CS 816 - Software Production Engineering

Mini Project - Scientific Calculator with DevOps

Uttam Hamsaraj - IMT2022524

Important Links

GitHub Repo Link: GitHub Repository
DockerHub Repo Link: DockerHub Repository

Introduction

The primary focus of this project is to demonstrate a modern software development lifecycle by first creating a functional command-line interface (CLI) scientific calculator. Following its development, a complete automation pipeline was built to handle all subsequent stages, including testing, packaging, and deployment. The entire process is designed to follow the core principles of DevOps, establishing a seamless and automated workflow.

What is DevOps?

DevOps is a modern approach to software production that combines the practices of software development (Dev) and IT operations (Ops). It focuses on improving collaboration between these two teams through shared processes and automation. The primary goal is to streamline the entire software delivery pipeline, from building and testing the code to its final release. This integration results in a more frequent and reliable software updates.

Why DevOps?

The importance of DevOps lies in its ability to solve the traditional delays that occur when development and operations teams work separately. By automating the release process and encouraging shared ownership, organizations can deliver software much faster and more efficiently. This increased speed enhances the stability and reliability of the final product. As a result, organizations can better respond to customer needs and market changes, fostering a culture of continuous improvement that consistently delivers value.

Tools Used

This project integrates several key DevOps tools to create a seamless pipeline. Each tool serves a specific purpose in the automation workflow:

- **Git & GitHub:** A version control system (Git) and a repository hosting platform (GitHub) used for source code management.
- **Python:** The core programming language used to develop the scientific calculator's logic and user interface.
- **unittest (PyUnit):** Python's built-in framework was used to write and execute unit tests to ensure the calculator's functions are correct.
- **Jenkins:** : An automated build, test, and deployment solution for continuous integration and continuous delivery (CI/CD).
- **Docker:** Used to containerize the Python application, packaging it with all its dependencies into a lightweight, portable Docker image.
- **Ansible:** A configuration management tool for managing configurations that automates infrastructure and application deployment and maintenance.
- **Ngrok:** A tunneling service that exposes a local server to the public internet. It provides a temporary URL for the local Jenkins server, making it possible to receive the incoming notification from the GitHub webhook.
- **GitHub Webhooks:** A feature within GitHub that automatically sends a notification when a specific event, like a `git push`, occurs in the repository. In this project, it was configured to send the trigger signal to start the Jenkins pipeline.

Directory Structure

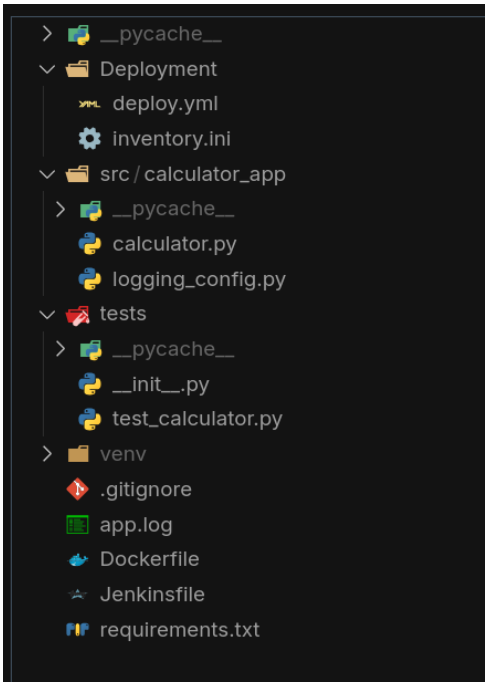


Figure 1: screenshot of Directory Structure

Description of Files and Folders

Folders

- **Deployment/**
  Contains deployment-related configuration files such as Ansible playbooks and inventory.
- **src/calculator_app/**
  Main source code folder for the scientific calculator application.
- **tests/**
  Holds unit tests and related test files for verifying the calculator's functionality.

Files

- **app.log**
  Log file generated by the application for debugging and tracking runtime activity.
- **Dockerfile**
  Contains instructions to containerize the calculator application using Docker.
- **Jenkinsfile**
  Defines the Jenkins CI/CD pipeline stages for building, testing, and deploying the app.
- **.gitignore**
  Specifies which files and folders should be ignored by Git version control.
- **requirements.txt**
  Lists the Python dependencies required for the project.

Steps to Run the Project

1. **Clone the Repository**
   ```
   git clone <repository-url>
   cd ScientificCalculator
   ```
2. **Create and Activate Virtual Environment**
   ```
   python3 -m venv venv
   source venv/bin/activate
   ```
3. **Install Dependencies**
   ```
   pip install -r requirements.txt
   ```
4. **Run the Application**
   ```
   python3 -m src.calculator_app.calculator
   ```
5. **Run Tests**
   ```
   python3 -m unittest discover -v tests
   ```

CLI Implementation and Testing

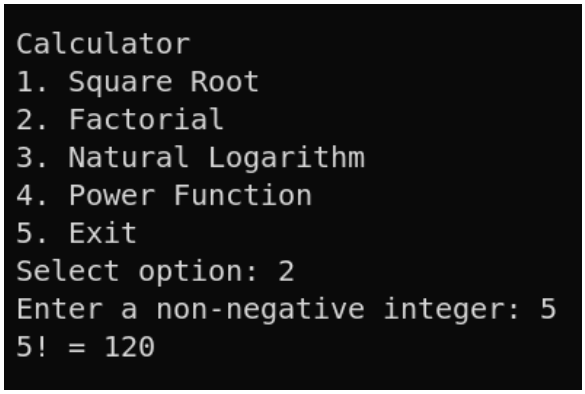
Firstly, I've written code for implementation of the command-line interface (CLI) for the calculator and the unit tests created to verify its functionality and a logging system to record its runtime events.

CLI Implementation

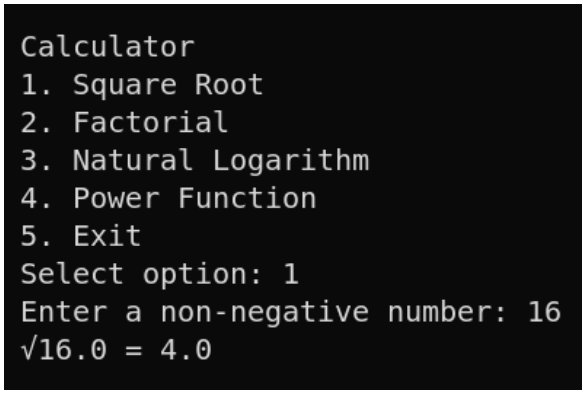
calculator.py : This file contains implementation of calculator. It operates as a menu-driven command-line interface (CLI) where user is presented with menu.
It has 4 main features

- Square Root ($\sqrt{x}$)
- Factorial ($x!$)
- Natural Logarithm, base e ($\ln(x)$)
- Power Function (x^b)

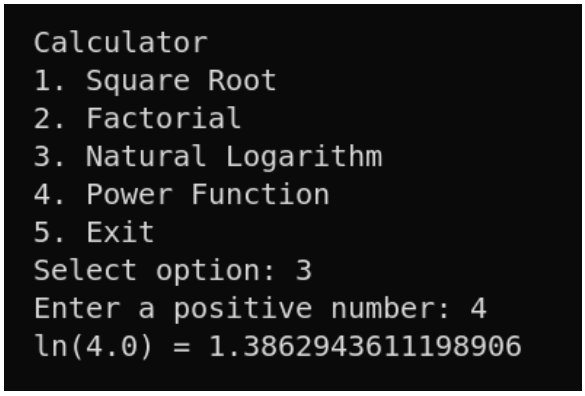
It handles all edge cases.



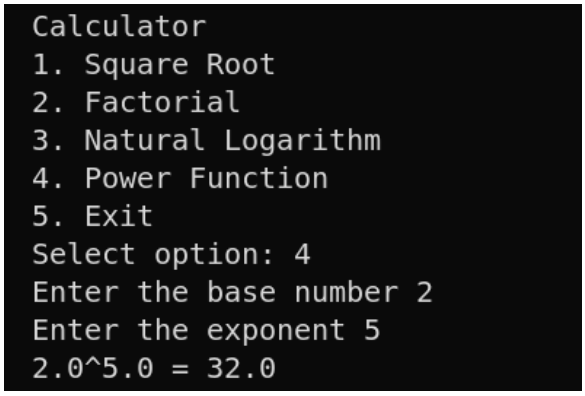
(a) Factorial



(b) Square root



(c) Natural Log



(d) Power

Figure 2: Screenshot of the scientific calculator application and its various operations

.

logging_config.py : This file configures the application's logging system. It is set up to record all runtime events and save them to the `app.log` file, which is created in the project's root directory.

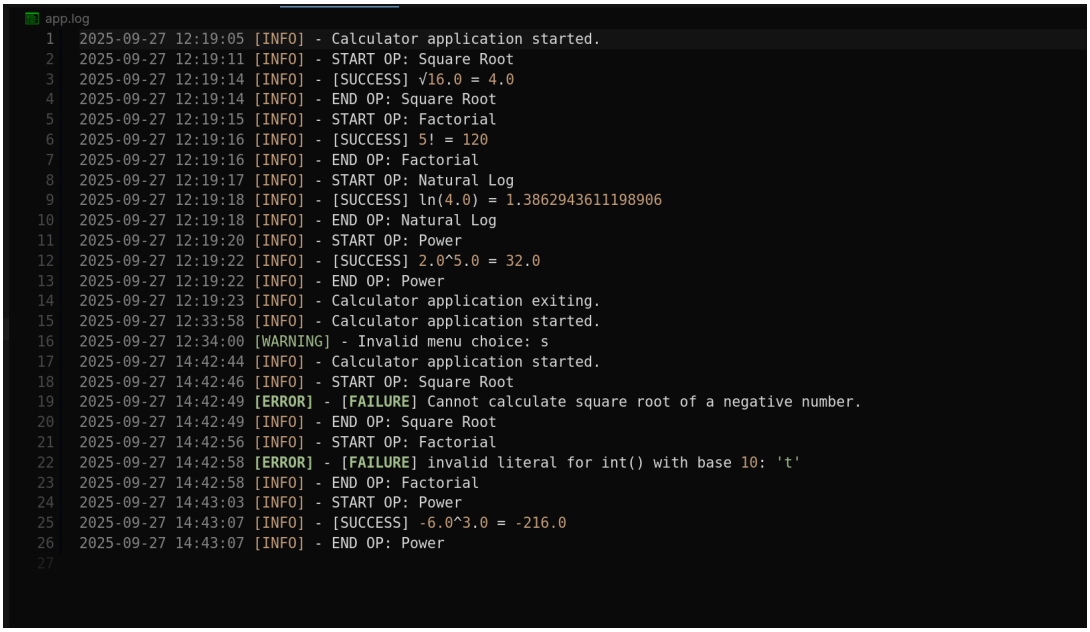


Figure 3: Screenshot of app.log

Testing

test_calculator.py : This file defines the unit tests created using Python's built-in unittest framework (PyUnit). The purpose of these tests is to verify the correctness and reliability of the core mathematical functions in calculator.py.
Within each test method, assertion functions are used to automatically check if the code's actual output matches an expected outcome.
This tests covers :

- Standard inputs to verify correct calculations.
- Edge cases such as zero, one, negative numbers etc.
- Error handling for mathematically invalid operations.
- Invalid data types.

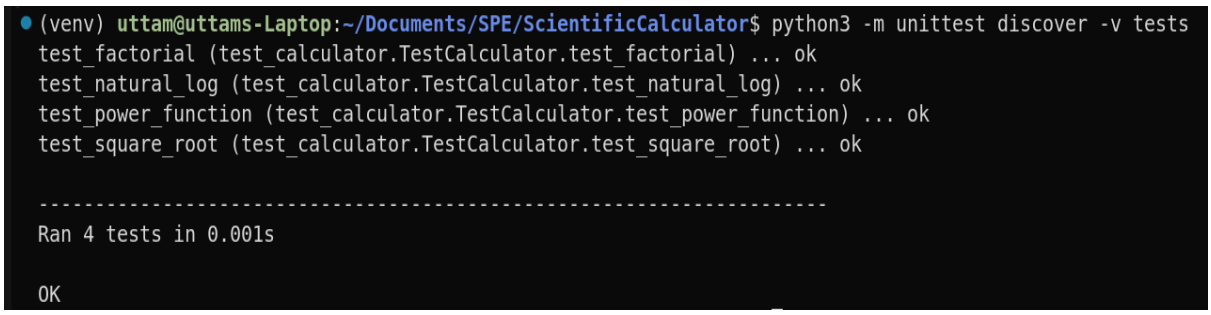


Figure 4: Screenshot of console output from test runner

Source Code Management

Source code management (SCM) is the practice of tracking, managing, and controlling changes in the codebase to ensure collaboration, versioning, and rollback capabilities.

In this project I have used Git as the version control system with GitHub as the remote repository hosting platform. GitHub was chosen because it is free, widely adopted, integrates seamlessly with Jenkins, and supports automated workflows via webhooks.

Local Repository Setup

The project folder was initialized as a local Git repository to track changes in the codebase. The command used for this initialization was $ git init..

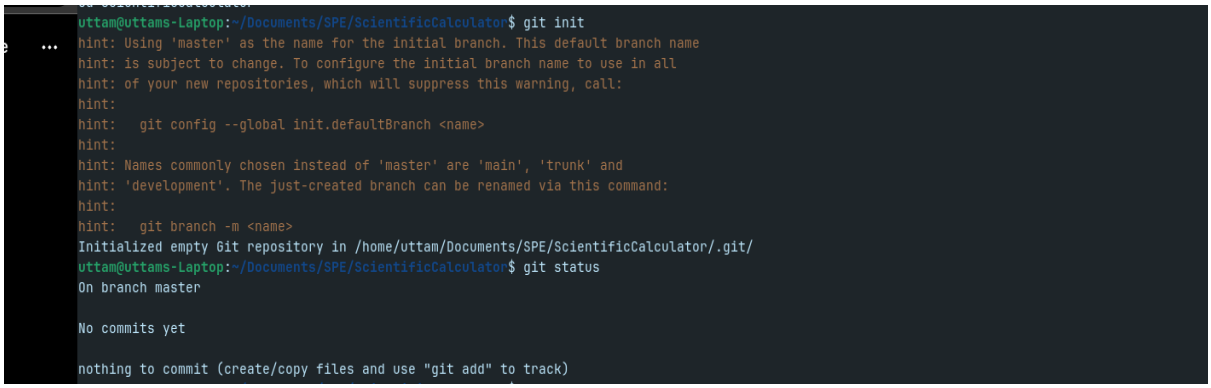


Figure 5: Screenshot of Local Git repository initialization

Remote GitHub Repository Creation

A remote repository was created on GitHub to host the project and serve as a central codebase.

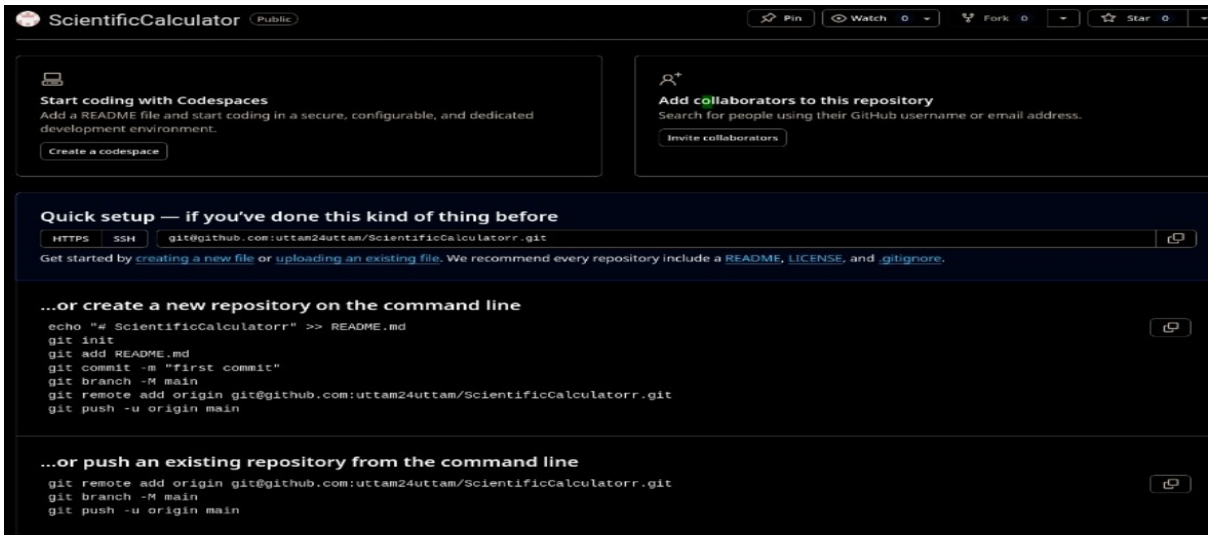


Figure 6: Screenshot of GitHub repo creation

Linking local repository to GitHub

The local repository was linked to the remote GitHub repository, and the initial code was pushed to the main branch.
In this project, SSH keys were configured to securely connect the local machine to GitHub.

To link the repository, the following command was used:
$ git remote add origin git@github.com:uttam24uttam/ScientificCalculator.git

To push the initial code, the following command was used
$ git push -u origin main

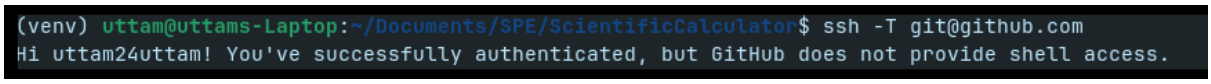


Figure 7: Screenshot of ssh key

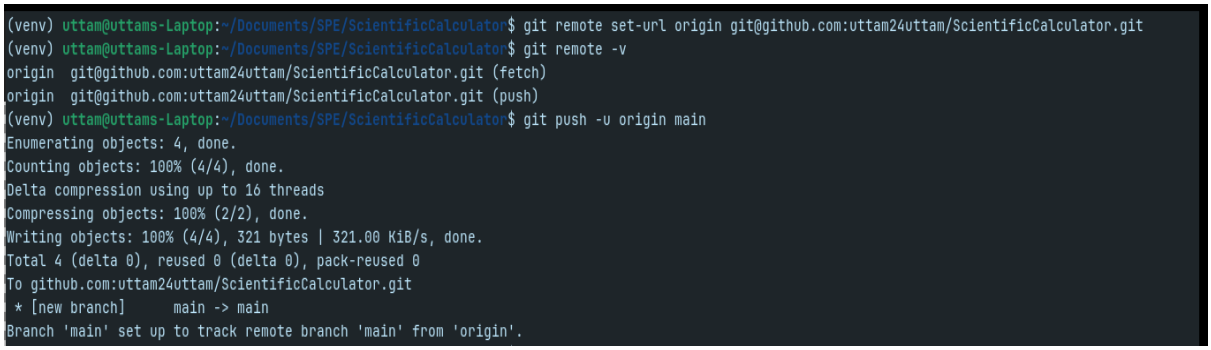


Figure 8: Screenshot of pushing initial code

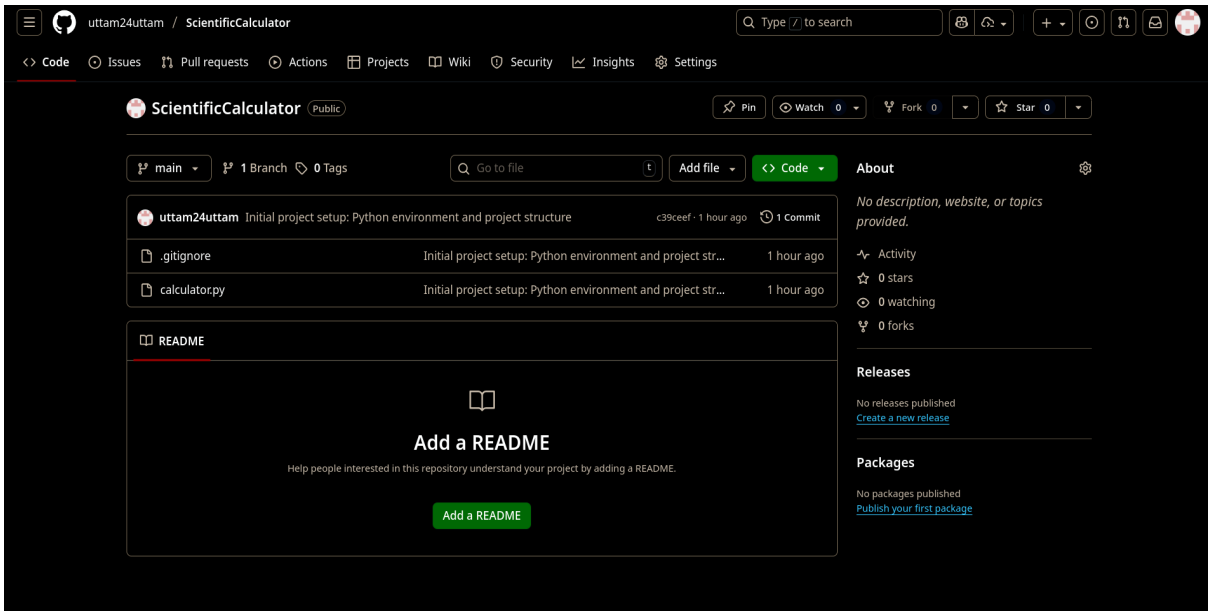


Figure 9: Screenshot of Newly created GitHub repository

.

For all future changes, I used the following commands to update the local repository and push changes to the remote repository.

$ git add
$ git commit -m "commit message"
$ git pull origin main
$ git push origin main

Docker

Docker and containers simplify software development and deployment by packaging an application along with all its dependencies into a single, lightweight environment called a container. Containers ensure that applications run consistently across different systems, from development to production.
Docker is a popular platform for creating, managing, and sharing containers. It helps developers and operations teams work together, improves resource usage, and makes applications easier to scale and move across environments. Containers are now a key part of modern software development, enabling faster and more reliable deployment of applications.

Installing Docker

Command used to install docker : $ sudo apt install docker.io

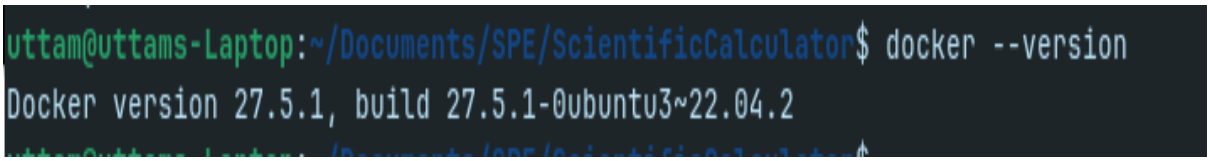


Figure 10: Screenshot of installed Docker version.

Dockerfile Creation

Dockerfile was created to package the Python application along with its dependencies into a container. The working directory, dependencies, and the command to run the calculator were defined

WORKDIR /app – Set working directory inside container to /app.
COPY requirements.txt . – Copy requirements.txt into /app.
RUN – Install Python dependencies without cache.
COPY ./src /app/src – Copy application source code into container.
ENV PYTHONPATH=/app – Set Python module search path to /app.

```
Dockerfile
1  FROM python:3.12-slim
2  WORKDIR /app
3  COPY requirements.txt .
4  RUN pip install --no-cache-dir -r requirements.txt
5  COPY ./src /app/src
6  ENV PYTHONPATH=/app
7  CMD ["python", "-m", "src.calculator_app.calculator"]
```

Figure 11: Screenshot of Dockerfile

Building Docker Image and Running Docker Container Locally

The Docker image was built from the Dockerfile, packaging the application and its dependencies into a portable container.

Command used for building : $ docker build -t scientific-calculator:1.0 .

The container was then run locally to verify the application works correctly in the Docker environment.

```
uttam@uttams-Laptop:~/Documents/SPE/ScientificCalculator$ sudo docker images
[sudo] password for uttam:
REPOSITORY                              TAG       IMAGE ID       CREATED         SIZE
uttamhamsaraj24/scientific-calculator   1.0       45ffb3fde02b   22 hours ago    127MB
uttamhamsaraj24/scientific-calculator   <none>    083cd9cb8b55   23 hours ago    127MB
uttamhamsaraj24/scientific-calculator   <none>    051128fbb5d3   24 hours ago    127MB
uttamhamsaraj24/scientific-calculator   <none>    9443dbafd941   32 hours ago    127MB
<none>                                  <none>    9760a9210cca   32 hours ago    127MB
uttamhamsaraj24/scientific-calculator   <none>    5cc59371e285   32 hours ago    127MB
```

Figure 12: Screenshot of Docker image, which is the containerized version of app

```
uttam@uttams-Laptop:~/Documents/SPE/ScientificCalculator$ docker run -it scientific-calculator:1.0

Calculator
1.Square Root
2.Factorial
3.Natural Logarithm
4.Power Function
5.Exit

Select option
2

Enter a non-negative integer
4
4! = 24
```

Figure 13: Screenshot of container running locally

.

Docker hub Login and repo creation

A remote public repository was created on Docker Hub to store the project's Docker image and make it accessible for deployment.
Personal Access Token was used for Login.

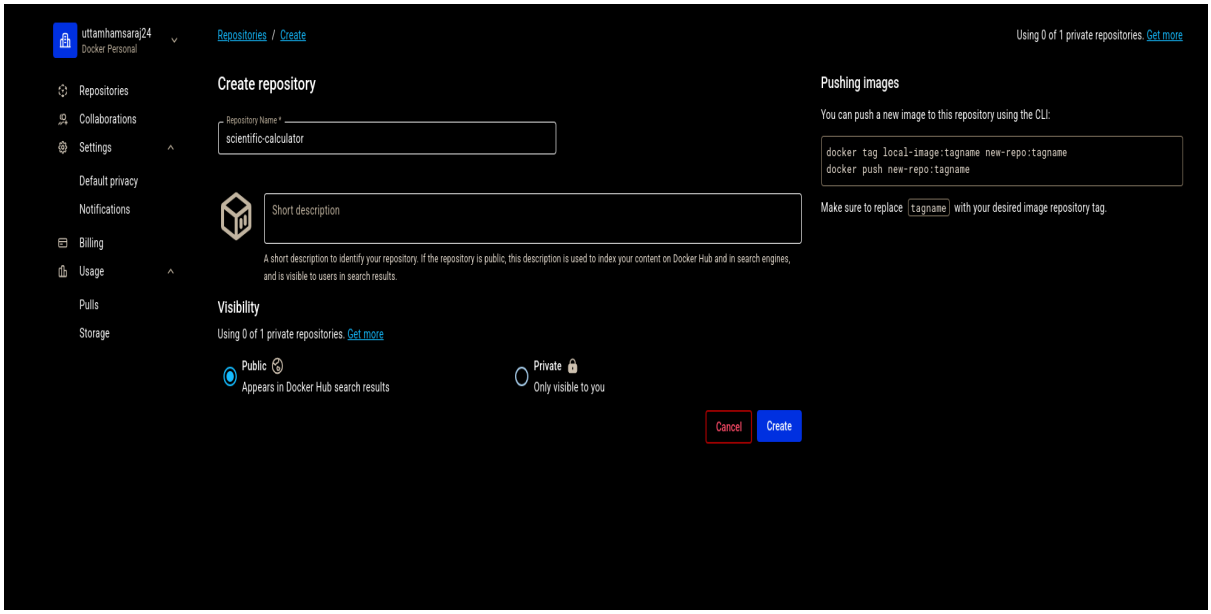


Figure 14: Screenshot of creating a public Docker hub repository

.

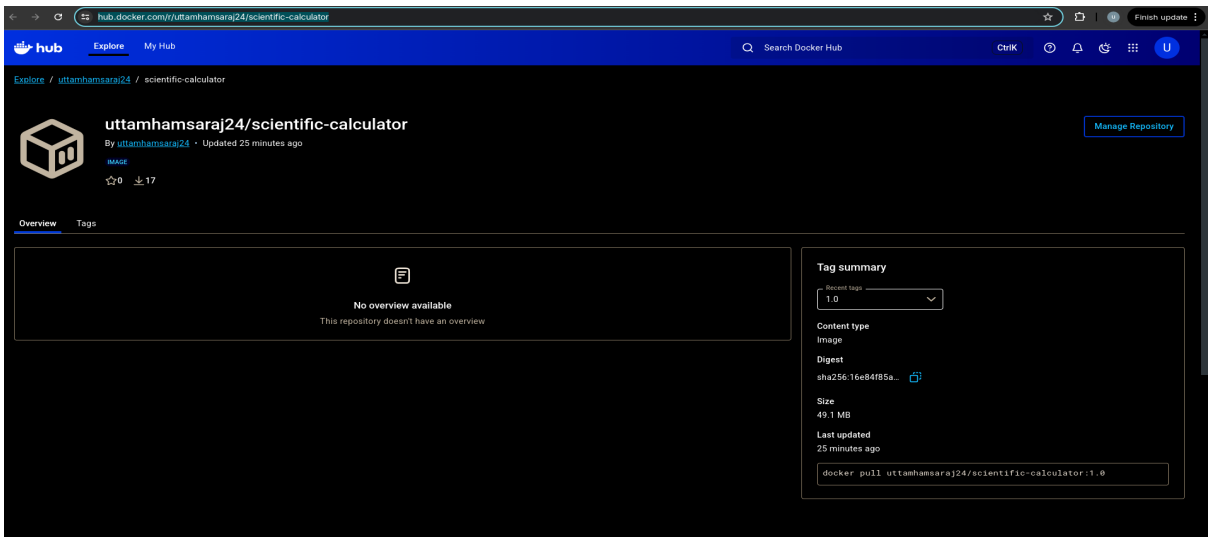


Figure 15: Screenshot of Newly created Docker hub repository

.

Tagging the Docker image and pushing it to Docker Hub.

The local Docker image which was built was first tagged to match the remote Docker Hub repository. After tagging, the image was pushed to Docker Hub.

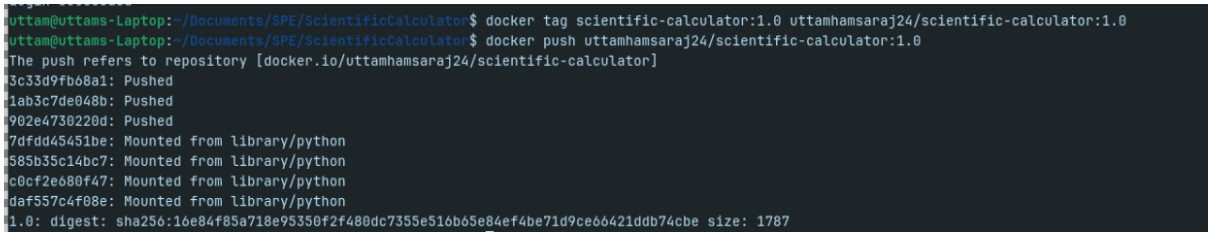


Figure 16: Screenshot of tagging the Docker image and pushing it to Docker Hub.

So to summarize the flow of Docker:

-We first write a Dockerfile which prepares the instructions for containerizing app.
-We then containerize the application by building a Docker image that contains the app along with all its dependencies..
-We then do tagging, which assigns a repository name and tag to the image for pushing and prepares the image for registry upload.
-We then push container into Docker Hub.
Note : Jenkins automates all the above steps. I have integrated these steps to Jenkins pipeline which is explained in later sections

Ansible

Ansible is an open-source automation tool used for configuration management, application deployment, and task automation across multiple servers. It is agentless, easy to use, and works with a wide range of platforms and cloud providers.

Installing Ansible

$ sudo apt install -y software-properties-common
$ sudo add-apt-repository –yes –update ppa:ansible/ansible
$ sudo apt install -y ansible

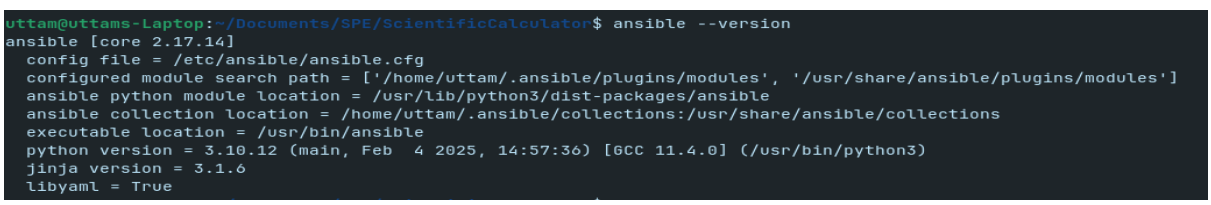


Figure 17: Screenshot of Installed Ansible version.

Playbook

Ansible Playbooks provide a simple, reusable, repeatable method of configuration management. Playbooks are collections of one or more plays performed in a specific sequence. An ordered series of tasks against selected hosts from your inventory is called a play. Plays outline the necessary tasks. Every play has a list of tasks to be completed and a set of hosts to configure.
This below given playbook (deploy.yml) does three main tasks : removing the old container, pulling the latest image, and running the scientific calculator application in Docker on the target hosts which are defined in inventory file (inventory.ini). Target host defined is localhost.

Deployment > deploy.yml

```yaml
- name: Deploy Scientific Calculator Docker Image
  hosts: myservers
  become: true
  tasks:
    - name: Remove old container
      community.docker.docker_container:
        name: scientific_calculator
        state: absent
        force_kill: yes

    - name: Pull latest Docker image
      community.docker.docker_image:
        name: uttamhamsaraj24/scientific-calculator
        tag: "1.0"
        source: pull

    - name: Run the interactive Docker container
      community.docker.docker_container:
        name: scientific_calculator
        image: uttamhamsaraj24/scientific-calculator:1.0
        state: started
        restart_policy: "no"
        detach: yes
        interactive: yes

```

Figure 18: Screenshot of Ansible Playbook (deploy.yml)

Target Host Connectivity and Playbook Deployment

First, I used Ansible to verify connectivity with the target hosts (LocalHost) to ensure they were reachable. After confirming connectivity, the Ansible playbook (deploy.yml) was executed to automate deployment of the Scientific Calculator Docker container. As mentioned earlier, this includes removing any existing container, pulling the latest Docker image from Docker Hub, and running the container on the target hosts.

Commands Used:
$ ansible -i inventory.ini myservers -m ping
$ ansible-playbook -i inventory.ini deploy.yml

Note: Tasks performed using Ansible, including verifying host connectivity and playbook deployment is integrated to Jenkins pipeline and automated, which is shown later.

Continuous Integration Using Jenkins

Jenkins is an open-source automation server used to streamline software development through continuous integration and continuous delivery (CI/CD). It automates the building, testing, and deployment of applications, helping teams work faster and reduce errors. With its plugins and easy-to-use web interface, Jenkins allows efficient monitoring and management of automation tasks.

I created a 6-stage Jenkins pipeline to implement continuous integration for the project which automates code checkout, testing, Docker image creation and deployment using Ansible.
Jenkinsfile is shown below which defines the stages and steps to run in each stage. Jenkins automatically fetches this file from the GitHub repository when the pipeline is triggered and executes the defined stages. We need to configure these during pipeline setup.

Jenkinsfile

```
pipeline {
    agent any
    environment {
        PYTHON = "python3"
        DOCKER_IMAGE = "uttamhamsaraj24/scientific-calculator:1.0"
        DOCKER_CRED = "docker-hub-cred"
    }
    stages {
        //STAGE 1
        stage('Checkout Code') {
         steps {
            git branch: 'main', url: 'https://github.com/uttam24uttam/ScientificCalculator.git'
         }
        }
        //STAGE 2
        stage('Install Requirements') {
         steps {
            sh "${PYTHON} -m pip install --no-cache-dir -r requirements.txt"
            }
        }
        //STAGE 3
        stage('Run Tests') {
            environment {
                PYTHONPATH = 'src'
            }
            steps {
                sh "${PYTHON} -m unittest discover -s tests"
            }
        }
        //Stage 4
        stage('Build Docker Image') {
            steps {
                sh "docker build -t ${DOCKER_IMAGE} ."
            }
        }
        //Stage 5
        stage('Push Docker Image') {
            steps {
                withDockerRegistry([credentialsId: "${DOCKER_CRED}", url: '']) {
                    sh "docker push ${DOCKER_IMAGE}"
                }
            }
        }
        //Stage 6
        stage('Deploy with Ansible') {
            steps {
                ansiblePlaybook(
                    playbook: 'Deployment/deploy.yml',
                    inventory: 'Deployment/inventory.ini',
                    installation: 'Ansible-default'
                )
            }
        }
```

Figure 19: Screenshot of the Jenkinsfile used to configure the CI/CD pipeline

Jenkins Configuration and Pipeline Setup

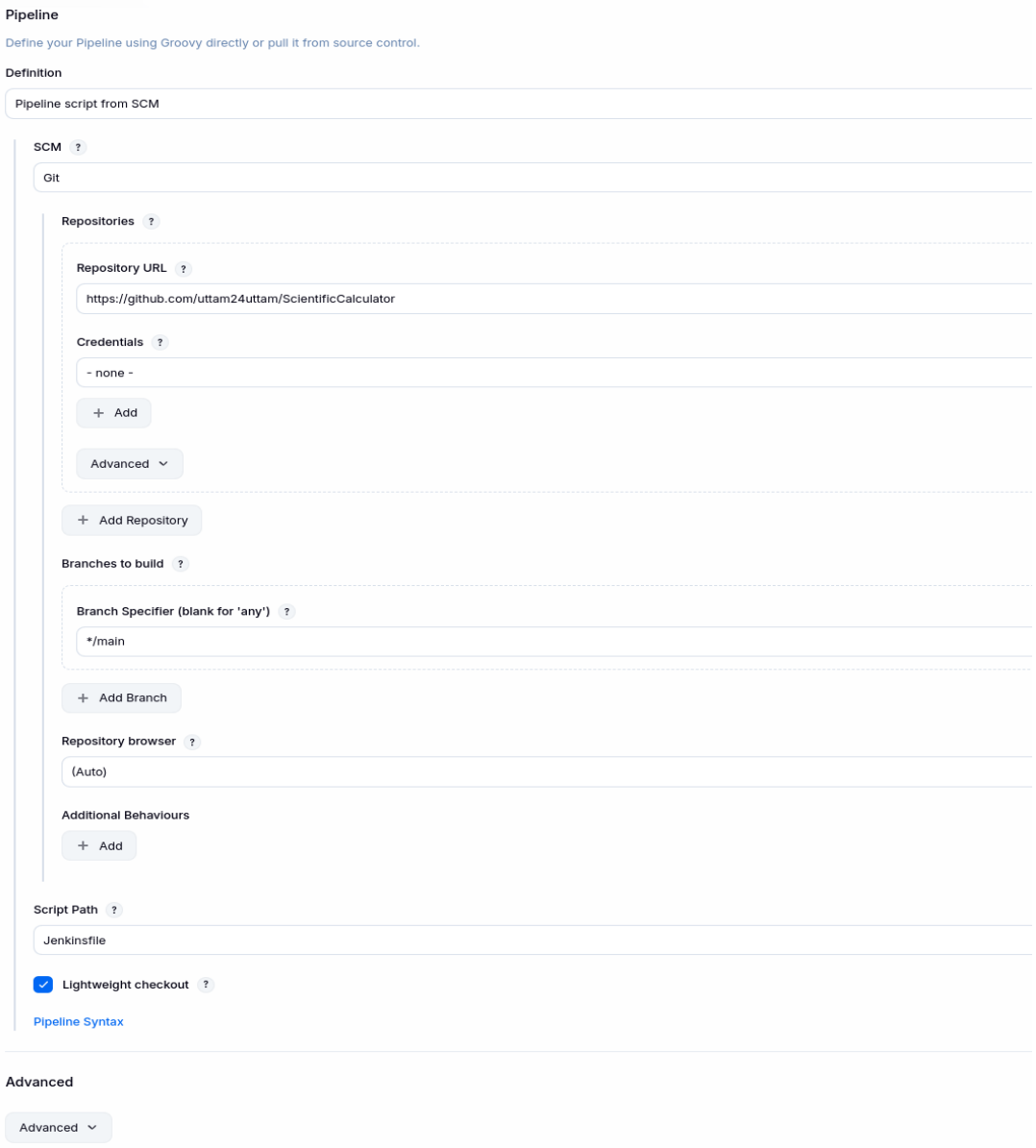


Figure 20: Screenshot of pipeline configuration

The following steps were performed to configure the Jenkins pipeline:

- Created a new Pipeline job in Jenkins.
- Selected Pipeline script from SCM.
- Chose Git as the source control management tool.
- Provided the GitHub repository URL
- Specified the main branch to build.
- Set the Script Path to "Jenkinsfile".

This setup allows Jenkins to automatically fetch the pipeline script from the GitHub repository and execute it whenever changes are pushed.

Jenkins Global Credentials

GitHub SSH Credential: This credential allows Jenkins to securely access your GitHub repository using SSH without needing a password every time. It is required to fetch the pipeline (Jenkinsfile) and project code during builds. When Jenkins runs the pipeline, it uses this credential to securely clone the repository and fetch the latest code from GitHub.

ID : github-ssh is used in the Jenkinsfile to tell Jenkins which credential to use.
Username : GitHub account username.
Private Key : The SSH private key
Scope : Global so that all jobs on this Jenkins instance can use it

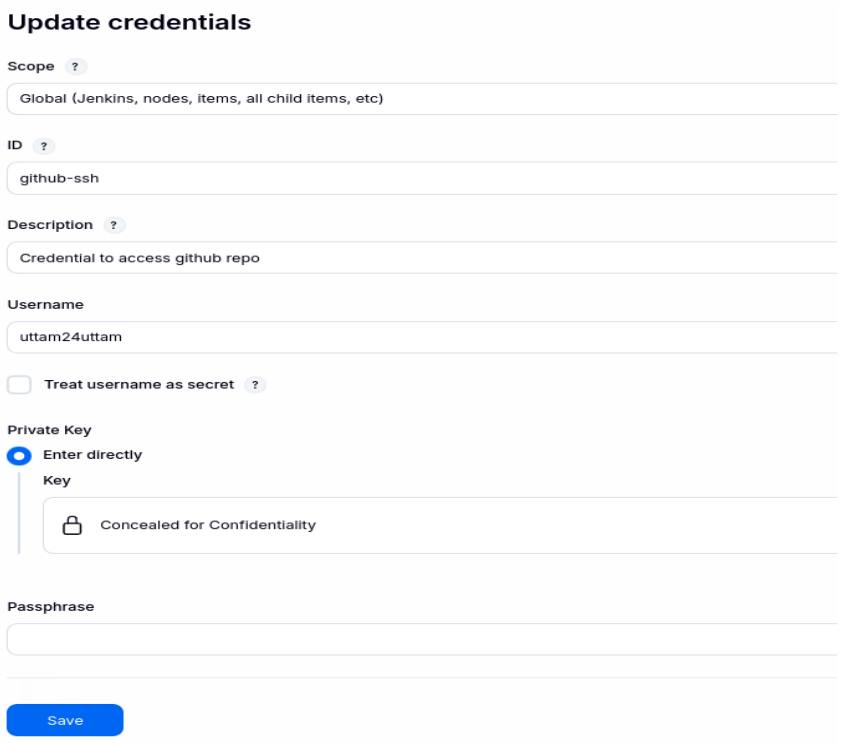


Figure 21: Screenshot of the GitHub SHH credentials

.

Docker Hub Credential: This credential allows Jenkins to authenticate with Docker Hub to push Docker images built in the pipeline.
In the pipeline's Push Docker Image stage (Stage 5), Jenkins uses this credential to log in to Docker Hub and push the built image (scientific-calculator:1.0) to repository.
.

ID : docker-hub-cred used in the Jenkinsfile for Docker steps.
Username : Docker Hub account username.
Private Key : Personal access token
Scope : Global so that it can be used by any pipeline job needing Docker Hub access.

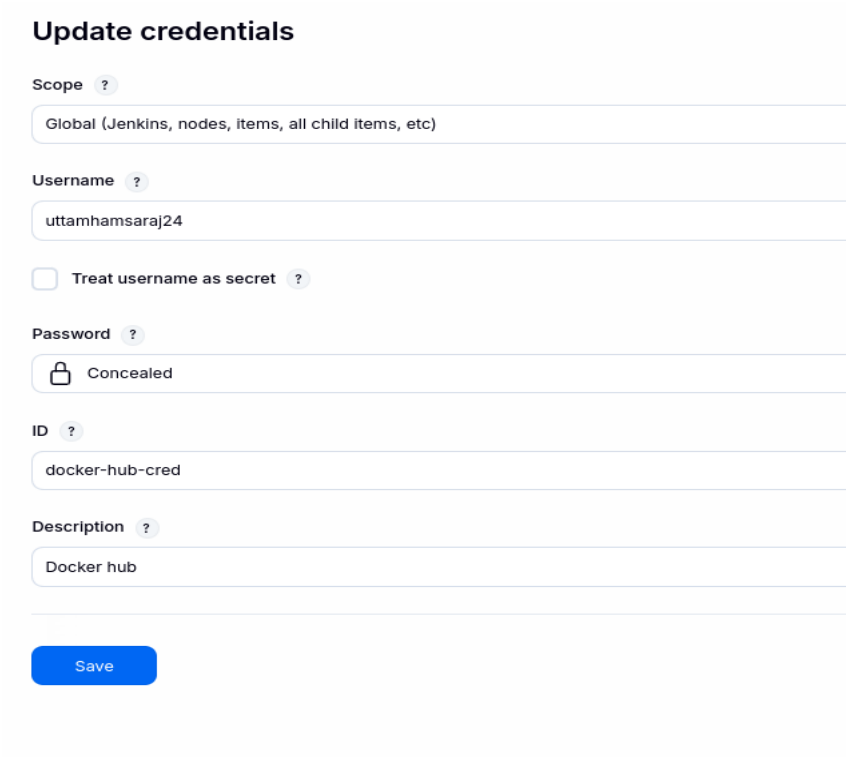
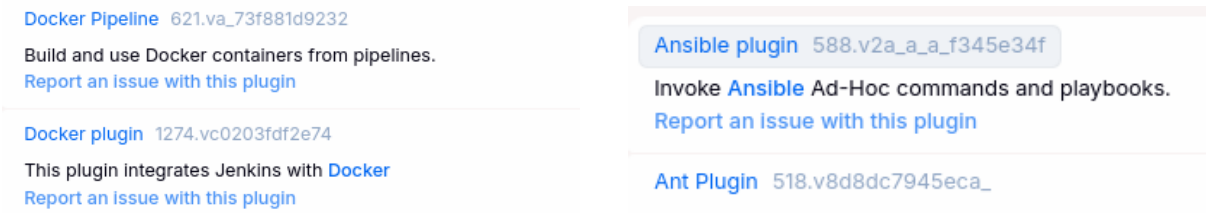


Figure 22: Screenshot of the Docker Hub credentials

Plugins installed in Jenkins

Installed Docker plugin to allow Jenkins to build and push Docker images directly from the pipeline.
Installed Ansible plugin to enable Jenkins to run Ansible playbooks for automated deployment.



(a) Docker Plugins (b) Ansible Plugins

Figure 23: Sreenshot of installed plugins

.

Now let us look at all the stages one by one.

STAGE 1: Checkout Code

This stage fetches the latest source code from the GitHub repository . Jenkins automatically checks out the code from the specified branch (main) that we have specified during pipeline configuration to use in subsequent stages.
The code for this stage in Jenkinsfile can be seen in **Figure** 19.

STAGE 2: Install Requirements

In this stage, all Python dependencies required for the Scientific Calculator are installed. This ensures that the Jenkins agent has the necessary libraries for the application to run correctly.

Jenkins executes the command: sh "${PYTHON} -m pip install –no-cache-dir -r requirements.txt"

The code for this stage can be seen in **Figure** 19.

STAGE 3: Run Tests

In this stage, the pipeline runs all unit tests defined in the project using Python's unittest framework. The environment variable PYTHONPATH is set to ensure the src folder is included, allowing Python to locate the calculator modules.

Jenkins executes the command: sh "${PYTHON} -m unittest discover -s tests"

This discovers all test files in the tests directory and runs them automatically, validating that the calculator functions work as expected. The code for this stage can be seen in **Figure** 19.

STAGE 4: Build Docker Image

In this stage, Jenkins builds a Docker image of the Scientific Calculator application using the provided Dockerfile.
The image is tagged with the name defined in the environment variable ${DOCKER_IMAGE} which is define in Jenkinsfile . This ensures that the application is packaged in a consistent, portable format that can be deployed across any environment.

Jenkins executes the command: sh "docker build -t ${DOCKER_IMAGE}

The code for this stage can be seen in **Figure** 19.

STAGE 5: Login and Push Docker Image

In this stage, the Docker image built in the previous step is pushed to Docker Hub. To handle the secure login and authentication, Jenkins uses the Docker Hub credentials (with ID docker-hub-cred) configured in the Jenkins global credentials store.

The withDockerRegistry block in the Jenkinsfile, which is provided by the Docker Pipeline plugin, ensures that the pipeline logs in to Docker Hub, pushes the tagged image, and then logs out automatically once the operation is completed.

The code for this stage can be seen in **Figure** 19.

STAGE 6: Deploy with Ansible

In this stage, Jenkins triggers Ansible to handle the deployment of the Scientific Calculator container. The Ansible plugin is used to run the deploy.yml playbook against the hosts defined in the inventory.ini file.
The playbook first removes any existing container,then pulls the latest image from Docker Hub, and finally starts the container on the target host(s). This ensures that the latest version of the application is always deployed automatically after a successful build and push.

The code for this stage can be seen in **Figure** 19.

In the code pluggin,
playbook → specifies the Ansible playbook to run.
inventory → points to the inventory file with target hosts.
installation → refers to the Ansible installation configured in Jenkins.

All 6 stages of the pipeline including building, testing, Docker image creation, pushing to Docker Hub, and deployment using Ansible ran successfully. This indicates that the CI/CD workflow is fully automated.

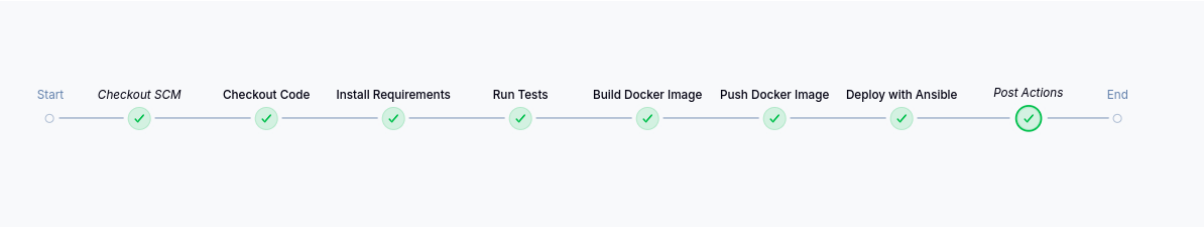


Figure 24: Screenshot of Jenkins pipeline execution showing all stages successfully completed.

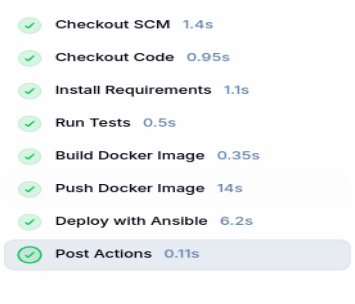


Figure 25: Screenshot of Jenkins pipeline execution showing all stages successfully completed.

Docker image successfully built and pushed to Docker Hub via Jenkins pipeline. It can be seen in the below **Figure** 26.

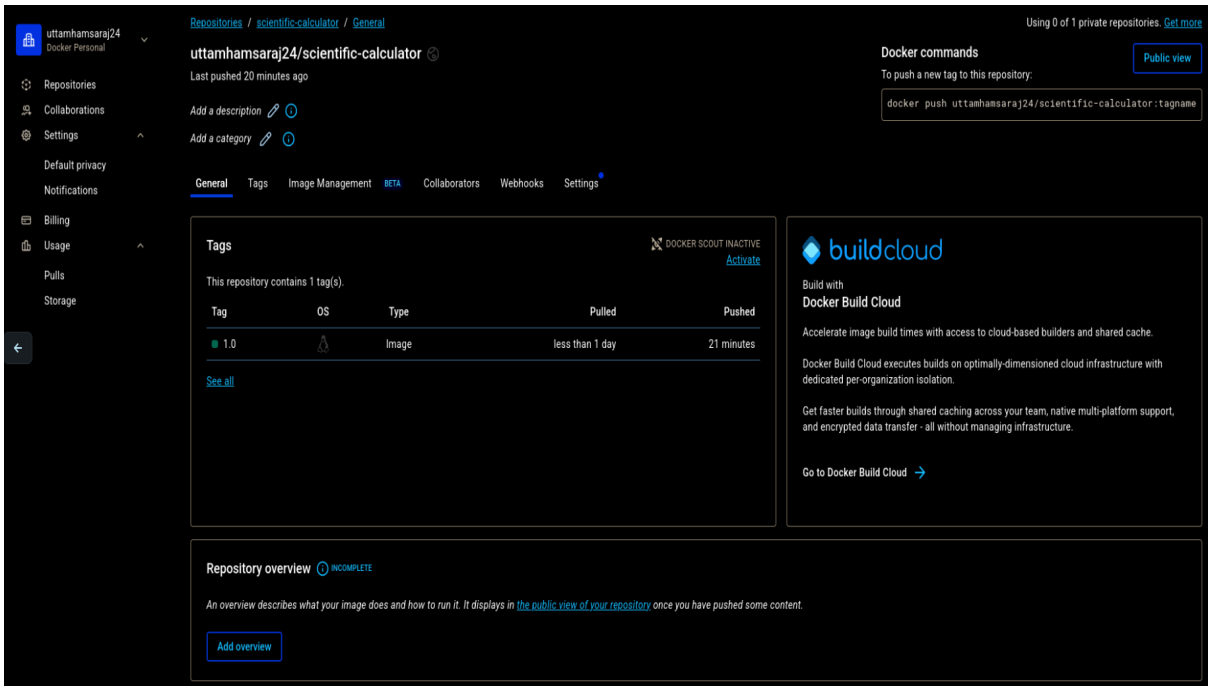


Figure 26: Screenshot showing the Docker image successfully pushed to Docker Hub as part of the Jenkins pipeline execution

Sending Email Notification in Post Action

In our pipeline, the post section of the Jenkinsfile executes after all deployment stages are complete. It's configured with success and failure conditions to automatically send an email notification detailing the final outcome of the build.
So after completion of the pipeline , the result is sent to the configured EMail id automatically.

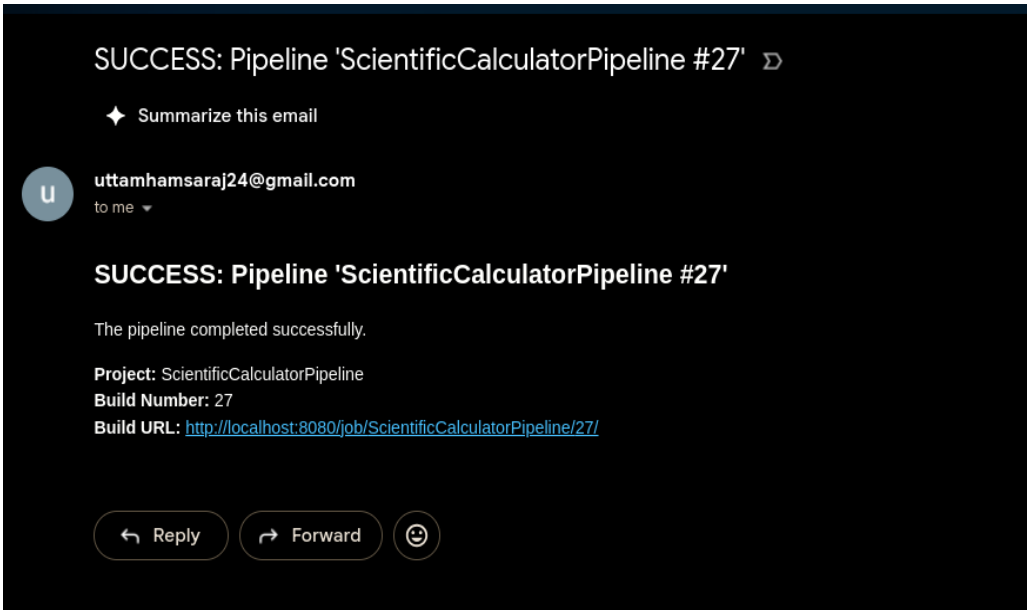


Figure 27: Screenshot of the automated email notification after a successful build

.

.

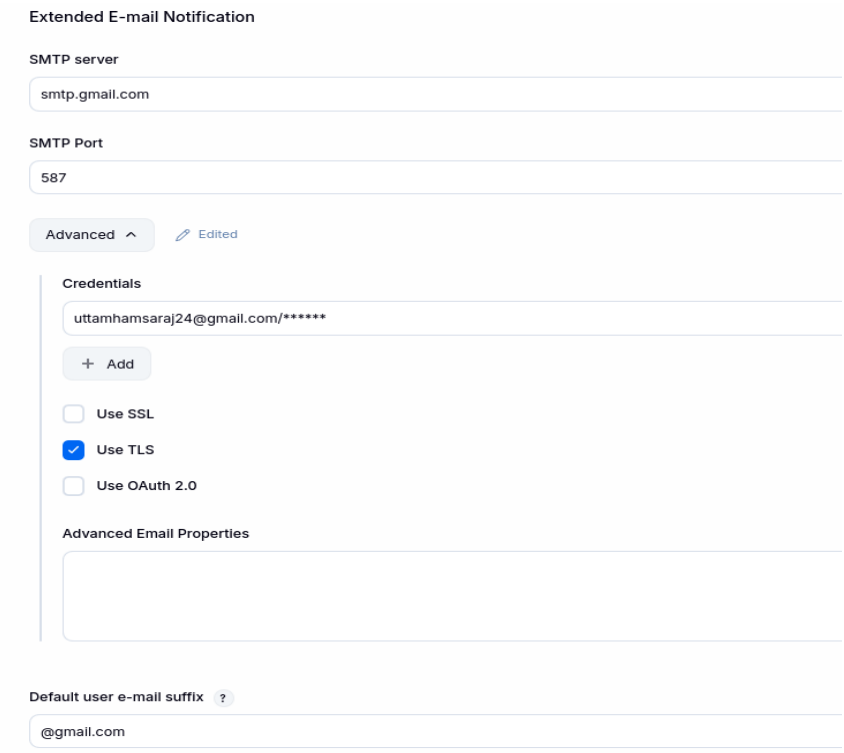


Figure 28: Screenshot of the Jenkins email notification configuration.

```
    //Sending Email Notification
post {
    success {
        emailext (
            to: 'uttamhamsaraj24@gmail.com',
            subject: "SUCCESS: Pipeline '${currentBuild.fullDisplayName}'",
            mimeType: 'text/html',
            body: """
                <h2>SUCCESS: Pipeline '${currentBuild.fullDisplayName}'</h2>
                <p>The pipeline completed successfully.</p>
                <p>
                    <strong>Project:</strong> ${env.JOB_NAME}<br>
                    <strong>Build Number:</strong> ${env.BUILD_NUMBER}<br>
                    <strong>Build URL:</strong> <a href="${env.BUILD_URL}">${env.BUILD_URL}</a>
                </p>
            """
        )
    }
    failure {
        emailext (
            to: 'uttamhamsaraj24@gmail.com',
            subject: "FAILED: Pipeline '${currentBuild.fullDisplayName}'",
            mimeType: 'text/html',
            body: """
                <h2>FAILED: Pipeline '${currentBuild.fullDisplayName}'</h2>
                <p>The pipeline failed. Please check the console output.</p>
                <p>
                    <strong>Project:</strong> ${env.JOB_NAME}<br>
                    <strong>Build Number:</strong> ${env.BUILD_NUMBER}<br>
                    <strong>Error:</strong> ${currentBuild.result}<br>
                    <strong>Console Output:</strong> <a href="${env.BUILD_URL}">Click here to view the logs</a>
                </p>
            """
        )
    }
}
```

Figure 29: Screenshot of Post section code in Jenkinsfile responsible for Email Notification.

Ngrok

Ngrok is a tool that creates a secure tunnel from the internet to a local server running on our computer. It allows us to share our local web application publicly without exposing our whole network. Ngrok is useful for testing webhooks or accessing local services from anywhere safely. In our project, we used Ngrok to expose the Jenkins server to the internet temporarily, allowing GitHub to trigger the pipeline via webhooks and test the CI/CD workflow without hosting Jenkins on a public server.
Command used after installing and unzipping ngrok : ./ngrok http 8080

As it can be seen in terminal in **Figure** 30. , Ngrok creates a temporary public URL for the local Jenkins server. It allows external services, such as GitHub, to access Jenkins for triggering builds via webhooks, without needing to host Jenkins on a public server permanently.

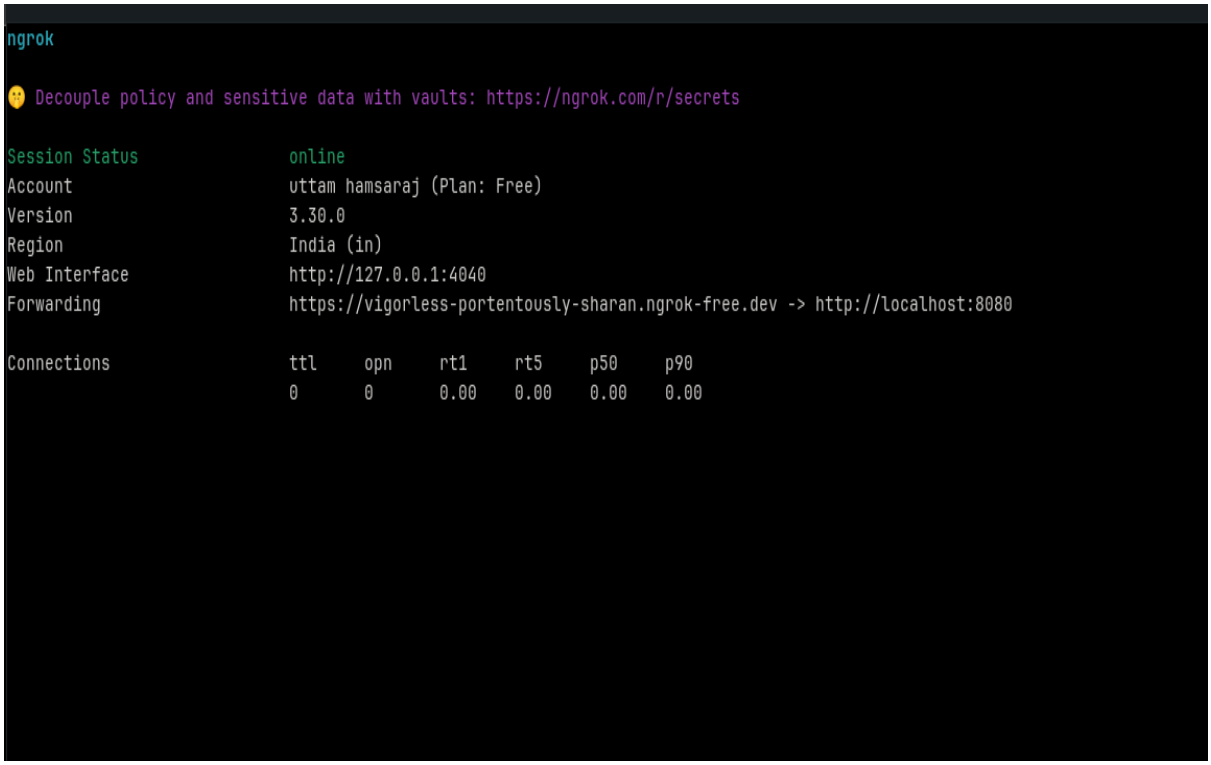


Figure 30: Screenshot showing Ngrok exposing the local Jenkins server.

Webhooks

Webhooks are a mechanism that allows one system to notify another in real time when a specific event occurs. In our project, GitHub webhooks are used to automatically trigger the Jenkins pipeline whenever code is pushed to the repository. Since our Jenkins server runs locally and is not publicly accessible , Ngrok was used to create a temporary public URL that points to the local Jenkins instance as seen in the previous section.
The temporary URL is configured in GitHub repository as the webhook endpoint, allowing github to notify Jenkins securely whenever there is a code push.

We configure the webhook in github as follows: (Can be seen in Figure
Payload URL: This is the public URL generated by Ngrok that points to the local Jenkins server. GitHub sends a POST request to this URL whenever the specified event occurs.

Content Type: This specifies that the data sent by GitHub in the webhook payload will be in JSON format, which Jenkins can parse and use to trigger the pipeline.

Triggers / Event: The webhook is triggered only when code is pushed to the GitHub repository. This ensures Jenkins builds run automatically on code changes. .

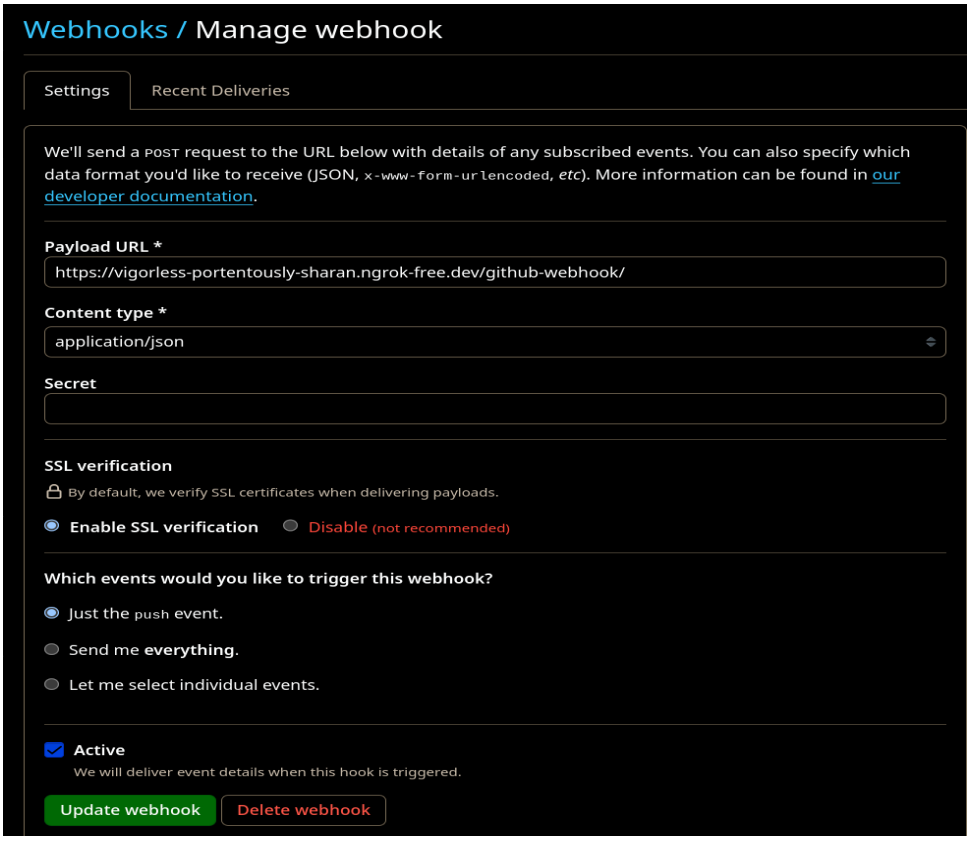


Figure 31: Screenshot of GitHub webhook configuration with Ngrok URL

Trigger Configuration on Jenkins (GITScm polling)

This setting allows Jenkins to listen for incoming webhook POST requests from GitHub. When a push is made to the repository, Jenkins automatically triggers the pipeline execution, enabling continuous integration without manual intervention.

Figure 32: Screenshot of Jenkins pipeline trigger configuration (GITScm polling)

.

Once we make sure that

- Ngrok is running
- Webhook is configured with the Ngrok URL
- Jenkins pipeline trigger (GITScm polling) properly set up,

Any push to the GitHub repository automatically triggers the pipeline. The webhook sends an HTTP POST request to the Ngrok public URL, which then triggers the Jenkins pipeline.

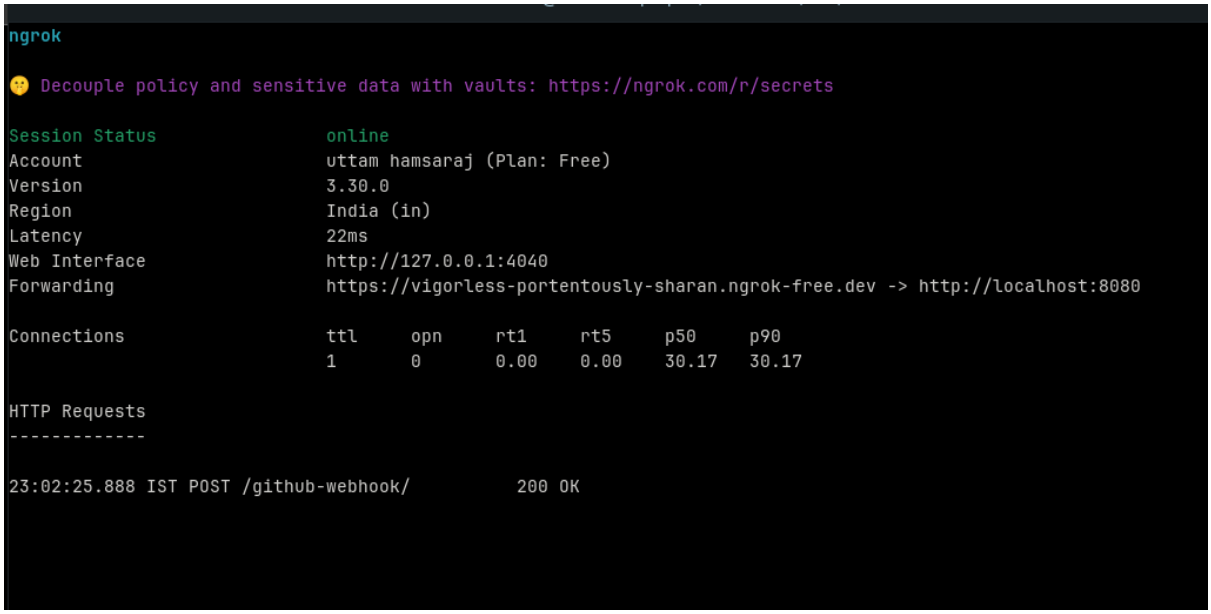


Figure 33: Screenshot of GitHub webhook POST requests received by Ngrok

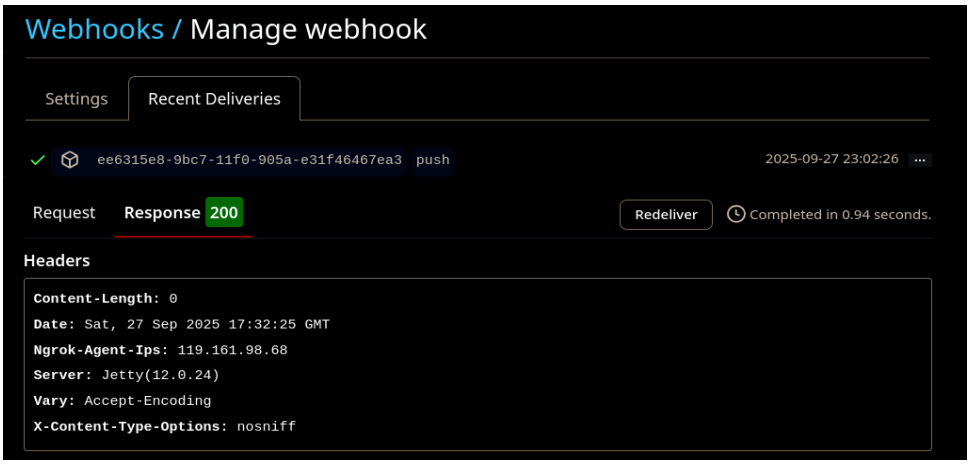


Figure 34: Screenshot of GitHub webhook delivery Response

.

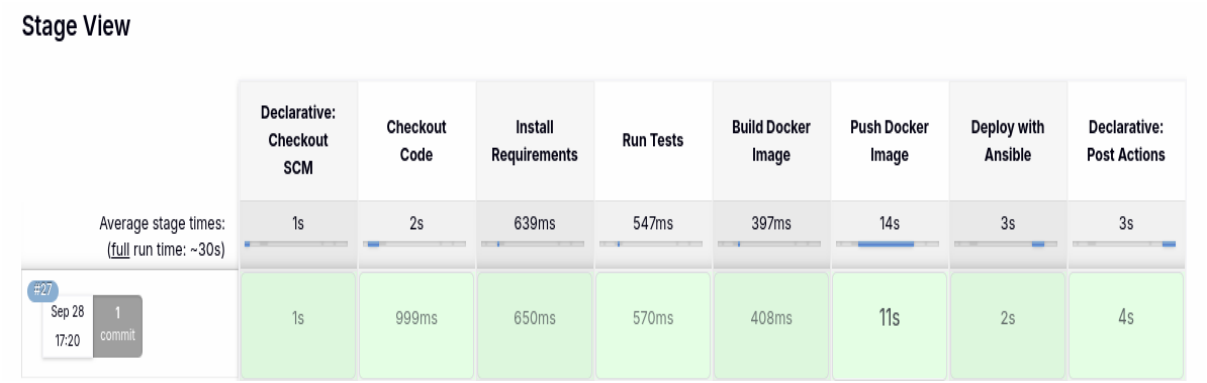


Figure 35: Screenshot of Jenkins pipeline execution showing all stages successfully completed after 1 push change.

With Ngrok and the GitHub webhook configured, every push to the repository triggers the Jenkins pipeline automatically, completing all stages without manual intervention.

Important Links

GitHub Repo Link: GitHub Repository
DockerHub Repo Link: DockerHub Repository