CSE 816 - Software Production Engineering
Mini Project Report

Scientific Calculator with DevOps

Anurag Ramaswamy
IMT2022103

October 10, 2025

Project Links

GitHub Repository

Docker Hub Repository

Contents

1 Introduction **3**
- 1.1 What and Why of DevOps? . . . 3
- 1.2 How to Implement DevOps . . . 3

2 Requirement Analysis **4**
- 2.1 Functional Requirements . . . 4
- 2.2 Non-Functional Requirements . . . 4

3 Tools Used & Setup **5**
- 3.1 Java & Maven . . . 5
- 3.2 Git & GitHub . . . 6
- 3.3 Docker . . . 6
- 3.4 Jenkins . . . 7
- 3.5 Ansible . . . 7
- 3.6 Ngrok . . . 8

4 Project Workflow & Setup **8**
- 4.1 Folder Structure . . . 8
- 4.2 Build & Test with Maven . . . 8
- 4.3 Setting Up Git and GitHub . . . 10
  - 4.3.1 Initializing Git in my Project . . . 11
  - 4.3.2 Creating a GitHub Repository and Pushing Code . . . 11
  - 4.3.3 Creating a .gitignore File . . . 12
- 4.4 Containerizing the Project using Docker . . . 13
  - 4.4.1 Dockerfile Setup . . . 13
  - 4.4.2 Building and Pushing the Docker Image to Docker Hub 13
- 4.5 Ansible Setup . . . 13
  - 4.5.1 Creating the Ansible Inventory . . . 13
  - 4.5.2 Writing the Ansible Playbook . . . 13
- 4.6 Jenkins Pipeline Setup . . . 15
  - 4.6.1 Adding Docker Hub Credentials . . . 16
  - 4.6.2 Creating the Jenkins Pipeline Job . . . 16
  - 4.6.3 Jenkinsfile . . . 19
- 4.7 GitHub Webhooks and Ngrok Setup . . . 19

5 Challenges Faced **21**

6 Conclusion **22**

1 Introduction

1.1 What and Why of DevOps?

DevOps is a set of practices that brings together development and operations teams to work across the entire software development lifecycle. It promotes shared responsibility and faster delivery through automation and continuous integration. The core principles of DevOps can be summarized by the acronym CALMS: Culture, which emphasizes people and collaboration, Automation, which enables infrastructure and deployment as code, Lean, focusing on efficiency and delivering value in small batches, Measurement, to track progress and drive improvement and Sharing, which strengthens transparency and cooperation between development and operations. These principles help teams deliver faster and more reliably software.

The previously used Waterfall model uses a rigid sequential flow which lacks feedback between stages and Agile model uses shorter iterative cycles but still keeps development and operations largely separate. However DevOps overcomes these deficiencies by unifying all phases into a single continuous workflow, promoting automation, collaboration and continuous delivery, ensuring faster and more reliable software.

1.2 How to Implement DevOps

To implement DevOps effectively, a series of well-defined stages and tools are used to automate and streamline the software development and deployment process. They include:

- **Integrated Development Environment (IDE):** Provides built-in support for Git and build tools, allowing developers to write, manage and test code efficiently within a unified workspace (VSCode).
- **Build Automation and Dependency Management:** Automates the process of compiling, testing and packaging the application while managing dependencies for consistent builds across environments (Maven).
- **Version Control:** Tracks code changes and facilitates collaboration through local and remote repositories, ensuring proper version management (Git & Github).
- **Continuous Integration and Deployment:** Automates build, test and deployment processes using pipelines, ensuring reliable software delivery (Jenkins).

- **Containerization:** Packages applications and their dependencies into portable containers for consistent execution across different systems (Docker).
- **Container Registry:** Stores and manages container images, allowing developers to share and deploy them easily across multiple environments (Docker Hub).
- **Configuration and Deployment Automation:** Simplifies and automates environment setup, configuration and deployment to ensure repeatability and reduce manual errors (Ansible).

2 Requirement Analysis

2.1 Functional Requirements

The aim of this project was to create a Scientific Calculator with the following user-friendly menu-driven operations:

- Square root function ($\sqrt{x}$)
- Factorial function ($x!$)
- Natural logarithm (base - e) ($\ln(x)$)
- Power function (x^b)

Proper error handling must be implemented to manage invalid inputs such as negative integers for square root and fractional numbers for factorial.

2.2 Non-Functional Requirements

The project should satisfy the following non-functional requirements:

- **Usability:** The system should provide a simple and intuitive interface that assists users with clear prompts, guidance and error messages to ensure smooth interaction.
- **Code Quality:** Code should be clean, modular and well-documented to ensure maintainability.
- **Version Control:** Git and GitHub (or another version control tool) must be used for source code management.

- **Build Automation:** Maven (or another build automation tool) should handle compilation, testing and packaging.
- **Continuous Integration/Deployment:** Jenkins (or another CI/CD tool) must automate builds, tests and deployments through CI/CD pipelines.
- **Containerization:** Docker (or another containerization tool) must be used to containerize the application for consistent deployment.
- **Environment Management:** Ansible (or another deployment tool) should automate environment setup and deployment tasks.
- **Security:** The system should safeguard data and credentials, ensuring secure integrations with platforms such as GitHub and Docker Hub.
- **Documentation:** Detailed documentation and screenshots must be provided for each step.

3 Tools Used & Setup

I worked on the project in WSL (Ubuntu), but the same code should work fine in any Linux environment. After installing the tools, I verified their availability using the `--version` tag and added their outputs to the respective tool sections.

3.1 Java & Maven

Java is a high level object-oriented programming language used here to implement the scientific calculator logic. I used Java because I had used JUnit before and thought it would be easier to test the code since JUnit is a testing package for Java. Maven is a build automation tool that manages compilation, testing, and packaging, while also handling dependencies consistently.

```
$ sudo apt update
$ sudo apt install openjdk-17-jdk maven
$ java --version
openjdk 17.0.16 2025-07-15
OpenJDK Runtime Environment (build 17.0.16+8-Ubuntu-0
   ubuntu122.04.1)
OpenJDK 64-Bit Server VM (build 17.0.16+8-Ubuntu-0ubuntu122
   .04.1, mixed mode, sharing)
$ mvn --version
Apache Maven 3.6.3
Maven home: /usr/share/maven
```

```
Java version: 17.0.16, vendor: Ubuntu, runtime: /usr/lib/jvm/
    java-17-openjdk-amd64
Default locale: en, platform encoding: UTF-8
OS name: "linux", version: "6.6.87.2-microsoft-standard-wsl2"
    , arch: "amd64", family: "unix"
```

3.2 Git & GitHub

Git is a distributed version control system, while GitHub is used as the remote hosting service to store the repository, manage commits and integrate with Jenkins for CI/CD.

```
$ sudo apt update
$ sudo apt install git
$ git --version
git version 2.34.1
```

3.3 Docker

Docker provides lightweight containers that package the application and its dependencies. This ensures portability and consistency across environments.

```
$ sudo apt remove docker docker-engine docker.io containerd
    runc
$ sudo apt update
$ sudo apt install ca-certificates curl gnupg
$ sudo install -m 0755 -d /etc/apt/keyrings
$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg \
  | sudo gpg --dearmor -o /etc/apt/keyrings/docker.gpg
$ echo "deb [arch=$(dpkg --print-architecture) \
  signed-by=/etc/apt/keyrings/docker.gpg] \
  https://download.docker.com/linux/ubuntu $(lsb_release -cs)
      stable" \
  | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
$ sudo apt update
$ sudo apt install docker-ce docker-ce-cli containerd.io \
  docker-buildx-plugin docker-compose-plugin
$ sudo usermod -aG docker $USER
$ newgrp docker
$ docker ps
CONTAINER ID   IMAGE   COMMAND   CREATED   STATUS   PORTS
    NAMES
$ docker --version
Docker version 27.5.1, build 27.5.1-0ubuntu3~22.04.2
```

3.4 Jenkins

Jenkins is used for continuous integration, automating builds, tests and deployment stages with the Jenkinsfile.

```
$ sudo wget -O /usr/share/keyrings/jenkins-keyring.asc \
  https://pkg.jenkins.io/debian-stable/jenkins.io-2023.key
$ echo "deb [signed-by=/usr/share/keyrings/jenkins-keyring.
   asc] \
  https://pkg.jenkins.io/debian-stable binary/" \
  | sudo tee /etc/apt/sources.list.d/jenkins.list > /dev/null
$ sudo apt update
$ sudo apt install jenkins
$ sudo systemctl start jenkins
$ sudo systemctl status jenkins
jenkins.service - Jenkins Continuous Integration Server
     Loaded: loaded (/lib/systemd/system/jenkins.service;
        enabled; vendor preset: enabled)
     Active: active (running) since Sun 2025-10-05 13:12:07
        IST; 4h 59min ago
   Main PID: 283 (java)
      Tasks: 78 (limit: 9354)
     Memory: 834.0M
        CPU: 3min 2.385s
     CGroup: /system.slice/jenkins.service
             283    /usr/bin/java -Djava.awt.headless=true
              -jar /usr/share/java/jenkins.war --w
```

After following the above steps, we need to go to `http://localhost:8080` to complete the initial Jenkins setup by creating a new admin user and installing the suggested plugins. Also install the stage view plugin and Mailer plugin if not already installed.

3.5 Ansible

Ansible is used for configuration management and automated deployment. A YAML-based playbook is used here to automate the deployment of the Docker environment, ensuring a consistent execution environment for all application containers, thereby minimizing the risk of manual errors during deployment.

```
$ sudo apt update
$ sudo add-apt-repository --yes --update ppa:ansible/ansible
$ sudo apt install -y ansible
$ ansible --version
ansible [core 2.17.14]
  config file = None
  configured module search path = ['/home/anurag/.ansible/
     plugins/modules', '/usr/share/ansible/plugins/modules']
```

```
  ansible python module location = /home/anurag/.local/lib/
     python3.10/site-packages/ansible
  ansible collection location = /home/anurag/.ansible/
     collections:/usr/share/ansible/collections
  executable location = /home/anurag/.local/bin/ansible
  python version = 3.10.12 (main, Aug 15 2025, 14:32:43) [GCC
      11.4.0] (/usr/bin/python3)
  jinja version = 3.1.4
  libyaml = True
```

3.6 Ngrok

Ngrok creates a secure tunnel to expose local servers to the internet. In this project, a reserved ngrok domain is used to expose the Jenkins server so that GitHub webhooks can trigger builds.

```
$ curl -sSL https://ngrok-agent.s3.amazonaws.com/ngrok.asc \
  | sudo tee /etc/apt/trusted.gpg.d/ngrok.asc >/dev/null
$ echo "deb https://ngrok-agent.s3.amazonaws.com jammy main"
    \
  | sudo tee /etc/apt/sources.list.d/ngrok.list
$ sudo apt update
$ sudo apt install ngrok
```

After following the above commands we also need to add our authentication token from `https://dashboard.ngrok.com/get-started/your-authtoken` by using the command below:

```
$ ngrok config add-authtoken $YOUR_AUTHTOKEN
```

4 Project Workflow & Setup

4.1 Folder Structure

I used the standard Maven-Java folder structure. The calculator logic code was written in `ScientificCalculator.java`, and the corresponding JUnit 5 test code was written in `ScientificCalculatorTest.java`.

4.2 Build & Test with Maven

Before integrating the code into the CI/CD pipeline, I first compiled, tested and packaged it locally using Maven. This helped me verify that all implemented operations worked correctly and that the JUnit test cases passed without errors.

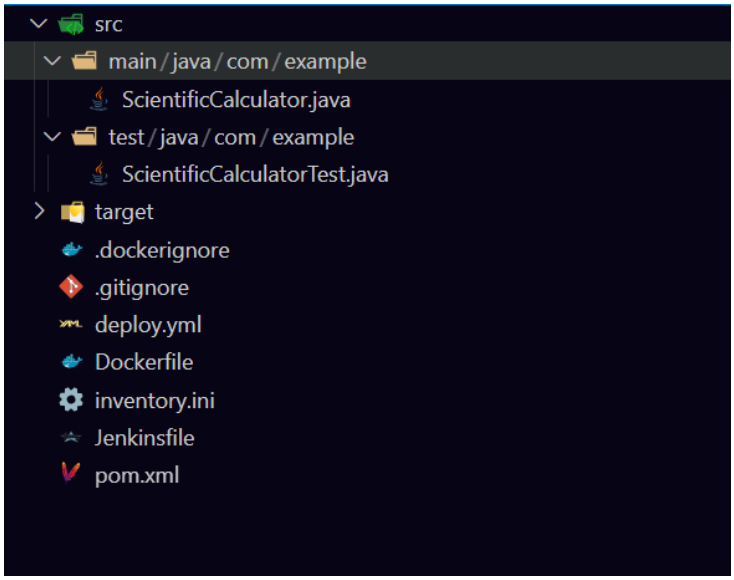

Figure 1: Project Folder Structure

```
# Remove any leftover compiled files from previous builds
$ mvn clean

# Run all unit tests
$ mvn test

# Package the calculator into an executable JAR file
$ mvn package

# Run the packaged application from the generated target/
   folder
$ java -jar target/scientific-calculator-1.0-SNAPSHOT.jar
```

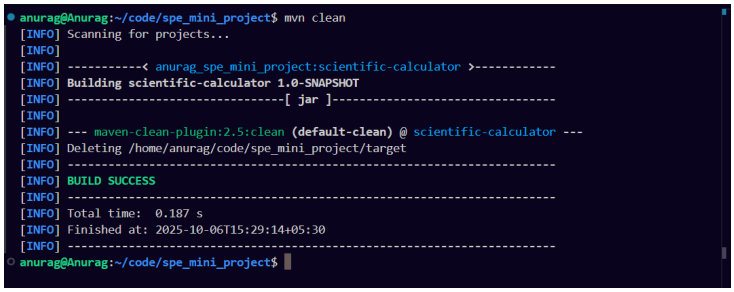

Figure 2: mvn clean

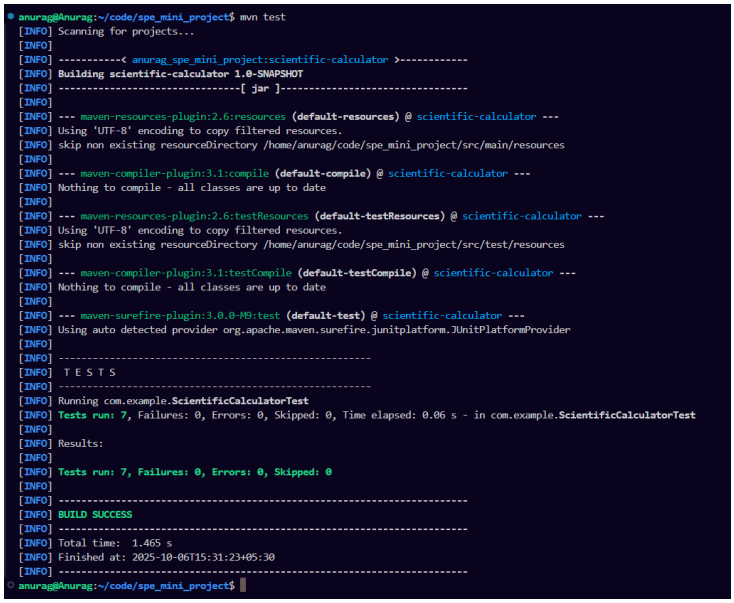

Figure 3: mvn test

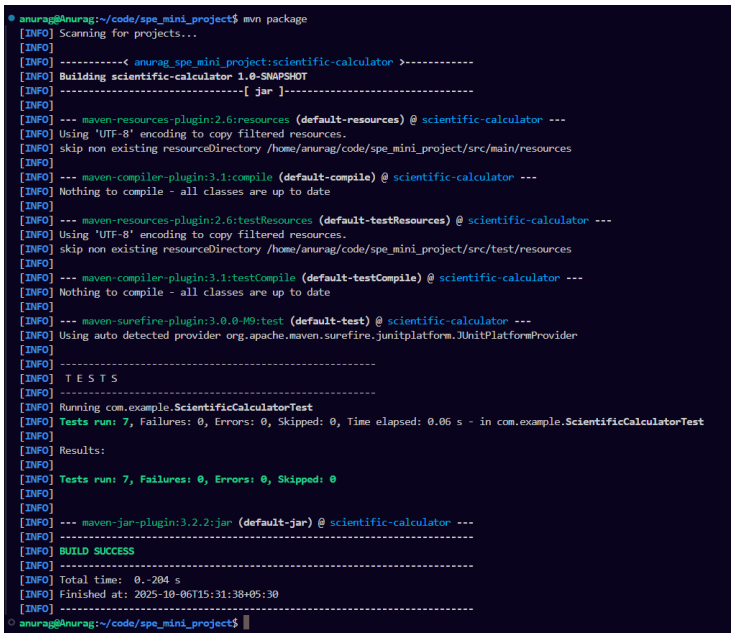

Figure 4: mvn package

4.3 Setting Up Git and GitHub

I took the following steps to initialize a Git repo for my project, connect it to a remote GitHub repo and push changes to it.

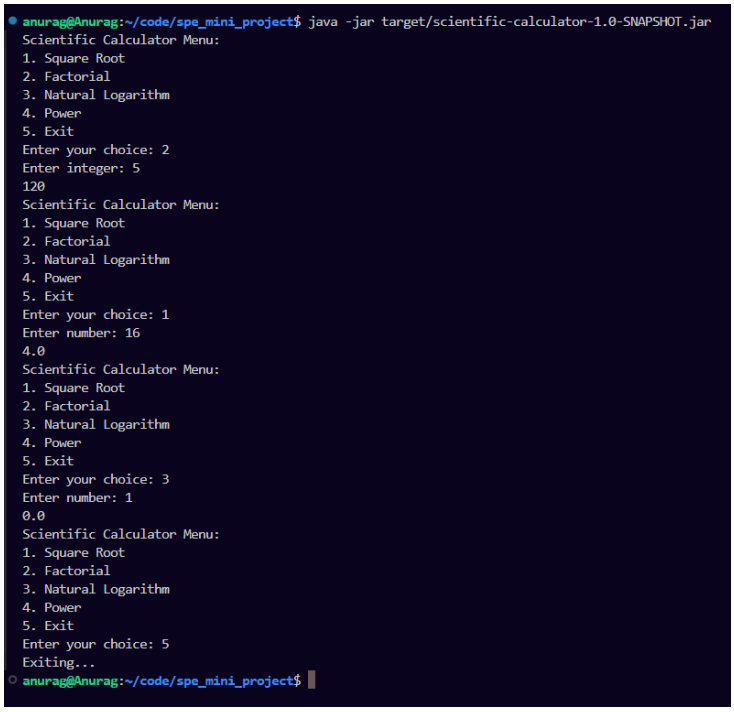

Figure 5: Running the calculator

4.3.1 Initializing Git in my Project

Run these commands in the root directory of the project.

```
# Initialize Git
$ git init

# Stage and commit all files
$ git add .
$ git commit -m "initial commit"
```

4.3.2 Creating a GitHub Repository and Pushing Code

I created a new public repository on Github for my project. The local repository was linked with the remote GitHub repository, and the code was pushed to the `main` branch.

```
$ git remote add origin https://github.com/Anurag9507/
   spe_mini_project.git

# Push the main branch to the remote repository
$ git push -u origin main
```

```
# Push updates to GitHub
$ git add .
$ git commit -m "update"
$ git push
```

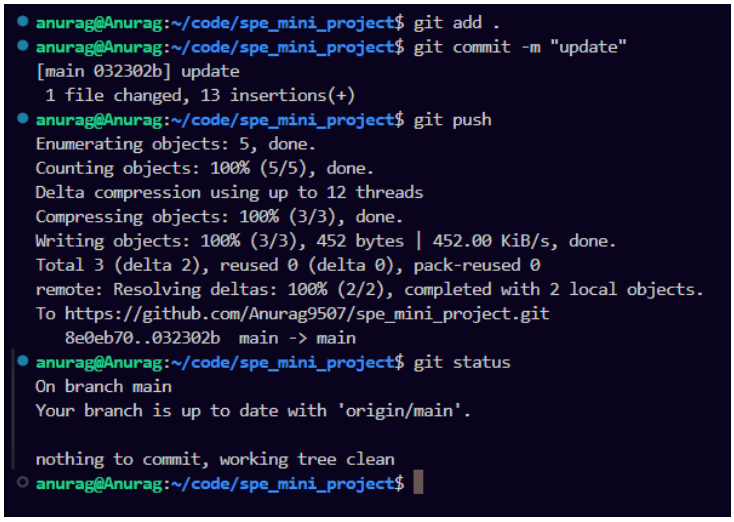

Figure 6: Pushing updates to GitHub

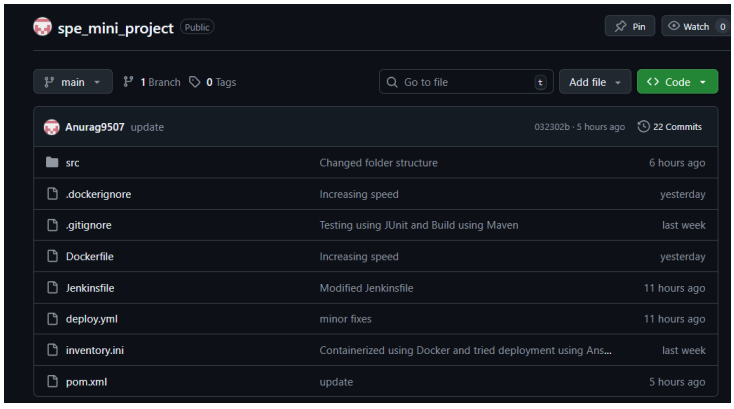

Figure 7: GitHub repository after the above commit

4.3.3 Creating a .gitignore File

I also created a `.gitignore` file to ignore all class files and all generated files within the `target` directory.

4.4 Containerizing the Project using Docker

4.4.1 Dockerfile Setup

A `Dockerfile` was created to containerize the scientific calculator application into a lightweight and portable image. The container uses `OpenJDK 17` as the base, copies the compiled JAR file into the working directory, and executes the calculator using the Java runtime. This ensures the same environment is used across different systems, eliminating dependency issues and making deployment simpler and consistent.

```
FROM openjdk:17-jdk-slim
WORKDIR /app
COPY target/scientific-calculator-1.0-SNAPSHOT.jar app.jar
ENTRYPOINT ["java", "-jar", "app.jar"]
```

4.4.2 Building and Pushing the Docker Image to Docker Hub

The Docker image was then built using the above Dockerfile, tagged and pushed to a public Docker Hub repository using the following commands:

```
$ docker build -t anurag9507/scientific-calculator:latest .
$ docker login -u anurag9507
$ docker push anurag9507/scientific-calculator:latest
```

After building, it was verified locally using:

```
$ docker run -it anurag9507/scientific-calculator:latest
```

4.5 Ansible Setup

4.5.1 Creating the Ansible Inventory

The inventory file (`inventory.ini`)(given below) defines which hosts Ansible should manage, in our project it only needs to manage localhost.

```
[anurag_pc]
localhost ansible_connection=local
```

4.5.2 Writing the Ansible Playbook

The Ansible playbook (`deploy.yml`) defines and automates the deployment tasks across the inventory hosts. It ensures Docker is available, pulls the latest image from Docker Hub, removes any existing container and starts a new instance in detached mode.

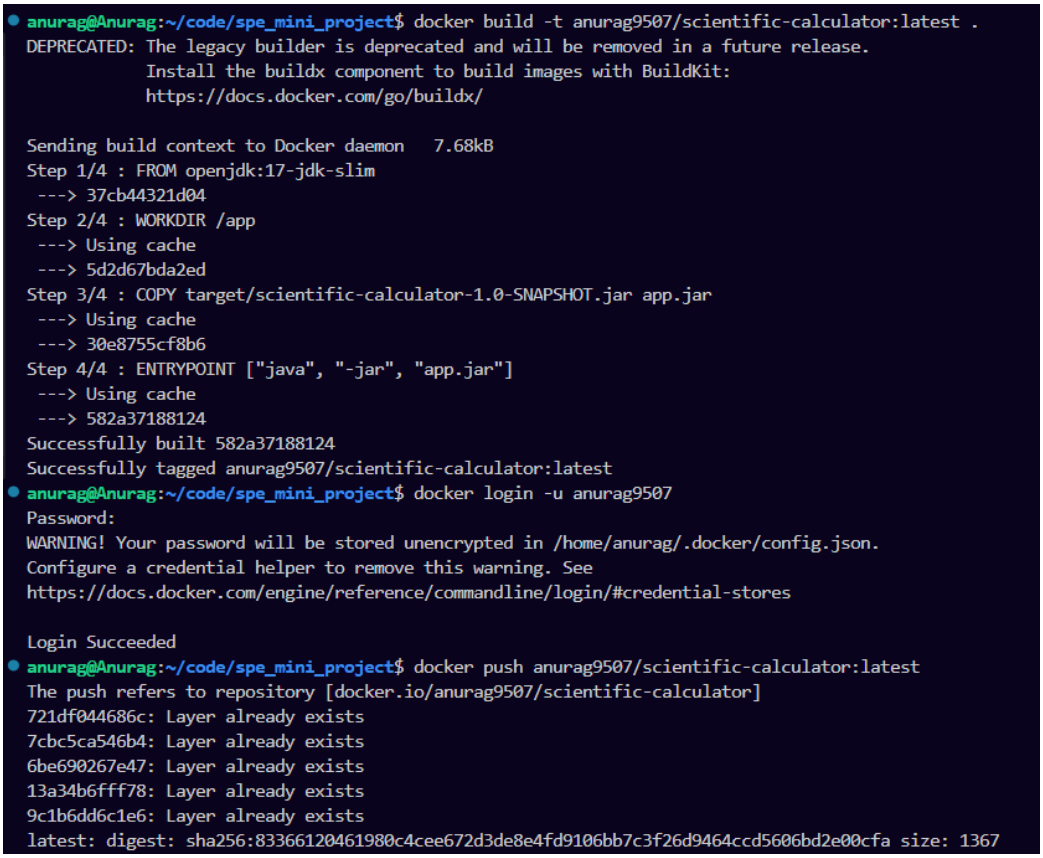

Figure 8: Containerizing using Docker

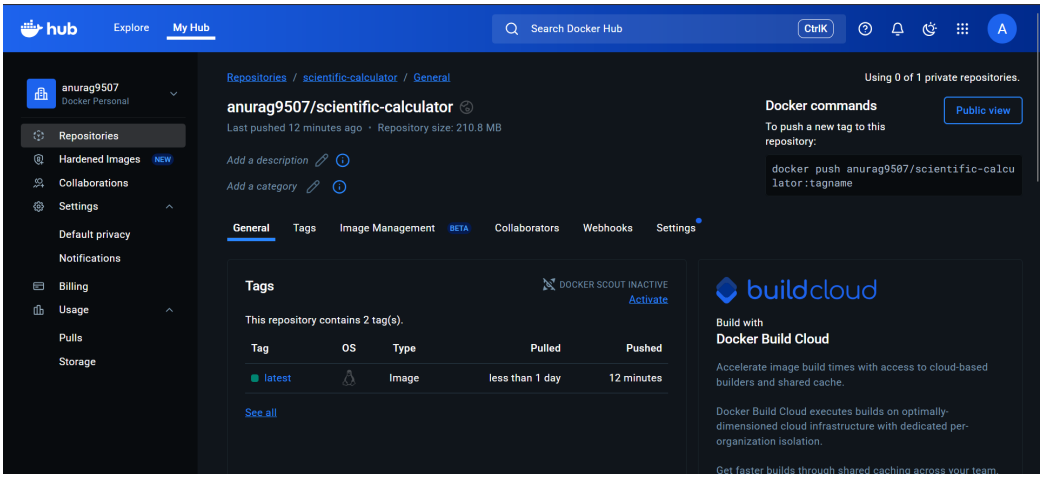

Figure 9: Docker Hub Repository

```
---
- name: Deploying using Ansible
  hosts: anurag_pc
  become: true

  vars:
    image_name: "anurag9507/scientific-calculator:latest"
    container_name: "scientific-calculator"

  tasks:
    - name: 0. Ensure Docker is installed
      ansible.builtin.pip:
        name: docker
        state: present

    - name: 1. Pull latest Docker image from Docker Hub
      community.docker.docker_image:
        name: "{{ image_name }}"
        source: pull

    - name: 2. Remove any old container if it exists
      community.docker.docker_container:
        name: "{{ container_name }}"
        state: absent

    - name: 3. Run calculator in detached mode
      community.docker.docker_container:
        name: "{{ container_name }}"
        image: "{{ image_name }}"
        state: started
        detach: yes
        interactive: yes
        tty: yes
        recreate: true
```

The playbook was verified locally via the below command using `sudo` since the playbook utilized the `become: true` for privilege escalation.

```
$ sudo ansible-playbook -i inventory.ini deploy.yml
```

4.6 Jenkins Pipeline Setup

To automate all stages of building, testing, containerizing and deploying the scientific calculator application, a Jenkins pipeline was configured.

Figure 10: Ansible playbook verification

4.6.1 Adding Docker Hub Credentials

To allow Jenkins to authenticate with Docker Hub and push images automatically:

1. Open Jenkins Dashboard → `Manage Jenkins` → `Credentials`.
2. Click on `System` → `Global credentials` → `Add Credentials`.
3. Select credential type as `Username with password` and enter Docker Hub username and password.
4. Set `ID` as `dockerhub-creds` and click save.

4.6.2 Creating the Jenkins Pipeline Job

1. In Jenkins Dashboard, click `New Item`.
2. Enter a job name, select `Pipeline`, and click `OK`.
3. Under the `Pipeline` section, choose `Pipeline script from SCM`. Select `SCM`: git.
4. Repository URL: `https://github.com/Anurag9507/spe_mini_project.git`
5. Select the main branch and set `Script Path` to `Jenkinsfile`.
6. In `Build Triggers`, check `GitHub hook trigger for GITScm polling`.
7. Save the job and click `Build Now` to test the setup.

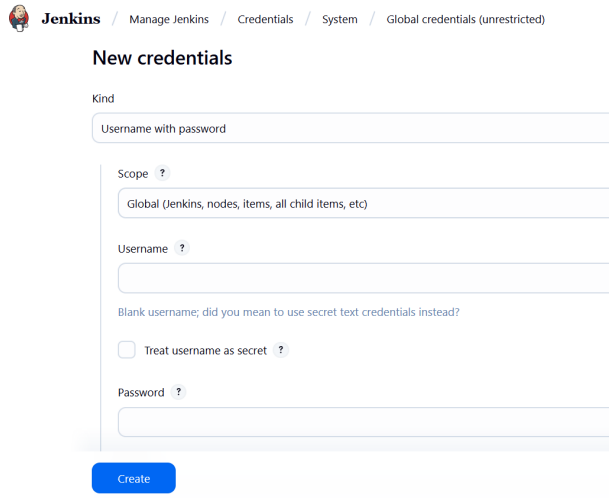

Figure 11: Adding Docker Hub credentials

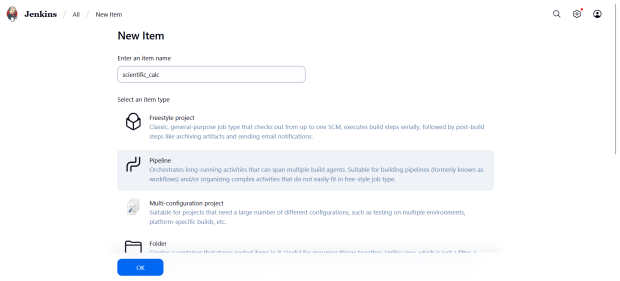

Figure 12: Creating a new Jenkins pipeline

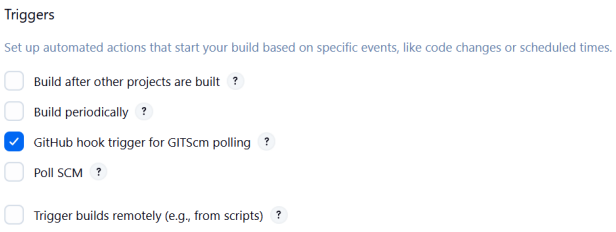

Figure 13: Setting up GitHub hook trigger

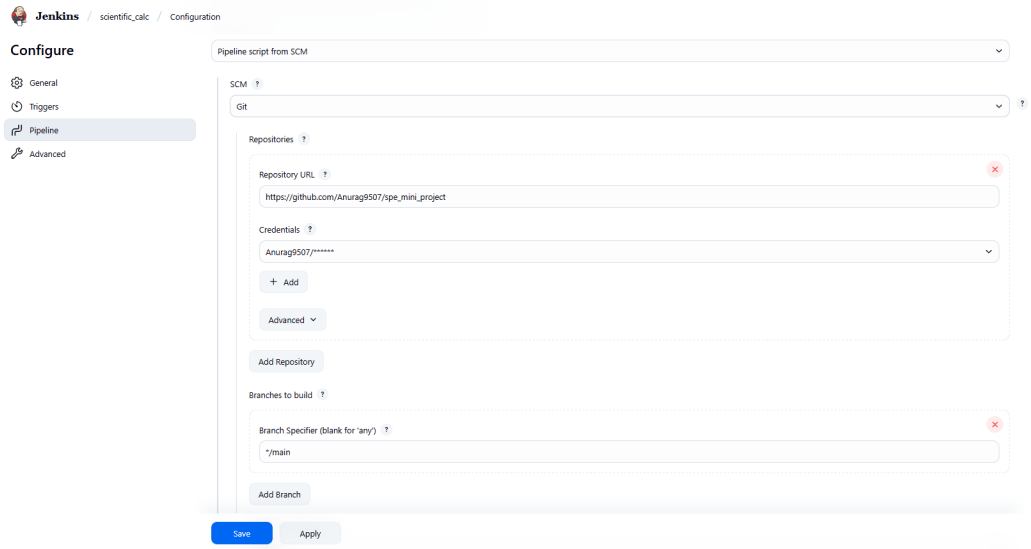

Figure 14: Configuring the pipeline settings

Figure 15: Stage View of the Jenkins pipeline

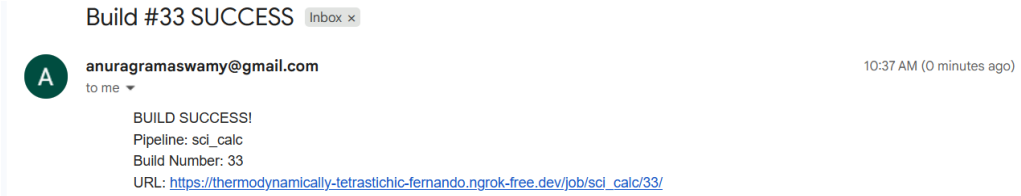

Figure 16: E-mail sent on Build Success

4.6.3 Jenkinsfile

The Jenkins pipeline in the `Jenkinsfile` automates the entire CI/CD process for the project. Each stage executes sequentially to ensure that the code is built, tested, containerized and deployed in a consistent way.

- **Build & Test:** In this stage, Jenkins uses Maven to compile the Java source code, execute JUnit test cases, and package the application into a runnable JAR file using the commands given in the Maven setup section.
- **Build Docker Image:** After successful testing, the pipeline builds a Docker image containing the packaged application, containerizing both the application and its runtime environment for consistent execution across systems.
- **Push Docker Image to Docker Hub:** In this stage, Jenkins authenticates with Docker Hub using the previously stored credentials (`dockerhub-creds`) and pushes the newly built image to the public Docker Hub repository.
- **Deploy with Ansible:** The final stage automates the deployment of the Docker container using Ansible. The playbook `deploy.yml` is then executed, which pulls the Docker image from Docker Hub, removes any old container and starts a new instance.
- **Post Actions:** Upon completion of the pipeline, Jenkins automatically sends an email including the build status(success or failure), pipeline name, build number, and the local Jenkins URL for quick access to build details.

The `Checkout SCM` stage appears automatically in Jenkins due to the GitHub webhook trigger via the configured Ngrok URL, making a Git checkout step in the Jenkinsfile unnecessary.

4.7 GitHub Webhooks and Ngrok Setup

To enable continuous integration whenever new code is pushed to GitHub, webhooks were configured using Ngrok. The following steps were followed:

1. Obtain a free static domain from `ngrok.com`.
2. Run the Jenkins server locally on port `8080`.

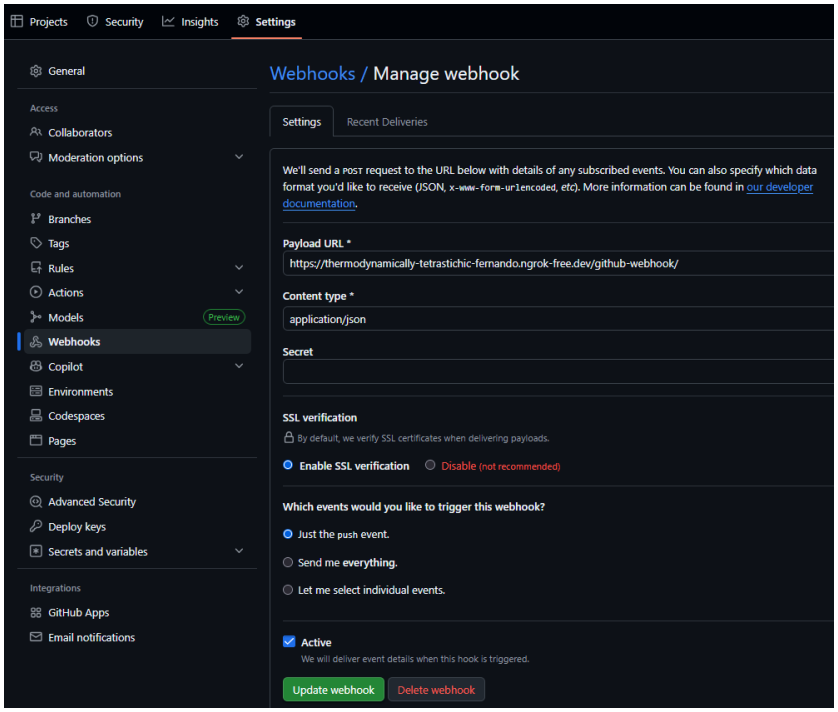

Figure 17: GitHub Webhook

3. Start Ngrok to expose Jenkins to the internet:

```
$ ngrok http --domain=ngrok-static-domain 8080
```

4. In GitHub repository → `Settings` → `Webhooks` → `Add Webhook`.

5. Set `Payload URL` to the ngrok static domain obtained followed by `/github-webhook/`

6. Set `Content type` to `application/json` and select `Just the push event`.

7. Click `Add Webhook`.

Now, every time there is a push to the GitHub repository, a build will be automatically triggered in the Jenkins pipeline through the webhook from the ngrok static domain.

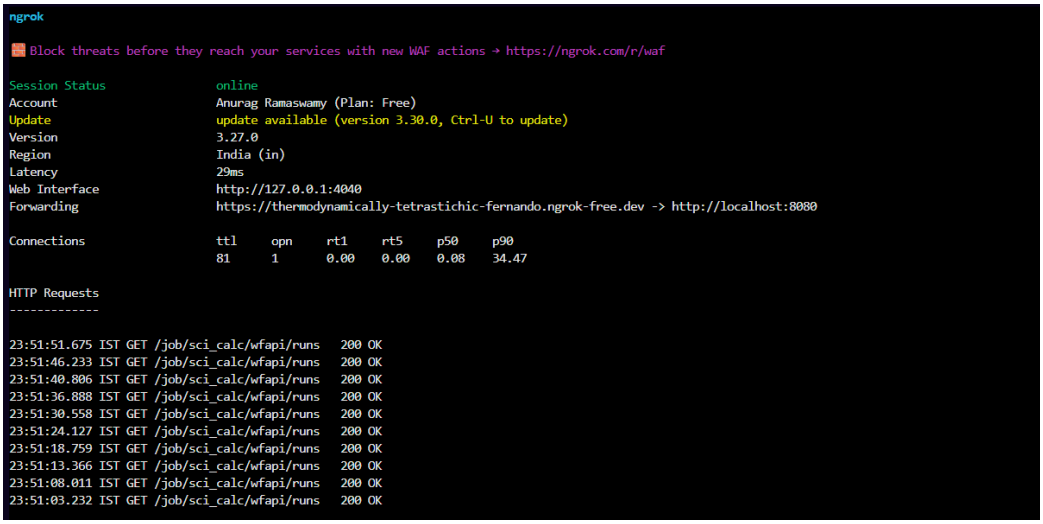

Figure 18: Accessing Jenkins via the Ngrok url

5 Challenges Faced

Throughout the implementation of this project, I encountered several technical and configuration challenges that helped me gain deeper insights into the DevOps workflow:

1. Docker build times were significantly higher at first, which was resolved by updating the `.dockerignore` file to exclude unnecessary files and retain only the compiled JAR.

2. The GitHub webhook triggered automatically even when Ngrok was not running, resulting in failed build delivery. This required manual redelivery of the webhook event to Jenkins.

3. The project initially failed to build because the Maven directory structure was not followed correctly. Once the standard `src/main/java` and `src/test/java` hierarchy was used, it compiled successfully.

4. Running the JAR inside Docker was tricky — it initially worked only with the `-cp` option instead of `-jar`. This was fixed by adding the Maven JAR plugin to include the main class in the manifest.

5. The Ansible deployment failed initially because the Docker Python module was missing. Adding the task to install Docker via `ansible.builtin.pip` resolved the issue.

6 Conclusion

This project successfully demonstrated the end-to-end implementation of a CI/CD pipeline for a Java application using modern DevOps tools. Starting from coding and version control with Git and GitHub, to automated builds with Jenkins, containerization with Docker and deployment automation using Ansible, each phase reflected a key DevOps principle of integration, automation and consistency. The use of Ngrok for webhook integration allowed automatic triggering of Jenkins pipelines, ensuring continuous integration. Docker and Ansible together provided reproducible environments and streamlined deployment, eliminating manual configuration errors.

Overall, this mini project gave me hands-on experience with the DevOps life cycle and reinforced its benefits including faster delivery, reliable automation, consistent environments and improved collaboration between development and operations. All the code used can be found in the GitHub repository and the latest Docker image can be found in the Docker Hub Repository.