 [INFO] BUILD SUCCESS
57 [INFO]
58 [INFO] Total time: 0.974 s
59 [INFO] Finished at: 2025-10-09T16:07:58+05:30
60 [INFO]
  
```

Figure 9: The successful Maven build log in Jenkins.

5.3 Stage 3: Build Docker Image

This stage takes the JAR file created in the previous step and packages it into a standardized, portable Docker image as defined by the `Dockerfile`.

- **Purpose:** To containerize the application, bundling it with its Java runtime environment and dependencies.
- **Command:** `docker build -t nonachadcp/scientific-calculator:latest .` creates a new Docker image and tags it with the specified name and version.

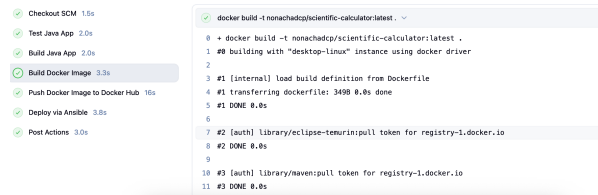


Figure 10: Successful completion of the Docker image build.

5.4 Stage 4: Push Docker Image to Docker Hub

After the image is built, it is pushed to a central registry, making it available for deployment.

- **Purpose:** To store the versioned container image in a public or private registry (in this case, Docker Hub).
- **Process:** The pipeline securely logs into Docker Hub using credentials stored in Jenkins. It then pushes the tagged image to the specified repository.

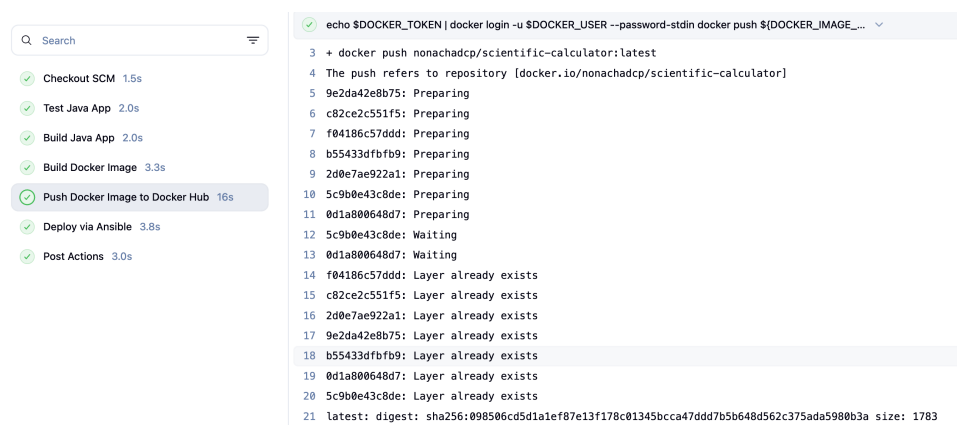


Figure 11: The Docker image being successfully pushed to Docker Hub.

5.5 Stage 5: Deploy via Ansible

This is the final stage, where the application is deployed into a "production" environment (in this case, the local machine).

- **Purpose:** To automate the release and running of the application.

- **Process:** The stage first activates an Ansible virtual environment. It then executes the `ansible-playbook` command, which reads the `playbook.yml` file to pull the latest image from Docker Hub and start the container.

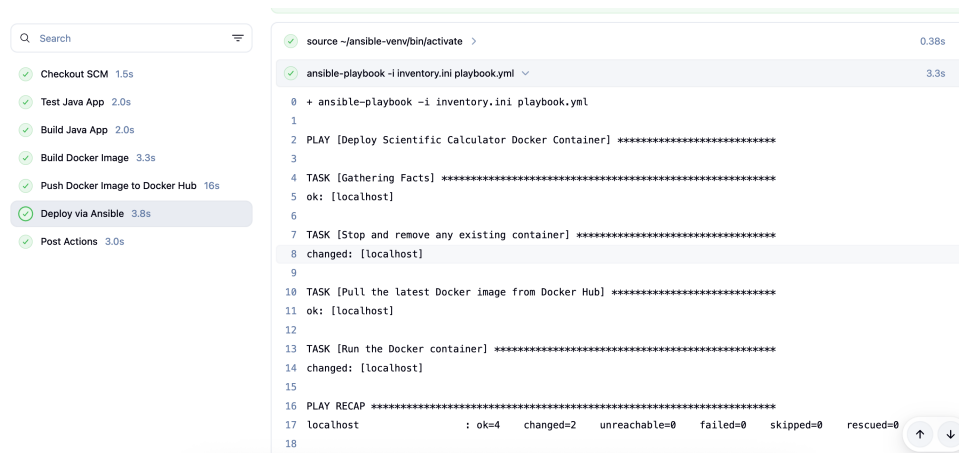


Figure 12: Successful deployment recap from the Ansible playbook.

5.6 Post-Build Actions: Email Notifications

After all stages are complete, the `post` block is executed to provide feedback on the pipeline's outcome.

- **Purpose:** To automatically notify stakeholders of the build status.
- **Process:** Depending on whether the pipeline succeeded or failed, a custom email is sent to the specified recipient with a summary of the build and a link back to the Jenkins logs.

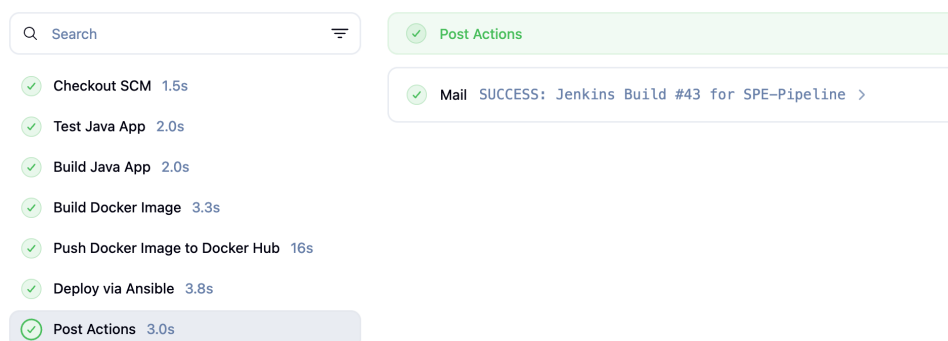


Figure 13: Successful mailing of the results to us

6 Steps to Be Followed For Installation

This section provides a comprehensive, systematic guide detailing all the steps taken to set up the environment, develop the application, and build the full CI/CD pipeline from start to finish. It is intended to be a complete reference for replicating the project.

6.1 Phase 1: Local Environment Setup and Installation

The first phase involved installing all the necessary tools on the local development machine (a macOS M3 system) using the Homebrew package manager.

1. **Install Homebrew:** The package manager for macOS.
2. **Install DevOps Tools:** The following commands were run in the terminal:

```
1
2 brew install git
3 brew install openjdk@17
4 brew install maven
5 brew install --cask docker
6 brew install jenkins-lts
7 brew install ansible
```

3. **Start Services:** Docker Desktop was launched, and the Jenkins service was started.

```
1 brew services start jenkins-lts
```

6.2 Phase 2: Project Scaffolding and Code Development

With the tools installed, the project structure and all necessary configuration files were created.

1. **Create Project Structure:** A standard Maven project layout was created.
2. **Develop Java Application:** The core scientific calculator logic was written in `App.java` and the corresponding unit tests were created in `AppTest.java`.
3. **Create Maven POM (pom.xml):** This file was created to manage dependencies (JUnit) and configure the Java build process.
4. **Create Dockerfile:** A multi-stage `Dockerfile` was written to define how to build an efficient container image for the application.
5. **Create Ansible Playbook (playbook.yml):** A playbook was written to automate the deployment of the Docker container.
6. **Create Jenkinsfile:** The pipeline-as-code script was created to define all CI/CD stages.

6.3 Phase 3: Repository and Service Configuration

This phase involved setting up the external services that the pipeline interacts with.

1. **Create GitHub Repository:** A new public repository was created on GitHub to host the project's source code.

2. **Initial Git Push:** The local project was initialized as a Git repository and all files were pushed to the GitHub remote.

```
1 git init -b main
2 git add .
3 git commit -m "Initial project setup"
4 git push -u origin main
```

3. **Create Docker Hub Repository:** A public repository was created on Docker Hub.

- **Note (Initial Problem):** An early pipeline failure was caused by this repository not existing. The solution was to manually create a public repository on Docker Hub with a name that exactly matched the one specified in the Jenkinsfile.

6.4 Phase 4: Jenkins Configuration

Configuring Jenkins correctly was the most critical and complex part of the setup, involving several troubleshooting steps.

1. **Configure System PATH Variable (Troubleshooting Step):**

- **Problem:** Initial builds failed with ‘mvn: command not found’ and ‘docker: command not found’.
- **Solution:** The Jenkins ”Built-In Node” was configured to include the paths for Homebrew-installed tools. This was done in Manage Jenkins Nodes Built-In Node Configure by adding an environment variable:

- **Name:** PATH+EXTRA
- **Value:** /opt/homebrew/bin:/usr/local/bin

After this change, the Jenkins service was restarted.

2. **Manage Jenkins Credentials:**

- **Docker Hub Credentials:** A ”Username with password” credential with the ID `dockerhub-credentials` was created in *Manage Jenkins > Credentials* to allow the pipeline to log in to Docker Hub.
- **Gmail Credentials for Email:** Another ”Username with password” credential with the ID `gmail-credentials` was created.
 - **Problem:** Using the standard Gmail password failed due to Google’s security policies.
 - **Solution:** A 16-character **App Password** was generated from the Google Account security settings and used as the password in this Jenkins credential.

3. **Configure Email System:**

- The ”Email Extension Plugin” was installed.

- In *ManageJenkins* > *System*, the "Extended E-mail Notification" section was configured with Gmail's SMTP server (`smtp.gmail.com`), port 465, SSL enabled, and the newly created `gmail-credentials`.

4. Create and Configure the Jenkins Pipeline Job:

- A new "Pipeline" job was created in Jenkins.
- The job was configured to use "Pipeline script from SCM," pointing to the project's GitHub repository URL and specifying `Jenkinsfile` as the script path.

6.5 Phase 5: Enabling Automation with Webhooks

To allow GitHub to automatically trigger the Jenkins pipeline, a webhook was set up using ngrok.

1. **Expose Local Jenkins Server:** The `ngrok` tool was used to create a public URL that tunnels to the local Jenkins instance.

```
1 ngrok http 8080
```

2. **Configure GitHub Webhook:** The public HTTPS URL provided by ngrok was copied. In the GitHub repository's settings, under **Webhooks**, a new webhook was created by pasting the ngrok URL into the "Payload URL" field and appending `/github-webhook/`.

6.6 Phase 6: Final Pipeline Execution and Verification

With all configurations in place, the final step was to run the pipeline and verify the result.

1. **Triggering the Pipeline:** A final `git push` to the repository triggered the pipeline automatically via the webhook.
2. **Keeping the Container Alive (Troubleshooting Step):**
 - **Problem:** The deployed Docker container was stopping as soon as the Jenkins build finished.
 - **Solution:** The Ansible deployment command in the `Jenkinsfile` was updated to:

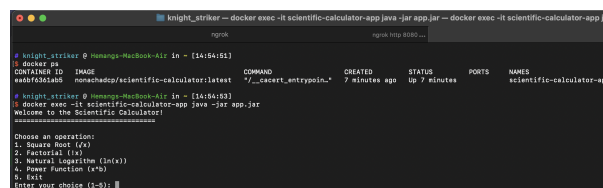
```
sh 'BUILD_ID=dontKillMe ansible-playbook playbook.yml'
```

This command prevents Jenkins from terminating the container after the step completes.
3. **Verifying the Deployment:** The successful deployment was verified by running `docker ps` to see the running container and then attaching to it to use the application.

```
1 # Check that the container is running
2 docker ps
3 # Attach to the container to interact with the application
4 docker attach scientific-calculator-app
```

7 Final Results and Observations

After the successful execution of the entire Jenkins pipeline, the scientific calculator application was automatically deployed and running in a Docker container on the local machine. The `docker ps` command confirms the container is up and running, and the `docker run -it ...` command (used here for a clean demonstration) shows the interactive menu of the application, proving the deployment was successful.



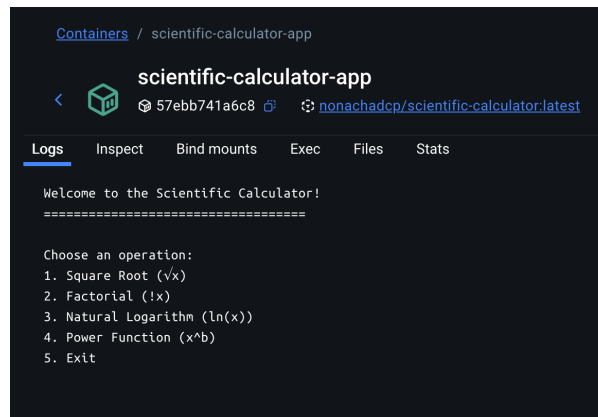
```
knight_striker ~$ docker exec -it scientific-calculator-app java -jar app.jar -- docker exec -it scientific-calculator-app java
nigrok http 8080 ...

# knight_striker@Homemaps-MacBook-Air in ~ [14:54:51]
# docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS              NAMES
eada9f363a05       nonachadp/scientific-calculator:latest   "/_execert_entrypoint..." 7 minutes ago       Up 7 minutes                scientific-calculator-app

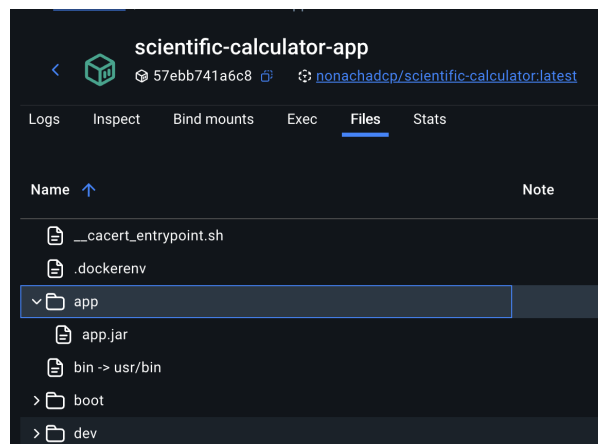
# knight_striker@Homemaps-MacBook-Air in ~ [14:54:53]
# docker exec -it scientific-calculator-app java -jar app.jar
Welcome to the Scientific Calculator!
=====

Choose an operation:
1. Square Root (sqrt)
2. Factorial (fact)
3. Natural Logarithm (ln(x))
4. Power Function (x^y)
5. Exit
Enter your choice (1-5):
```

Figure 14: The scientific calculator running interactively inside the Docker container after successful deployment.



(a) The Docker container being successfully created



(b) Contents inside the Docker container

Figure 15: Docker container creation and contents overview

This result validates the entire CI/CD pipeline. A change pushed to GitHub successfully travels through all automated stages and results in a test-passing, deployed, and running application, with a success notification sent via email.

8 Conclusion

This project successfully demonstrated the implementation of a complete, end-to-end CI/CD pipeline using a modern DevOps toolchain. By integrating Git, Maven, Jenkins, Docker, and Ansible, the entire process from code commit to application deployment was fully automated. This automation not only accelerates the development lifecycle but also significantly improves reliability and consistency. The project serves as a practical showcase of how DevOps principles can be applied to transform software delivery.

External Links:

- **GitHub Repository:** [Github/scientific-calculator](https://github.com/nonachadcp/scientific-calculator)
- **Docker Hub Repository:** [Docker/scientific-calculator](https://hub.docker.com/r/nonachadcp/scientific-calculator)

8.1 Manual Execution Steps

For local development and testing, the application can be compiled, containerized, and run manually without using the full Jenkins CI/CD pipeline. This section provides the necessary commands to execute these steps from a local terminal.

Prerequisites

Ensure the following tools are installed on your local machine, along with commands to install them if missing:

- **Java 17 (JDK)**

Install via Homebrew: `brew install openjdk@17`

- **Apache Maven**

Install via Homebrew: `brew install maven`

- **Docker**

Install via Homebrew: `brew install --cask docker`

(Then start Docker Desktop)

- **Ansible**

Install via Homebrew: `brew install ansible`

Or via Python virtual environment: `python3 -m venv /ansible-venv && source /ansible-venv/bin/activate && pip install ansible requests docker`

- **ngrok (for exposing local webhooks)**

Install via Homebrew: `brew install ngrok/ngrok/ngrok`

Execution Commands

The following commands should be executed in sequence from your terminal.

```
1 # 1. Clone the repository from GitHub to your local machine
2 git clone https://github.com/Hemang-2004/scientfic-calculator.git
3
4 # 2. Navigate into the project directory
5 cd scientfic-calculator
6
7 # 3. Build the Java application using Maven
8 mvn clean install
9
10
11 # 4. Deploy the Docker container using Ansible
12 # Ensure inventory.ini points to the correct Python interpreter for Ansible
13 ansible-playbook -i inventory.ini playbook.yml
14
15 # 5. Verify the running Docker container
16 docker ps
17
```

```
18 # 6. Execute the application inside the running container
19 docker exec -it scientific-calculator-app java -jar app.jar
20
21 # 7. (Optional) Expose local webhook endpoints using ngrok (For Jenkins Pipeline)
22 ngrok http 8080
```

After running these commands, the scientific calculator application will be up and running in a Docker container, accessible for testing, and ready to receive any webhook requests if ngrok is configured.