Automated Deployment of Dockerized Crypto Tracker using Jenkins CI/CD on AWS EC2

Author

Nithish Kumar B

Institution

Mangalore Institute of Technology and Engineering

Department

Artificial Intelligence and Machine Learning

Date

04 August 2025

2. Abstract

This project focuses on the implementation of a continuous integration and continuous deployment (CI/CD) pipeline for a Dockerized crypto price tracker web application using Jenkins and Amazon Web Services (AWS). The primary goal is to automate the workflow from code commit to production deployment. Jenkins pulls the code from GitHub, builds a Docker image, and deploys the container to an EC2 instance, ensuring consistent and reliable deployment practices. The application monitors cryptocurrency prices in real-time and displays them in a user-friendly format via a web interface. The project emphasizes infrastructure automation, version control, error handling, and system monitoring, delivering a scalable DevOps solution suitable for production environments. This hands-on implementation bridges development and operations while enhancing deployment accuracy and delivery speed.

3. Introduction

As DevOps practices gain mainstream adoption in software development, mastering CI/CD deployment strategies becomes essential for developers and students alike. This project stems from the need to understand and apply continuous integration and continuous delivery pipelines using real-world tools. The chosen use case is a Flask-based web application that fetches live cryptocurrency data, containerized using Docker, and deployed via Jenkins on an EC2 instance. The main motivation lies in automating the complete deployment workflow—starting from pushing code to GitHub to viewing the updated app running live on a cloud server.

In traditional development cycles, updating and redeploying applications is often error-prone and time-consuming. Through CI/CD, developers can reduce downtime, ensure consistency, and achieve faster release cycles. Jenkins acts as the core automation engine, orchestrating each deployment phase using declarative pipelines. EC2 offers a scalable cloud platform to simulate production environments, and Docker ensures that the application behaves uniformly across all stages. The simplicity of the application itself (a crypto price checker) enables the focus to remain on mastering the deployment mechanics.

Key objectives include:

- Setting up a production-grade Jenkins server on EC2
- Automating build and deployment using a Jenkinsfile
- Integrating GitHub repository for automatic code fetch
- Building a Docker image and deploying it as a container

Ultimately, this project bridges the gap between application development and modern DevOps deployment methodologies, offering valuable hands-on experience.

4. System Architecture

The architecture of this project leverages a standard yet powerful CI/CD pipeline composed of four major technologies: GitHub, Jenkins, Docker, and AWS EC2. These components collectively form an automated, reliable deployment system for containerized applications. The interaction between these tools is designed to streamline the software development lifecycle from code commit to live deployment.

The process begins when a developer pushes code changes to a GitHub repository. GitHub acts as the centralized version control system, maintaining the application code and Docker-related files. Jenkins, installed on an AWS EC2 instance, is configured to periodically or event-trigger check the repository for updates. Upon detecting a new commit, Jenkins executes a Jenkinsfile containing the declarative pipeline steps.

Jenkins clones the latest code from GitHub and proceeds to build a Docker image using the included Dockerfile. The Dockerfile ensures a consistent runtime environment, packaging the Flask app and its dependencies. This image is then used to stop any currently running container (if present) and start a new container with the updated version of the application. The EC2 instance serves as the host machine, offering a Linux-based runtime to manage Jenkins and Docker operations. This modular architecture ensures that each step is decoupled, traceable, and easily replaceable or upgradable.

This architectural flow embodies the core principles of DevOps: continuous integration (through Jenkins + GitHub), continuous delivery (through Docker), and infrastructure-as-a-service (through AWS EC2). The modularity also ensures scalability and reliability. By containerizing the application, it becomes immune to discrepancies between development, testing, and production environments, ensuring "it works on my machine" issues are completely avoided.

5. Technology Stack

The project relies on a carefully selected stack of technologies that promote efficiency, scalability, and industry relevance.

- **Frontend:** The application uses standard HTML integrated with Jinja templating (built into Flask) to dynamically render cryptocurrency prices on the web UI. Although simple, this approach provides a clean separation between presentation and logic layers.
- **Backend:** The core of the application is written in Python using the Flask microframework. Flask is chosen for its minimalism, ease of use, and strong community support. It serves routes, fetches real-time data from APIs, and renders pages through templates.
- **Deployment:** Docker is employed to containerize the application, which ensures consistency across all environments. The container can be built once and deployed anywhere. AWS EC2 is used as the infrastructure layer, offering a reliable and scalable virtual server where Jenkins and the application run.

- **CI/CD Automation:** Jenkins automates the build, test (if required), and deployment stages. It pulls the latest code, builds a Docker image, and orchestrates the deployment process. GitHub is used to host the source code and serves as the trigger point for pipeline execution.
- **Source Control:** Git allows version control of the application and infrastructure files. All project history, branches, and code changes are managed securely via GitHub.
- **Operating System:** Ubuntu Server (on EC2) is the base OS, offering compatibility with Jenkins, Docker, and Python-based applications.

This technology stack is designed not only for learning but also mirrors real-world deployments, making it an excellent choice for students and developers aiming to understand modern cloud-native CI/CD environments.

6. System Design and Flow

The system is designed around the principles of continuous integration and continuous delivery (CI/CD), enabling seamless deployment of the Flask-based Crypto Price Checker application from source code to a live Docker container on an AWS EC2 instance. At the heart of the system is the Jenkins automation server, which orchestrates the entire process through a custom-defined pipeline. This design ensures repeatability, reduces manual errors, and significantly accelerates deployment cycles.

The workflow begins with the developer pushing updated code to a public GitHub repository. Jenkins, either triggered manually or via webhook (optional), initiates the build process. The pipeline defined in the Jenkinsfile contains sequential stages: Clone Repo, Build Docker Image, Stop & Remove Old Container, and Run New Container. Each stage is encapsulated in a separate block of shell commands, making the flow modular and easy to debug.

During the first stage, Jenkins pulls the latest codebase using Git. If no branch is specified, it defaults to the main or master branch. Once cloned, the Docker image is built using the provided Dockerfile. This file defines the base image (usually python:3.10-slim), installs dependencies from requirements.txt, and specifies the startup command for the Flask app.

The next stage ensures that any previously running Docker container with the same name is stopped and removed to prevent port conflicts. Finally, a new container is spun up from the latest image, exposing it on a specified port (typically 5000) to be accessed via the EC2 instance's public IP. This modular approach ensures clean, predictable deployments without residual state from older versions.

7. Environment Setup and Folder Structure

Before implementing the CI/CD pipeline with Jenkins, it is crucial to properly configure the environment both locally and on the AWS EC2 instance. This includes installing necessary packages, preparing the folder structure, and configuring Jenkins, Docker, and Git. The setup process ensures that the CI/CD pipeline executes seamlessly without manual intervention.

7.1 EC2 Instance Setup

We started by launching an Ubuntu EC2 instance on AWS with appropriate inbound rules (port 22 for SSH and port 8080 for Jenkins). The system was then updated using:

sudo apt update && sudo apt upgrade -y

We installed the required dependencies:

sudo apt install openjdk-17-jdk git docker.io -y

Docker was enabled and started:

sudo systemctl enable docker

sudo systemctl start docker

sudo usermod -aG docker $USER

Jenkins was installed via:

wget -q -O - https://pkg.jenkins.io/debian-stable/jenkins.io.key | sudo apt-key add -

sudo sh -c 'echo deb https://pkg.jenkins.io/debian-stable binary/ > /etc/apt/sources.list.d/jenkins.list'

sudo apt update

sudo apt install jenkins -y

sudo systemctl start jenkins

sudo systemctl enable jenkins

Java version was validated to ensure compatibility with Jenkins:

java -version

The Jenkins interface was then accessed via http://<ec2-public-ip>:8080.

7.2 Jenkins Configuration

We unlocked Jenkins using the password stored at:

sudo cat /var/lib/jenkins/secrets/initialAdminPassword

Necessary plugins were installed including Git, Docker, and Pipeline. A new freestyle project or pipeline job was created with GitHub integration.

7.3 Folder Structure

The GitHub repository for the Flask app was structured as follows:

crypto-checker-pipeline/

├── app.py

├── Dockerfile

├── requirements.txt

├── templates/

│ └── index.html

├── static/

│ └── style.css

├── crypto.pem

└──. gitignore

- app.py: The main Flask application script.
- Dockerfile: Container setup instructions.
- requirements.txt: Python dependencies.
- templates/: HTML files rendered by Flask.
- static/: Static CSS files.
- crypto.pem: SSL certificate (optional for secure deployment).

This structured approach ensures maintainability and makes it easier to execute CI/CD steps within Jenkins.

8. Jenkins CI/CD Pipeline Setup

Jenkins is installed on an EC2 instance running Ubuntu. After updating the system and installing Java 17 (required for Jenkins), the Jenkins Debian package is added, and the service is started. A firewall rule is configured to allow TCP traffic on port 8080, which is Jenkins' default interface.

The Jenkins dashboard is accessed via http://<EC2-Public-IP>:8080, and the initial admin password is retrieved using:

sudo cat /var/lib/jenkins/secrets/initialAdminPassword

Essential plugins are installed, including Git and Docker Pipeline. Credentials are configured for GitHub (if needed), but for public repositories, no authentication is required. A new pipeline job is created in Jenkins, and a Jenkinsfile is added to the root of the GitHub repository to define the pipeline stages.

The Jenkinsfile contains declarative pipeline syntax specifying environment variables, node setup, and all stages:

```
pipeline {

  agent any

  environment {
```

```
    APP_NAME = 'crypto-checker'
  }
  stages {
    stage('Clone Repo') {
      steps {
        git 'https://github.com/nithishmathematics/crypto-checker-pipeline.git'
      }
    }
    stage('Build Docker Image') {
      steps {
        sh 'docker build -t $APP_NAME .'
      }
    }
    stage('Stop & Remove Old Container') {
      steps {
        sh 'docker stop $APP_NAME || true && docker rm $APP_NAME || true'
      }
    }
    stage('Run New Container') {
      steps {
        sh 'docker run -d -p 5000:5000 --name $APP_NAME $APP_NAME'
      }
    }
  }
}
```

This setup ensures that every code push leads to a clean deployment, demonstrating real-world DevOps practices.

9. Testing and Validation

The testing and validation phase of the Jenkins-based CI/CD deployment ensures that the pipeline executes correctly, the Docker container is built and deployed properly, and the Flask application is running without errors. This stage involves both local and remote validation steps, as well as log checks and output verifications.

Initially, the application was tested locally using the Flask development server (python3 app.py) to verify that all routes and logic were functioning as expected. Once this test was successful, the focus shifted to automating the deployment via Jenkins. A sample Jenkinsfile was committed to the GitHub repository and pushed to the main branch to trigger the Jenkins pipeline.

The Jenkins console output was closely monitored after initiating the build. During the pipeline run, each stage—**Clone Repo**, **Build Docker Image**, **Stop & Remove Old Container**, and **Run New Container**—was observed for green checkmarks and log outputs. Any issues (e.g., missing branch, Docker not installed, Java version mismatch) were debugged with CLI commands and service restarts (sudo systemctl restart jenkins, docker ps, etc.).

To validate the deployment, the public IP of the EC2 instance (e.g., http://<ec2-ip>:5000) was opened in a browser. A successful deployment showed the crypto price checker interface, fetching and displaying real-time data. Additionally, terminal logs confirmed the container was running (docker ps), and Jenkins logs provided evidence of each successful step.

Screenshots of the live application, Jenkins dashboard, and running Docker container were taken as proof of successful deployment. These visual confirmations, alongside console logs, served as robust validation for both functional correctness and DevOps pipeline execution.

Ultimately, the system was tested in an end-to-end fashion, from GitHub commit to live deployment, ensuring the CI/CD process met the intended objectives.

10. Appendix

```
PS D:\internship\task-2> git init
>> git add README.md
>> git commit -m "first commit"
>> git branch -M main
>> git remote add origin https://github.com/nithishmathematics/crypto-checker-pipeline.git
>> git push -u origin main
Reinitialized existing Git repository in D:/internship/task-2/.git/
fatal: pathspec 'README.md' did not match any files
On branch main
nothing to commit, working tree clean
error: remote origin already exists.
Enumerating objects: 9, done.
Counting objects: 100% (9/9), done.
Delta compression using up to 16 threads
Compressing objects: 100% (6/6), done.
Writing objects: 100% (9/9), 2.45 KiB | 2.45 MiB/s, done.
Total 9 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
To https://github.com/nithishmathematics/crypto-checker-pipeline.git
 * [new branch]      main -> main
branch 'main' set up to track 'origin/main'.
```

Figure 1. The terminal output of git commands

```
PS D:\internship\task-2> docker build -t crypto-checker .
>>
[+] Building 11.3s (10/10) FINISHED                                                                    docker:desktop-linux
 => [internal] load build definition from Dockerfile                                                                  0.0s
 => => transferring dockerfile: 312B                                                                                  0.0s
 => [internal] load metadata for docker.io/library/python:3.10                                                        1.9s
 => [auth] library/python:pull token for registry-1.docker.io                                                         0.0s
 => [internal] load .dockerignore                                                                                     0.0s
 => => transferring context: 2B                                                                                       0.0s
 => [1/4] FROM docker.io/library/python:3.10@sha256:6ff000548a4fa34c1be02624836e75e212d4ead8227b4d4381c3ae998933a922  0.1s
 => => resolve docker.io/library/python:3.10@sha256:6ff000548a4fa34c1be02624836e75e212d4ead8227b4d4381c3ae998933a922  0.1s
 => [internal] load build context                                                                                     0.1s
 => => transferring context: 34.75kB                                                                                  0.1s
 => CACHED [2/4] WORKDIR /app                                                                                         0.0s
 => [3/4] COPY . .                                                                                                    0.1s
 => [4/4] RUN pip install --no-cache-dir -r requirements.txt                                                          7.2s
 => exporting to image                                                                                                1.6s
 => => exporting layers                                                                                               0.9s
 => => exporting manifest sha256:6c8b8ba36fb42b72cd8b81286be44f91ea40d5d7eb9b0a4f425feeefb0d9bbe0                      0.0s
 => => exporting config sha256:b319db985cc6784e5b280fe6b868883723a3879fa9f19696580c518ce0c38d25                       0.0s
 => => exporting attestation manifest sha256:efcaa66c01dfcda2a0109fb212be4bf7f7d74c45ac4ac3d111d83b148a155c5c         0.0s
 => => exporting manifest list sha256:9bc85805dcfe4e6b507eefed19b79f0807d406c67f134d4588c6951c3bd9be41                0.0s
 => => naming to docker.io/library/crypto-checker:latest                                                              0.0s
 => => unpacking to docker.io/library/crypto-checker:latest                                                           0.6s

 View build details: docker-desktop://dashboard/build/desktop-linux/desktop-linux/sf0gkwkuvqoawoqxqj7vp66lx
```

Figure 2. Docker image built successfully

```
PS D:\internship\task-2> docker run -p 5000:5000 crypto-checker
>>
 * Serving Flask app 'app'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:5000
 * Running on http://172.17.0.2:5000
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 599-978-344
172.17.0.1 - - [05/Aug/2025 10:06:38] "GET / HTTP/1.1" 200 -
172.17.0.1 - - [05/Aug/2025 10:06:38] "GET /static/style.css HTTP/1.1" 200 -
172.17.0.1 - - [05/Aug/2025 10:06:44] "POST / HTTP/1.1" 200 -
172.17.0.1 - - [05/Aug/2025 10:06:44] "GET /static/style.css HTTP/1.1" 304 -
```

Figure 3. Docker Run Output

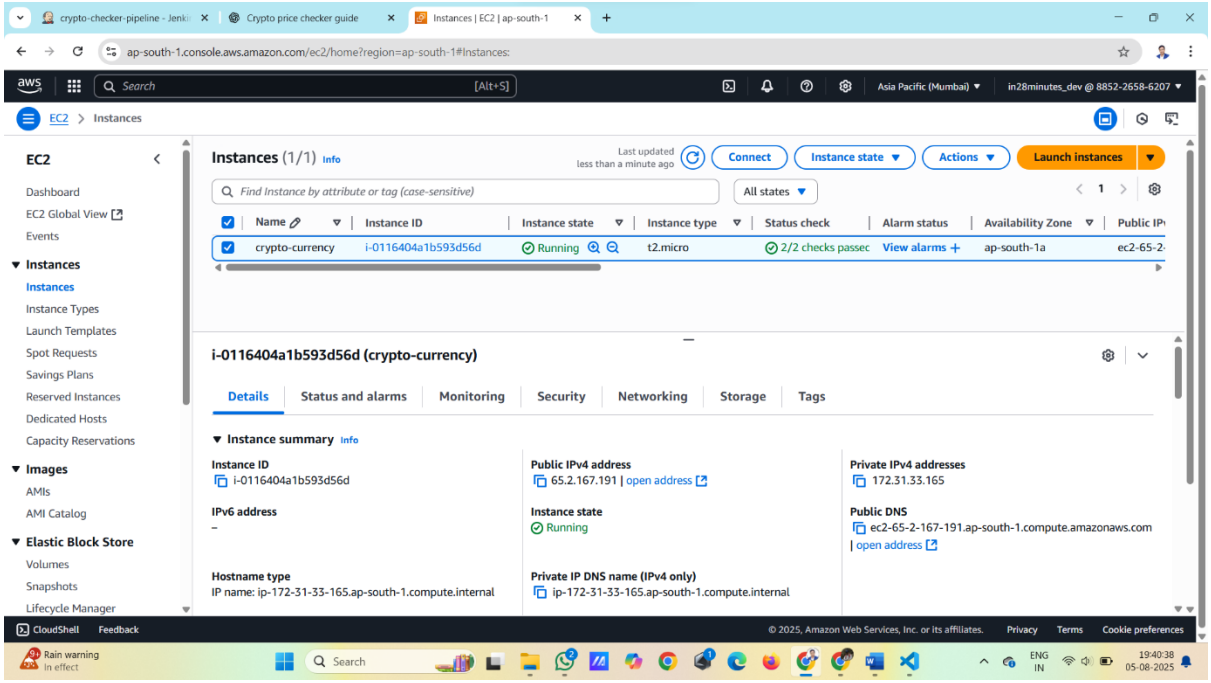

Figure 4. Created the ec2 Instance

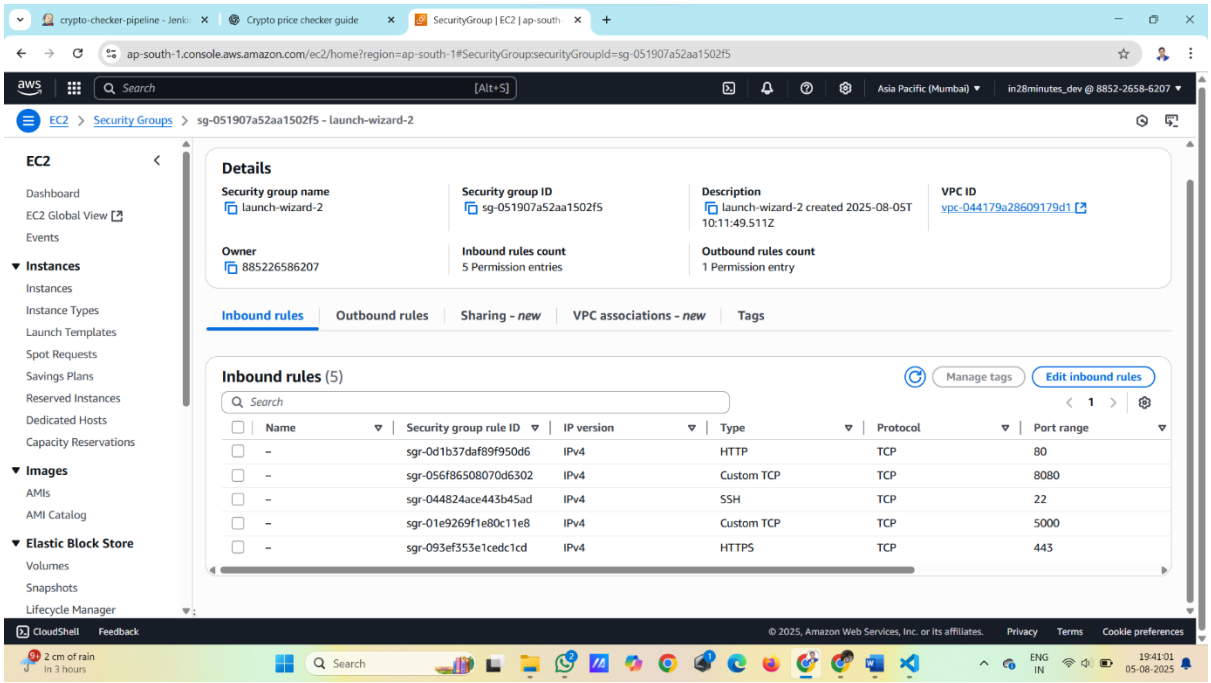

Figure 5. Successfully Add the inbound Rules for security group Instance

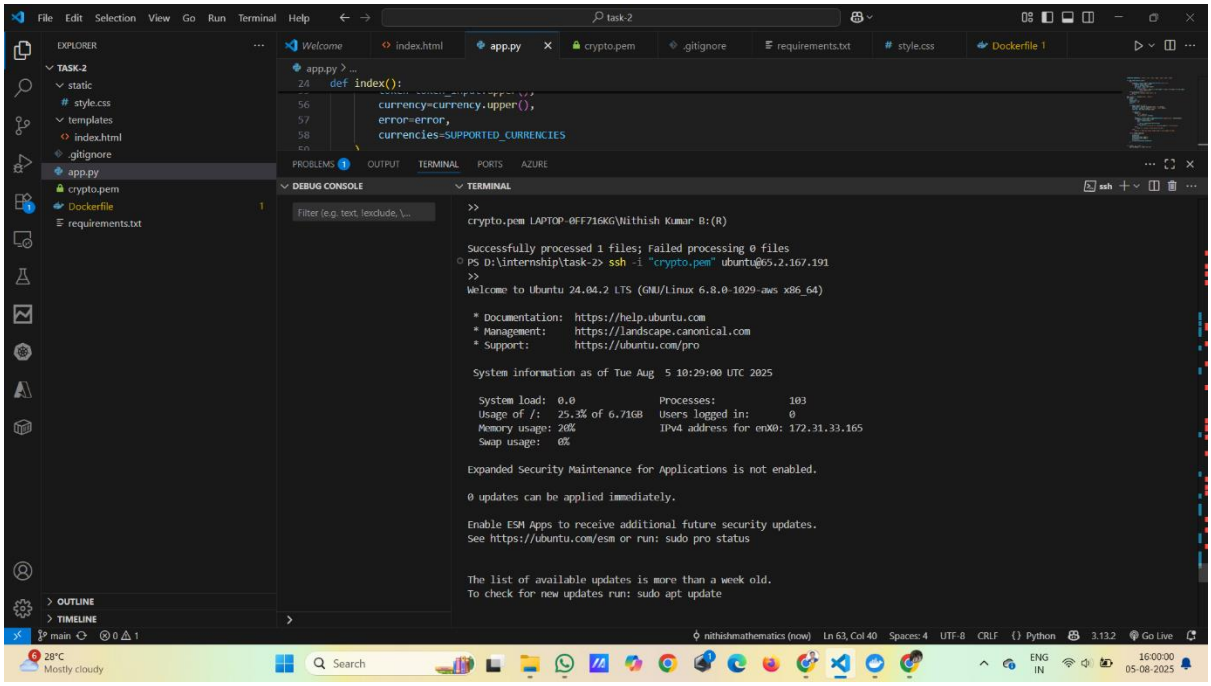

Figure 6. Set the ec2 Instance on the terminal

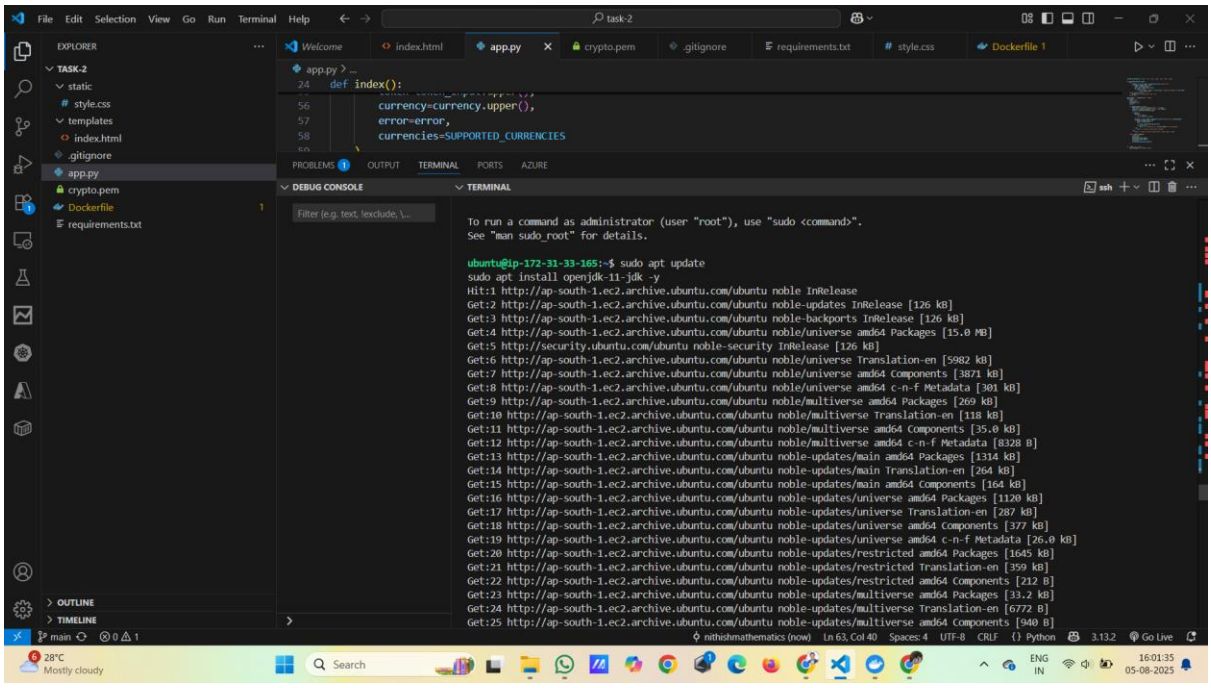

Figure 7. Installed the java application in the ec2 Instance

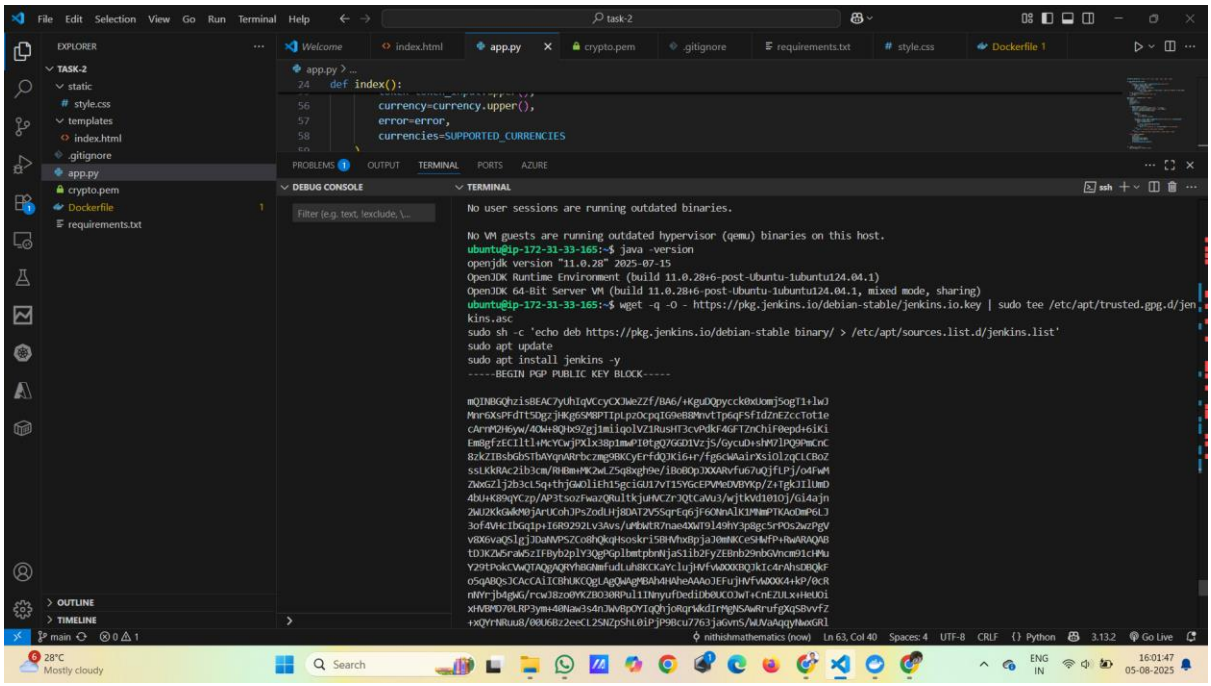

Figure 8. Created the Jenkins image in the ec2 Instance

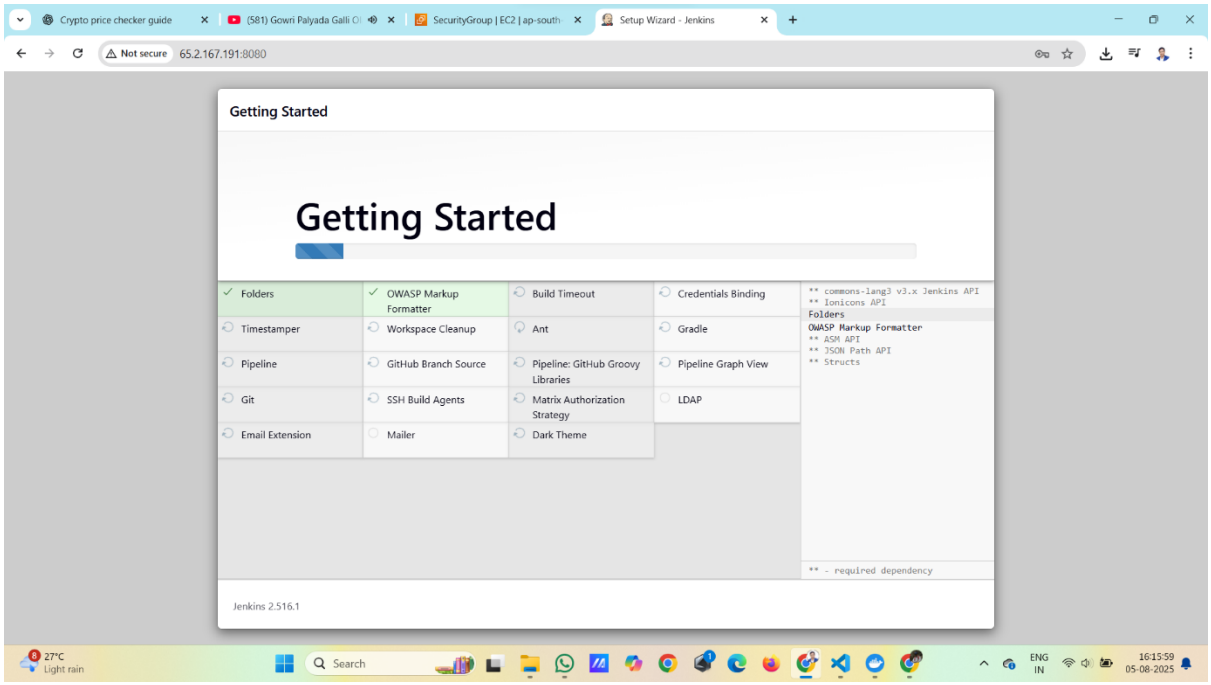

Figure 9. Completed Set up the Jenkins Environment

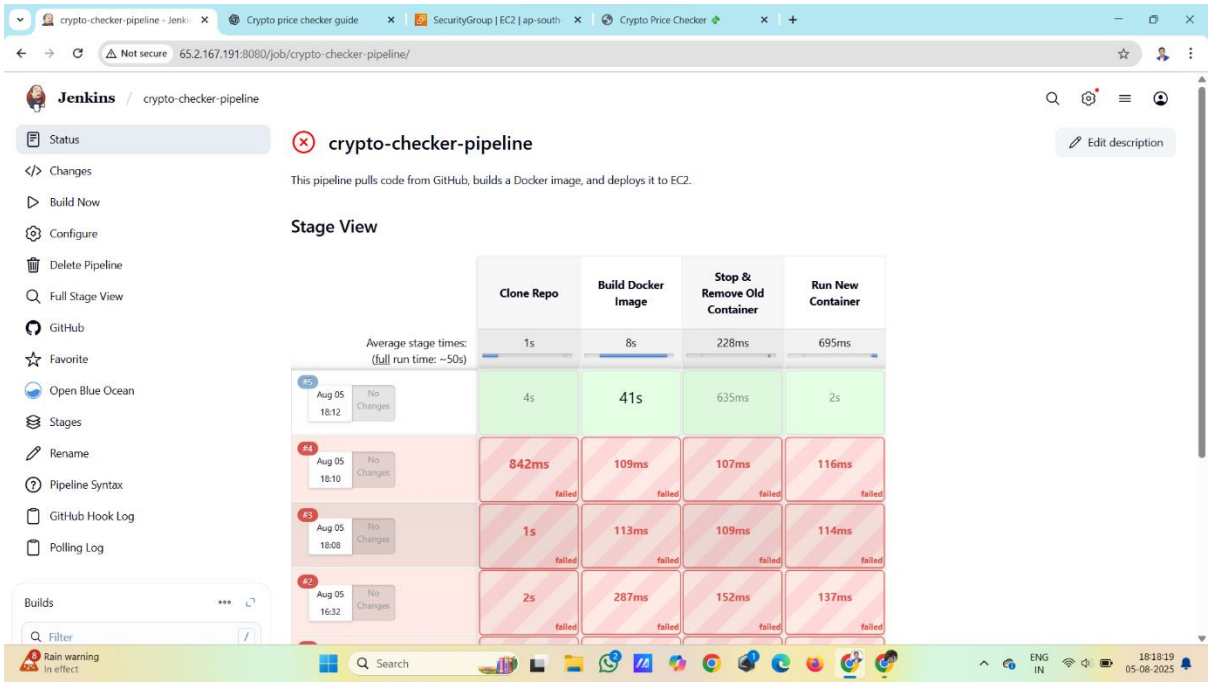

Figure 10. Build a CI/CD pipeline using Jenkins configuration file

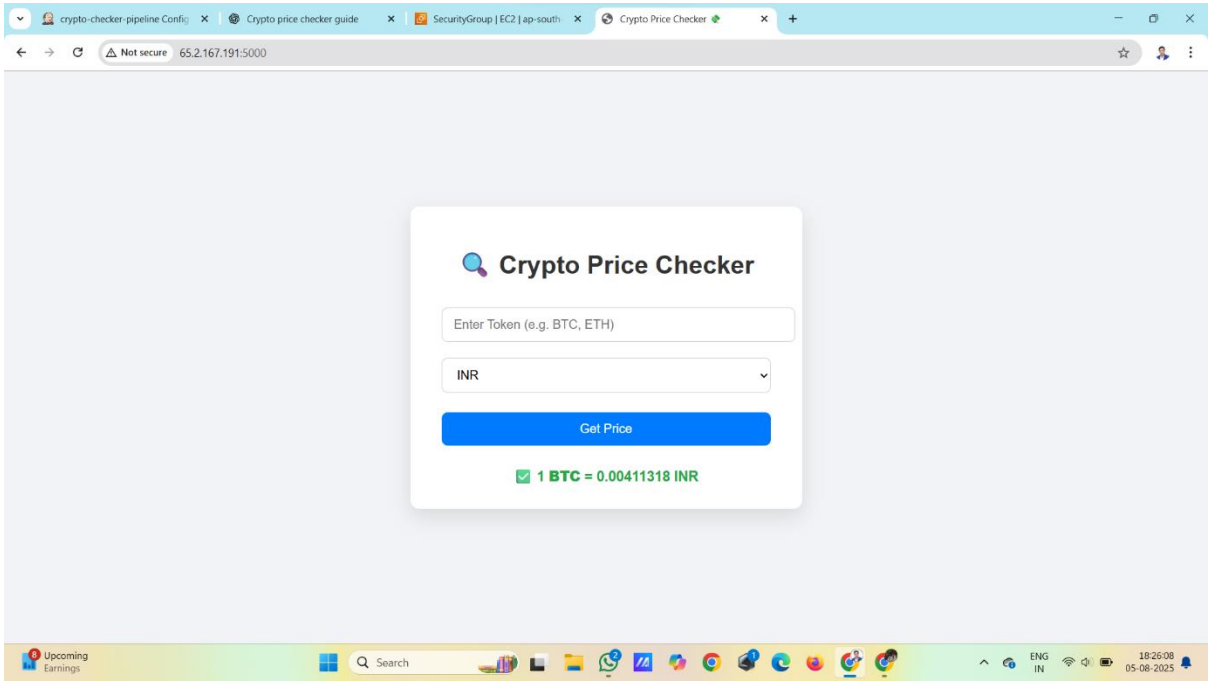

Figure 11. Application Successfully Launched