International Institute of Information Technology Bangalore

Software Production Engineering

CSE-816

MINI PROJECT

Ajitesh Kumar Singh
(IMT2022559)

GitHub: github.com/Transyltooniaa/Calculator-App
Docker Hub: hub.docker.com/r/transyltoonia/calculator-app

October 10, 2025

Contents

Contents

Chapter 1

Executive Summary

Overview

This project demonstrates a complete end-to-end **DevOps lifecycle** implementation for a sample **Node.js Scientific Calculator Application**. The objective is to integrate development, testing, containerization, deployment, and automation processes within a single CI/CD pipeline achieving reliable, repeatable, and consistent delivery from code to deployment.

Purpose

The goal is to practically showcase how each DevOps component fits into a modern software delivery workflow:

- Efficient source control and collaboration using **GitHub**.
- Automated testing via **Jest** ensuring code quality.
- Seamless containerization using **Docker** for reproducible builds.
- Continuous integration and delivery through **Jenkins**.
- Automated configuration and deployment using **Ansible**.

Together, these demonstrate how DevOps principles improve reliability, speed, and scalability of software releases.

Scope

The pipeline covers:

1. **Code Management:** Version-controlled project repository on GitHub.
2. **Testing:** Unit tests written with Jest to verify core calculator functions.
3. **Build & Packaging:** Container image creation using Docker.
4. **Continuous Integration:** Jenkins automating testing and image builds.

5. **Deployment:** Ansible provisioning and deploying on a local environment.
6. **Monitoring & Feedback:** Basic verification and logs post-deployment.

Outcomes

At the end of the pipeline:

- The application builds, tests, and packages successfully with zero manual intervention.
- Docker images are built and optionally published to Docker Hub.
- The local system is automatically configured and the container deployed using Ansible.
- CI/CD feedback is visible within Jenkins dashboards.

Key Deliverables

- **Source Code:** GitHub repository containing the Node.js calculator app.
- **Automation Scripts:** Jenkinsfile, Dockerfile, and Ansible playbook.
- **Documentation:** Step-by-step reproducibility guide and screenshots.
- **Container Image:** Built and pushed to Docker Hub.

Chapter 2

Tools Used (Overview)

Introduction

This project demonstrates a complete DevOps pipeline built around modern open-source tools. Each tool plays a key role in automating one or more phases of the software development lifecycle — from coding and testing to deployment. Together, these tools enable the creation, validation, packaging, and automated deployment of the **Scientific Calculator Application**.

2.1 Node.js

Figure 2.1: nodejs

Purpose: Application Development and Runtime Environment. Node.js provides the core runtime for building the calculator using JavaScript and managing dependencies with npm.

Role in Pipeline:

- Implements calculator operations and business logic.
- Provides a testable runtime for Jest.
- Forms the base layer for containerization.

Version: Node.js LTS (latest stable version). **Project Entry Point:** `app.js`

2.2 GitHub

Figure 2.2: Github

Purpose: Source Control and Collaboration Platform. GitHub manages version control and acts as the central repository for source code and automation scripts.

Role in Pipeline:

- Stores source code, test scripts, and DevOps configuration files.
- Tracks version history and changes.
- Integrates with Jenkins for automated builds on code push.

Typical Workflow:

```
git add .
git commit -m "Add factorial functionality"
git push origin main
```

2.3 Jest

Figure 2.3: Jest

Purpose: Testing Framework for Node.js Applications. Jest ensures reliability of the scientific calculator's mathematical operations by validating functionality through unit tests.

Role in Pipeline:

- Executes automated tests during the Jenkins build stage.
- Generates pass/fail reports and coverage data.

Command:

```
npm test
```

2.4 Jenkins

Figure 2.4: Jenkins

Purpose: Continuous Integration and Delivery Tool. Jenkins automates the full workflow from pulling code to deployment.

Role in Pipeline:

- Pulls latest commits from GitHub.
- Runs automated tests through Jest.
- Builds and pushes Docker images.
- Triggers deployment using Ansible.

Configuration: Pipeline stages and automation logic are defined in the `Jenkinsfile` *(to be detailed in Chapter 7: Continuous Integration).*

2.5 Docker

Figure 2.5: Docker

Purpose: Containerization Platform. Docker ensures the calculator runs consistently across environments.

Role in Pipeline:

- Packages the Node.js application with its dependencies.
- Provides isolated and reproducible runtime environments.
- Produces an image that can be pushed to Docker Hub.

Base Image Used: `FROM node:lts-alpine` *(full Dockerfile will be discussed in Chapter 8: Containerization).*

2.6 Docker Hub

Purpose: Centralized Image Repository. Docker Hub stores versioned Docker images that can be pulled during deployment.

Role in Pipeline:

- Acts as a remote registry for project images.
- Enables image sharing and retrieval via Jenkins and Ansible.

(Details covered in Chapter 9: Image Publishing).

2.7 Ansible

Figure 2.6: ansible

Purpose: Configuration Management and Deployment Automation. Ansible automates pulling the Docker image and deploying it on the local host.

Role in Pipeline:

- Manages application deployment and environment setup.
- Uses playbooks to run Docker containers locally.

(The full playbook will be included in Chapter 10: Configuration and Deployment).

Summary

Each tool plays a distinct yet interconnected role:

- **GitHub** — Source control and integration trigger.
- **Jest** — Automated testing.
- **Jenkins** — Continuous integration and delivery.
- **Docker** — Containerization and packaging.
- **Docker Hub** — Image registry for versioned containers.
- **Ansible** — Automated deployment on local environment.

This toolchain demonstrates the full DevOps lifecycle — from code to automated deployment in a simple and reproducible setup.

Chapter 3

Source Control (GitHub)

Overview

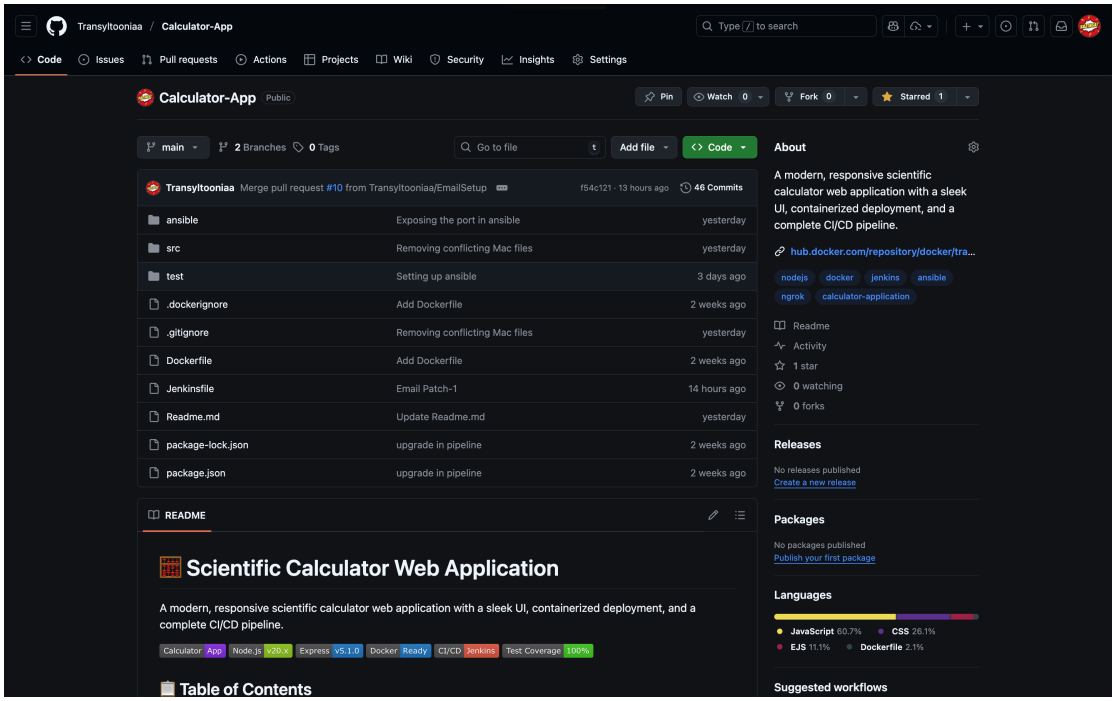

Figure 3.1: Repository

Source control forms the backbone of every DevOps pipeline by providing version tracking, collaboration, and integration triggers. For this project, **GitHub** was used as the Source Control Management (SCM) system to host, manage, and synchronize the entire codebase of the **Scientific Calculator Web Application**.

The public repository is available at: https://github.com/Transyltooniaa/Calculator-App

3.1 Repository Description

The repository contains the full implementation of the Scientific Calculator along with all DevOps automation scripts.

- **Frontend:** HTML, CSS, and EJS templates providing a responsive UI.
- **Backend:** Node.js + Express.js handling core logic and REST API endpoints.
- **Testing:** Jest test suite ensuring correctness of all calculator operations.
- **DevOps:** Jenkinsfile for CI/CD, Dockerfile for containerization, and Ansible playbook for deployment automation.

Repository Details:

- **Name:** Calculator-App
- **Visibility:** Public
- **Branch Strategy:** Single-branch workflow (`main`)
- **Owner:** `@Transyltooniaa`

3.2 Repository Structure

```
/Calculator-App
 Dockerfile              # Containerization configuration
 Jenkinsfile             # CI/CD pipeline definition
 package.json            # Project metadata and dependencies
 ansible/                # Deployment automation
    deploy.yml          # Ansible playbook (detailed in Chapter 10)
    inventory.ini       # Target configuration
 src/                    # Source code (Node.js + Express)
    app.js
    index.js
    controllers/
    routes/
    services/
    views/
 test/                   # Jest test files
```

This approach simplifies automation since Jenkins always pulls the latest stable version from the main branch.

Once pushed, Jenkins automatically detects new commits on the main branch and triggers a pipeline build.

3.3 GitHub Webhook via `ngrok` (Local Jenkins)

Purpose

To trigger Jenkins builds *immediately* on every push to `main`, we expose the local Jenkins instance (port `8080`) to the internet using **ngrok** and configure a GitHub webhook that points to Jenkins' GitHub endpoint.

Prerequisites

- Jenkins running locally on `http://localhost:8080`
- GitHub repository: Transyltooniaa/Calculator-App
- Jenkins job uses **"Pipeline script from SCM"** and targets branch `main`
- Jenkins GitHub integration enabled (build trigger: **GitHub hook trigger for GITScm polling**)
- `ngrok` installed and authenticated with your account

Step 1 — Install & Authenticate `ngrok`

```
# macOS (Homebrew):
brew install ngrok/ngrok/ngrok

# Add your auth token (from dashboard.ngrok.com)
ngrok config add-authtoken <YOUR_NGROK_AUTHTOKEN>
```

Step 2 — Start a Tunnel to Jenkins

```
ngrok http 8080
```

This prints a public forwarding URL, e.g. `https://<random>.ngrok.io`. **Note:** The URL changes on each run unless you use a reserved domain; update the GitHub webhook if it changes.

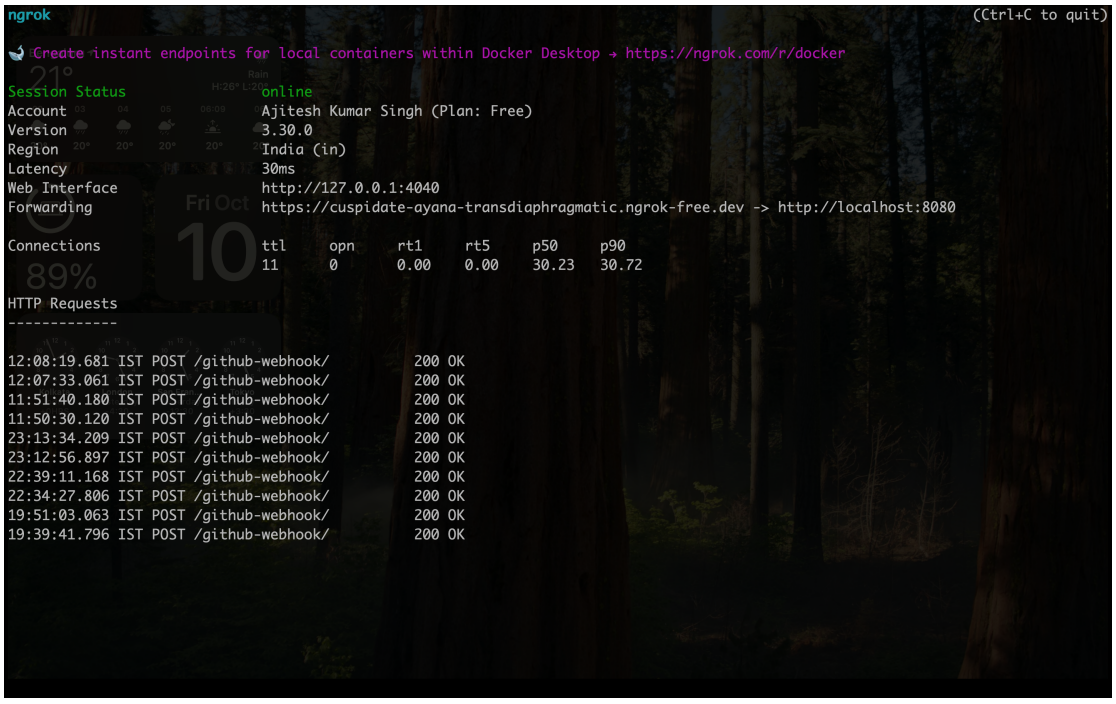

Figure 3.2: ngrok program runnning

Step 3 — Configure Jenkins Job Trigger

In your Pipeline job:

1. Open **Jenkins → Your Job → Configure**.
2. In **Build Triggers**, check **GitHub hook trigger for GITScm polling**.
3. Save.

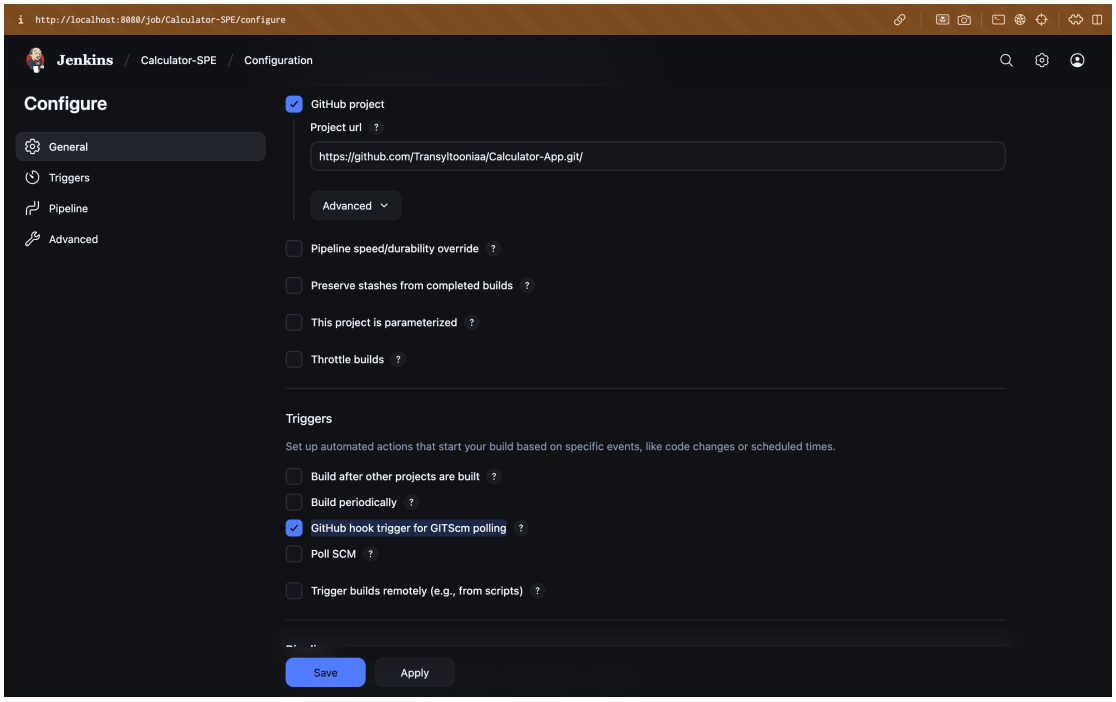

Figure 3.3: Setup in Jenkins

Step 4 — Add GitHub Webhook

On the repository:

1. **GitHub → Settings → Webhooks → Add webhook**
2. **Payload URL:** `https://<your-ngrok>.ngrok.io/github-webhook/`
3. **Content type:** `application/json`
4. **Secret (recommended):** set a random string; store the same secret in Jenkins if applicable (GitHub plugin / multibranch setups that validate HMAC)
5. **Which events: Just the push event** (and **Pull requests** if you want PR builds)
6. Click **Add webhook**

Step 5 — Verify Delivery

GitHub sends a *Ping* event immediately.

- In **GitHub → Webhooks → Recent Deliveries**, status should be 200.
- In **Jenkins**, a new build should trigger on the next push to `main`.

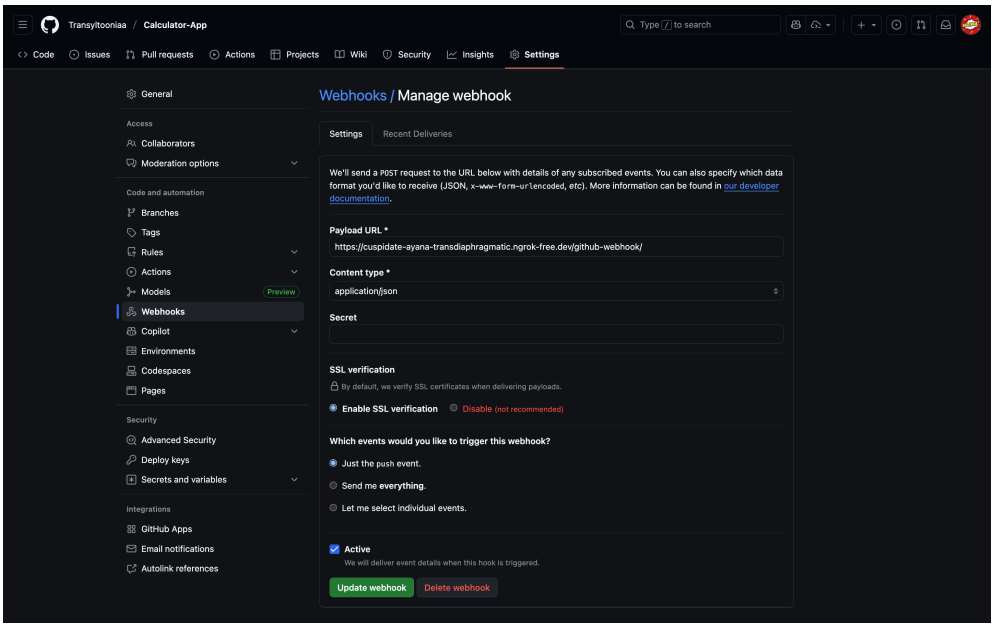

Figure 3.4: Jenkins job trigger configured for GitHub webhooks

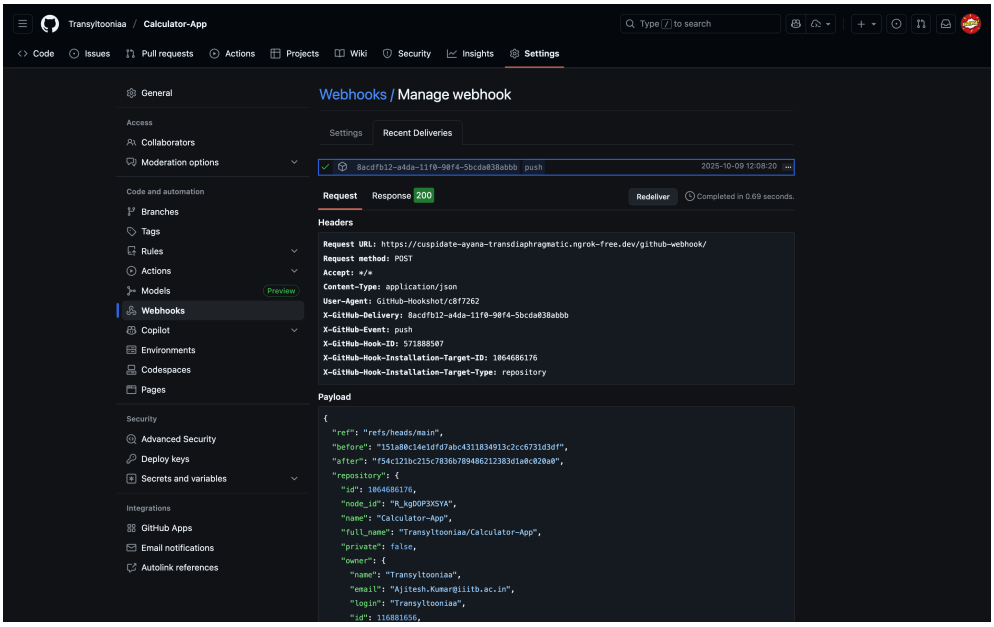

Figure 3.5: GitHub Webhook pinging to the ngrok URL

Chapter 4

Testing (Jest)

Overview

Automated testing is a key aspect of the DevOps lifecycle, ensuring that application logic remains correct and stable throughout continuous integration and deployment. This project uses the **Jest** testing framework to verify all core functionalities of the Scientific Calculator — including power, factorial, logarithmic, and square-root operations.

4.1 Purpose of Testing

The primary goals of testing in this project are:

- To validate all mathematical operations for accuracy.
- To catch potential errors (invalid inputs, unsupported operations).
- To enable automated test execution as part of the Jenkins pipeline.

By running tests automatically on each commit, the pipeline maintains confidence that all calculator operations behave as expected before proceeding to build and deployment stages.

4.2 Testing Framework: Jest

Jest was chosen for its simplicity, zero-configuration setup with Node.js, and fast execution speed. It provides features such as snapshot testing, watch mode, and integrated coverage reports.

Installed Version: Jest v30.1.3 **Execution Command:**

```
npm test
```

4.3 Configuration

The test configuration resides in the project's `package.json` file. Jest is run using the experimental VM modules flag for ES modules support.

```
"scripts": {
  "test": "node --experimental-vm-modules node_modules/jest/bin/jest.js --runInBand",
  "test:watch": "npm test -- --watch",
  "test:cov": "npm test -- --coverage"
}
```

These scripts allow developers to:

- Run a single full test cycle.
- Continuously re-run tests on file changes (watch mode).
- Generate coverage reports for validation.

4.4 Test File Structure

All test files are located inside the `/test` directory:

```
test/
 calcService.test.js
```

Each test imports the calculator's main compute logic from `src/services/calcService.js` and validates a set of mathematical operations.

4.5 Example Test Cases

Below is an excerpt from the implemented test suite validating the `compute()` function. These examples confirm the correctness of basic and edge-case scenarios.

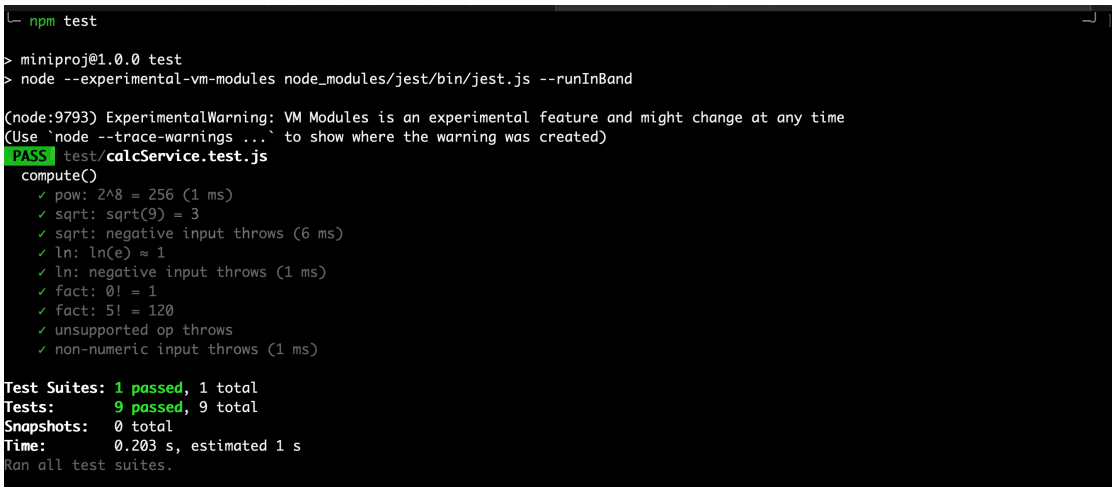

Figure 4.1: Jest

4.6 Test Execution

To execute the test suite:

```
npm test
```

Jenkins automatically runs this command during the CI stage. If any test fails, the pipeline halts, preventing faulty builds from being packaged or deployed.

4.7 Test Reporting

Running coverage reports provides visibility into which lines of code are covered by tests:

```
npm run test:cov
```

This generates an HTML coverage summary inside the `coverage/` folder, which can be reviewed locally or archived by Jenkins as part of the build artifacts.

Chapter 5

Continuous Integration (Jenkins)

Overview

Continuous Integration (CI) ensures that every change to the source code is automatically built, tested, and verified before being deployed. In this project, **Jenkins** was used as the CI/CD orchestrator to automate the full workflow for the **Scientific Calculator Web Application**. It handles everything — fetching code from GitHub, installing dependencies, running tests, building and pushing Docker images, and deploying using Ansible.

5.1 Jenkins Setup and Configuration

1. Jenkins Installation

Jenkins LTS was installed on the local machine and configured with the required system dependencies:

- **Node.js** – Installed globally using Jenkins plugin.
- **Docker Engine and CLI** – Installed via system package manager (Homebrew on macOS).
- **Ansible** – Installed locally and added to the Jenkins system `PATH`.

2. Plugin Installation

The following plugins were installed via **Manage Jenkins → Plugins**:

- **NodeJS Plugin** – Provides Node.js environments for build steps.
- **Pipeline Plugin** – Enables declarative and scripted pipelines.
- **Email Extension Plugin (Email-ext)** – For build success/failure notifications.
- **HTML Publisher Plugin** – Publishes Jest HTML coverage reports.
- **Ansible Plugin** – Allows invoking Ansible playbooks directly.

After installation, Jenkins was restarted to activate the plugins.

3. Node.js Tool Configuration

Node.js was configured under **Manage Jenkins → Global Tool Configuration**:

- Tool Name: `node`
- Installation Method: "Install automatically"
- Version: `Node.js 24.9.0 LTS`

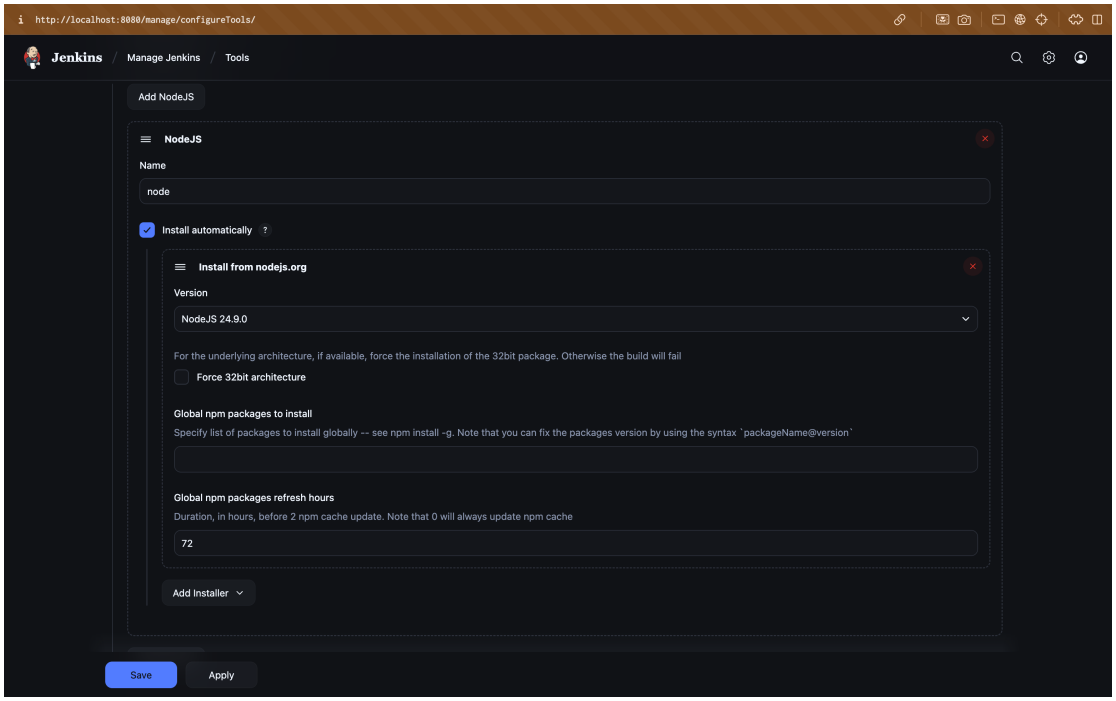

Figure 5.1: Node.Js installation for Global too configutaion

This allows the pipeline to reference Node.js using:

```
tools {
  nodejs 'node'
}
```

4. Docker Hub Credentials Setup

To securely push Docker images, Docker Hub credentials were added via: **Manage Jenkins → Credentials → Global → Add Credentials**

- **Kind:** Username with Password
- **ID:** `docker_creds`
- **Username:** `transyltoonia`
- **Password:** Docker Hub Access Token
- **Scope:** Global

The credential ID `docker_creds` is later referenced in the pipeline during the **Push Image** stage

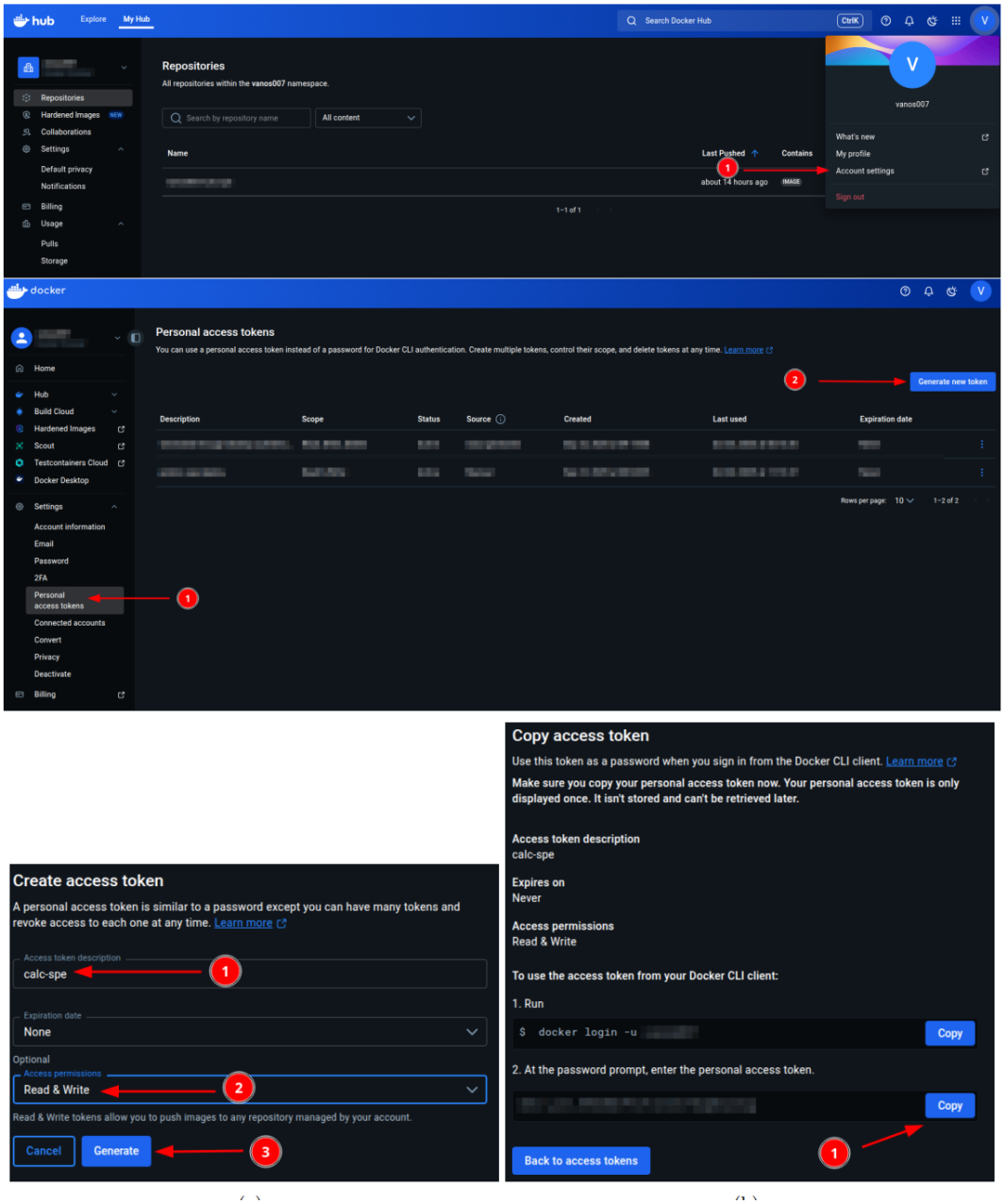

Figure 5.2: Dockerhub personal access token generation

.

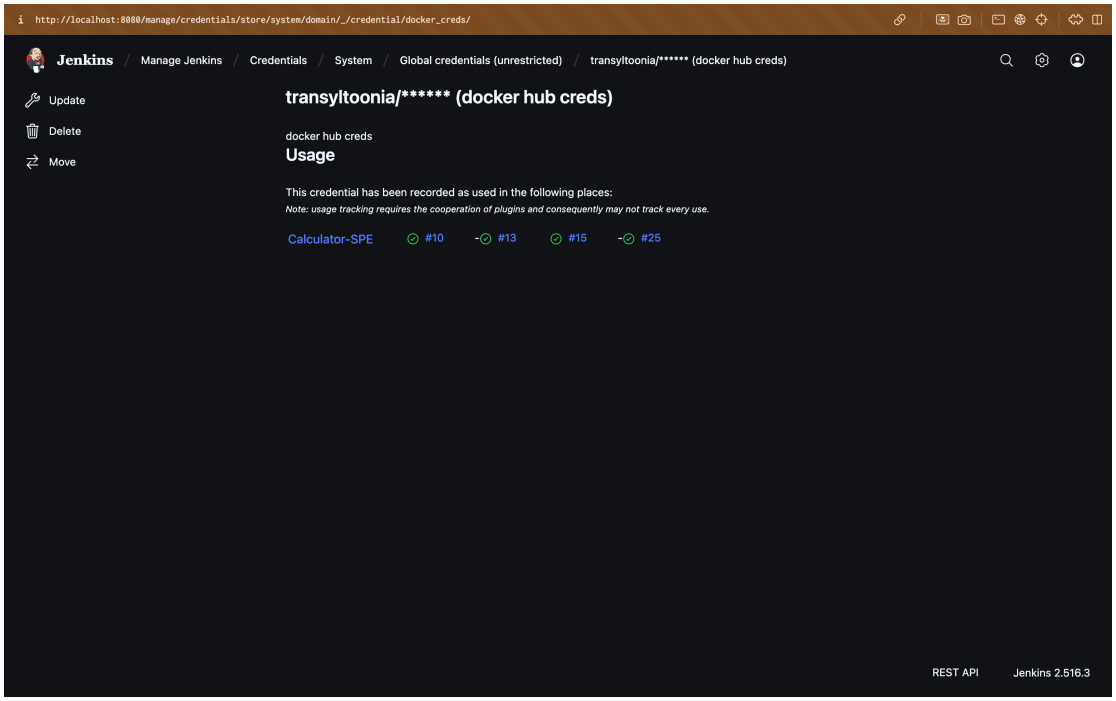

Figure 5.3: DockerHub Credentials in Jenkins Global Credentials

5. GitHub Integration

The repository Calculator-App was integrated by creating a **Pipeline** project in Jenkins with the following setup:

- Pipeline Definition: `Pipeline script from SCM`
- SCM: `Git`
- Repository URL: `https://github.com/Transyltooniaa/Calculator-App.git`
- Branch Specifier: `*/main`
- Script Path: `Jenkinsfile`

This ensures Jenkins automatically pulls the latest code from the `main` branch.

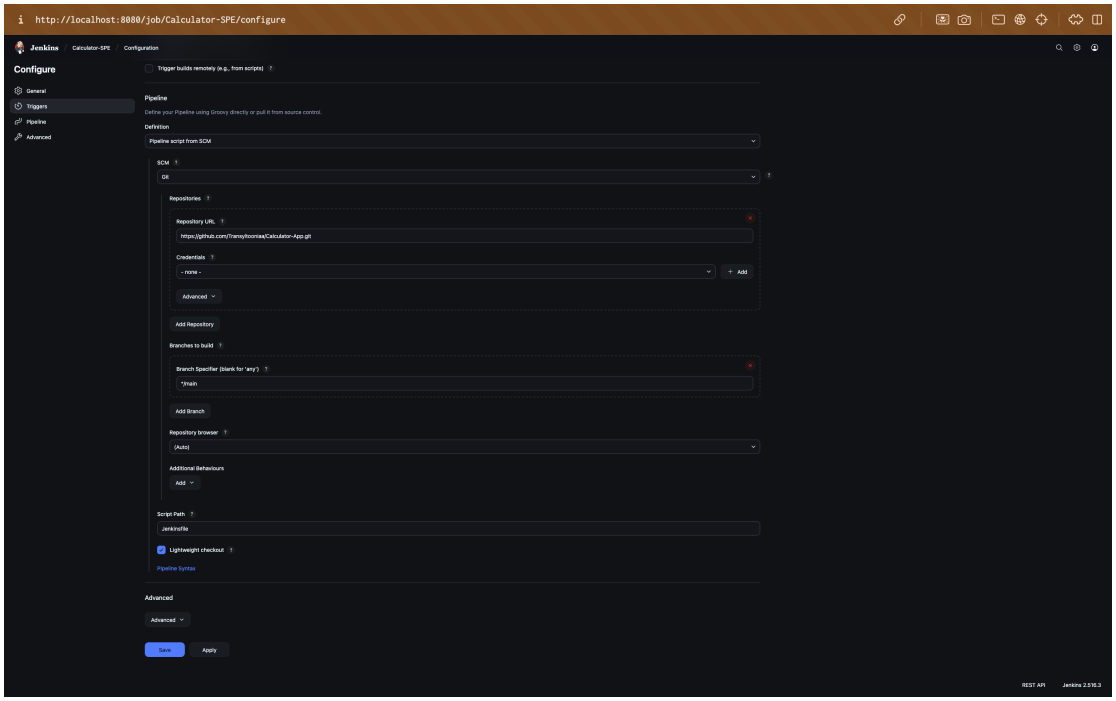

Figure 5.4: Pipeline Configuration

5.2 Pipeline Definition

The pipeline is written in declarative syntax and executes seven main stages, each corresponding to one CI/CD step. Below is the structural overview:

```
pipeline {
  agent any
  tools { nodejs 'node' }
  environment {
    DOCKER_USER = 'transyltoonia'
    DOCKER_IMAGE_NAME = 'calculator-app'
    DOCKER_TAG = 'latest'
    DOCKER_IMAGE = "${DOCKER_USER}/${DOCKER_IMAGE_NAME}:${DOCKER_TAG}"
  }

  stages {
    stage('Checkout') { ... }
    stage('Install') { ... }
    stage('Test') { ... }
    stage('Docker Diagnostics') { ... }
    stage('Build Image') { ... }
    stage('Push Image') { ... }
    stage('Use ansible to deploy') { ... }
  }
  post { ... }
}
```

5.3 Detailed Stage Descriptions

Stage 1 — Checkout SCM

Duration: 2.2 seconds **Purpose:** Fetches the latest code from GitHub. Jenkins uses its internal Git plugin to clone the repository and check out the `main` branch.

Stage 2 — Tool Install

Duration: 0.3 seconds Verifies that Node.js is available using the configured NodeJS plugin. This ensures subsequent npm commands run correctly.

Stage 3 — Install

Duration: 4–5 seconds Installs project dependencies using npm. If a lock file is found, `npm ci` is used for clean installs; otherwise, `npm install` is executed.

Stage 4 — Test

Duration: 2–3 seconds Runs Jest tests with coverage. After test completion, the coverage directory is archived as a build artifact. If any test fails, the pipeline halts.

Stage 5 — Docker Diagnostics

Duration: 1 second Validates Docker installation and displays environment details:

```
docker version
which docker
echo "PATH=$PATH"
```

Stage 6 — Build Image

Duration: 4–7 seconds Builds the Docker image for the Scientific Calculator application using:

```
/usr/local/bin/docker build -t ${DOCKER_IMAGE} .
```

Outputs build logs and confirms successful image creation.

Stage 7 — Push Image

Duration: 16–28 seconds Authenticates to Docker Hub using stored credentials and pushes the built image:

```
echo "$PSW" | docker login -u "$USR" --password-stdin
docker push ${DOCKER_IMAGE}
```

Once the push is complete, the image becomes publicly available at:

`https://hub.docker.com/r/transyltoonia/calculator-app`

Stage 8 — Use Ansible to Deploy

Duration: 11–13 seconds Triggers the Ansible playbook for deployment:

```
ansible-playbook -i ansible/inventory.ini ansible/deploy.yml
```

Ansible pulls the Docker image and launches it as a running container on the local host.

Stage 9 — Post Actions

Duration: 4–5 seconds Includes:

- Cleaning up workspace.
- Publishing Jest coverage HTML report (if available).
- Sending email notifications to developers.

5.4 Email Notification Setup and Delivery

Purpose

Automated email notifications are configured in Jenkins to alert the developer of build results immediately after each pipeline run.

Plugin Used

The feature is enabled using the **Email Extension Plugin (Email-ext)** in Jenkins, which allows HTML-formatted emails and conditional triggers for success and failure events.

Configuration Steps

1. Navigate to **Manage Jenkins → Configure System**.
2. Under the **Extended E-mail Notification** section, fill in:
    - **SMTP Server:** `smtp.gmail.com`
    - **Use SMTP Authentication:** Enabled
    - **User Name:** Jenkins service email (e.g., `ajitesh710@gmail.com`)
    - **Password:** App-specific password or access token
    - **Use SSL:** Checked
    - **SMTP Port:** 465
    - **Default Content Type:** HTML
3. Click **Advanced** and add a default recipient list (e.g., `Ajitesh.Kumar@iiitb.ac.in`).
4. Test configuration using "Send Test E-mail".

Pipeline Integration

The Jenkinsfile defines notification logic inside the `post` block to send emails on both success and failure outcomes.

```
post {
  success {
    emailext(
      subject: "Build #${env.BUILD_NUMBER} succeeded - ${env.JOB_NAME}",
      body: """
        <p>Hi Team,</p>
        <p>The Jenkins build <b>#${env.BUILD_NUMBER}</b> for <b>${env.JOB_NAME}</b> w
        <p><b>Details:</b></p>
        <ul>
          <li>Job: ${env.JOB_NAME}</li>
          <li>Build Number: ${env.BUILD_NUMBER}</li>
          <li>URL: <a href="${env.BUILD_URL}">${env.BUILD_URL}</a></li>
        </ul>
        <p>Regards,<br>Jenkins CI</p>
      """,
      mimeType: 'text/html',
      to: 'Ajitesh.Kumar@iiitb.ac.in'
    )
  }

  failure {
    emailext(
      subject: " Build #${env.BUILD_NUMBER} failed - ${env.JOB_NAME}",
      body: """
        <p>Hi Team,</p>
        <p>The Jenkins build <b>#${env.BUILD_NUMBER}</b> for <b>${env.JOB_NAME}</b> h
        <p>Check logs: <a href="${env.BUILD_URL}console">${env.BUILD_URL}console</a><
        <p>Regards,<br>Jenkins CI</p>
      """,
      mimeType: 'text/html',
      to: 'Ajitesh.Kumar@iiitb.ac.in'
    )
  }
}
```

Email Content

Each email includes:

- Build number and project name.
- Status (success/failure emoji included in subject).
- Direct link to build logs.
- Jenkins dashboard URL.

- Build time and duration (visible via $env.BUILD\_DURATION if added$).

Sample Email (Successful Build)

Subject: Build #45 succeeded — Calculator-App **From:** Jenkins CI **To:** Ajitesh.Kumar@iiitb.ac.in **Body:**

```
Hi Team,

The Jenkins build #45 for Calculator-App was successful!

Details:
Job: Calculator-App
Build Number: 45
URL: http://localhost:8080/job/Calculator-App/45/

Regards,
Jenkins CI
```

Sample Email (Failed Build)

Subject: Build #46 failed — Calculator-App **Body:**

```
Hi Team,

The Jenkins build #46 for Calculator-App has failed.

Check logs:
http://localhost:8080/job/Calculator-App/46/console

Regards,
Jenkins CI
```

Verification

To confirm email functionality:

- Run a test build (either force success or failure).
- Check Gmail inbox for the message from Jenkins CI.
- Verify that links direct to the correct build page.

Summary

Automated emails add a vital feedback loop for the CI/CD system. They notify developers instantly of any build status change, allowing quick response to failures and assurance of successful deployments without needing to log into Jenkins manually.

5.5 Pipeline Runtime Summary

From multiple runs, the pipeline shows an average total runtime of approximately **1 minute 1 second**. Table 5.1 summarizes average durations per stage.

Stage	Average Time
Checkout SCM	2.2 s
Tool Install	0.3 s
Install	4.5 s
Test	2.9 s
Docker Diagnostics	1.0 s
Build Image	4.5 s
Push Image	16–21 s
Use Ansible to Deploy	11–13 s
Post Actions	4.4 s
Total Runtime	**1 min 1 s**

Table 5.1: Average Execution Time per Pipeline Stage

5.6 Visualization of Pipeline Execution

Figure 5.5 shows the graphical overview of the Jenkins pipeline execution with all stages succeeding.

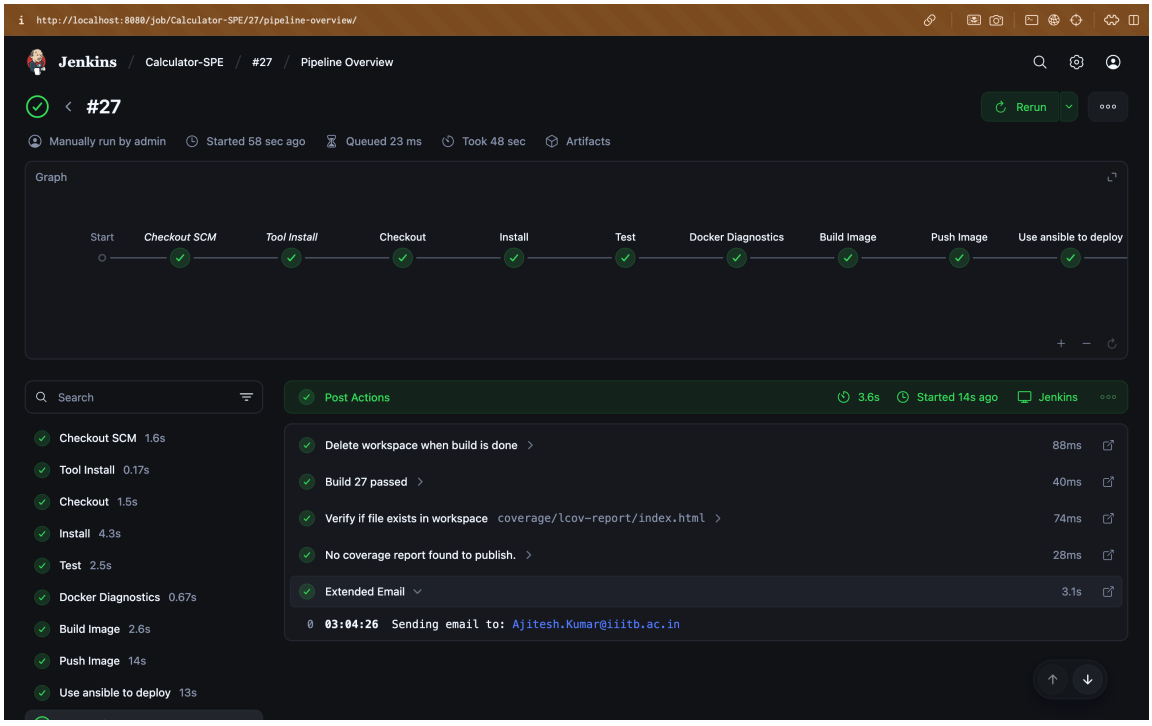

Figure 5.5: Jenkins Pipeline Overview (All stages passed)

Figure 5.6 presents the detailed stage view with timing metrics for multiple builds.

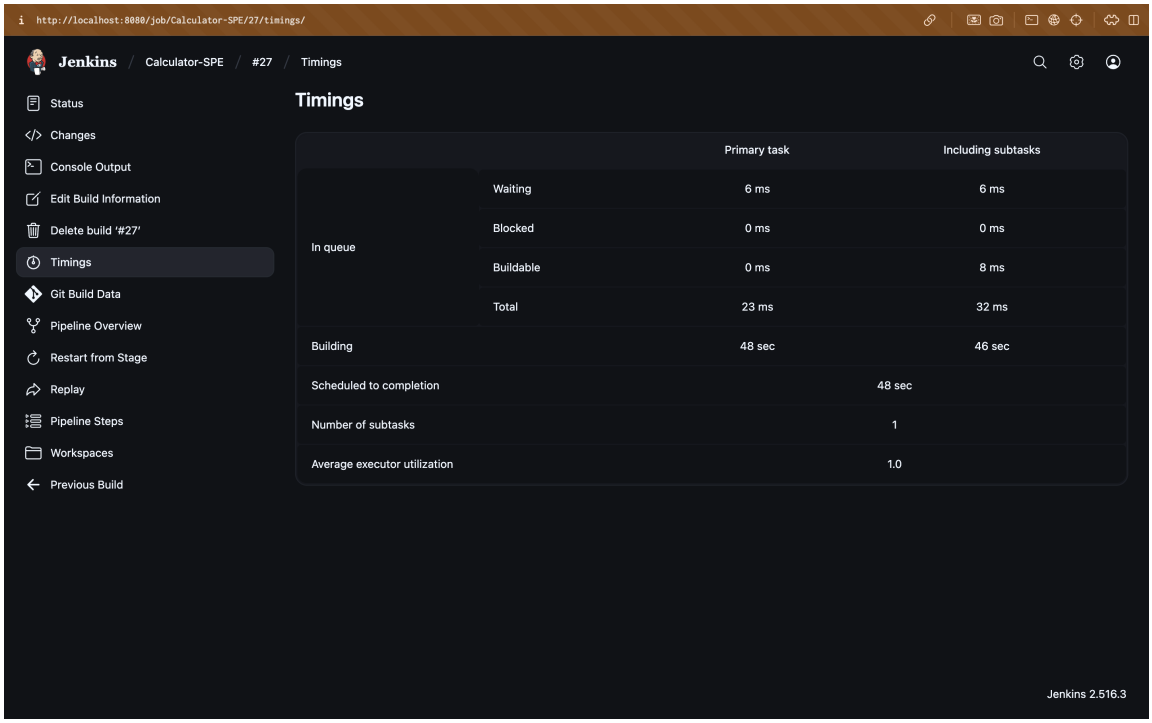

Figure 5.6: Jenkins Stage View with Stage Execution Times

Summary

The Jenkins pipeline seamlessly integrates every component of the DevOps workflow:

- Pulls source code from GitHub.
- Installs dependencies and runs Jest tests.
- Builds and pushes Docker images to Docker Hub.
- Deploys the latest container using Ansible.
- Sends notifications and archives reports automatically.

With a total runtime of just about one minute, this pipeline achieves full CI/CD automation—ensuring every code change is tested, built, and deployed with minimal human intervention.

Chapter 6

Containerization (Docker)

Overview

Containerization is a key pillar of modern DevOps workflows. It enables applications to run in consistent, isolated environments, ensuring that software behaves identically across all stages — development, testing, and production.

For this project, **Docker** was used to containerize the **Scientific Calculator Web Application** built on Node.js. The Dockerized setup ensures lightweight portability, reproducibility, and seamless integration with the CI/CD pipeline managed by Jenkins.

6.1 Objective of Containerization

The main objectives of containerizing the calculator app were:

- To package the Node.js application with its runtime and dependencies into a single deployable unit.
- To ensure consistent builds and eliminate "it works on my machine" issues.
- To simplify deployment via automated pulling and execution using Ansible.

6.2 Docker Setup

Docker is installed and configured on the local host machine (same node used by Jenkins and Ansible). It is authenticated using Docker Hub credentials for image pushes during CI/CD.

a minimal, production-ready container image.

```
# Use small base image
FROM node:lts-alpine

# Set working directory inside container
WORKDIR /app

# Copy only package.json and package-lock.json first (better caching)
COPY package*.json ./
```

```
# Install dependencies (omit devDependencies for smaller image)
RUN npm ci --omit=dev

# Copy the rest of your source code
COPY src/ ./src/

# Set environment variables
ENV NODE_ENV=production PORT=3000

# Expose the app port
EXPOSE 3000

# Start the app
CMD ["npm", "start"]
```

6.3 Explanation of Instructions

- **Base Image:** Uses `node:lts-alpine` — a lightweight, secure Node.js image ideal for production.
- **Working Directory:** Sets `/app` as the workspace within the container.
- **Layer Caching:** Copies only the package files first to leverage Docker's layer cache during rebuilds.
- **Dependency Installation:** Uses `npm ci --omit=dev` for faster, cleaner, and reproducible installs while excluding development dependencies.
- **Source Copy:** Copies application source files into the image.
- **Environment Variables:** Defines production environment and exposes port 3000.
- **Entrypoint:** Launches the app with `npm start`.

6.4 Building the Image

The Docker image is built using the following command:

```
docker build -t calculator-app .
```

Upon successful build, Docker outputs a new image ID and layer summary:

```
Successfully built <image_id>
Successfully tagged calculator-app:latest
```

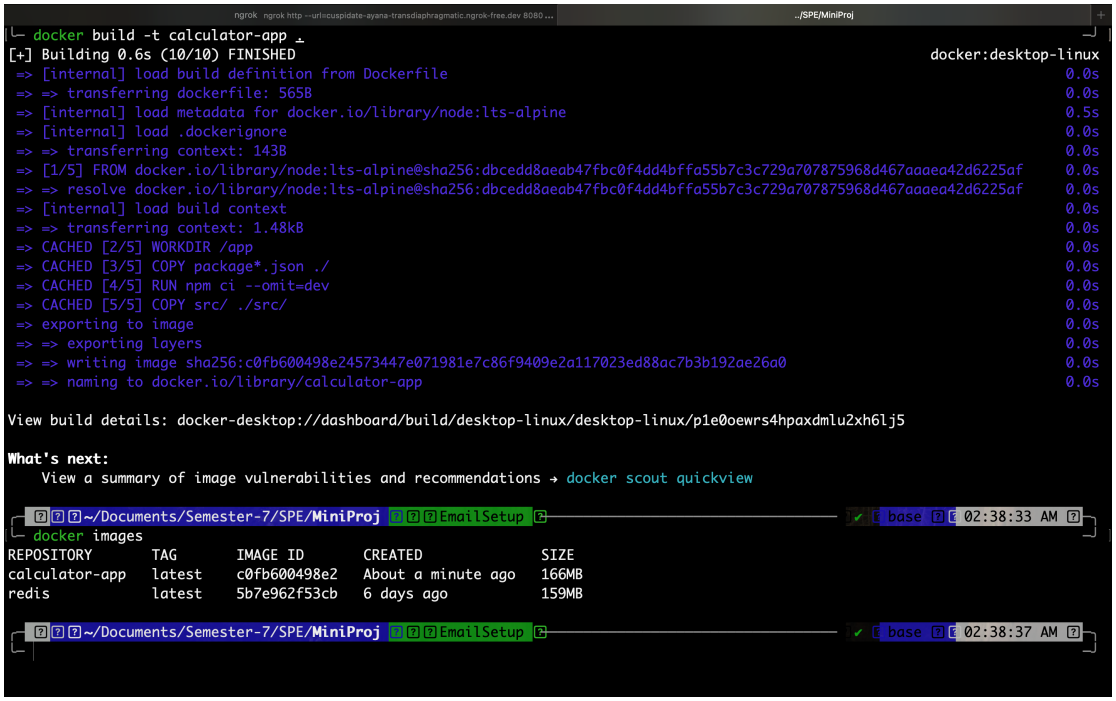

Figure 6.1: Building Image

6.5 Running the Container

Once the image is built, it can be executed locally with:

```
docker run -d -p 3000:3000 calculator-app
```

This binds the container's port 3000 to the host's port 3000, making the application accessible at:

`http://localhost:3000`

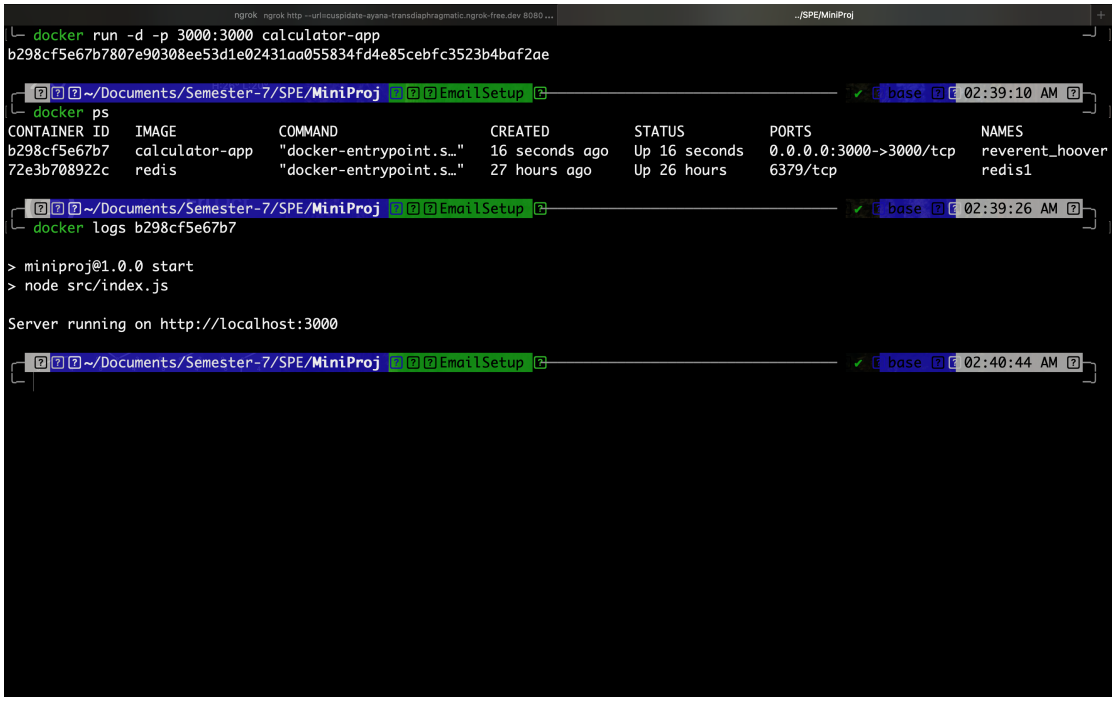

Figure 6.2: Running Container

6.6 Integration with Jenkins

The Docker build process is fully automated through Jenkins. In the pipeline, the following stages are responsible for containerization:

- **Build Image:** Executes the Docker build command to create the application image.
- **Push Image:** Logs in to Docker Hub using stored credentials and pushes the image for deployment.

Excerpt from the Jenkins pipeline:

```
stage('Build Image') {
  steps {
    script {
      sh '''
        /usr/local/bin/docker build -t ${DOCKER_IMAGE} .
      '''
    }
  }
}

stage('Push Image') {
  steps {
    withCredentials([usernamePassword(credentialsId: 'docker_creds',
                                      usernameVariable: 'USR',
                                      passwordVariable: 'PSW')]) {
      sh '''
```

```
        echo "$PSW" | /usr/local/bin/docker login -u "$USR" --password-stdin
        /usr/local/bin/docker push ${DOCKER_IMAGE}
        /usr/local/bin/docker logout
      '''
    }
  }
}
```

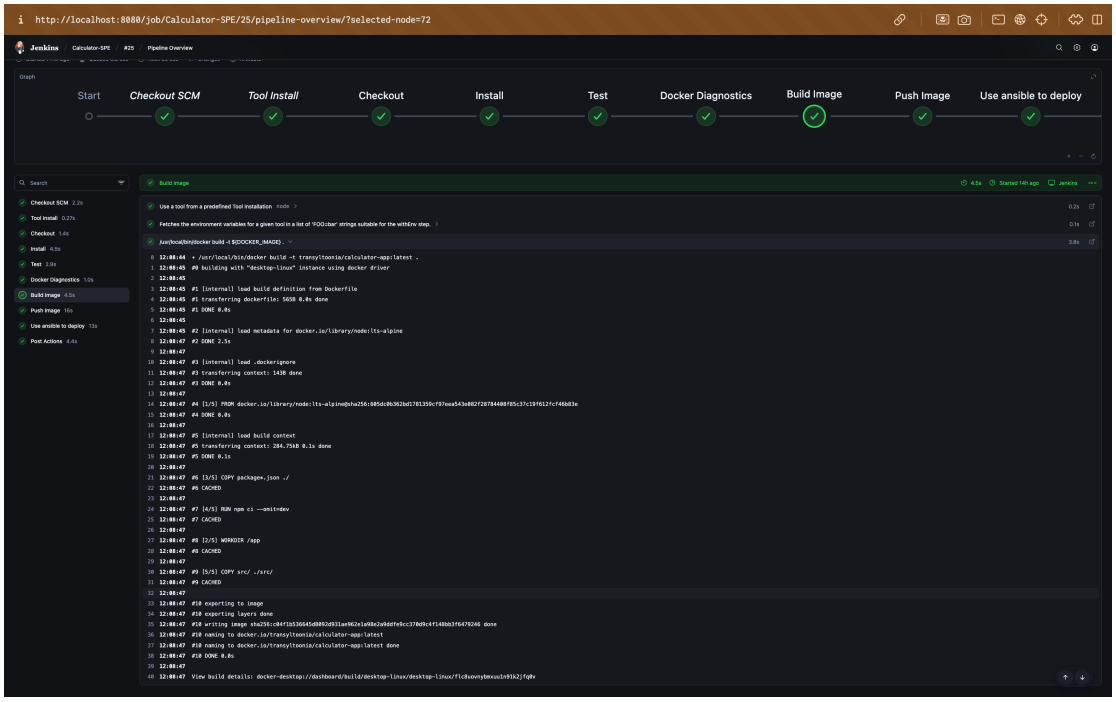

Figure 6.3: Build stage logs on Jenkins

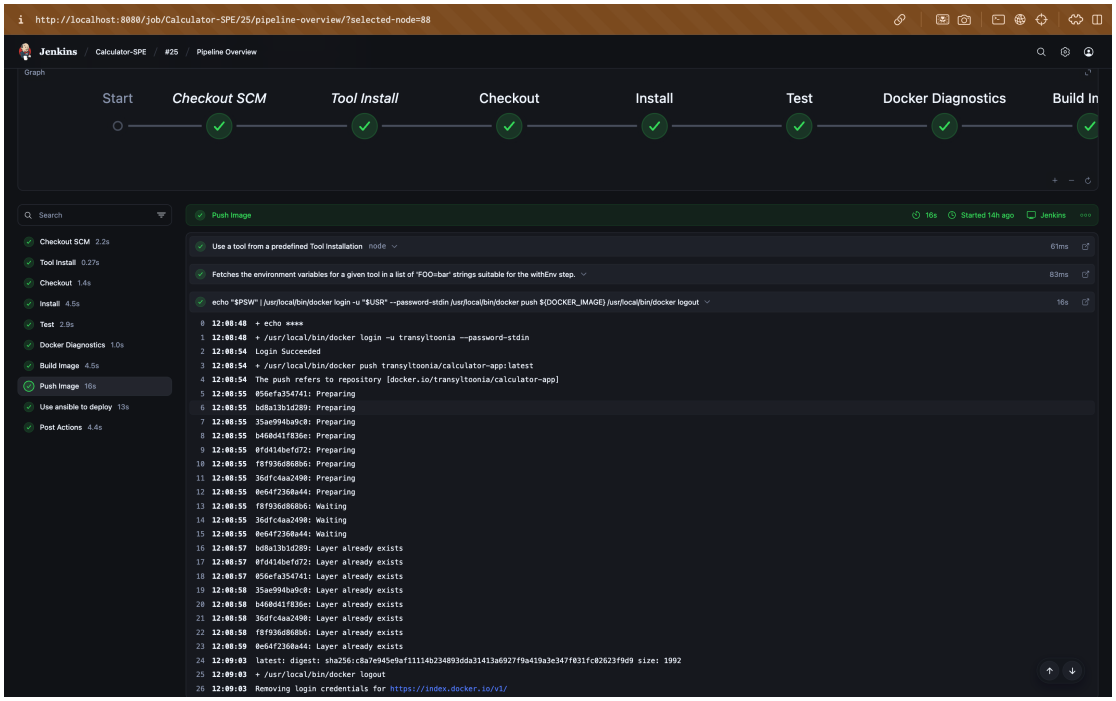

Figure 6.4: Jest

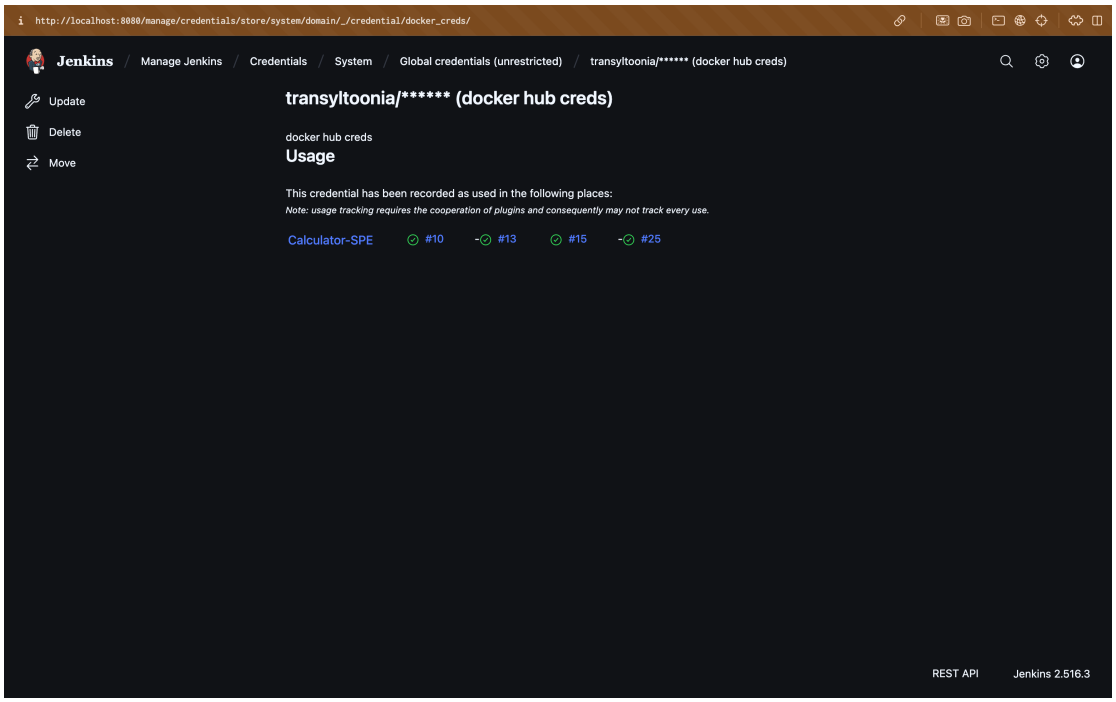

Figure 6.5: DockerHub Credentials

6.7 Image Publishing

After the image is successfully built, Jenkins pushes it to Docker Hub for deployment and archival:

```
docker push transyltoonia/calculator-app:latest
```

The published image can later be pulled by Ansible during deployment.

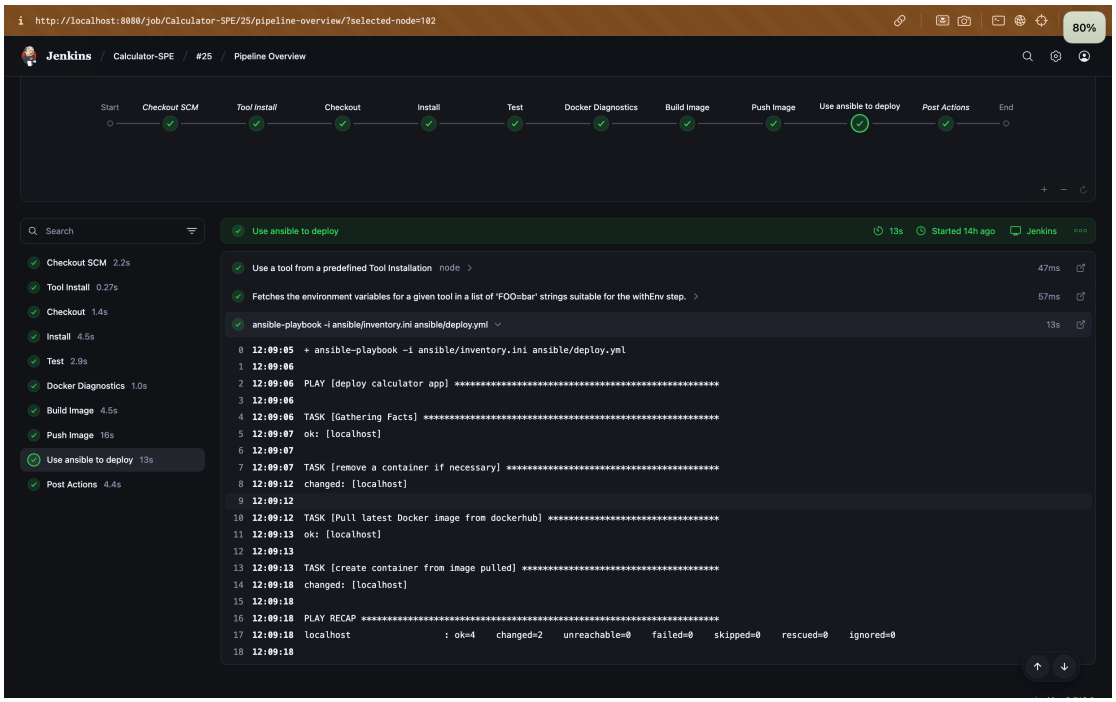

Figure 6.6: Ansible Logs

Chapter 7

Image Publishing (Docker Hub)

Overview

Once the Docker image is successfully built within Jenkins, it must be stored in a centralized registry to enable versioned distribution and automated deployment. For this project, **Docker Hub** was used as the image registry to host and distribute the container image for the **Scientific Calculator Application**.

7.1 Purpose of Image Publishing

- Provide a central repository for storing container images.
- Enable the Ansible deployment stage to pull the latest image directly from the registry.
- Maintain versioning and traceability between builds.
- Allow re-use of the same image across different environments (local or cloud).

7.2 Docker Hub Repository

- **Repository Name:** `calculator-app`
- **Owner:** `transyltoonia`
- **URL:** `https://hub.docker.com/r/transyltoonia/calculator-app`
- **Visibility:** Public (accessible for pull operations)

Each new Jenkins build pushes an updated version of the image tagged as:

`transyltoonia/calculator-app:latest`

7.3 Authentication and Credentials

To securely push images, Jenkins uses Docker Hub credentials configured globally:

- **Credentials ID:** `docker_creds`
- **Username:** `transyltoonia`
- **Password/Token:** Docker Hub Personal Access Token
- **Scope:** Global

These credentials are accessed in the pipeline using the `withCredentials` directive:

```
withCredentials([usernamePassword(credentialsId: 'docker_creds',
                                  usernameVariable: 'USR',
                                  passwordVariable: 'PSW')]) {
  sh '''
    echo "$PSW" | docker login -u "$USR" --password-stdin
    docker push ${DOCKER_IMAGE}
    docker logout
  '''
}
```

7.4 Tagging the Image

Before pushing, Jenkins tags the built image using environment variables defined at the top of the pipeline:

```
environment {
  DOCKER_USER = 'transyltoonia'
  DOCKER_IMAGE_NAME = 'calculator-app'
  DOCKER_TAG = 'latest'
  DOCKER_IMAGE = "${DOCKER_USER}/${DOCKER_IMAGE_NAME}:${DOCKER_TAG}"
}
```

This results in a standardized tag naming convention that can easily be extended for versioned releases (e.g., v1.0, v1.1).

7.5 Push Stage in Pipeline

During the **Push Image** stage, Jenkins performs the following steps sequentially:

1. Log in to Docker Hub using stored credentials.
2. Push the tagged image to the repository.
3. Log out securely to clear session tokens.

Excerpt from the pipeline:

```
stage('Push Image') {
  steps {
    withCredentials([usernamePassword(credentialsId: 'docker_creds',
                                      usernameVariable: 'USR',
                                      passwordVariable: 'PSW')]) {
      sh '''
        echo "$PSW" | /usr/local/bin/docker login -u "$USR" --password-stdin
        /usr/local/bin/docker push ${DOCKER_IMAGE}
        /usr/local/bin/docker logout
      '''
    }
  }
}
```

7.6 Verification of Image Push

After each successful Jenkins run, verification can be done in multiple ways:

- Jenkins console output shows:

  ```
  The push refers to repository [docker.io/transyltoonia/calculator-app]
  latest: digest: sha256:<digest-id> size: <bytes>
  ```

- The new image appears under the Docker Hub repository's **Tags** tab.
- Manual verification from CLI:

  ```
  docker pull transyltoonia/calculator-app:latest
  docker run -d -p 3000:3000 transyltoonia/calculator-app:latest
  ```

7.7 Integration with Deployment Stage

Once the image is published, the deployment stage in Jenkins triggers Ansible, which uses this exact tag to pull the latest image:

```
docker_image:
  name: transyltoonia/calculator-app:latest
  source: pull
```

This ensures that the container deployed locally or on any host always corresponds to the most recent CI build.

7.8 Screenshots and Logs

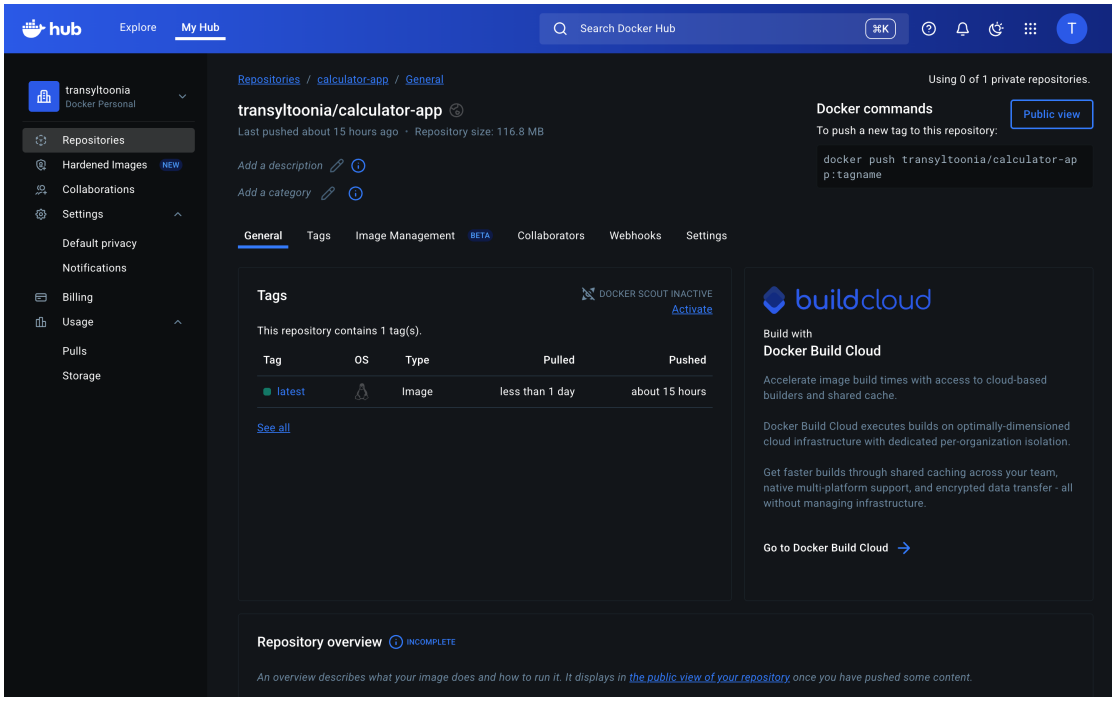

Figure 7.1: Image on DockerHub

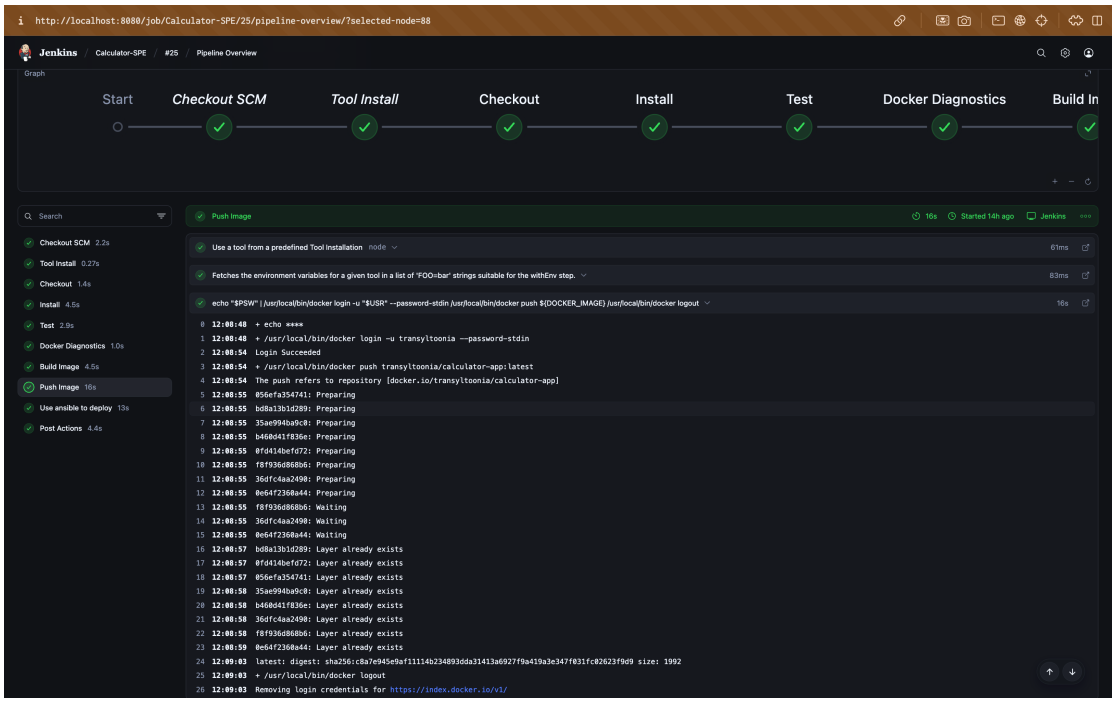

Figure 7.2: Push Stage Logs

Chapter 8

Configuration & Deployment (Ansible)

Overview

After Jenkins builds and publishes the Docker image to Docker Hub, deployment is automated with **Ansible** on the local machine. Ansible orchestrates the full container lifecycle: stop/remove any previous container, pull the latest image, and start the updated container bound to port `3000`.

8.1 Objectives

- Ensure reproducible deployments using idempotent Ansible tasks.
- Pull the exact image pushed by Jenkins (`transyltoonia/calculator-app:latest`).
- Run the app locally, exposing it at `http://localhost:3000`.

8.2 Prerequisites

Tools

- **Ansible** installed on the control node (same host).
- **Docker Engine & CLI** up and running.
- **Python** available to Ansible (interpreter path set in inventory).

Ansible Collections & Python SDK

The playbook uses modules from the `community.docker` collection. Install once:

```
ansible-galaxy collection install community.docker
```

On some systems, the Docker Python SDK may be required:

```
pip install docker
```

8.3 Inventory

File: `ansible/inventory.ini` — targets the local host and pins the Python interpreter.

```
[myhosts]
localhost ansible_connection=local ansible_python_interpreter=/opt/anaconda3/bin/pyth
```

8.4 Playbook

File: `ansible/deploy.yml` — declares variables for image/name/tag and performs three tasks.

Complete Playbook

```
- name: deploy calculator app
  hosts: myhosts
  vars:
    calc_image_name: "transyltoonia/calculator-app"
    calc_image_tag: "latest"
    calc_container: "calculator-app"

  tasks:
    - name: remove a container if necessary
      community.docker.docker_container:
        name: "{{ calc_container }}"
        state: absent

    - name: Pull latest Docker image from dockerhub
      community.docker.docker_image:
        name: "{{ calc_image_name }}"
        tag: "{{ calc_image_tag }}"
        source: pull

    - name: create container from image pulled
      community.docker.docker_container:
        name: "{{ calc_container }}"
        image: "{{ calc_image_name }}:{{ calc_image_tag }}"
        state: started
        tty: true
        interactive: true
        detach: true
        restart_policy: "no"
        ports:
          - "3000:3000"
```

Task-by-Task Explanation

1. **Remove container if necessary** Uses `docker_container` with `state: absent` to cleanly stop and remove any existing container named `calculator-app`. Idem-

potent—does nothing if absent.

2. **Pull latest Docker image** `docker_image` pulls `transyltoonia/calculator-app:latest` from Docker Hub to ensure the newest CI build is deployed.

3. **Create/Start container** Starts the container with the pulled image, attaches a TTY, and publishes `3000:3000`. The app becomes available at `http://localhost:3000`.

8.5 Running the Playbook

Manual Invocation

```
ansible-playbook -i ansible/inventory.ini ansible/deploy.yml
```

Via Jenkins (Automated)

The pipeline includes a deploy stage that invokes the same playbook:

```
stage('Use ansible to deploy'){
  steps{
    sh 'ansible-playbook -i ansible/inventory.ini ansible/deploy.yml'
  }
}
```

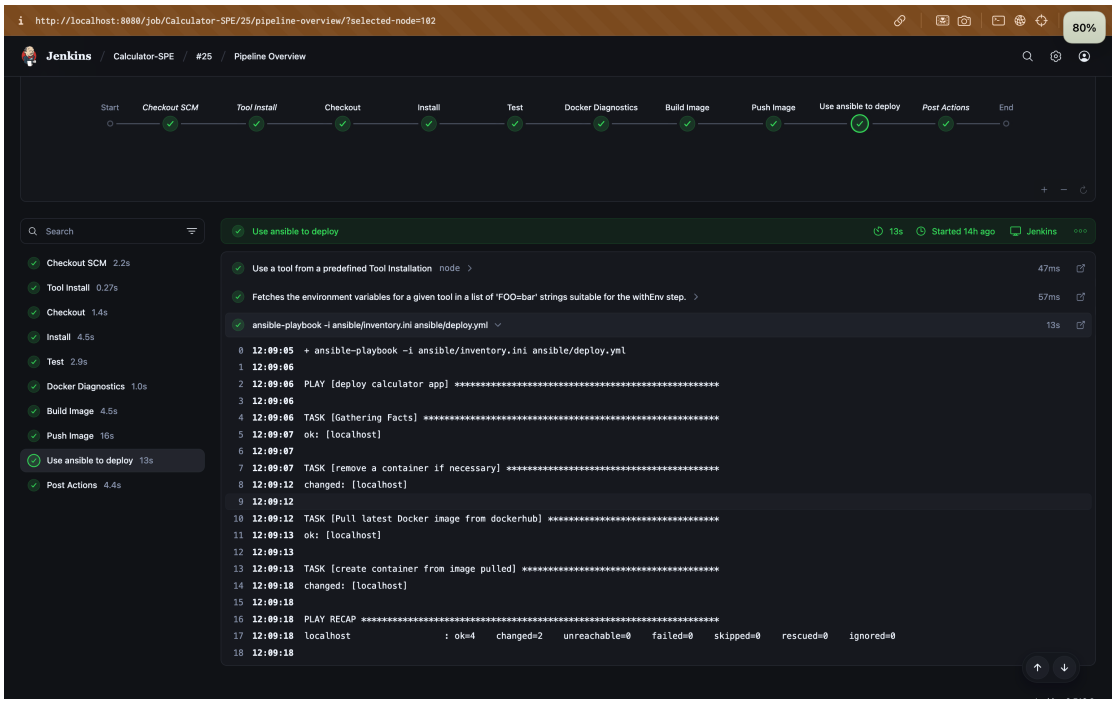

Figure 8.1: Ansible Stage Logs

8.6 Verification

- Confirm the container is running:

  ```
  docker ps --filter "name=calculator-app"
  ```

- Access the application: `http://localhost:3000`
- Review Ansible output for changed/ok status and any task failures.

8.7 Rollback & Re-Deploy

- Re-run the playbook after pushing a fixed image—idempotent tasks ensure safe convergence.
- If needed, manually stop/remove the container:

  ```
  docker rm -f calculator-app
  ```

- Pin a specific tag by overriding `calc_image_tag` (e.g., `v1.0`) for controlled rollbacks.

Chapter 9

Application Demo & Screenshots

Overview

This chapter showcases the final deployed **Scientific Calculator Web Application** in action. The application is fully containerized, deployed via Ansible, and accessible locally at `http://localhost:3000`. Screenshots illustrate the UI, operations, and successful deployment outcomes.

9.1 Application Interface

The calculator provides a modern, responsive interface supporting multiple scientific operations with real-time computation. It is built using HTML, CSS, JavaScript, and EJS templates rendered through Express.js.

Key Interface Features

- Clean gradient background and responsive layout.
- Dynamic input fields that adjust per selected operation.
- Clear output area displaying the computed result or error message.
- Smooth transitions and error validation.

9.2 Supported Operations

1. Square Root ($\sqrt{x}$)

- Input: $x \geq 0$
- Output: $\sqrt{x}$
- Example: $\sqrt{9} = 3$

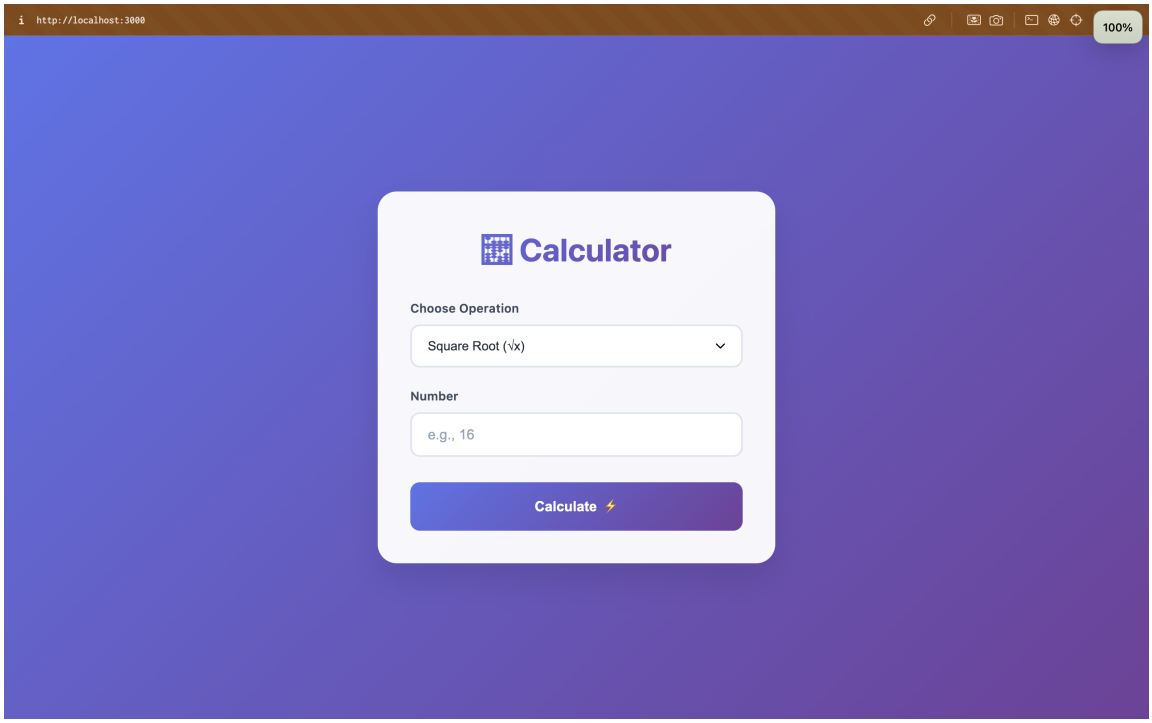

Figure 9.1: Main interface of the Scientific Calculator Web Application

2. Factorial ($x!$)

- Input: non-negative integer
- Output: product of all positive integers x
- Example: $5! = 120$

3. Natural Logarithm ($\ln x$)

- Input: $x > 0$
- Output: $\ln x$
- Example: $\ln e \approx 1$

4. Power Function (x^y)

- Input: base x and exponent y
- Output: x^y
- Example: $2^8 = 256$

9.3 Backend Verification

The backend API is accessible through two endpoints:

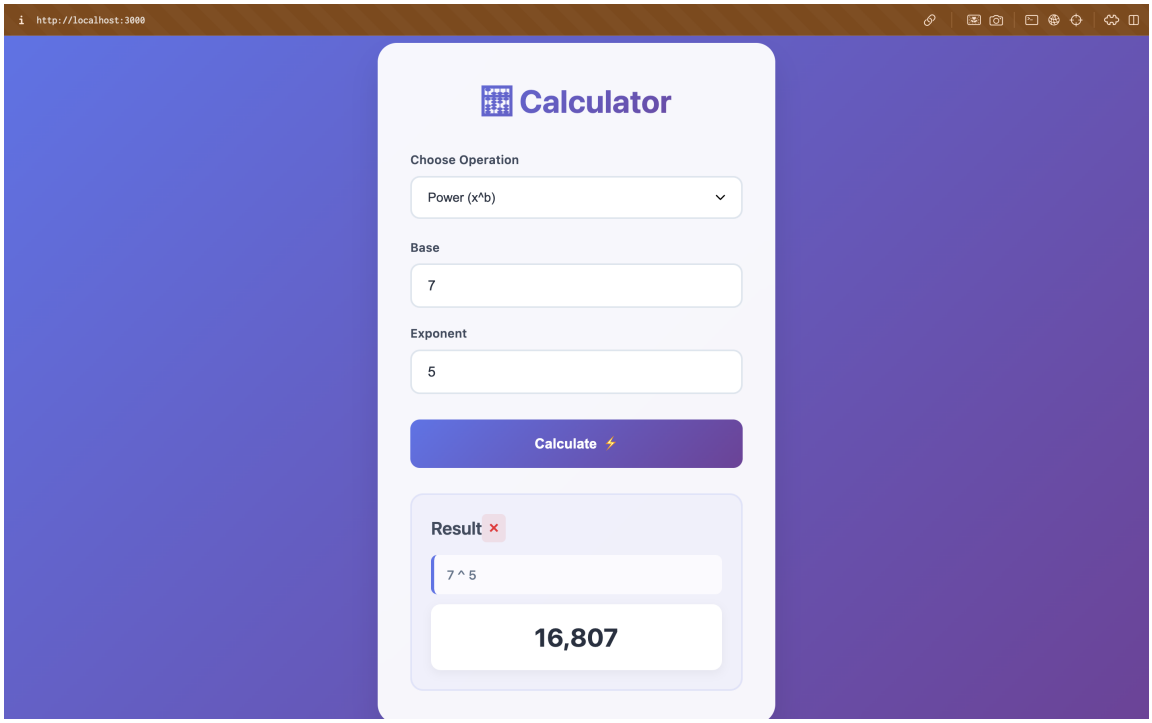

Figure 9.2: Example of power operation performed using the web interface

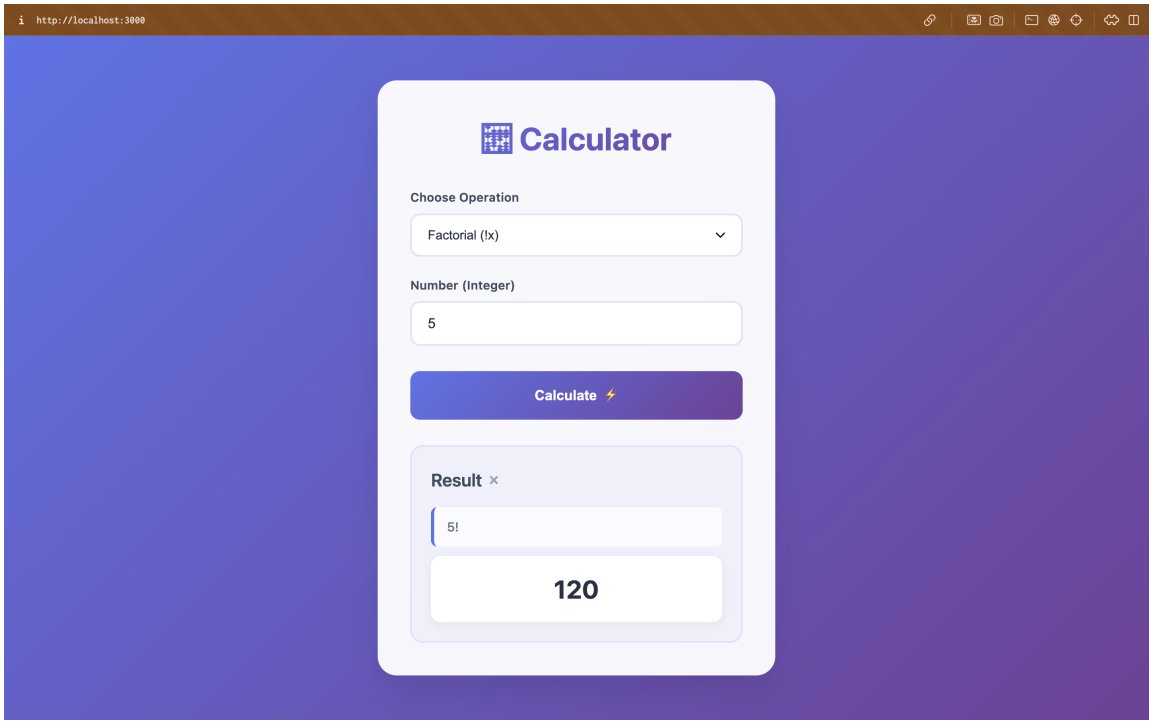

Figure 9.3: Factorial computation example displayed on the calculator

- **GET /calc** — performs operations via query parameters.
- **POST /calc** — accepts JSON payload for computations.

Example request:

```
GET /calc?op=pow&a=2&b=8
```

```
Response:
{
  "op": "pow",
  "a": 2,
  "b": 8,
  "result": 256
}
```

9.4 Deployment Confirmation

After the Jenkins pipeline completes, Ansible pulls the latest Docker image and starts the container automatically. Verification commands and outputs:

```
$ docker ps
CONTAINER ID   IMAGE                                  PORTS
abcd1234efgh   transyltoonia/calculator-app:latest   0.0.0.0:3000->3000/tcp
```

Accessing the app at `http://localhost:3000` confirms successful deployment.

9.5 Live Pipeline Verification

The Jenkins dashboard confirms that deployment and testing stages completed successfully with full automation. All pipeline stages (Checkout, Install, Test, Docker Build, Push, Deploy) show as completed in green.

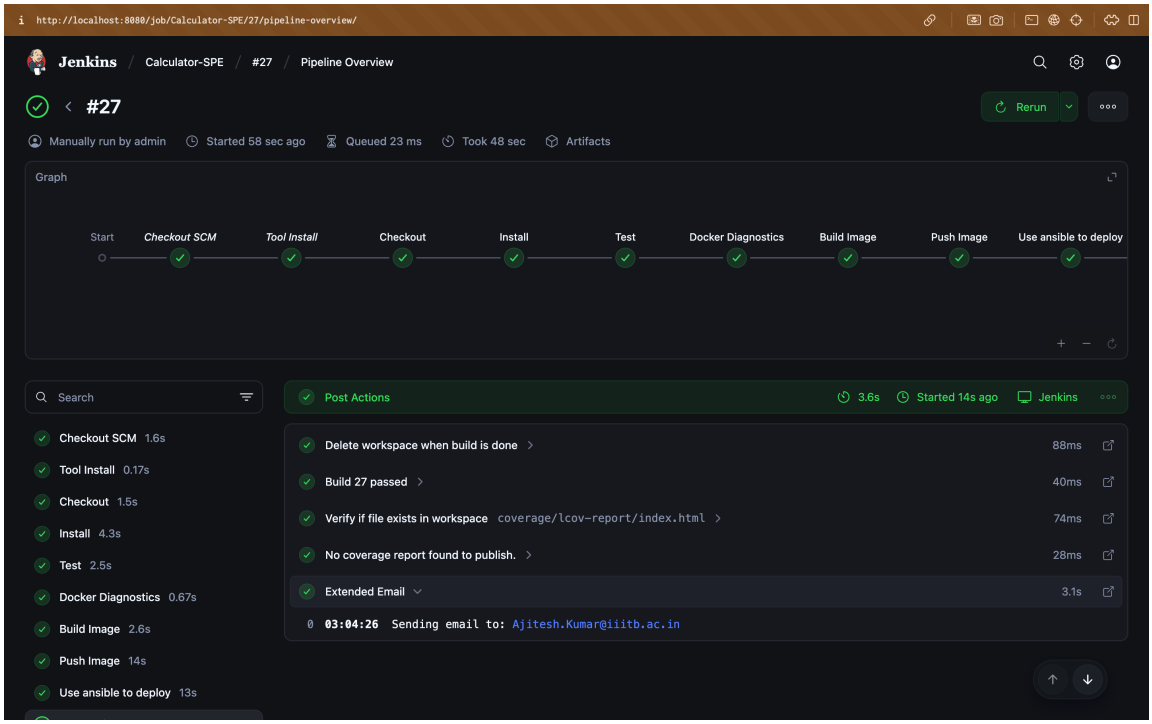

Figure 9.4: Jenkins pipeline view confirming automated deployment

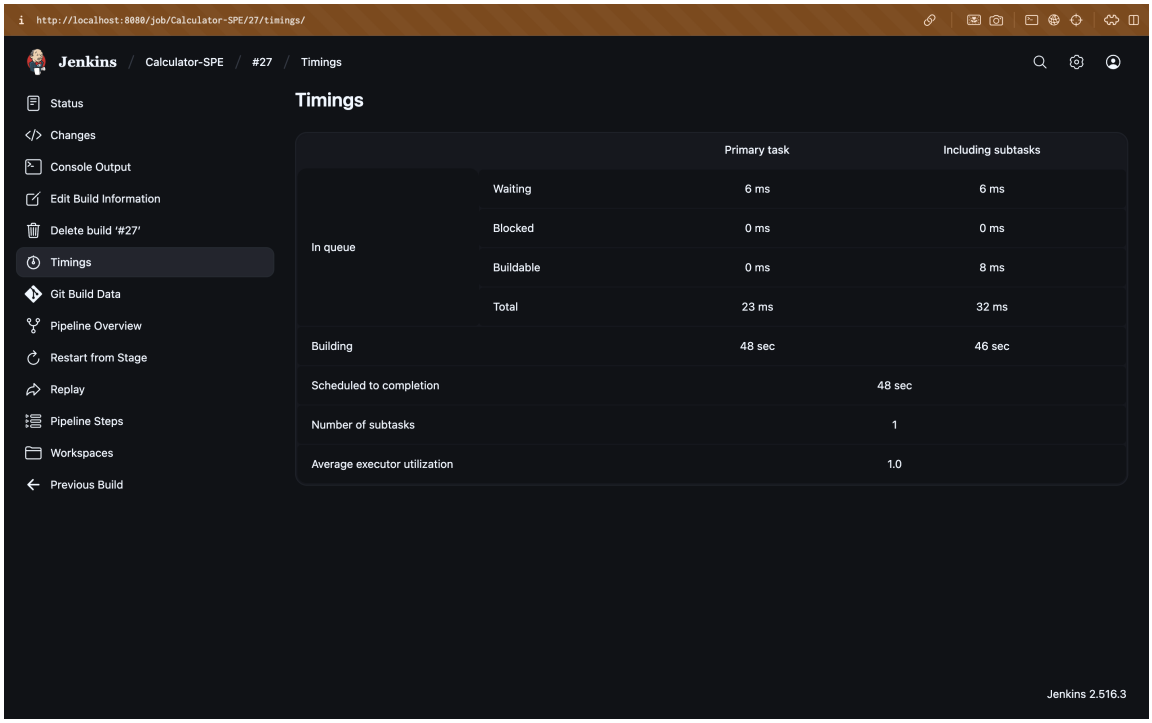

Figure 9.5: Detailed Jenkins stage timings during automated deployment

Summary

The application demonstration validates that:

- The Node.js web app builds and runs flawlessly within a Docker container.
- Jenkins automates the entire pipeline up to deployment.
- Ansible deploys the image locally and starts the container without manual steps.
- The final application is reachable and functions correctly, completing the full DevOps lifecycle.