CS 816 - Software Production Engineering: Mini Project Report

Project Title: Scientific Calculator with End-to-End CI/CD Pipeline

Name: Aayush Bhargav

Roll Number:IMT2022089

GitHub Repository Link: https://github.com/Aayush-Bhargav/SPE_MiniProject

DockerHub Repository Link: https://hub.docker.com/u/aayushbhargav57

1. Introduction: What and Why of DevOps?

What is DevOps?

DevOps (Development and Operations) is a modern approach that integrates software development and IT operations to enable faster, more reliable software delivery. It focuses on automating the entire software life cycle — from code integration and testing to deployment and monitoring — while fostering close collaboration between developers, testers, and operations teams.

Rather than functioning as isolated departments, DevOps encourages shared ownership and continuous feedback throughout the development pipeline. This collaboration ensures that code changes move seamlessly from development to production while maintaining stability and quality. DevOps is both a **culture** and a **set of practices** that emphasize automation, continuous integration and deployment (CI/CD), monitoring, and rapid feedback loops. The ultimate goal is to deliver high-quality software quickly, consistently, and aligned with user needs.

Why Use DevOps?

Organizations adopt DevOps because it addresses several long-standing challenges in traditional software delivery models. The primary benefits include:

- **Speed and Agility:**
  DevOps enables faster and more frequent releases through automation and continuous integration. This agility allows teams to respond quickly to customer feedback and changing market demands.

- **Reliability:**
  Automated testing, continuous monitoring, and early error detection ensure that software is stable before reaching users. This reduces the likelihood of failures in production environments.

- **Scalability:**
  With tools like Infrastructure as Code (IaC) and containerization, DevOps allows teams to scale systems efficiently. Environments can be replicated and configured automatically, minimizing human error.

- **Collaboration and Culture:**
  DevOps bridges the gap between development, testing, and operations teams. It builds a culture of shared responsibility, where all members are collectively accountable for performance, stability, and user satisfaction.

- **Continuous Feedback:**
  Through automated monitoring and logging, teams receive real-time insights about system performance and user behavior. This feedback drives constant improvement in both code and processes.

Why DevOps is Preferred Over Traditional Models

Traditional models like **Waterfall** and even **Agile** have limitations that DevOps overcomes. In the **Waterfall model**, the software development process is linear and sequential. Each phase — from requirement gathering to deployment — happens one after the other, leading to long release cycles and delayed feedback. Problems often surface only at the final stages, making them expensive and time-consuming to fix.

The **Agile model** improved on this by introducing iterative development and faster delivery through sprints. However, Agile primarily focuses on collaboration and flexibility within the development team. It still often leaves a gap between development and operations — meaning code may be ready faster, but deployment and maintenance still involve manual steps and delays.

DevOps goes beyond Agile by extending the principles of iteration, automation, and collaboration to the entire software delivery pipeline. It eliminates the friction between development and operations by using automated deployment, continuous monitoring, and shared workflows. As a result, updates can be released multiple times a day rather than a few times a month. Moreover, since testing and deployment are automated, quality assurance becomes part of the ongoing process rather than a final checkpoint.

In essence, DevOps combines the development speed of Agile with the operational stability that traditional models aimed for. It transforms software delivery into a continuous, integrated process — reducing time to market while improving reliability and scalability.

2. Problem Statement and Tools Used

2.1 Problem Statement

The goal of this project was to implement a command-line Scientific Calculator program in Java, featuring the following menu-driven operations:

1. Square root function (sqrt(x))
2. Factorial function (!x)
3. Natural logarithm (ln(x))
4. Power function (x^b)

This application must be deployed via a complete **8-step CI/CD pipeline**.

2.2 Tools Used

Category	Tool	Description
Language	Java 17	Programming language used for the core application logic.
SCM	GitHub	Source Code Management and collaboration.
Build Tool	Apache Maven	Used to compile, package (JAR), and run unit tests.
Testing	JUnit 5	Framework used for writing and executing unit tests.
CI/CD Orchestration	Jenkins	Automation server to orchestrate the entire pipeline via the `Jenkinsfile`.

Containerization	Docker	Used to package the Java application and its environment into a portable image.
Registry	Docker Hub	Public cloud repository for storing and distributing the Docker image.
Deployment	Ansible	Configuration Management tool used to deploy the final container onto the local managed host.

3. Initial Development, Testing Setup, and Maven Configuration

Brief: The project began by establishing the core Java application logic and integrating the JUnit 5 testing framework, all managed and standardized by Apache Maven. This stage was essential for creating the tested, executable artifact required for the subsequent CI/CD pipeline stages.

Application Development and Structure: Development was initiated within the **IntelliJ IDEA** environment. The use of IntelliJ was deliberate, as its powerful, integrated features for **running and debugging JUnit test cases** provided immediate, superior feedback essential for test-driven development.

The project structure relied on a crucial partnership between the IDE and the build tool:

- **Apache Maven's Role:** Maven is a **Project Management and Comprehension Tool**. Its primary function is to define the project's entire lifecycle (dependencies, compiling, testing, and packaging) in a standardized, repeatable way via the `pom.xml` file.
- **IntelliJ's Partnership:** While Maven handles the build logic, IntelliJ acts as the user interface, reading the `pom.xml` to automatically configure the project, download dependencies, and provide shortcuts to execute Maven phases (like `mvn test`) without needing command-line interaction.

The source code was organized into the standard Maven directory structure using the package `com.spe.calculator`:

- **`Main.java`**: Located in `src/main/java`, implements the **menu-driven Command-Line Interface (CLI)**. It serves as the primary **entry point** for the application, handling user choices (1-5) and managing the application flow. Screenshots have not been included to compact the report.
- **`ScientificCalculator.java`**: Located in `src/main/java,` contains the **core business logic** for the required mathematical functions (sqrt(x), !x, ln(x), x^b). This file was built with robust exception handling, including a custom

`FactorialOverflowException` to manage large input limits, ensuring stability. Screenshots have not been included to compact the report.

- **`ScientificCalculatorTest.java`**: Located in `src/test/java`, this file houses the comprehensive **JUnit 5 test suite**. Test cases were designed to cover boundary conditions, zero, negative inputs, and large input overflow cases, proving the functional correctness of the code.

Maven (`pom.xml`) Configuration: The Build Gatekeeper The `pom.xml` serves as the central control file, configuring how the code is compiled, tested, and packaged. The following plugins and dependencies were crucial for linking the development phase to the DevOps pipeline:

Element	Plugin/Dependency	CI/CD Purpose
Testing Framework	`junit-jupiter-api` & `junit-jupiter-engine`	These dependencies satisfy the **Testing** requirement. They provide the necessary APIs for writing tests and the execution engine for running them, which is vital for the CI pipeline's immediate validation step.
Compiler Standard	`maven-compiler-plugin:3.11.0`	Configured the `<source>` and `<target>` to **Java 17**. This ensured the compiled code was standardized, guaranteeing consistent behavior whether it was run locally, by Jenkins, or inside the Docker container.
Test Runner	`maven-surefire-plugin:3.2.3`	**Critical for the Continuous Integration (CI) gate.** This plugin automatically discovers and executes all JUnit tests during the Maven `test` lifecycle phase. If tests fail, the Maven build (and thus the Jenkins pipeline) fails, preventing unstable code from moving forward.
Executable Packaging	`maven-jar-plugin:3.3.0`	**Crucial for the Build and Containerization.** The nested `<mainClass>com.spe.calculator.Main</mainClass>` configuration makes the final JAR file **self-executable** (`java -jar ...`). This eliminates the need to specify the main class when the Docker container runs the application.

1. The Maven/Directory Separation

The primary separation is handled by the build tool (Maven) and the directory structure:

- `src/main/java`: This folder is for production code (the code that gets deployed and run by the end-user).
- `src/test/java`: This folder is strictly for test code (JUnit files) and is *not* included in the final executable JAR.

This directory separation is standardized by Maven and is critical for the Continuous Integration (CI) pipeline, as it allows Jenkins to run tests (`mvn test`) independently before packaging the final application code (`mvn clean package`).

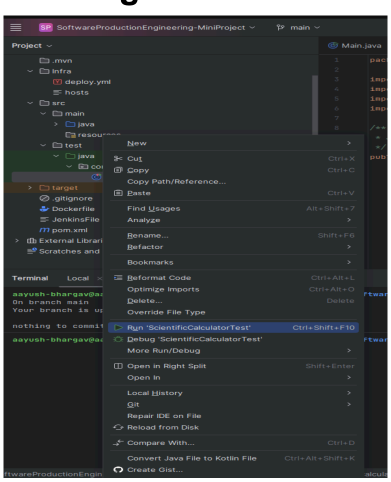
---

2. The Package Naming Convention (`com.spe.calculator`)

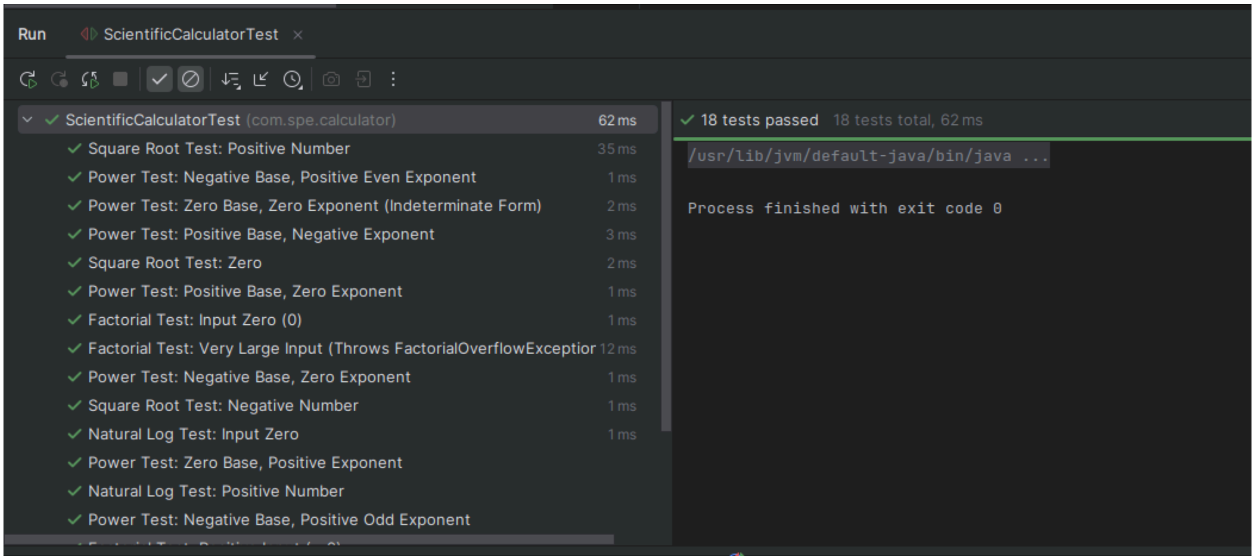
The Java package name handles the logical grouping and identity:

- Logical Grouping: By declaring `package com.spe.calculator;` in both `Main.java` and `ScientificCalculatorTest.java`, you are telling the Java compiler that these files are logically part of the same unit.
- No Imports Required: Since both classes belong to the same package, you do not need to write an `import` statement in `ScientificCalculatorTest.java` to use the `ScientificCalculator` class. This simplifies development and reduces boilerplate code.

Running The Tests:

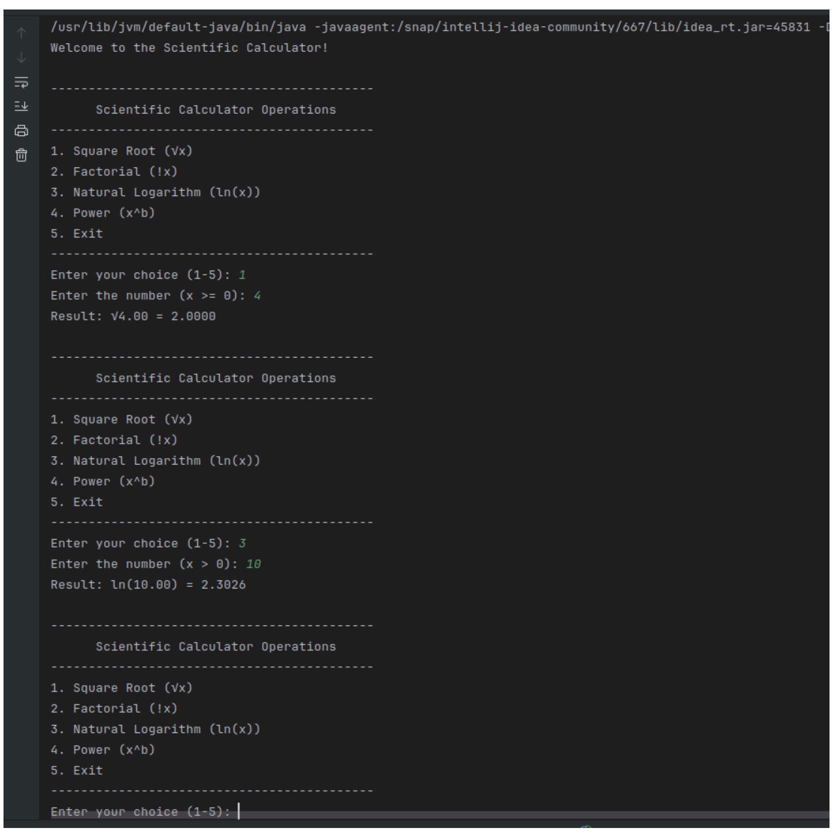


Output:



All 18 tests passed successfully.

Running the Program:



4. Source Control Management (SCM) and Initial Push

Brief: Source Control Management (SCM) is the foundational step in any DevOps pipeline. We utilized GitHub to host the project repository, allowing for collaborative development, version tracking, and, most importantly, providing the centralized source from which the Continuous Integration (CI) server (Jenkins) pulls the code.

Setup and Configuration:

1. **Repository Creation:** The process began by creating a new public repository on GitHub named `SPE_MiniProject`.

   Link to the GitHub repository:
   https://github.com/Aayush-Bhargav/SPE_MiniProject

2. **Local Repository Initialization:** In the IntelliJ project directory, Git was initialized, and the remote GitHub repository was linked.

Action	Command Used	Purpose
Initialize local Git	`git init`	Converts the local project directory into a Git repository.
Link Remote URL	`git remote add origin https://github.com/Aayush-Bhargav/SPE_MiniProject.git`	Links the local `origin` branch to the remote GitHub repository URL.
Stage Files	`git add .`	Adds all existing project files (`.java`, `pom.xml`, etc.) to the staging area.
Commit Changes	`git commit -m "First commit"`	Records the staged changes permanently in the local repository history.

Authentication via Personal Access Token (PAT): When executing the final push command, standard username and password **authentication failed, as GitHub** deprecated password usage for Git operations over HTTPS in 2021 for security reasons.

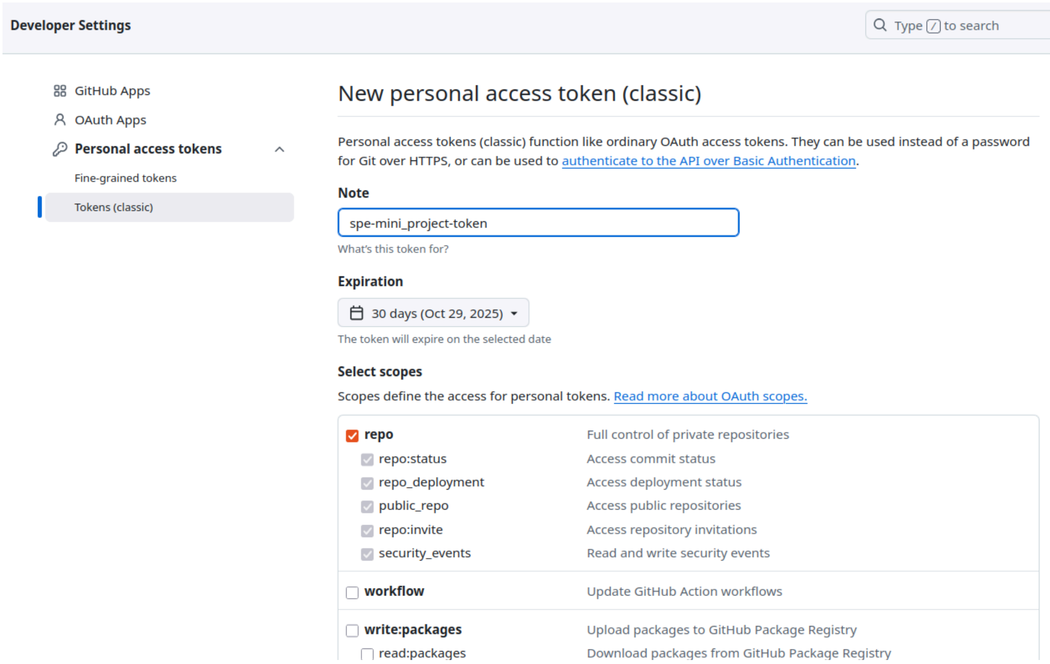
Action	Command Used	Authentication Required
Push to GitHub	`git push -u origin main`	Prompts for Username and Password/PAT.

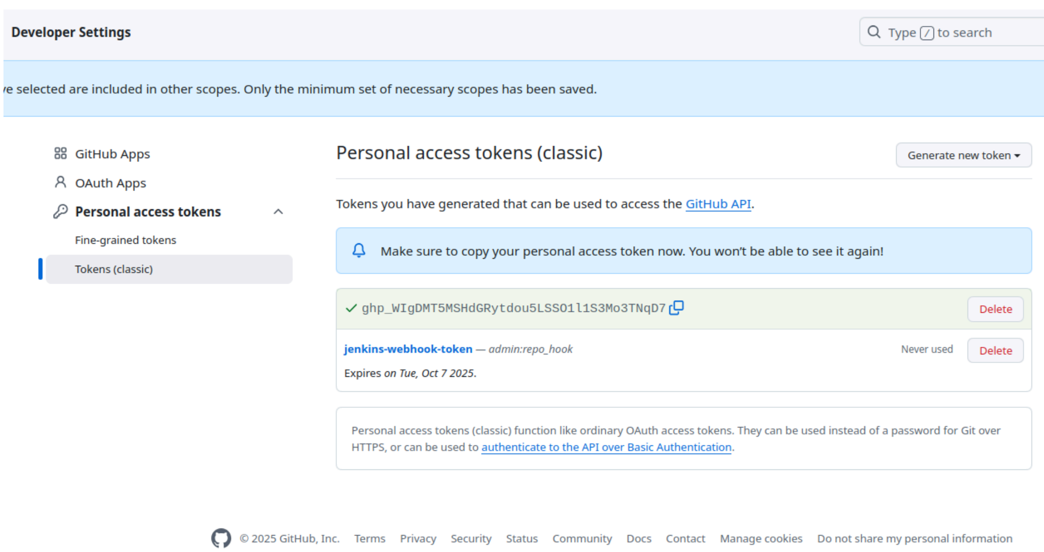
Explanation of Personal Access Token (PAT): A Personal Access Token (PAT) is a secure, temporary string that serves as a modern, revocable, and audited alternative to a full account password for accessing the GitHub API or performing Git operations.

- Why it is Used: PATs enhance security by allowing specific permissions (scopes, e.g., 'repo' access only) and set expiration dates, limiting potential damage if the token is compromised, unlike a permanent account password.
- PAT Generation: The token was generated through **Settings → Developer settings → Personal access tokens -> Tokens (classic)** on the GitHub website, with the necessary `repo` scope checked to allow code pushes.

The PAT was entered when prompted for the password, successfully completing the initial push and ensuring the project source code was ready for the next phase.

Screenshots for the creation of PAT:



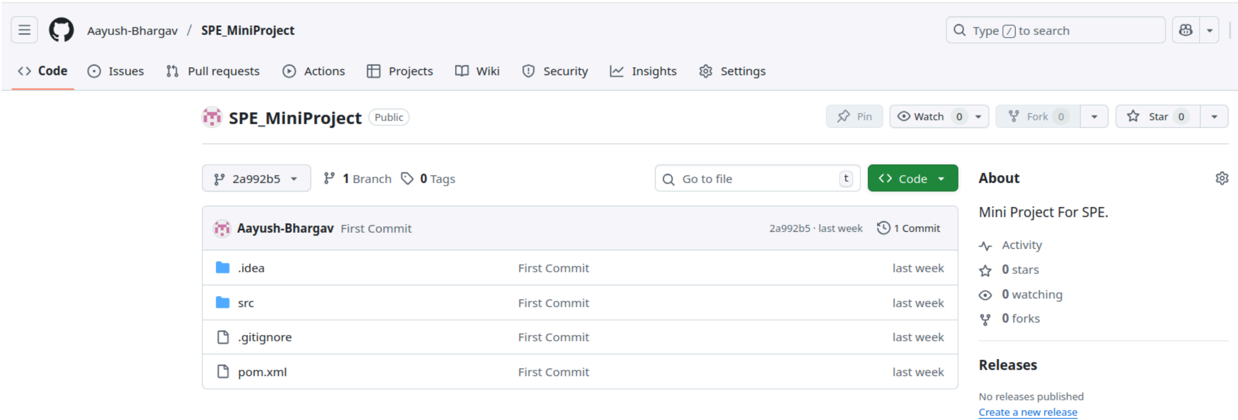


Screenshots of actions in the local repository leading to the first commit:

```
aayush-bhargav@aayush-bhargav-Inspiron-15-3520:~/IdeaProjects/SoftwareProductionEngineering-MiniProject$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   .gitignore
        new file:   .idea/.gitignore
        new file:   .idea/encodings.xml
        new file:   .idea/misc.xml
        new file:   .idea/vcs.xml
        new file:   pom.xml
        new file:   src/main/java/com/spe/calculator/Main.java
        new file:   src/main/java/com/spe/calculator/ScientificCalculator.java
        new file:   src/test/java/com/spe/calculator/ScientifcCalculatorTest.java
        new file:   src/test/java/com/spe/calculator/ScientificCalculatorTest.java

Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   pom.xml
        modified:   src/main/java/com/spe/calculator/Main.java
        deleted:    src/test/java/com/spe/calculator/ScientifcCalculatorTest.java
        modified:   src/test/java/com/spe/calculator/ScientificCalculatorTest.java
```

```
aayush-bhargav@aayush-bhargav-Inspiron-15-3520:~/IdeaProjects/SoftwareProductionEngineering-MiniProject$ git add .

aayush-bhargav@aayush-bhargav-Inspiron-15-3520:~/IdeaProjects/SoftwareProductionEngineering-MiniProject$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   .gitignore
        new file:   .idea/.gitignore
        new file:   .idea/encodings.xml
        new file:   .idea/misc.xml
        new file:   .idea/vcs.xml
        new file:   pom.xml
        new file:   src/main/java/com/spe/calculator/Main.java
        new file:   src/main/java/com/spe/calculator/ScientificCalculator.java
        new file:   src/test/java/com/spe/calculator/ScientificCalculatorTest.java

aayush-bhargav@aayush-bhargav-Inspiron-15-3520:~/IdeaProjects/SoftwareProductionEngineering-MiniProject$ git commit -m "First Commit"
[master (root-commit) 2a992b5] First Commit
 9 files changed, 499 insertions(+)
 create mode 100644 .gitignore
 create mode 100644 .idea/.gitignore
 create mode 100644 .idea/encodings.xml
 create mode 100644 .idea/misc.xml
```

```
aayush-bhargav@aayush-bhargav-Inspiron-15-3520:~/IdeaProjects/SoftwareProductionEngineering-MiniProject$ git push -u origin main
Username for 'https://github.com': Aayush-Bhargav
Password for 'https://Aayush-Bhargav@github.com':
Enumerating objects: 23, done.
Counting objects: 100% (23/23), done.
Delta compression using up to 12 threads
Compressing objects: 100% (14/14), done.
Writing objects: 100% (23/23), 6.10 KiB | 2.03 MiB/s, done.
Total 23 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/Aayush-Bhargav/SPE_MiniProject.git
 * [new branch]      main -> main
branch 'main' set up to track 'origin/main'.

aayush-bhargav@aayush-bhargav-Inspiron-15-3520:~/IdeaProjects/SoftwareProductionEngineering-MiniProject$
```



State of the Repository after the First Commit.

5. Continuous Integration Setup (CI Trigger)

What is Jenkins?

Jenkins is an open-source automation server used to build, test, and deploy software automatically. It is a key DevOps tool that supports Continuous Integration (CI) and Continuous Delivery (CD), helping developers detect and fix issues early by automating the software build and deployment pipeline. You need to have Jenkins installed on your system and running locally for this project.

Brief: Continuous Integration requires an immediate trigger when source code changes. Since the Jenkins server was running locally, this step involved exposing the local server to the public internet using ngrok and configuring a GitHub Webhook to send events to that public address.

Tool: ngrok (Public Tunneling Service)

ngrok is a crucial tool used to create a secure, public tunnel to a locally running service (like Jenkins on `localhost:8080`). This public URL allows external services, such as GitHub's webhook mechanism, to communicate with the local CI server.

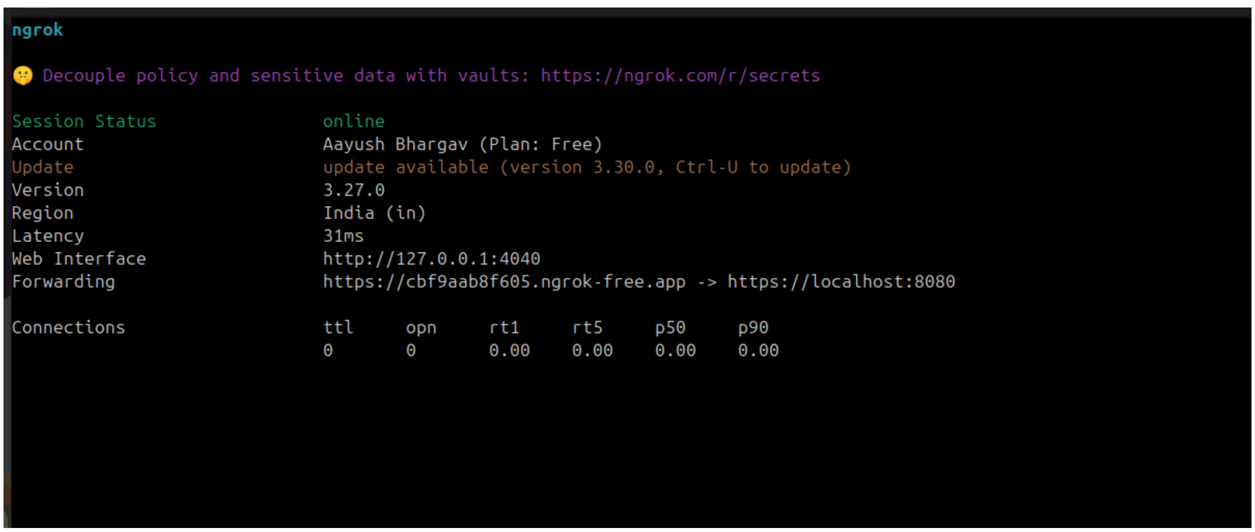
ngrok Setup and Execution:

Installation and Authentication: ngrok was downloaded, extracted, and authenticated using the personal auth token copied from the ngrok dashboard.
# 1. Extract ngrok to system path
$ sudo tar xvzf ~/Downloads/ngrok-stable-linux-amd64.tgz -C /usr/local/bin

# 2. Add Authtoken for secure connection
$ ngrok authtoken <token>

Exposing Jenkins: The local Jenkins instance, running on port 8080, was exposed to the public internet.
Execute ngrok and copy the generated public HTTPS URL

$ ngrok http 8080



The resulting HTTPS URL (e.g., `https://cbf9aab8f605.ngrok-free.app`) served as the Payload URL for the GitHub webhook.

Jenkins Server Global Configuration

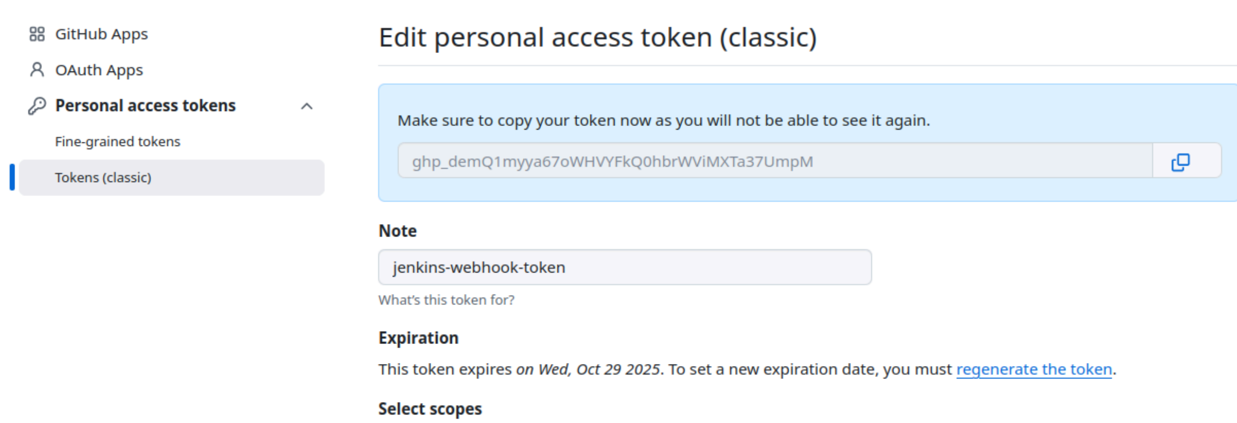
To ensure Jenkins correctly identifies its own public address and securely communicates with GitHub, global system settings were configured:

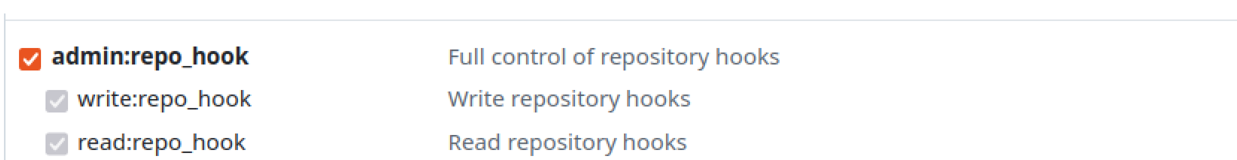
1. **Jenkins Location Update:** The public ngrok URL was set as the official Jenkins address.
    - Location: *Manage Jenkins → System → Jenkins Location*

    - Configuration: The ngrok address (`https://cbf9aab8f605.ngrok-free.app`) was entered into the Jenkins URL field. This allows Jenkins to generate correct links for webhooks and internal resources.

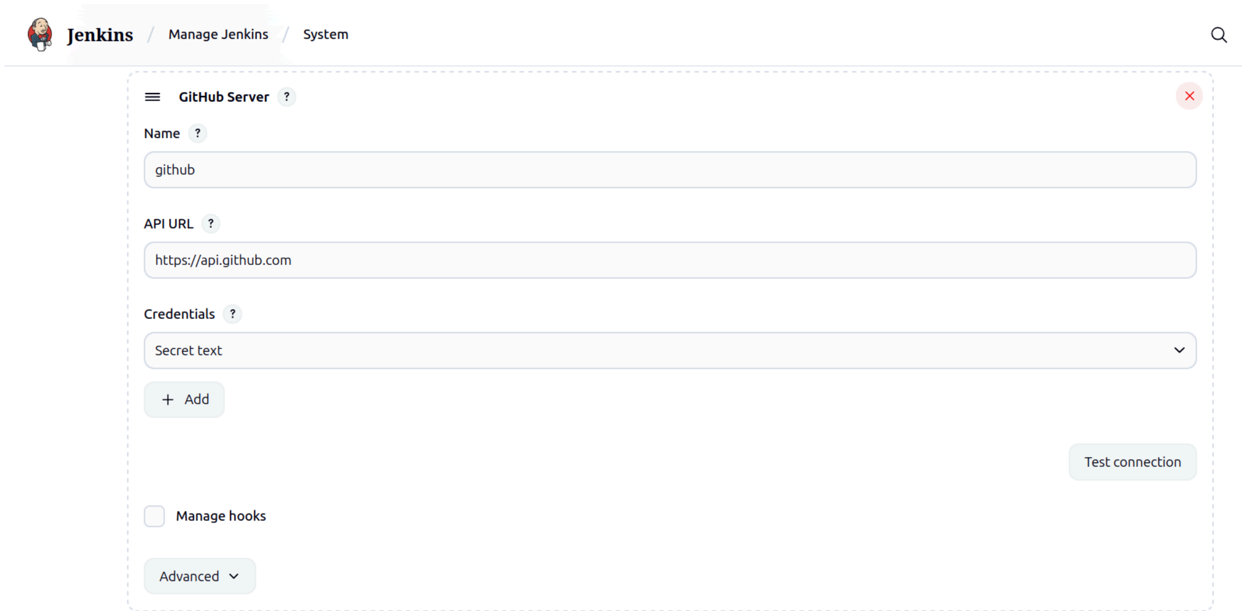
2. **GitHub Server Credential Setup:** The second PAT created for the webhook was registered in Jenkins.
    - Location: *Manage Jenkins → System → GitHub Servers*
    - Configuration: A new credential was added (Type: Secret Text) containing the PAT created with the `admin:repo_hook` scope. This is used by Jenkins plugins to make verified requests back to GitHub (e.g., updating commit statuses).

Second PAT screenshot:

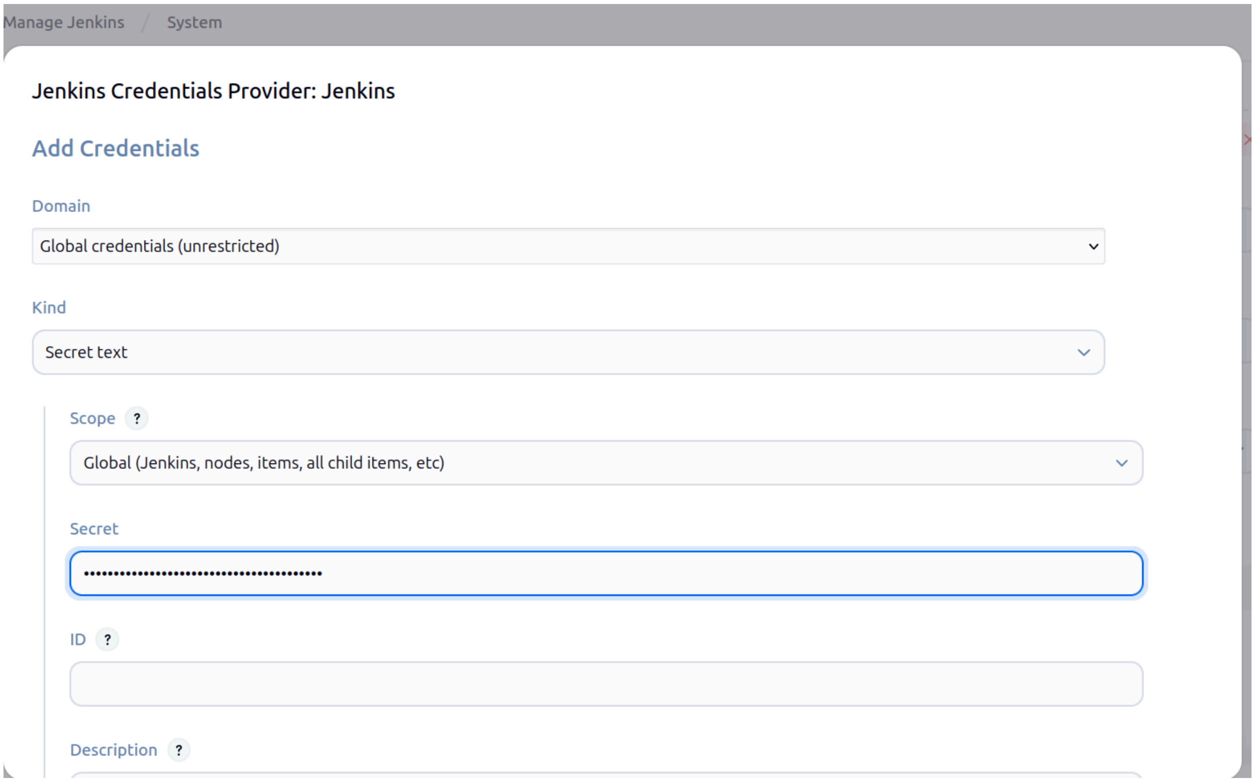




Adding the secret text into the credentials in Jenkins Settings:



Click on Add and use the Second PAT created as the Secret.



GitHub Webhook Configuration

To complete the trigger mechanism, a webhook was configured in the GitHub repository settings.

1. **Generating the Webhook PAT:** A second Personal Access Token (PAT), named `jenkins-webhook-token`, was created specifically for the webhook.
   - Scope: This PAT was granted the `admin:repo_hook` scope. This restricted permission is necessary because it allows GitHub to manage the webhook itself and verify the connection, without granting full repository write access.

2. **Webhook Settings:** The webhook was configured with the following parameters:

Parameter	Value	Purpose

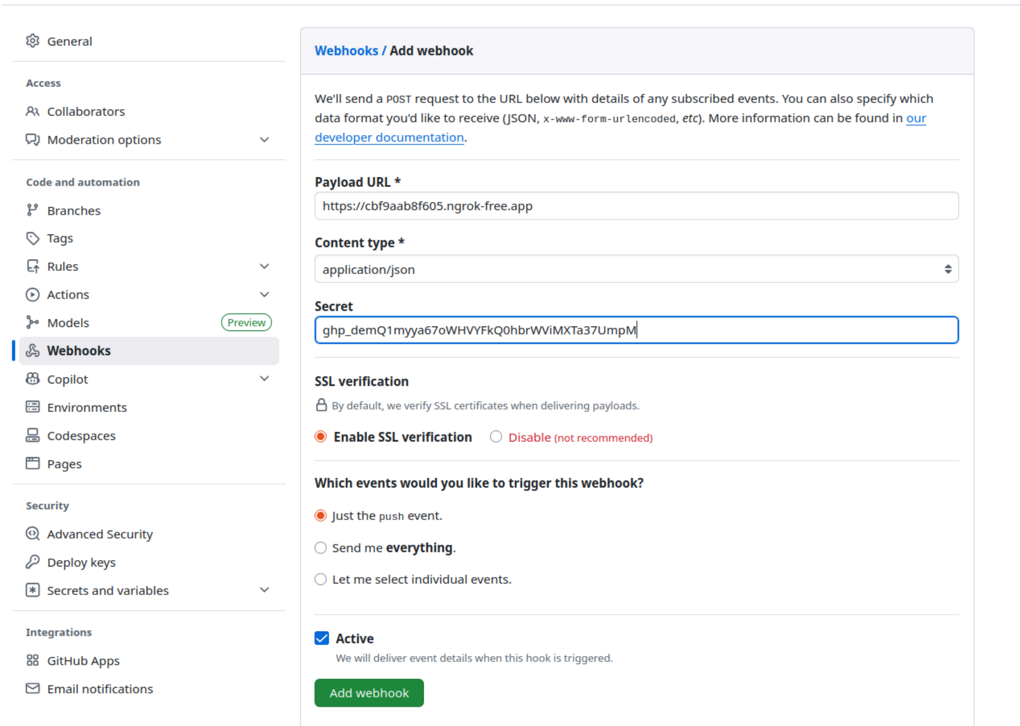
Payload URL	`https://52d64359d9cb.ngrok-free.app/github-webhook/`	The public address (from ngrok) plus the standard Jenkins endpoint for GitHub triggers.
Content type	`application/x-www-form-urlencoded`	Specifies the format of data GitHub sends to Jenkins.
Secret	The generated `jenkins-webhook-token` PAT.	Secures the connection; Jenkins uses this token to verify the incoming request is genuinely from GitHub.
SSL Verification	Enabled	Ensures data integrity and security between GitHub and the ngrok tunnel.

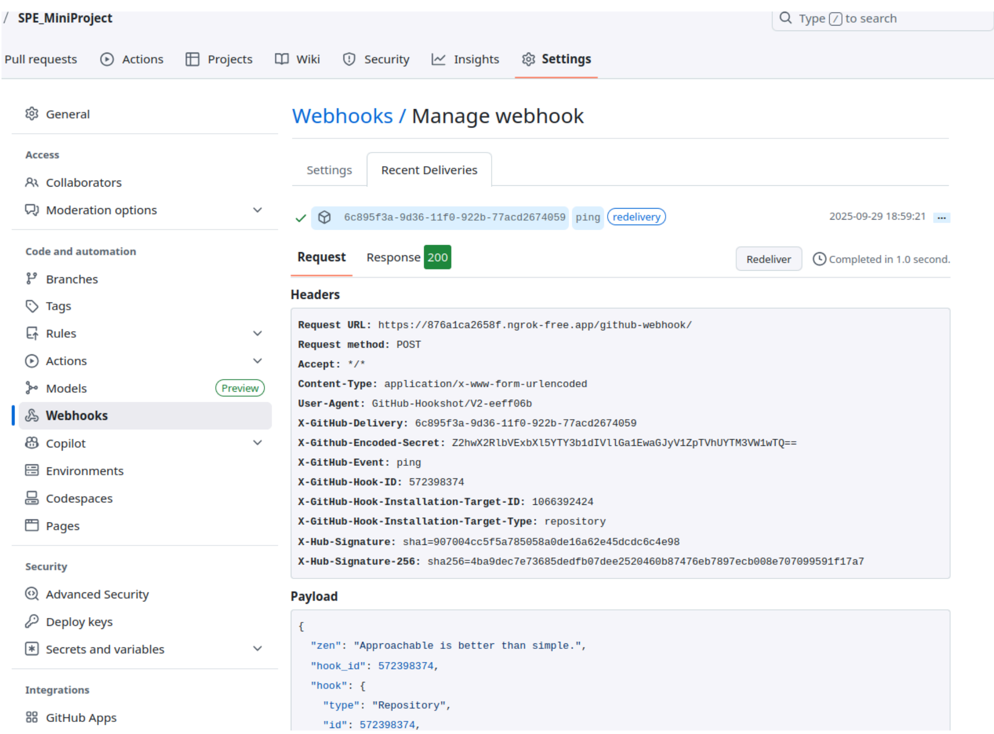
3. **Delivery Test:** A test delivery was executed, and the system returned a successful HTTP status code (200 OK), confirming the Continuous Integration trigger was successfully linked.

Screenshot of ngrok running:

```
ngrok

🧱 Block threats before they reach your services with new WAF actions → https://ngrok.com/r/waf

Session Status                online
Account                       Aayush Bhargav (Plan: Free)
Version                       3.27.0
Region                        India (in)
Latency                       28ms
Web Interface                 http://127.0.0.1:4040
Forwarding                    https://876a1ca2658f.ngrok-free.app -> http://localhost:8080

Connections                   ttl     opn     rt1     rt5     p50     p90
                              33      0       0.00    0.01    0.68    33.16

HTTP Requests
-------------

18:59:20.782 IST POST /github-webhook/                    200 OK
18:54:16.044 IST POST /widget/ExecutorsWidget/ajax        200 OK
18:54:16.016 IST POST /widget/BuildQueueWidget/ajax       200 OK
18:54:10.791 IST GET  /static/107d7d82/jsbundles/styles.css
18:54:10.631 IST GET  /                                   200 OK
18:54:10.959 IST GET  /i18n/resourceBundle                200 OK
18:54:09.587 IST POST /widget/ExecutorsWidget/ajax        200 OK
18:54:09.587 IST POST /widget/BuildQueueWidget/ajax       200 OK
18:49:39.084 IST POST /widget/ExecutorsWidget/ajax        200 OK
18:49:39.083 IST POST /widget/BuildQueueWidget/ajax       200 OK
```

Screenshot of GitHub Webhook Configuration:


General
Access
Collaborators
Moderation options
Code and automation
Branches
Tags
Rules
Actions
Models
Preview
Webhooks
Copilot
Environments
Codespaces
Pages
Security
Advanced Security
Deploy keys
Secrets and variables
Integrations
GitHub Apps
Email notifications
Webhooks / Add webhook
We'll send a POST request to the URL below with details of any subscribed events. You can also specify which data format you'd like to receive (JSON, x-www-form-urlencoded, etc). More information can be found in our developer documentation.
Payload URL *
https://cbf9aab8f605.ngrok-free.app
Content type *
application/json
Secret
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
SSL verification
By default, we verify SSL certificates when delivering payloads.
Enable SSL verification
Disable (not recommended)
Which events would you like to trigger this webhook?
Just the push event.
Send me everything.
Let me select individual events.
Active
We will deliver event details when this hook is triggered.
Add webhook


/ SPE_MiniProject
Type / to search
Pull requests
Actions
Projects
Wiki
Security
Insights
Settings
General
Access
Collaborators
Moderation options
Code and automation
Branches
Tags
Rules
Actions
Models
Preview
Webhooks
Copilot
Environments
Codespaces
Pages
Security
Advanced Security
Deploy keys
Secrets and variables
Integrations
GitHub Apps
Webhooks / Manage webhook
Settings
Recent Deliveries
6c895f3a-9d36-11f0-922b-77acd2674059
ping
redelivery
2025-09-29 18:59:21
Request
Response 200
Redeliver
Completed in 1.0 second.
Headers
Request URL: https://876a1ca2658f.ngrok-free.app/github-webhook/
Request method: POST
Accept: */*
Content-Type: application/x-www-form-urlencoded
User-Agent: GitHub-Hookshot/V2-eeff06b
X-GitHub-Delivery: 6c895f3a-9d36-11f0-922b-77acd2674059
X-Github-Encoded-Secret: Z2hwX2RlbVExbXl5YTY3b1dIVllGa1EwaGJyV1ZpTVhUYTM3VW1wTQ==
X-GitHub-Event: ping
X-GitHub-Hook-ID: 572398374
X-GitHub-Hook-Installation-Target-ID: 1066392424
X-GitHub-Hook-Installation-Target-Type: repository
X-Hub-Signature: sha1=907004cc5f5a785058a0de16a62e45dcdc6c4e98
X-Hub-Signature-256: sha256=4ba9dec7e73685dedfb07dee2520460b87476eb7897ecb008e707099591f17a7
Payload
{
"zen": "Approachable is better than simple.",
"hook_id": 572398374,
"hook": {
"type": "Repository",
"id": 572398374,

6. Jenkinsfile Creation

A **Jenkinsfile** is a text file that defines a **Jenkins pipeline** — a sequence of automated steps that enable continuous integration and continuous deployment (**CI/CD**). Our file is written using the **declarative** syntax based on Groovy and stored within the project's version control system. This allows the entire build and deployment process to be versioned alongside the source code, ensuring transparency, reproducibility, and collaboration.

In a **CI/CD pipeline**, the Jenkinsfile acts as the **blueprint** that automates the software development lifecycle — from fetching the latest code to testing, building, containerizing, and deploying it. It ensures that code changes are continuously integrated, verified, and delivered in a consistent and reliable manner without manual intervention.

A typical Jenkinsfile defines several key aspects:

- **Agent**: Specifies where the pipeline will run (e.g., any available node or a Docker container).
- **Environment Variables**: Store credentials and configuration values securely.
- **Stages**: Represent distinct steps such as *checkout*, *build*, *test*, *package*, *deploy*, etc.
- **Post Actions**: Define cleanup or notification steps that run after the pipeline completes.

By automating repetitive tasks, a Jenkinsfile enhances **developer productivity**, reduces **human errors**, and ensures **faster feedback** on code quality. It forms the backbone of modern DevOps workflows by integrating tools such as Maven for build management, Docker for containerization, and Ansible for deployment — as demonstrated in this project.

Our JenkinsFile can be seen on GitHub in the root of the project.

Detailed Stage Breakdown:

Stage Name	Command/ Instruction	Step/ Tool	Explanation
Checkout	`git branch: 'main', url: '...'`	SCM (Git)	Pulls the latest source code from the specified GitHub URL and ensures the pipeline is building from the **`main`** branch, guaranteeing synchronization.
Test	`sh 'mvn test'`	Testing (Maven Surefire)	Executes the test phase defined in `pom.xml`, running all JUnit test cases within `ScientificCalculatorTest.java`. The pipeline fails immediately if any test asserts are violated.
Build JAR	`sh 'mvn clean package'`	Build (Maven)	Compiles the Java code and packages it, along with dependencies, into the final executable JAR file (`scientific-calculator-project-1.0-SNAPSHOT.jar`). This artifact is necessary for the next stage.
Build Docker Image	`sh 'docker build -t ... .'`	Containerize (Docker)	Executes the `Dockerfile` using the Jenkins workspace as the build context. This creates the final application image.
Push Docker Image	`sh 'echo $PSW	docker login ...'`	Push the image to Docker Hub

Deploy with Ansible	`sh 'ansible-playbook -i ...'`	Deployment (Ansible)	Executes the configuration management tool, which manages the lifecycle of the container on the target host (`localhost`).

The `post` block is the final section of the pipeline, defining actions that must run after all stages are complete, regardless of the overall pipeline result. Its primary role is to provide Continuous Feedback (email notification) and ensure Cleanup.

Continuous Feedback and Cleanup

The `post` section utilizes conditional blocks to execute specific commands based on the outcome of the preceding stages:

Condition	Action Executed	Purpose
`success`	The **`mail`** step sends a success notification, including the unique **`$BUILD_NUMBER`** for easy reference. This informs the developer that the new code has been tested, built, and deployed.	**Continuous Feedback** (Success Notification)
`failure`	The **`mail`** step sends an immediate failure notification, directing the developer to check the Jenkins console for the specific stage failure.	**Continuous Feedback** (Immediate Alerting)
`always`	Contains steps that run regardless of whether the pipeline passed or failed.	Essential Cleanup

`cleanWs()`	A built-in Jenkins step that **deletes the contents of the Jenkins workspace** (`/var/lib/jenkins/workspace/...`).	**Resource Management.** This prevents leftover files and build artifacts from interfering with subsequent builds, ensuring the next run starts from a clean state.

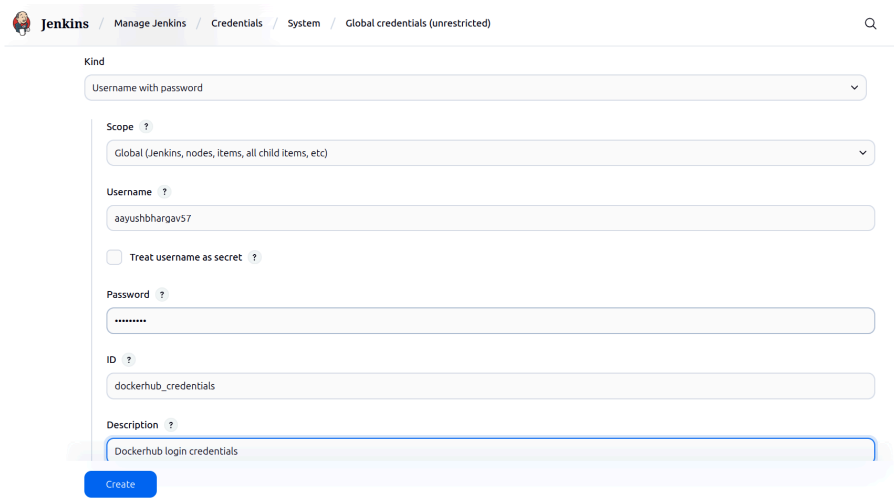
Docker Hub Credential Management and Security

What is DockerHub?

Docker Hub is a cloud-based registry service where Docker users can **store, share, and manage container images**. It acts like a “GitHub for Docker images,” allowing developers to pull pre-built images or push their own for use in different environments.

To execute the **Push Docker Image** stage, Jenkins requires secure access to Docker Hub. This was achieved by configuring a global credential.

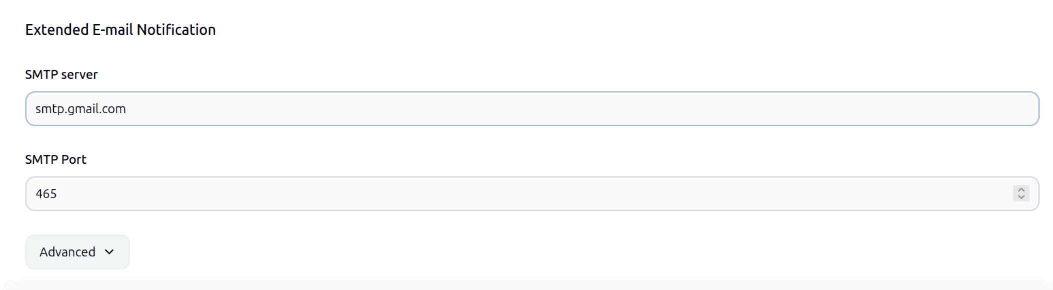
1. **Access:** *Manage Jenkins → Credentials → System → Global credentials (unrestricted) → Add Credentials*.
2. **Configuration:** A credential of kind **"Username with password"** was created with my DockerHub username and password.
3. **Identifier:** The **ID** was set to `dockerhub_credentials`. This identifier must match the name used in the `environment` block of the `Jenkinsfile`.
4. **Purpose:** This mechanism ensures the Docker Hub login details (username/password or PAT) are never exposed directly in the `Jenkinsfile` or the build logs, fulfilling a fundamental security requirement of CI/CD.

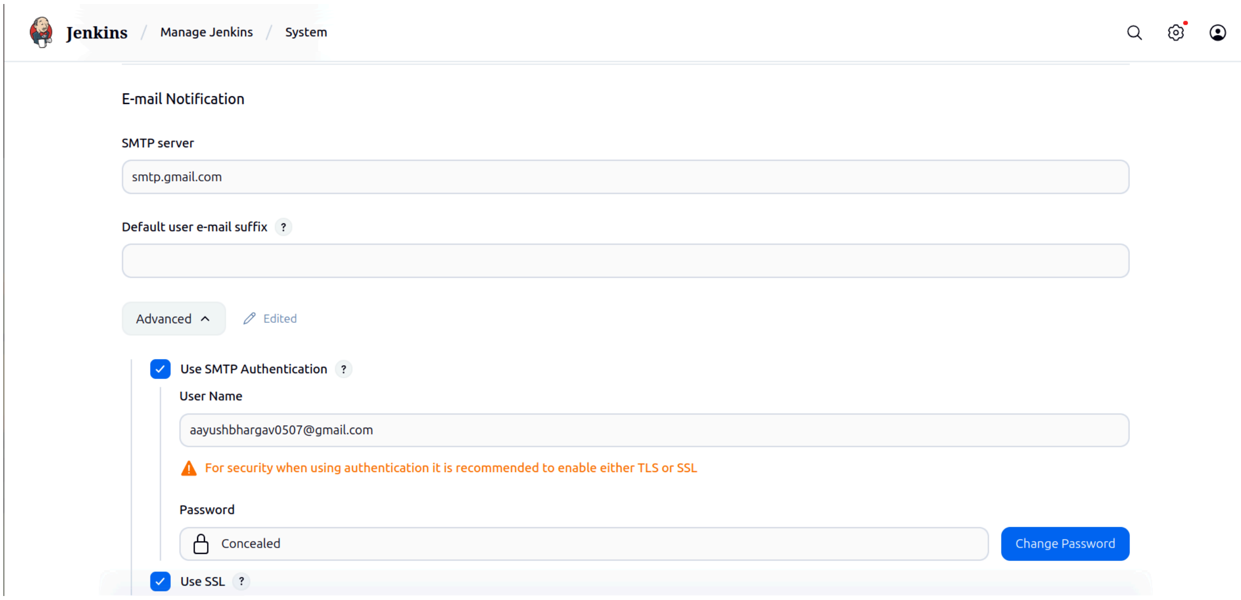


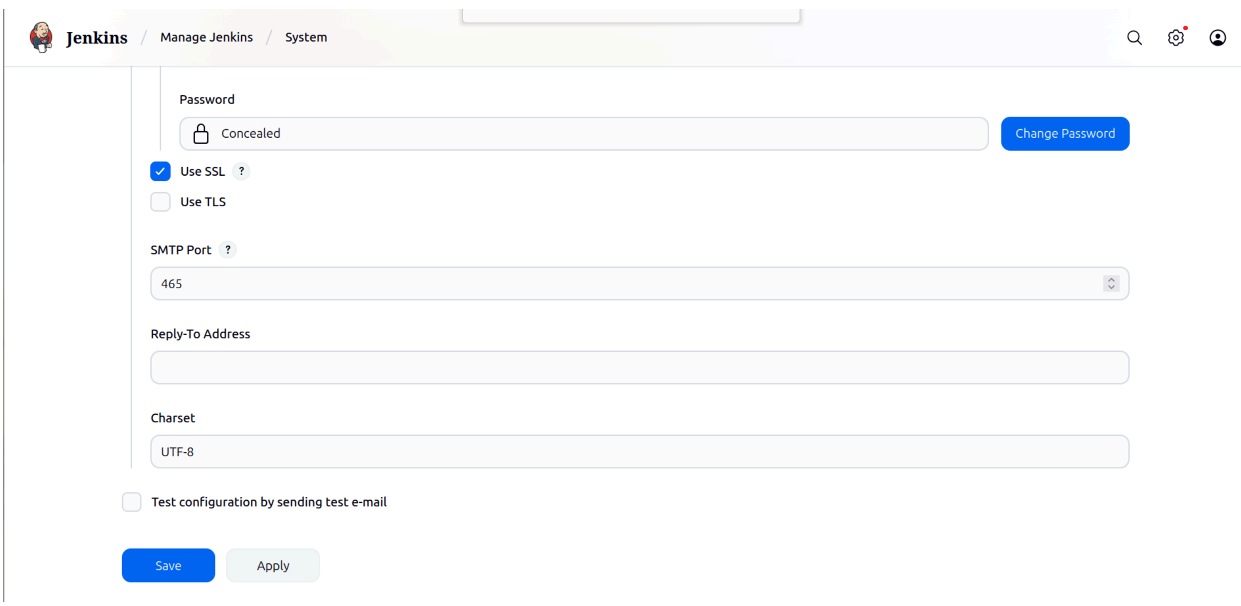
Jenkins Global Configuration (Setup)

Before the pipeline can send emails, the Jenkins server itself must be configured with an SMTP server (like Gmail, Outlook, or a company server).

1. Go to **Manage Jenkins** → **System**.
2. Scroll down to the **"Extended E-mail Notification"** section (or the standard **"E-mail Notification"** section).
3. **Configure the SMTP Server Details:**
   - **SMTP server:** (set it to `smtp.gmail.com` since we are using Gmail)
   - **Check "Use SMTP Authentication"** and enter your full email address and the corresponding **App Password**.
   - **Use SSL/TLS** and specify the correct port (465).
4. **Test Configuration:** Use the "Test Configuration" button at the bottom to ensure Jenkins can successfully send a test email.







7. Dockerfile Definition (Containerization)

What is Docker?

Docker is a platform that allows applications to run in **containers** — lightweight, isolated environments that package code along with all its dependencies. This ensures the application runs consistently across different systems, making deployment faster, more reliable, and easier to scale.

The **Dockerfile** is the blueprint for creating the portable, self-contained environment that wraps the Java application, fulfilling the **Containerization (Step 5)** requirement.

ulator.java   pom.xml (scientific-calculator-project)   .gitignore   ScientificCalculatorTest.java

```
# Use a lightweight official Java Runtime Environment (JRE) as the base image
FROM openjdk:26-slim

# Set the working directory inside the container
WORKDIR /app

# Copy the executable JAR file created by Maven's 'package' goal
# from the host's 'target/' directory into the container's '/app' directory.
# NOTE: The JAR name must match your pom.xml's artifactId and version.
COPY target/scientific-calculator-project-1.0-SNAPSHOT.jar app.jar

# Command to run the JAR file when the container starts
# The Main class (com.spe.calculator.Main) runs the command line menu.
ENTRYPOINT ["java", "-jar", "app.jar"]

```

Dockerfile Instruction	Code	Explanation
FROM	FROM openjdk:26-slim	Uses a minimal, lightweight **Java Runtime Environment (JRE)** based on OpenJDK 26.
WORKDIR	WORKDIR /app	Sets the application's root directory inside the container.
COPY	COPY target/scientific-calculator-project-1.0-SNAPSHOT.jar app.jar	This is the crucial step: it transfers the compiled Maven JAR (the build artifact) from the Jenkins workspace into the /app directory inside the container.

ENTRYPOINT	ENTRYPOINT \["java", "-jar", "app.jar"\]	Defines the command that executes by default when the container starts.

8. Configuration Management and Deployment - Ansible

The final phase of the pipeline uses **Ansible** to manage the target host and deploy the Docker containerized application. Since the deployment target is the local Jenkins host, Ansible's role is to ensure the latest tested image is pulled and run successfully.

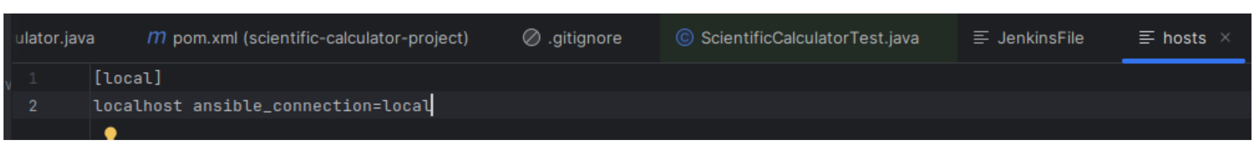
A. What is Ansible and Its Role?

Ansible is an open-source automation engine used for software provisioning, configuration management, and application deployment. Unlike other tools, it is **agentless**, meaning it manages remote hosts over SSH (or local connections) without requiring any special software installed on the target machine.

In this project, Ansible's role is to ensure the final deployment step—**running the Docker container**—is executed reliably, idempotently, and repeatably on the local host machine.

B. Ansible Inventory File (`Infra/hosts`)

The `hosts` file defines the inventory—the list of target machines Ansible will manage.



`[local]`: Defines a group named "local."
`localhost`: Specifies the target machine is the local host where Jenkins and Ansible are running.

`ansible_connection=local`: Instructs Ansible to execute commands directly on the local machine rather than connecting via SSH, making the deployment instantaneous for this single-host setup.

C. Ansible Playbook (`Infra/deploy.yml`)

The `deploy.yml` is the core configuration file written in YAML. It defines the sequence of tasks required to deploy the container.

```yaml
---
- name: Deploy Scientific Calculator Docker Container
  hosts: local
  # This requires the 'jenkins' user to have NOPASSWD access in /etc/sudoers
  # to run Docker commands as root (unless jenkins user is in the 'docker' group)
  become: yes

  vars:
    # IMPORTANT: The image name MUST match the one defined in your Jenkinsfile
    app_image: "aayushbhargav57/scientific-calculator:latest"
    container_name: "scientific_calculator_app"

  tasks:
    # FIX: Switching back to pip, using the 'ansible.builtin.pip' module with
    # the 'extra_args' flag to bypass the externally-managed-environment error
    # on modern Linux distributions. This is the cleanest fix for CI systems.
    - name: Ensure Python Docker module is installed (via pip bypass)
      ansible.builtin.pip:
        name: docker
        state: present
        executable: pip3
        extra_args: "--break-system-packages" # Flag to override PEP 668 restriction
      # This task requires elevated privileges (become: yes)
      # No 'when' condition needed as this fix is generally system-agnostic for CI

    - name: Stop and remove existing container (if running)
      # Use the fully qualified collection name for the docker_container module
      community.docker.docker_container:
        name: "{{ container_name }}"
        state: absent
      ignore_errors: yes

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

    - name: Run the new container
      community.docker.docker_container:
        name: "{{ container_name }}"
        image: "{{ app_image }}"
```

```yaml
- name: Run the new container
  community.docker.docker_container:
    name: "{{ container_name }}"
    image: "{{ app_image }}"
    state: started
    # Detach the container so the Jenkins job can finish
    interactive: true
    detach: true
    # The app is command-line only; it will run the entrypoint (Main class)
    # and exit after execution unless in an interactive environment.
    # For continuous running CLI, you'd use a loop in the Entrypoint/CMD,
    # but for this assignment, a started container is sufficient.
```

Detailed Task Explanation:

Task Name	Module Used	Purpose and Why
Play Setup	`hosts: local`, `become: yes`	The playbook targets the `local` group and runs tasks with elevated privileges (`become: yes`), which is essential for managing the Docker service.
Ensure Python Docker module is installed	`ansible.builtin.pip`	This addresses a common CI/CD environmental issue. Ansible requires the **Python `docker` library** to manage containers. We used `extra_args: "--break-system-packages"` to bypass system Python security restrictions (PEP 668), ensuring the deployment dependency is installed reliably.

Stop and remove existing container	`community.docker.docker_container`	This enforces **idempotency**. It guarantees that the new deployment starts clean by stopping and removing the previous version of the container. `ignore_errors: yes` handles the case where the container does not yet exist.
Pull the latest Docker image	`community.docker.docker_image`	Instructs the target Docker Daemon to pull the image (`aayushbhargav57/scientific-calculator:latest`) from Docker Hub, ensuring the host always runs the version that was just built and pushed by Jenkins.
Run the new container	`community.docker.docker_container`	**Final Deployment Step.** Starts the new container using the freshly pulled image. The setting `detach: true` ensures that the Jenkins pipeline does not hang, allowing the job to finish successfully immediately after the container starts running the Java application's `ENTRYPOINT`.

9. Jenkins Job Configuration (Orchestration)

The last step in establishing Continuous Integration (CI) was the creation and configuration of the Jenkins Pipeline job itself. This job acts as the central orchestrator, executing the `Jenkinsfile` and managing the entire flow from code commit to deployment.

A. Understanding the Pipeline Job

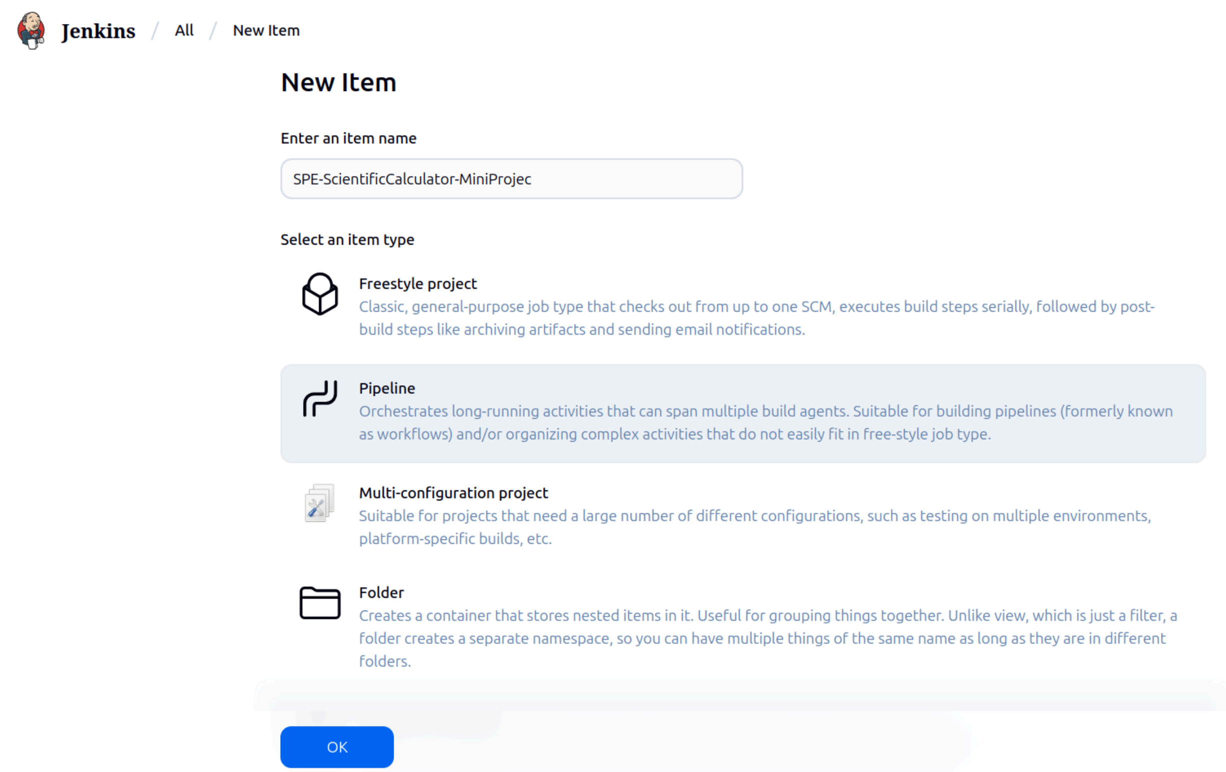
The job was created as a **Pipeline** type, which is superior to traditional Freestyle projects because it executes **Pipeline-as-Code** (the `JenkinsFile` stored in GitHub). This ensures the build process is always version-controlled and synchronized with the code it builds.

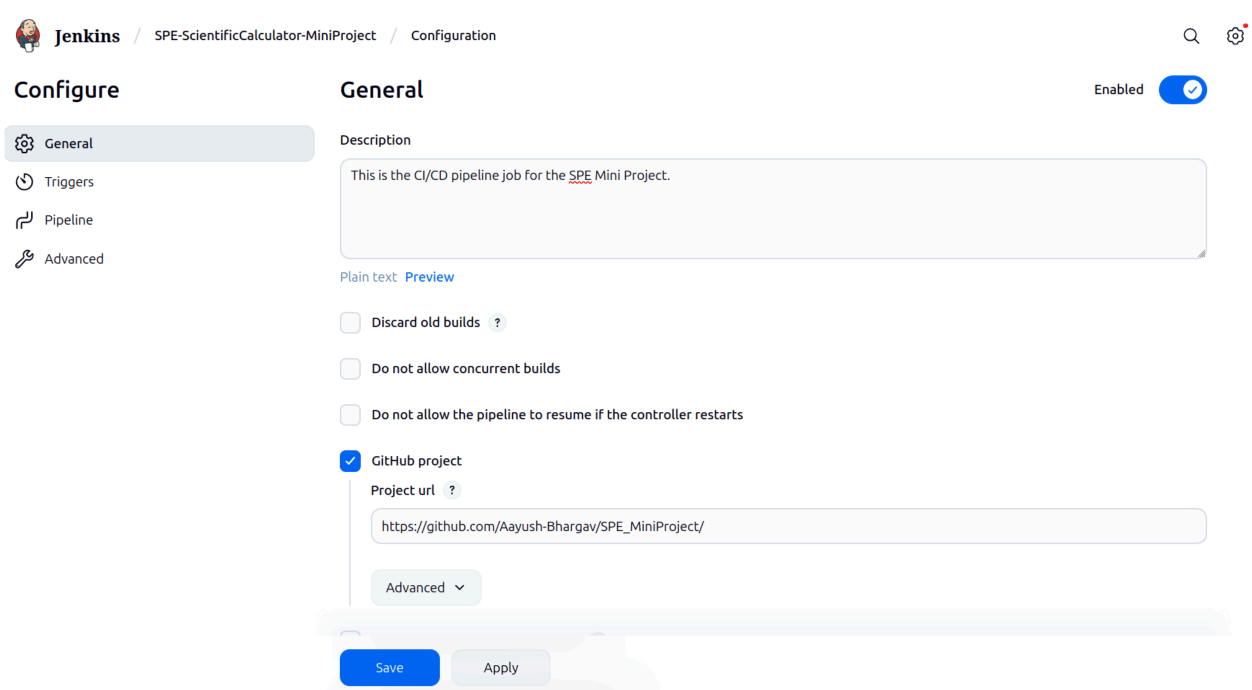
B. Step-by-Step Configuration

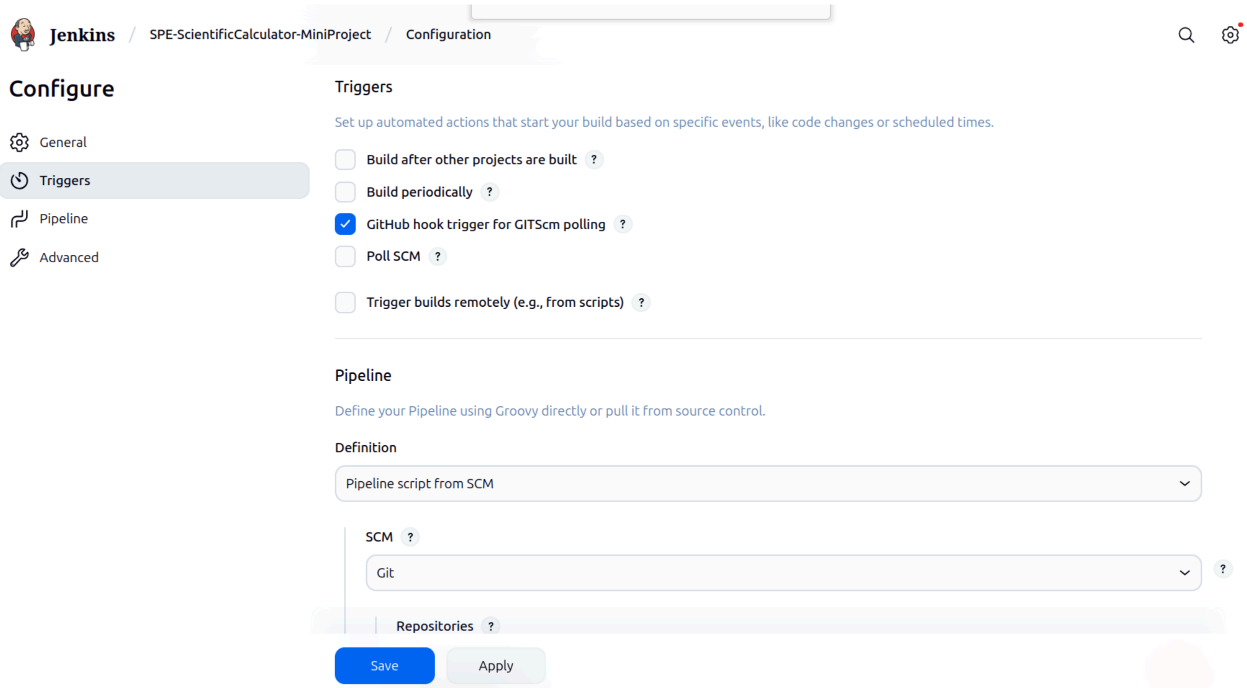
The job was named **SPE-ScientificCalculator-MiniProject**. The key settings required to link GitHub, the webhook, and the `JenkinsFile` are detailed below:

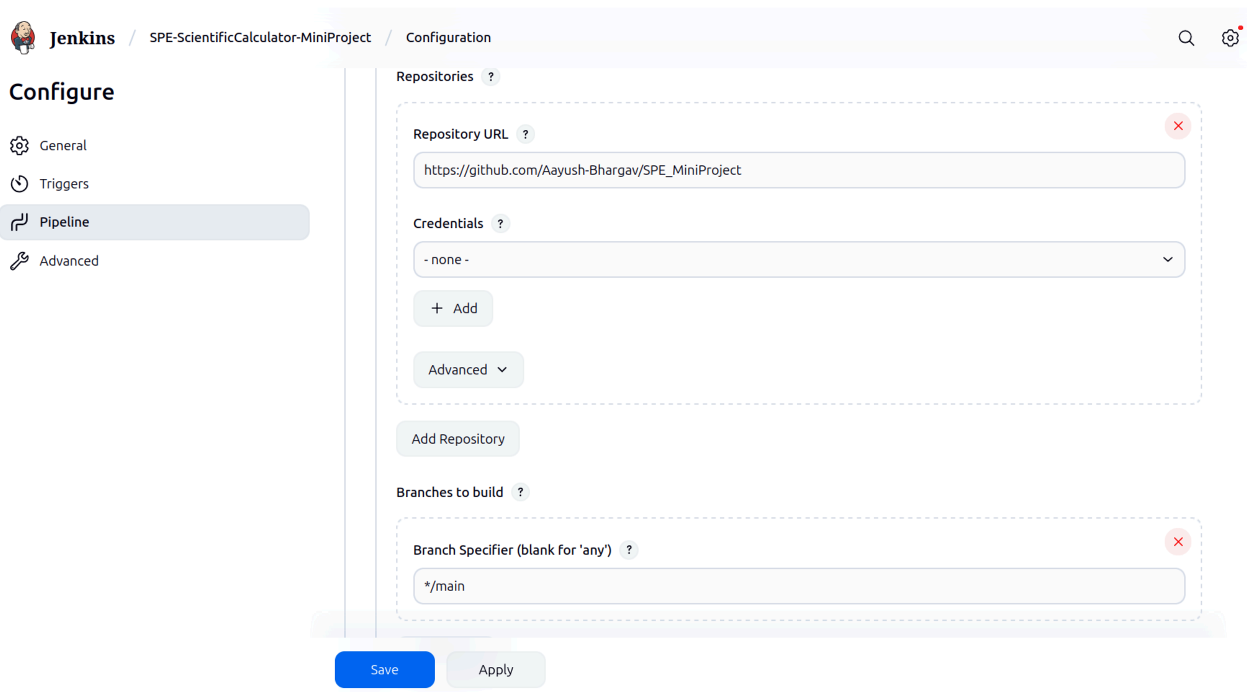
1. **General Configuration:**
    - **GitHub Project:** Checked, with the URL provided as `https://github.com/Aayush-Bhargav/SPE_MiniProject.git`. This links the job visually to the repository.
2. **Continuous Integration Trigger:**
    - **GitHub hook trigger for GITScm polling:** Selected. This setting is mandatory. It tells Jenkins to listen for the incoming push events sent by the webhook (via ngrok) and immediately start the pipeline when a commit is received.
3. **Pipeline Definition:** This section directs Jenkins to find and execute the build instructions stored in your SCM.
    - **Definition: Pipeline script from SCM** was chosen.
        - *Purpose:* This forces Jenkins to read the code's configuration from the repository, ensuring the CI process is governed by the version of the `JenkinsFile` checked into GitHub.
    - **SCM: Git** was selected.
    - **Repositories:**
        - **Repository URL:** `https://github.com/Aayush-Bhargav/SPE_MiniProject.git` (The public URL).
        - **Credentials:** Left as `(none)` since the repository is public.
    - **Branches to build:** Set to `*/main`. This restricts automated builds to the primary development branch.
    - **Script Path:** Set to `JenkinsFile`. This is the exact name of the file in the repository root containing the pipeline logic.

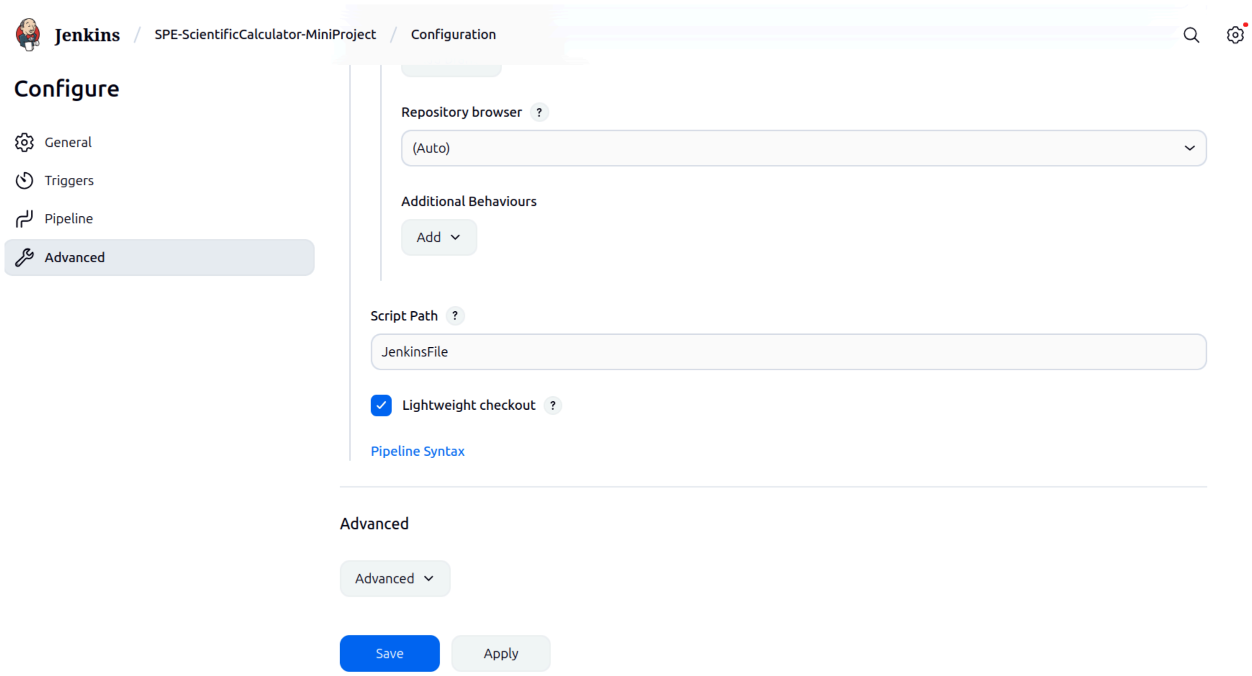
Summary: This configuration successfully automated the entire workflow. A commit to the `main` branch now instantly triggers Jenkins, which fetches the `JenkinsFile` and begins executing the eight stages of the CI/CD pipeline.





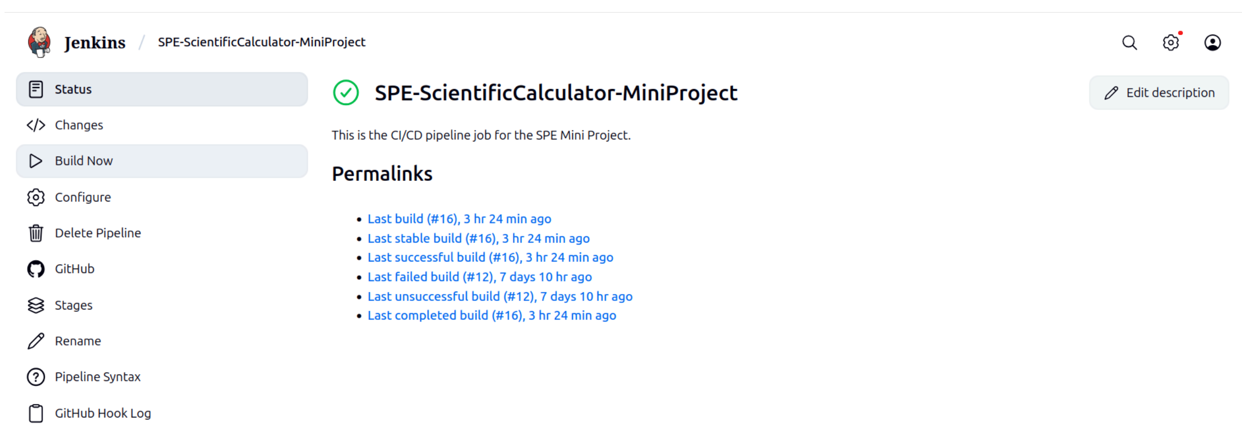

Jenkins
SPE-ScientificCalculator-MiniProject
Configuration
Configure
General
Triggers
Pipeline
Advanced
Triggers
Set up automated actions that start your build based on specific events, like code changes or scheduled times.
Build after other projects are built
Build periodically
GitHub hook trigger for GITScm polling
Poll SCM
Trigger builds remotely (e.g., from scripts)
Pipeline
Define your Pipeline using Groovy directly or pull it from source control.
Definition
Pipeline script from SCM
SCM
Git
Repositories
Save
Apply


Jenkins
SPE-ScientificCalculator-MiniProject
Configuration
Configure
General
Triggers
Pipeline
Advanced
Repositories
Repository URL
https://github.com/Aayush-Bhargav/SPE_MiniProject
Credentials
- none -
Add
Advanced
Add Repository
Branches to build
Branch Specifier (blank for 'any')
*/main
Save
Apply

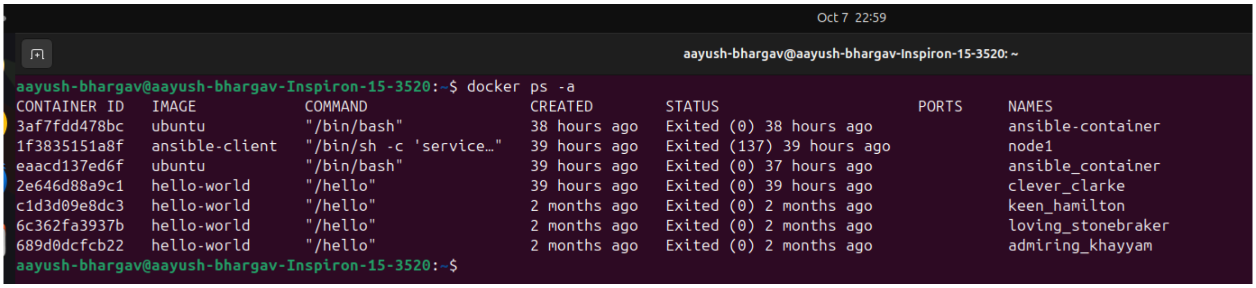


10. Running The Pipeline

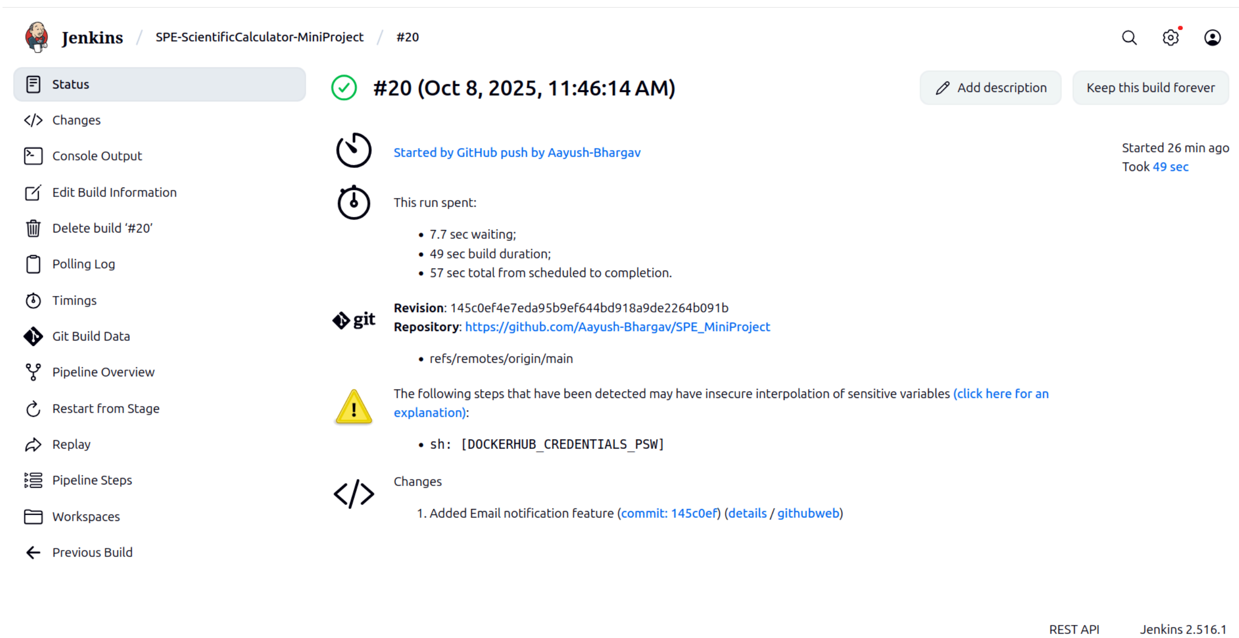
To run the pipeline, we can go to the created Jenkins pipeline job and click on the **'Build Now'** option.

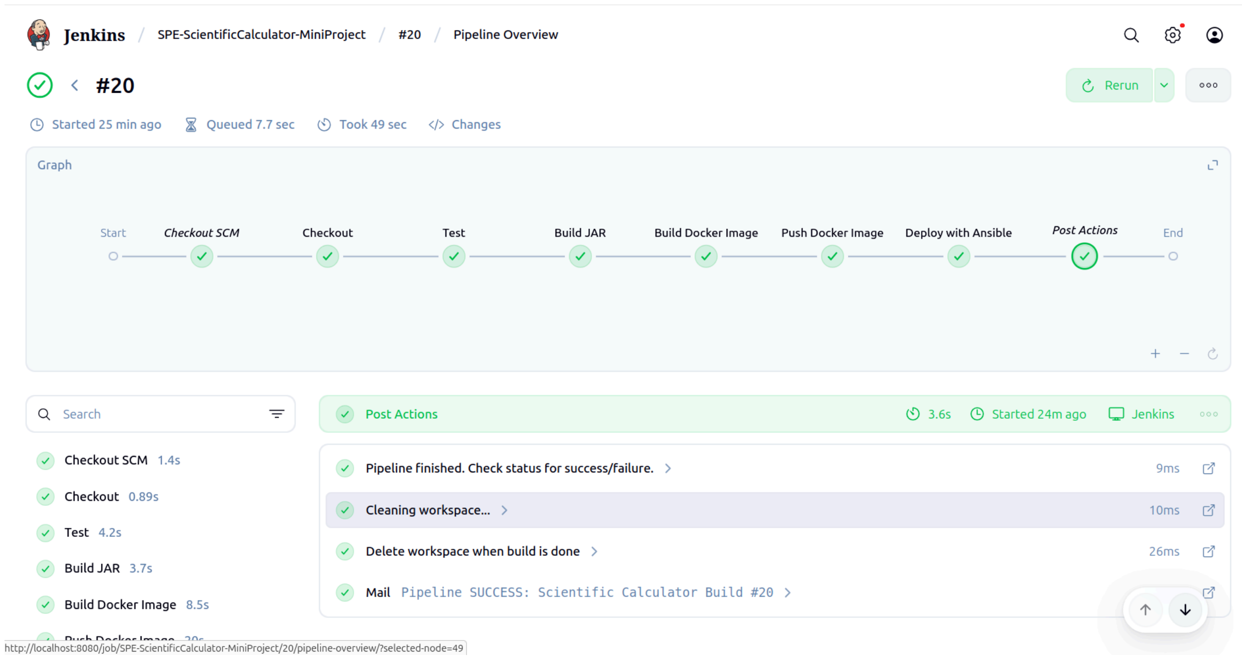


Before the build, the docker containers on my local host:

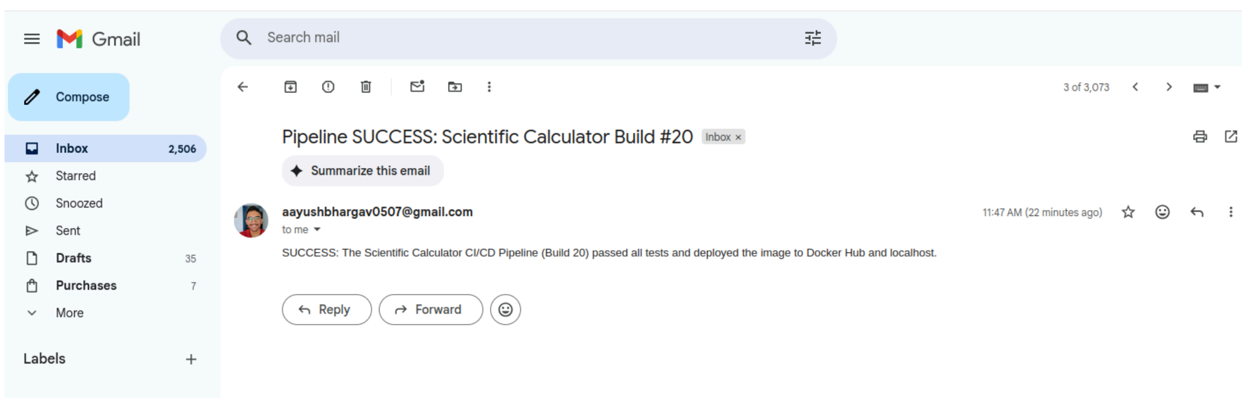


Pipeline job successfully completed:





Email Confirmation:



After the build, the docker containers on my local host:

The fact that you see the container status as **Up** means the Ansible playbook is working perfectly and the container is running in the background in the local host, waiting.

The next step you need to do: **use `docker exec` to attach a new interactive process to that running container.**

The Command to Connect

Since the application requires an interactive terminal (`Scanner`), you need to run a new process **inside** the container that provides that terminal connection.

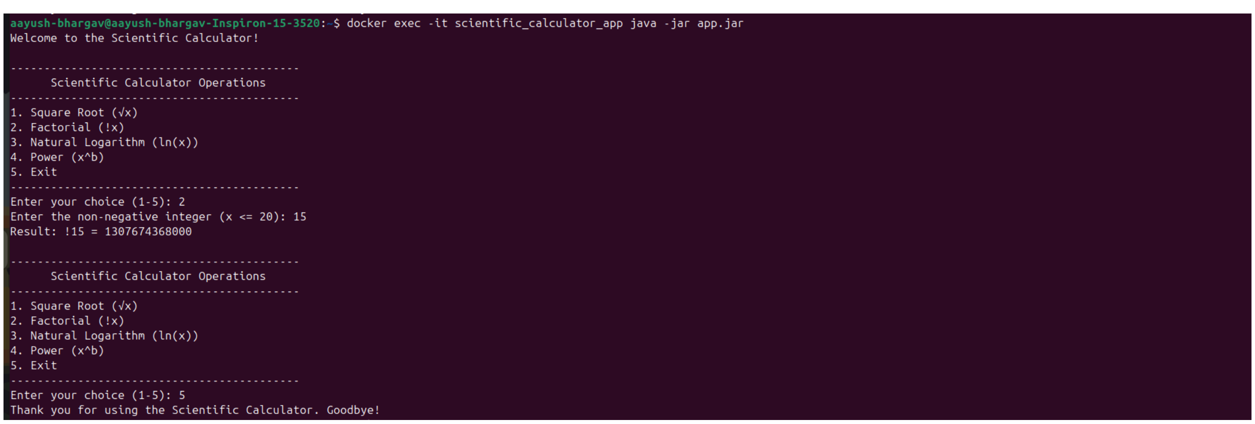
Here is the command to run on your host machine:

docker exec -it scientific_calculator_app java -jar app.jar

Explanation of the Process

1. **`docker exec`**: Executes a new command inside the already running container.
2. **`-it`**: This combination is what unlocks the **interactive terminal**. It connects your current keyboard/terminal to the container's input/output streams.
3. **`scientific_calculator_app`**: The target container ID or name.
4. **`java -jar app.jar`**: The command you want to run. This manually launches the Java application again inside the established interactive session.

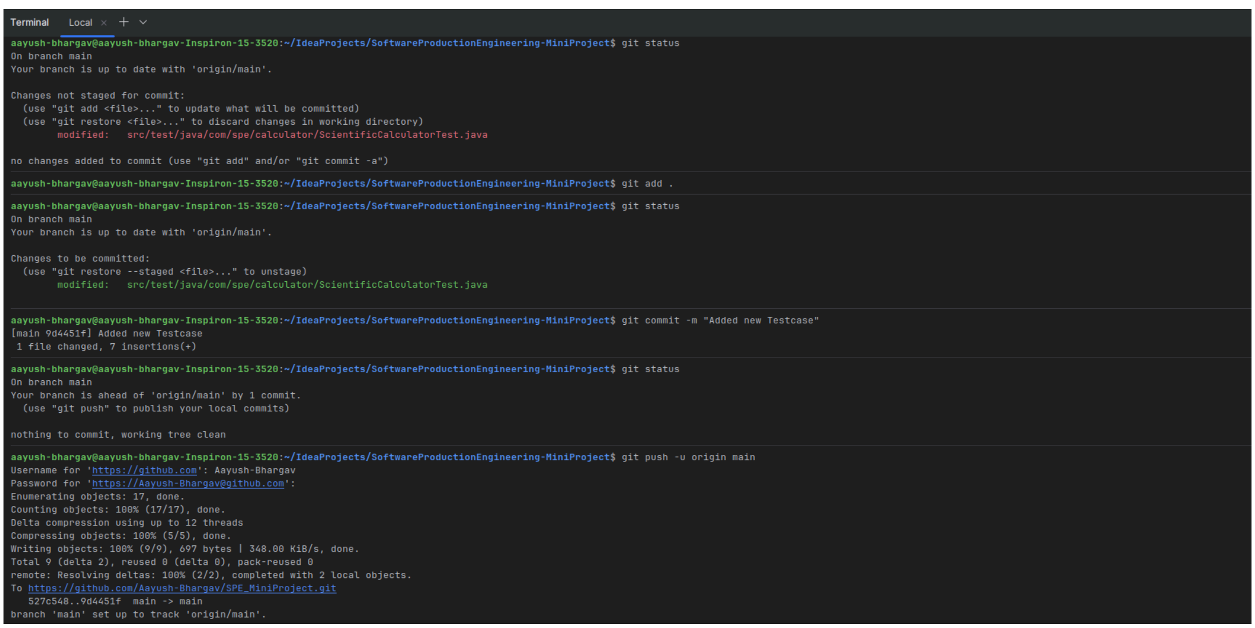
When you run this command, you will see your **calculator menu appear**, and you can then enter a choice (1-5) and interact with the application fully!



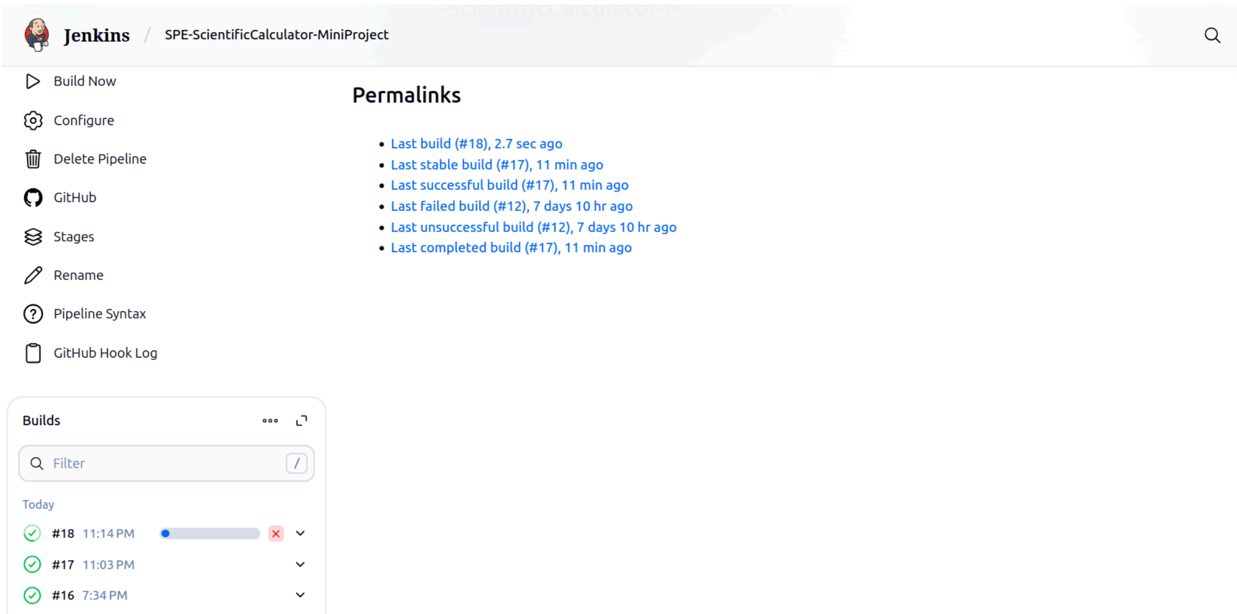
Instead of the build now option in jenkins, we can also make a change to the code ourselves, commit it and see the job being executed.

Writing a new test case:

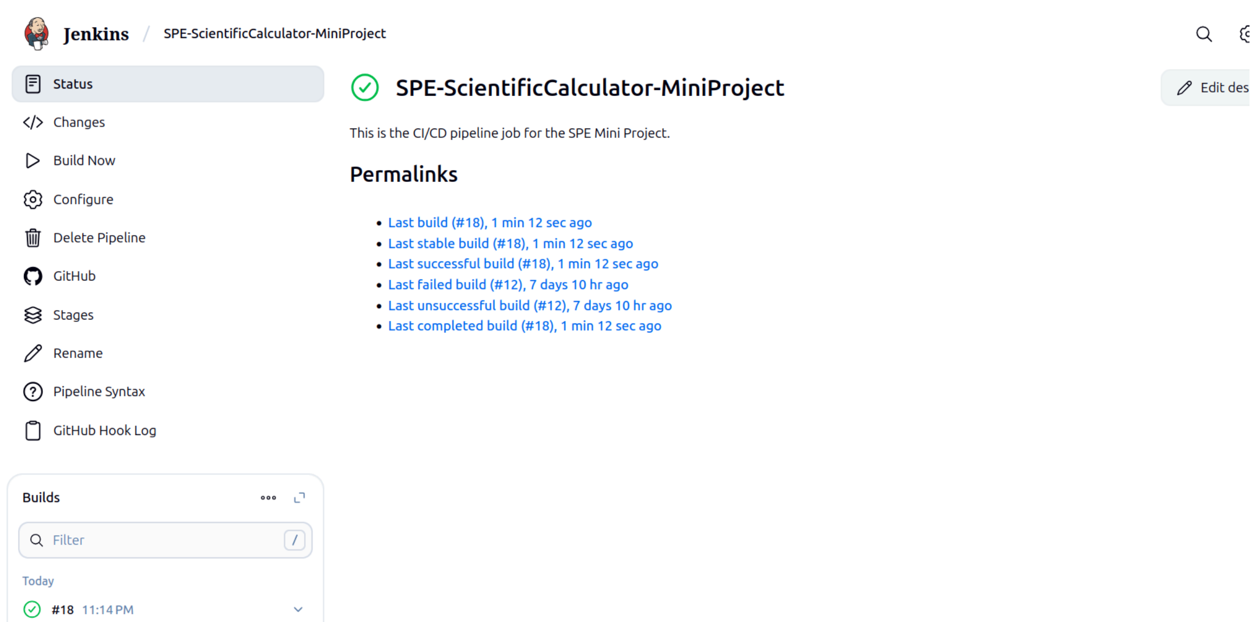
Let us commit the changes and push the changes to github.



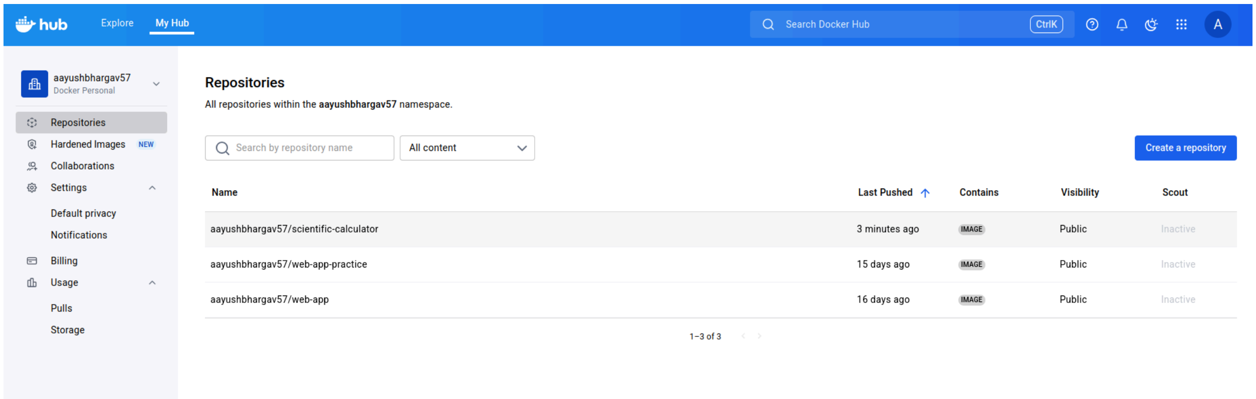
You can clearly see, that the jenkins job has been triggered:

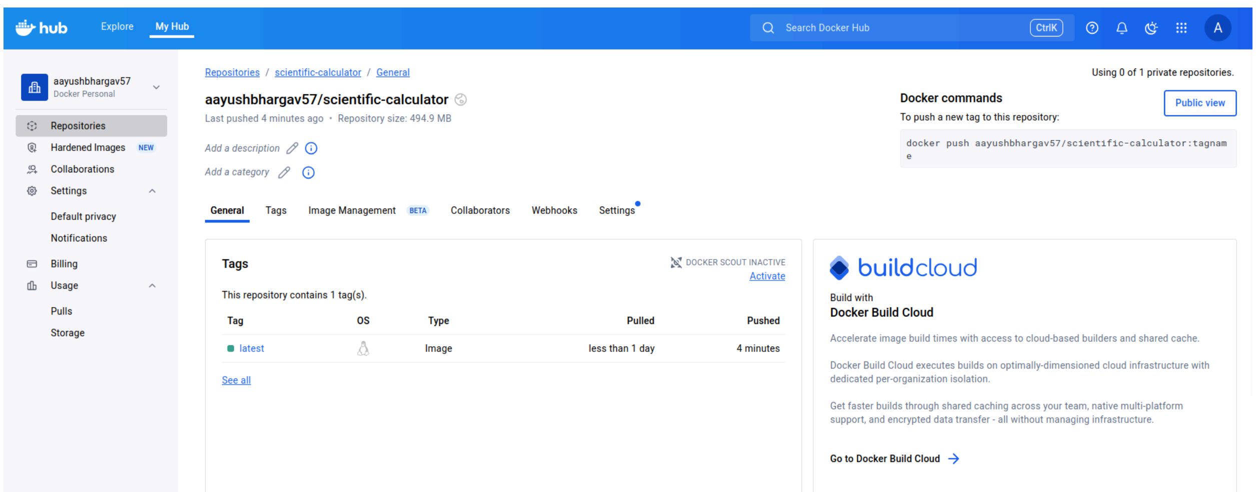


Build completed Successfully as well!



The docker container will be on the local host and we can run it again as before.
You can also verify that the image has been created and pushed to my dockerhub :





Link to public Github Repository:

https://github.com/Aayush-Bhargav/SPE_MiniProject

Link to Dockerhub:

https://hub.docker.com/u/aayushbhargav57