CI/CD Pipeline for Python-Based Rails Legal Application

Course Name: DevOps

Medicaps University – Datagami Skill Based Course

Student Name(s) & Enrolment Number(s):

Sr no	Student Name	Enrolment Number
1.	VISHAL RATHOD	EN22CS306059
2.	PALAK GUPTA	EN22CS306036
3.	JAY PRAKASH SINGH	EN22CS304031
4.	RAJVEER MUKATI	EN22CS304051
5.	KOMAL JADHAV	EN22CS303028
6.	DIYA GOYAL	EN22CS304025

Group Name: G11D11

Project Number: DO-37

Industry Mentor Name: Mr. Vaibhav Sir

University Mentor Name: Prof. Shyam Patel

Academic Year: 2025-26

1. Problem Statement & Objectives

1.1 Problem Statement

In traditional software development, application testing and deployment are often performed manually. This process is time-consuming, error-prone, and leads to inconsistent environments and delayed releases. Organizations require a reliable system that ensures automatic testing, fast deployment, and high-quality code delivery.

To solve this problem, an automated Continuous Integration and Continuous Deployment (CI/CD) pipeline is required for a Python-based Legal Application. The system should automatically test, build, and deploy the application to a cloud environment without manual intervention.

1.2 Project Objectives

The objective of this project is to develop an automated CI/CD pipeline for a Python-based Legal Application that ensures continuous testing, Docker-based containerization, secure image storage in Amazon ECR, and automatic deployment to Amazon EC2.

The project aims to improve code quality, enhance deployment speed, ensure security through IAM roles and GitHub Secrets, and implement enterprise-level DevOps practices for reliable and production-ready software delivery.

The main objectives of this project are:

- To implement automated unit testing on every code commit
- To build and containerize the application using Docker
- To store the Docker image securely in Amazon ECR
- To automatically deploy the application to Amazon EC2
- To ensure high code quality and reliability
- To enable faster and secure deployment
- To implement enterprise-level DevOps practices

1.3 Scope of the Project

The scope of this project covers the complete development and deployment lifecycle of a Python-based Legal Application using modern DevOps practices.

The project includes the development of a Legal Application using FastAPI framework in Python, along with the creation of a /health endpoint for monitoring purposes. Unit testing is implemented using Pytest to ensure application reliability and early bug detection.

The scope of the project includes:

- Development of a Python FastAPI-based Legal Application
- Writing unit tests using Pytest
- Version control using Git and GitHub
- CI/CD automation using GitHub Actions
- Containerization using Docker
- Cloud deployment using AWS (ECR and EC2)
- Secure access using IAM roles and GitHub Secrets
- Automated deployment without manual production steps

Version control is managed using Git and GitHub, enabling proper code tracking, collaboration, and workflow management. Continuous Integration and Continuous Deployment (CI/CD) automation is configured using GitHub Actions, which automatically triggers testing, building, and deployment processes whenever code is pushed to the repository.

The application is containerized using Docker to ensure environment consistency, portability, and dependency isolation. The Docker images are securely stored in Amazon Elastic Container Registry (ECR), acting as a centralized artifact repository.

For cloud deployment, Amazon EC2 is used as the execution environment where the Docker container runs and hosts the application. Secure access is maintained through IAM role-based permissions and GitHub Secrets to prevent exposure of sensitive credentials.

The deployment process is fully automated, eliminating manual production steps and ensuring faster, reliable, and production-ready software delivery.

2. Proposed Solution

2.1 Key features

- **Automated Testing using Pytest :-** Unit tests are automatically executed during the CI process to verify application functionality and ensure code quality before deployment.

- **Continuous Integration using GitHub Actions :-** GitHub Actions automatically triggers the workflow on every code push, performing code validation, dependency installation, and test execution.

- **Docker-based Containerization :-** The application is packaged into a Docker container to ensure a consistent runtime environment, easy deployment, and platform independence.

- **Automatic Docker Image Build and Versioning :-** After successful testing, the CI pipeline automatically builds the Docker image and tags it properly for version control and easy management.

- **Secure Image Storage in Amazon ECR :-** The built Docker images are securely stored in Amazon Elastic Container Registry, allowing reliable and centralized access for deployment.

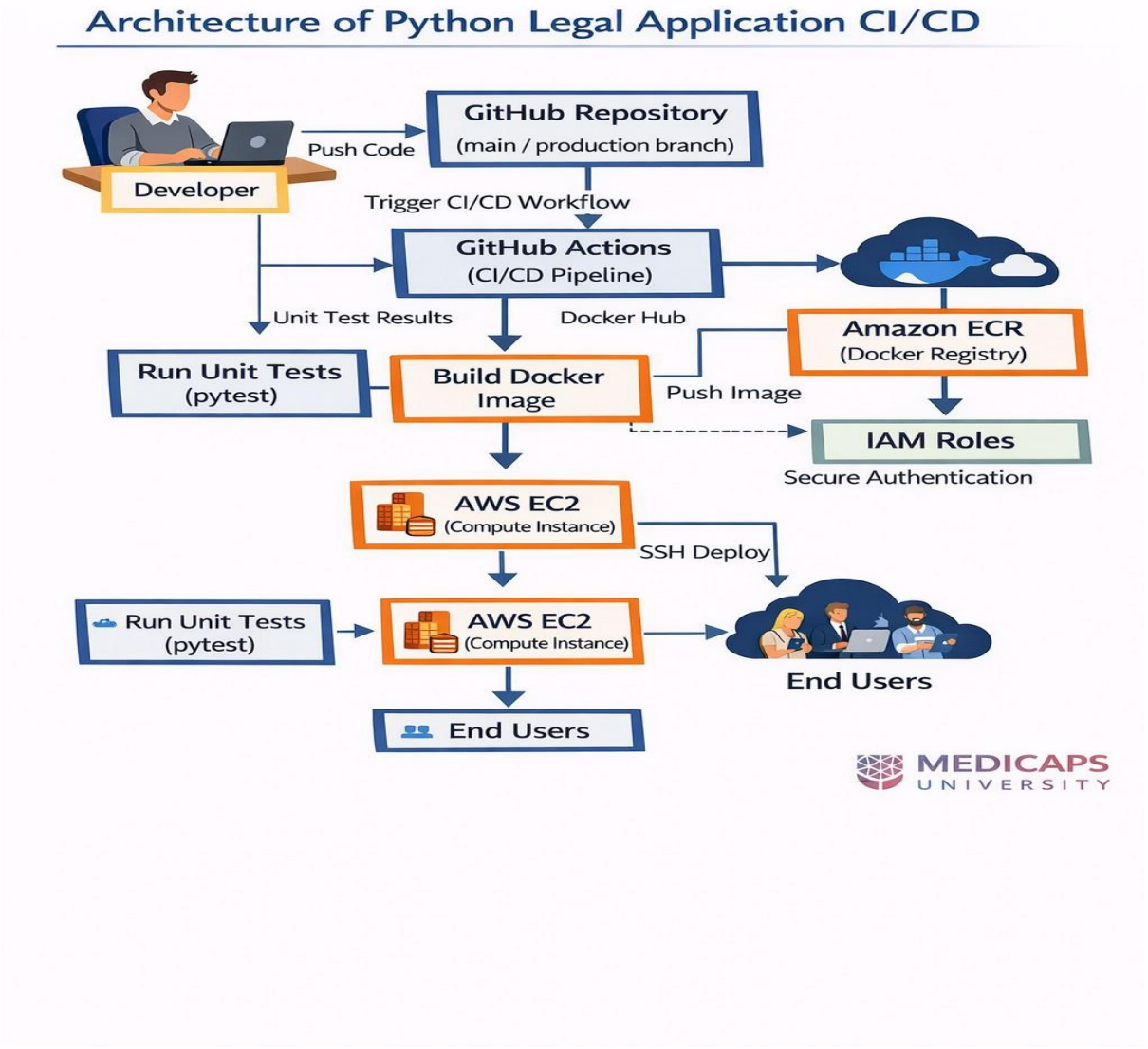
- **Automated Deployment to Amazon EC2 :-** The pipeline automatically connects to the EC2 instance via SSH, pulls the latest image from ECR, and runs the updated container without manual intervention.

- **Health Monitoring Endpoint (/health) :-** A health check API endpoint is implemented to verify that the application is running successfully after deployment.

- **Secure Credentials Management using GitHub Secrets :-** Sensitive information such as server details and keys is securely stored in GitHub Secrets to prevent exposure in the code repository.

- **IAM Role-Based Secure Access :-** IAM roles are used to grant secure and controlled access between AWS services, avoiding the use of hardcoded AWS credentials.

2.2 Overall Architecture / Workflow



- **Flow in One Line**

Developer → GitHub → GitHub Actions → Test → Build Docker → Push to ECR → Deploy to EC2 → Application Live

Overall Workflow

The project follows an automated step-by-step process:

- **Developer writes or updates the code :-** The developer makes changes in the FastAPI application.
- **Code is pushed to GitHub :-** The updated code is uploaded to the GitHub repository using Git.
- **GitHub Actions starts automatically :-** After the push, GitHub Actions triggers the CI/CD pipeline.
- **Run Unit Tests (CI Phase) :-** Dependencies are installed , Pytest runs to check code quality , If tests fail → process stops
- **Build Docker Image :-** If tests pass, a Docker image of the application is created.
- **Push Image to Amazon ECR :-** The Docker image is stored securely in Amazon ECR.
- **Deploy to EC2 (CD Phase) :-** GitHub connects to EC2 using SSH , Latest image is pulled from ECR , Old container is stopped and removed , New container is started
- **Application Goes Live :-** The application becomes accessible using:http://EC2_Public_IP:8000

2.3 Tools & Technologies Used

Category	Tool/Technology
Programming	Python (FastAPI)
Testing	Pytest
Version Control	Git
Repository	GitHub
CI/CD	GitHub Actions
Containerization	Docker
Cloud Platform	AWS
Container Registry	Amazon ECR
Compute Service	Amazon EC2
Security	IAM Roles, GitHub Secrets
Operating System	Ubuntu Linux
Infrastructure (Optional)	Terraform

- **Python (FastAPI)** – Used to develop the Legal Application and create API endpoints.
- **Pytest** – Used for writing and running automated unit tests to ensure code quality.
- **Git** – Used for version control to track code changes.
- **GitHub** – Used as a remote repository to store and manage the project code.
- **GitHub Actions** – Used to automate the CI/CD pipeline, including testing, building, and deployment.
- **Docker** – Used to containerize the application for consistent and portable deployment.
- **Amazon Web Services (AWS)** – Cloud platform used for hosting and deployment.
- **Amazon ECR** – Used to securely store Docker images.
- **Amazon EC2** – Used as a virtual server to run the application in production.
- **IAM Roles & GitHub Secrets** – Used to manage secure access and protect sensitive credentials.
- **Ubuntu Linux** – Operating system used on the EC2 server.
- **Terraform (Optional)** – Used for infrastructure automation (if implemented).

3. Results & Output

3.1 Screenshots / outputs

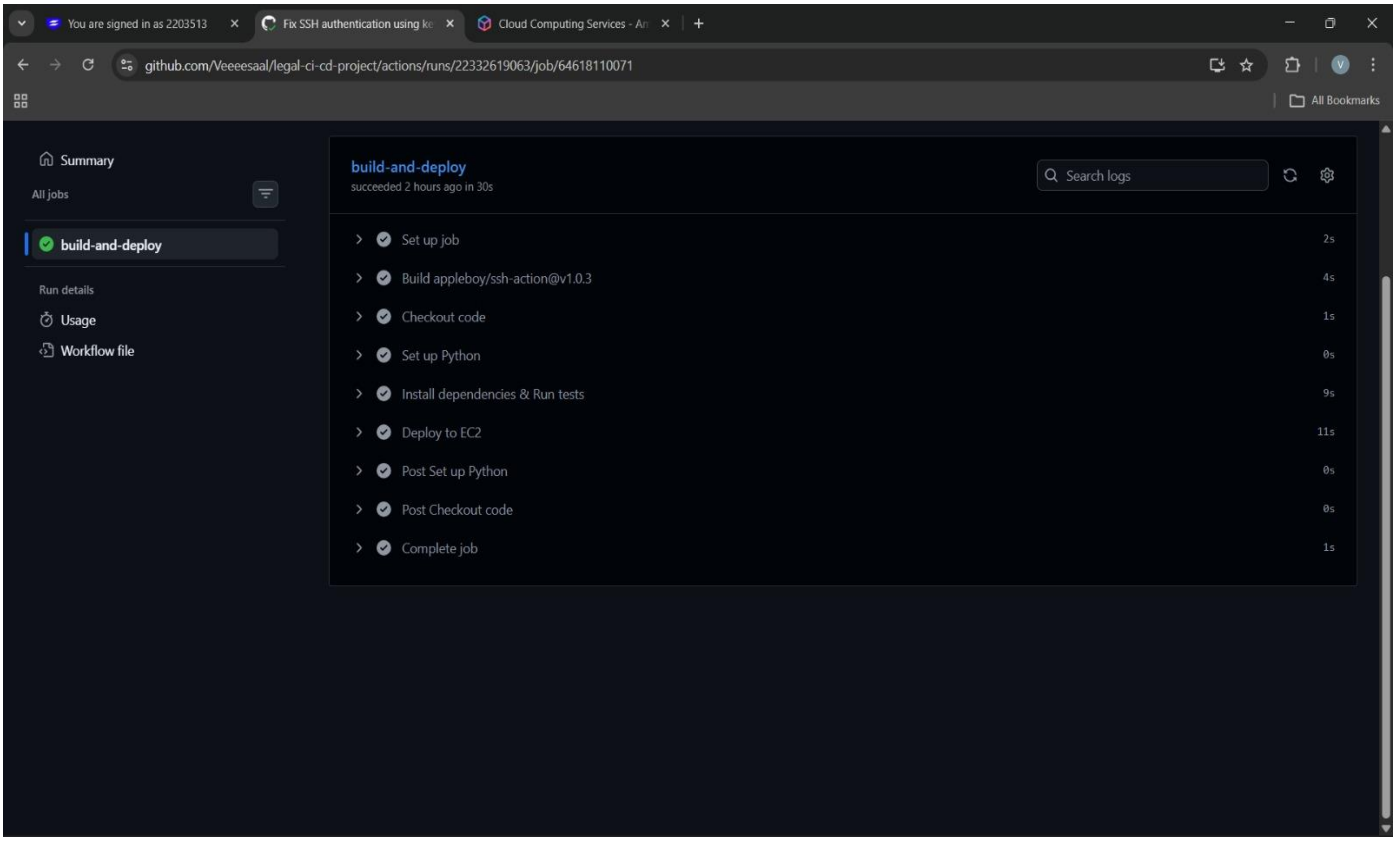


fig :- 1

- **GitHub Actions CI/CD Execution Output**

The above screenshot shows the successful execution of the CI/CD pipeline in GitHub Actions.

The workflow named **"build-and-deploy"** completed successfully without any errors. All stages of the pipeline were executed properly, including:

- Setting up the job environment
- Checking out the source code
- Setting up Python
- Installing dependencies
- Running unit tests
- Deploying the application to EC2
- Completing the job

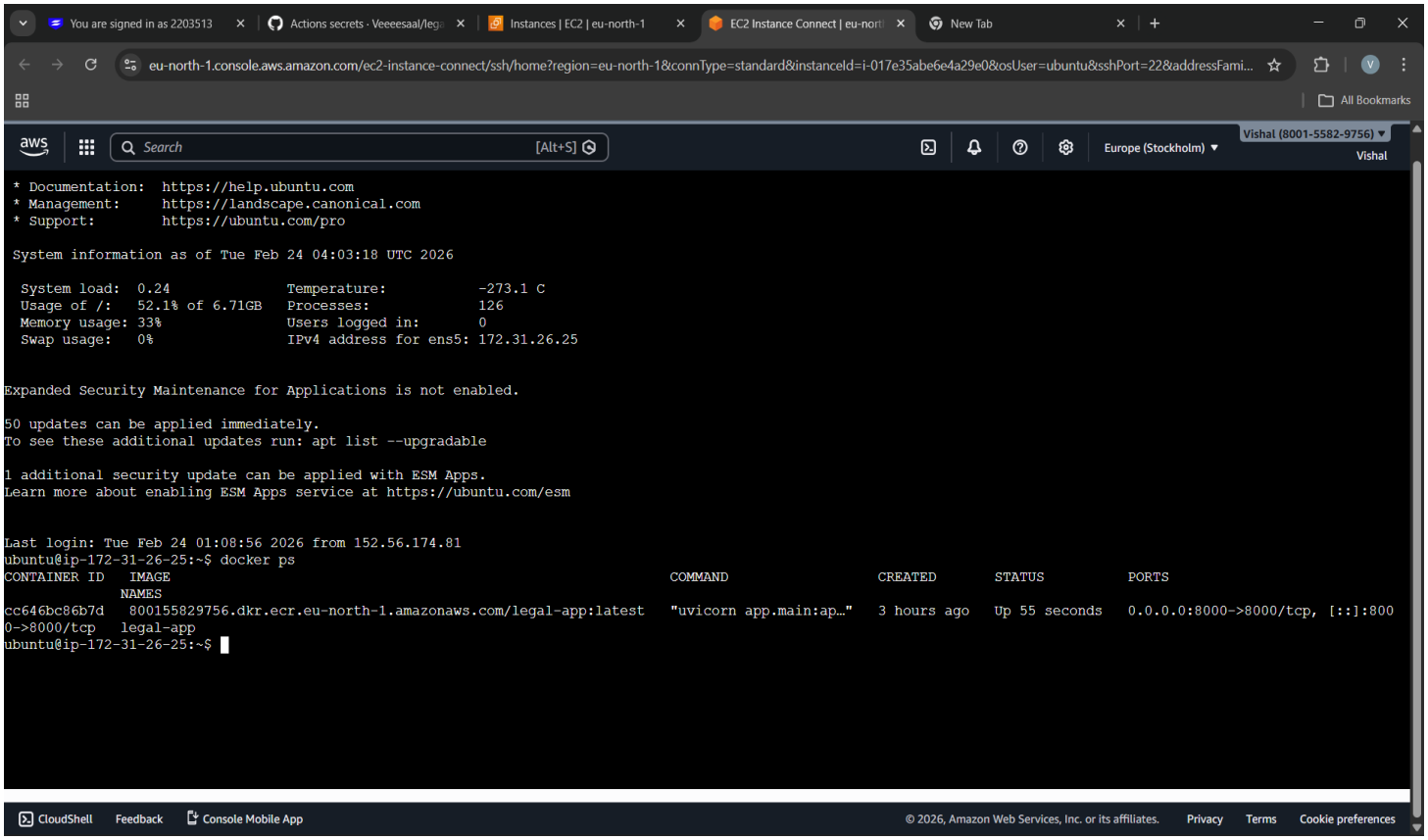


Fig :- 2

- **EC2 Deployment and Docker Container Status**

The above screenshot shows the Amazon EC2 instance terminal where the application is running successfully inside a Docker container.

The command docker ps is executed to verify the running containers. The output confirms that the container named legal-app is active and running using the Docker image stored in Amazon ECR.

Key observations from the output:

- The container status shows Up, which means the application is running successfully.
- The image is pulled from Amazon ECR.
- Port mapping 8000:8000 is configured, allowing external access to the application.
- The application is executed using the Uvicorn server.

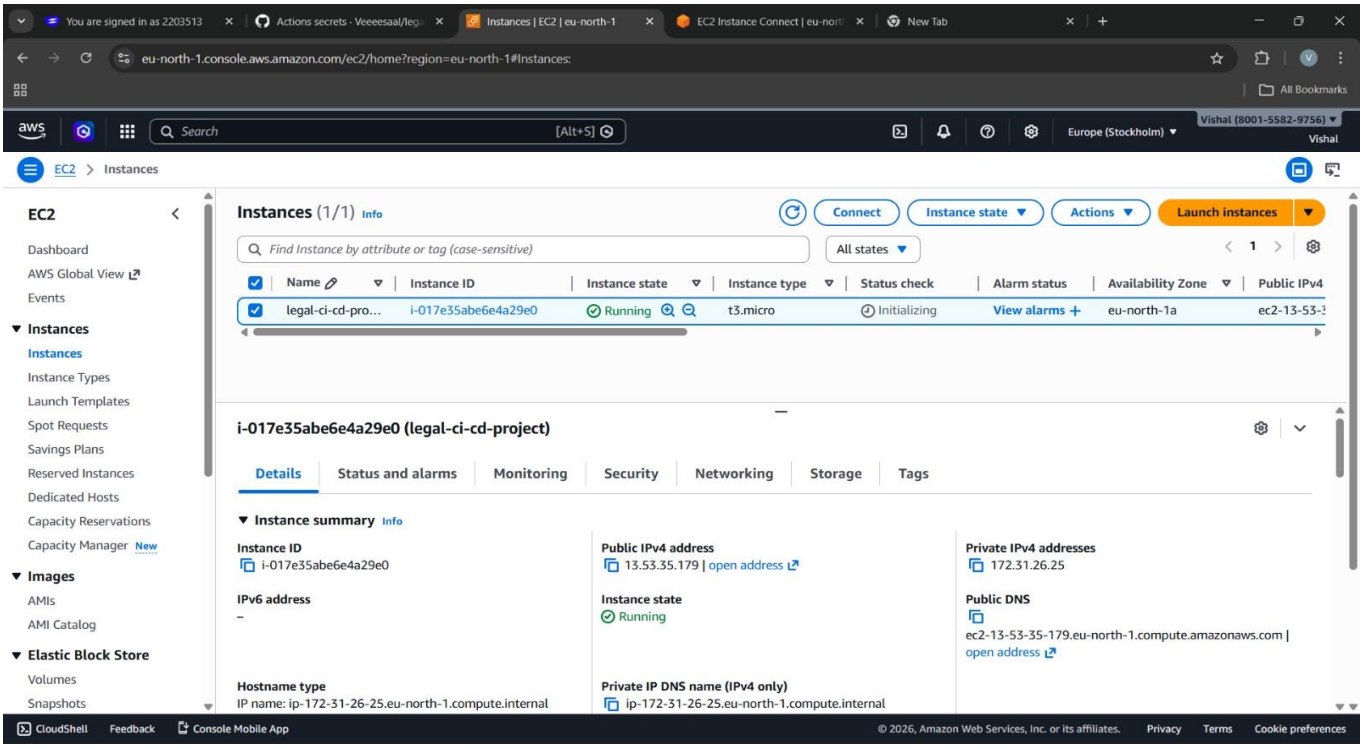


Fig :- 3

- **Amazon EC2 Instance Status**

The above screenshot shows the AWS EC2 dashboard displaying the instance details used for application deployment.

The instance named legal-ci-cd-project is currently in the Running state, which confirms that the server is active and available for hosting the application.

Key details from the dashboard:

- Instance Type: t3.micro
- Instance State: Running
- Public IPv4 Address is assigned for external access
- Public DNS is available to access the server from the internet
- Availability Zone: eu-north-1a

This confirms that the cloud infrastructure required for deployment is successfully created and operational. The EC2 instance acts as the production server where the Docker container runs and hosts the Legal Application.

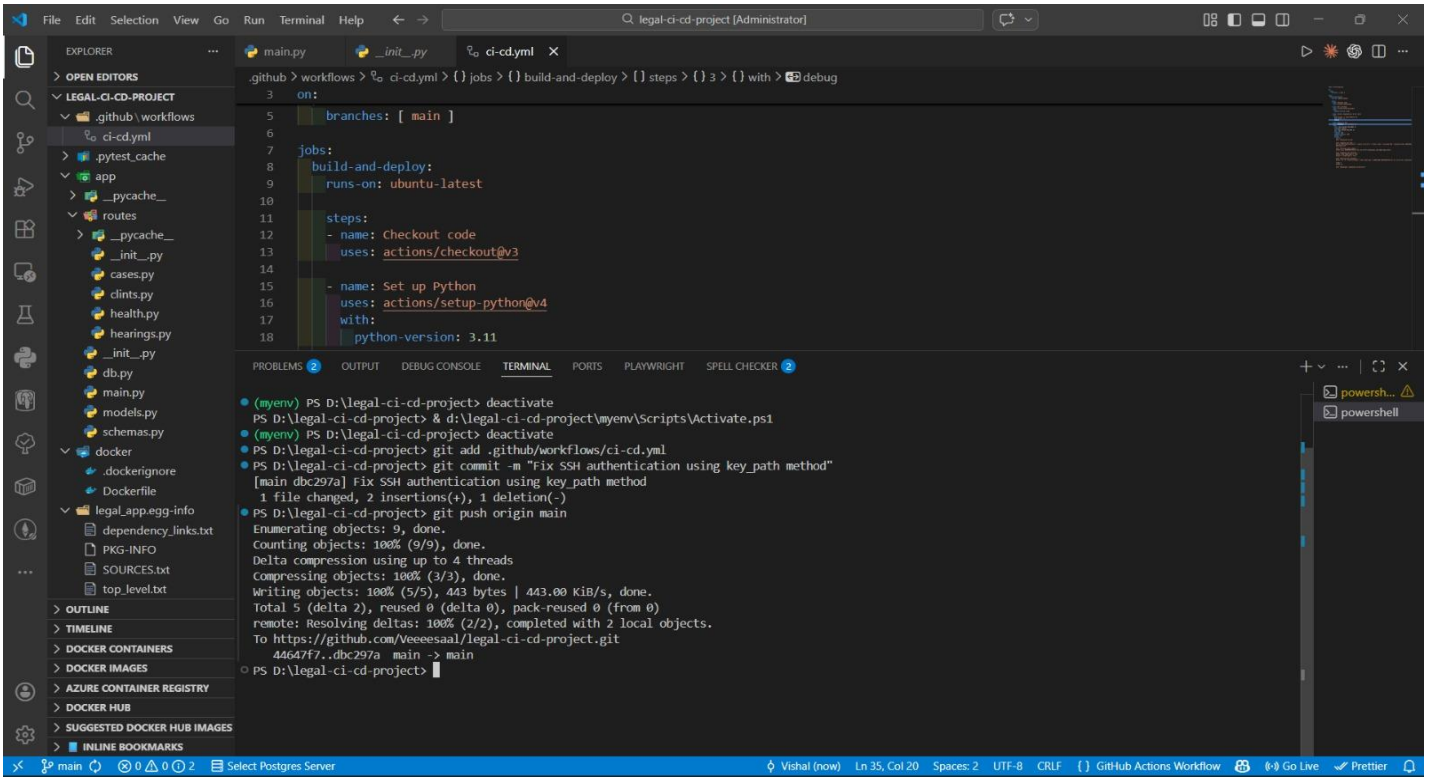


Fig :- 4

- **CI/CD Workflow Configuration in VS Code**

The above screenshot shows the CI/CD workflow configuration being developed and managed in Visual Studio Code.

The file **ci-cd.yml** is created inside the .github/workflows/ directory, which defines the GitHub Actions pipeline. This workflow is configured to trigger automatically when code is pushed to the **main branch**.

Key steps defined in the workflow:

- Checkout the source code from the repository
- Set up the Python environment (version 3.11)
- Install project dependencies
- Run unit tests

3.2 Reports / dashboards / models

- **Reports**
  Reports are the execution results and monitoring details generated during the CI/CD process. They help to track the performance, status, and success of each stage of the pipeline.

  Reports included in this project:

- **GitHub Actions Report**
  - Shows workflow status (Success / Failed)
  - Displays test execution results
  - Shows build and deployment steps
- **Test Execution Report (Pytest)**
  - Shows number of tests passed or failed
  - Confirms application code quality
- **Docker Image Report (ECR)**
  - Shows stored Docker images
  - Displays image versions and tags
- **Deployment Report (EC2)**
  - Shows container start/stop status
  - Confirms application is running
- **End-to-End Pipeline Status**
  - Confirms the complete flow: Code Push → Test → Build → Store → Deploy → Application Live

- **Dashboards**
  In this project, dashboards refer to the **visual interfaces** provided by tools where you can monitor the status of different services.

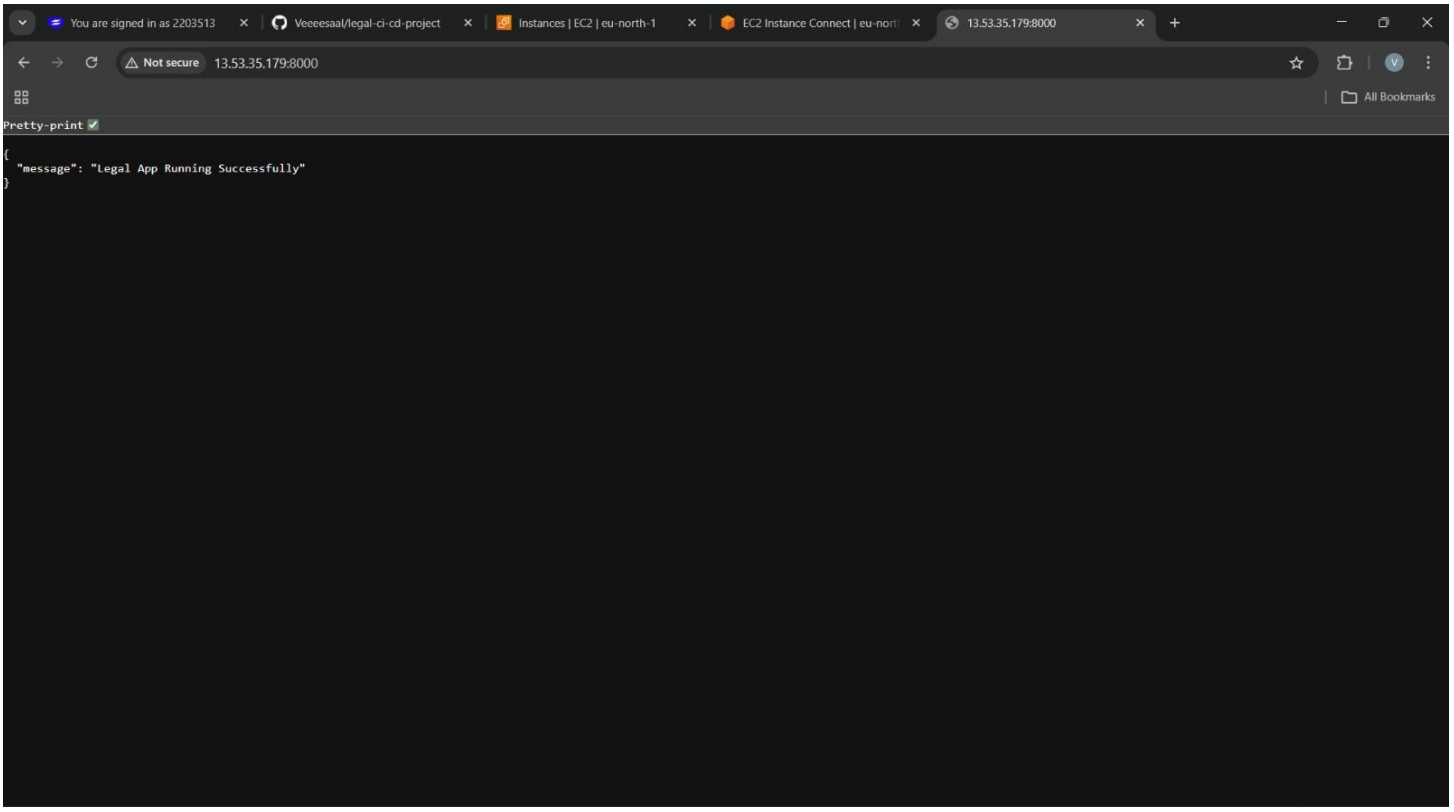


Fig :- 5

- **Application Running Output**
  The above screenshot shows the final output of the deployed application running successfully on the Amazon EC2 server.
  The application is accessed through the EC2 Public IP address on port 8000:

- The Docker container is running properly on EC2
- The application is successfully deployed through the CI/CD pipeline
- The server is accessible over the internet
- The FastAPI service is active and responding correctly
  This output verifies the successful end-to-end execution of the project, from code integration and automated deployment to live application availability.

  Dashboards used:

- **GitHub Actions Dashboard**
  - Shows workflow runs
  - Displays success or failure status
  - Provides step-by-step execution details
- **Amazon ECR Dashboard**
  - Shows stored Docker images

    - Displays image versions and tags
- **Amazon EC2 Dashboard**
    - Shows instance status (running/stopped)
    - Displays public IP and system details

- **Models**

  In this project, there are **no machine learning or data models**. Here, *models* refer to the **system structure or architecture design**.

  Models included:

- **CI/CD Workflow Model**

    - Developer → GitHub → GitHub Actions → Docker → ECR → EC2

- **Architecture Model**

    - Shows how different components (GitHub, Docker, AWS) are connected and work together.

3.3 Key outcomes

- Fully automated CI/CD pipeline implemented
- Zero manual deployment process
- Faster release and update cycle
- Reduced human errors
- Secure cloud-based deployment
- Production-ready DevOps architecture
- Improved application reliability and scalability

4. Conclusion

This project successfully implemented a complete CI/CD pipeline for a Python-based Legal Application using GitHub Actions and AWS cloud services. The system automates the entire software delivery process, including testing, containerization, artifact storage, and deployment.

The implementation ensures faster development cycles, improved code quality, and reliable production deployment. This project provided practical experience in DevOps practices such as Continuous Integration, Continuous Deployment, Docker containerization, cloud deployment, and secure infrastructure management.

Overall, the project demonstrates a real-world, enterprise-level automated deployment solution.

5. Future Scope & Enhancements

Future Scope

The future scope of this project focuses on improving scalability, reliability, and enterprise-level deployment capabilities.

- **Migration to Kubernetes :-** The application can be deployed on Kubernetes to support large-scale container orchestration and better resource management.

- **Auto-Scaling Implementation :-** Auto-scaling can be configured for EC2 instances to automatically adjust resources based on application load and traffic.

- **Integration with AWS Load Balancer :-** A load balancer can be added to distribute traffic across multiple instances, ensuring high availability and fault tolerance.

- **Full Infrastructure Automation using Terraform :-** The entire infrastructure (ECR, EC2, networking, security) can be managed using Infrastructure as Code (IaC) for better consistency and automation.

- **Advanced Monitoring and Logging :-** AWS CloudWatch can be integrated to monitor system performance, resource usage, and application health in real time.

Enhancements

The following enhancements can be implemented to improve system security, deployment safety, and operational efficiency.

- **Blue-Green Deployment Strategy :-** This approach allows deployment of new versions without downtime by maintaining two environments and switching traffic after validation.

- **Rollback Mechanism for Failed Deployments :-** Automatic rollback to the previous stable version can be implemented if a deployment fails or the application becomes unstable.

- **Integration with SonarQube :-** SonarQube can be added to perform advanced code quality analysis, security checks, and maintain coding standards.

- **Deployment Notifications :-** Email or Slack notifications can be configured to inform the team about build status, deployment success, or failure.

- **HTTPS Configuration using SSL Certificates :-** SSL certificates can be configured to enable secure HTTPS communication and improve application security.