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. Introduction

1.1 Scope of the Document

This High-Level Design (HLD) document describes the architecture and CI/CD implementation for the Python-based Legal Application. The focus of this document is to explain how the application is designed, built, tested, containerized, and deployed using automated DevOps practices.

The document specifically covers:

- Application architecture overview
- CI/CD process design
- Cloud deployment strategy
- Security considerations
- Data design overview
- Non-functional requirements

This document does not cover detailed business logic or UI implementation.

1.2 Intended Audience

This document is intended for:

- University Mentors
- Industry Mentors
- Project Evaluators

- DevOps Engineers
- System Architects
- Developers

It provides a clear understanding of the technical architecture and deployment model implemented in the project.

1.3 System Overview

The system is a Python-based Legal Application developed using FastAPI and deployed through a fully automated CI/CD pipeline.

The overall system works as follows:

- Developers push code to GitHub.
- GitHub Actions runs automated tests.
- Docker image is built after successful validation.
- The image is pushed to Amazon ECR.
- AWS EC2 pulls and runs the latest image.
- End users access the application via EC2.

The system ensures reliable deployments, improved code quality, and minimal manual intervention.

---

2. Application Design

2.1 Application Design

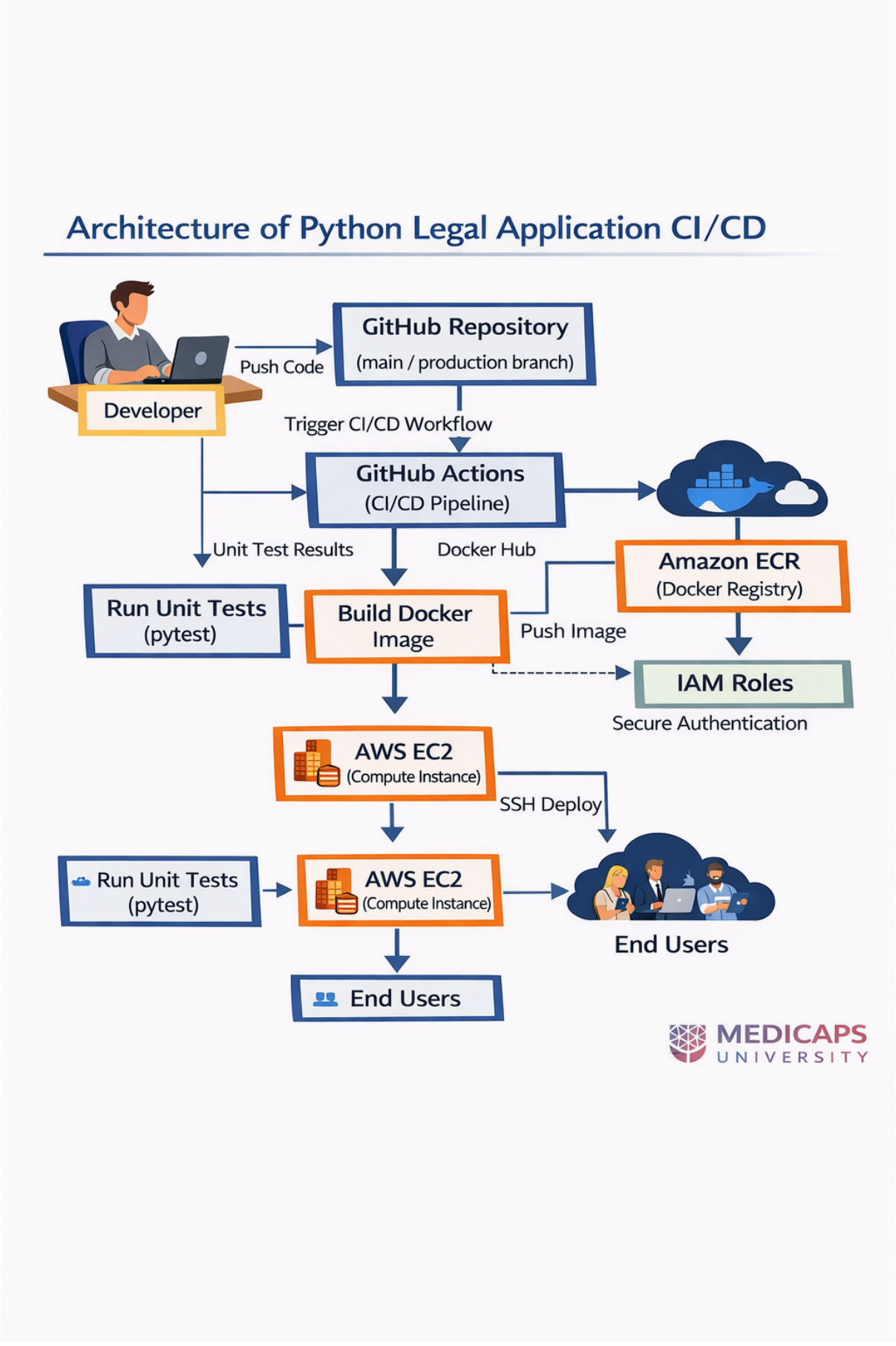
The Python-based Legal Application is developed using FastAPI and deployed through a fully automated CI/CD pipeline. The system architecture integrates source control, automation, containerization, and cloud infrastructure to ensure reliable and consistent deployments.

The application follows a containerized architecture where Docker is used to package the application along with its dependencies. This ensures that the application behaves consistently across development, testing, and production environments.

The architecture consists of the following layers:

- Presentation Layer (REST APIs exposed to users)
- Application Layer (Business logic in FastAPI)
- Container Layer (Docker runtime)
- Automation Layer (GitHub Actions CI/CD)
- Cloud Infrastructure Layer (AWS ECR + EC2)

This layered approach ensures separation of concerns, maintainability, and scalability.


Architecture of Python Legal Application CI/CD
Developer
Push Code
GitHub Repository
(main / production branch)
Trigger CI/CD Workflow
GitHub Actions
(CI/CD Pipeline)
Unit Test Results
Docker Hub
Amazon ECR
(Docker Registry)
Run Unit Tests
(pytest)
Build Docker Image
Push Image
IAM Roles
Secure Authentication
AWS EC2
(Compute Instance)
SSH Deploy
Run Unit Tests
(pytest)
AWS EC2
(Compute Instance)
End Users
End Users
MEDICAPS UNIVERSITY

2.2 Process Flow

The system follows an automated CI/CD-driven process flow from development to production deployment.

Development Phase

The developer writes and tests code locally before pushing it to the GitHub repository.

Continuous Integration Phase

When code is pushed to the main branch:

- GitHub Actions is triggered automatically.
- The workflow checks out the latest source code.
- Python environment is configured.
- Dependencies are installed.
- Unit tests are executed using pytest.
- If tests fail, the process stops.
- If tests pass, the code is approved for merge.

Continuous Deployment Phase

When code is merged into the production branch:

- Docker image is built.
- Image is tagged with commit SHA.
- Image is pushed to Amazon ECR.

- EC2 server pulls the updated image.

- Old container is stopped.
- New container is started.
- Health check endpoint verifies deployment.

This automated flow reduces manual intervention and deployment errors.

2.3 Information Flow

The information flow describes how data and artifacts move across the system.

1. Source Code Flow
   Developer → GitHub Repository

2. CI Results Flow
   GitHub Actions → Test Reports → Merge Decision

3. Artifact Flow
   GitHub Actions → Docker Image → Amazon ECR

4. Deployment Flow
   Amazon ECR → AWS EC2 → Running Container

5. User Request Flow
   End User → EC2 → FastAPI Application → Response

This structured information flow ensures transparency, traceability, and controlled deployment of application versions.

2.4 Components Design

The system is composed of several core components that work together to implement CI/CD.

GitHub Repository

- Stores source code.
- Maintains branching strategy.
- Contains workflow configuration files.

GitHub Actions

- Executes CI and CD pipelines.
- Runs automated tests.
- Builds Docker images.
- Deploys application to EC2.

Docker

- Packages application into container image.
- Ensures consistent runtime environment.

Amazon ECR

- Stores Docker images.
- Maintains version control using tags.

- Provides secure artifact storage.

AWS EC2

- Hosts containerized application.

- Executes Docker runtime.
- Serves requests from end users.

IAM and Secrets

- Secure authentication between GitHub and AWS.
- Implements least privilege access control.

2.5 Key Design Considerations

While designing the CI/CD architecture, the following principles were considered:

- Automation First: Minimize manual deployment steps.
- Immutable Artifacts: Each build generates a new Docker image.
- Security: No hardcoded credentials; use IAM roles.
- Traceability: Use commit SHA for tagging images.
- Scalability: Architecture can be extended to ECS/Kubernetes.
- Reliability: Fail-fast mechanism in CI pipeline.

- Maintainability: Clear separation of responsibilities.

These considerations ensure that the system aligns with enterprise DevOps standards.

2.6 API Catalogue

The Legal Application exposes REST APIs for interacting with the system.

Endpoint	Method	Description
/health	GET	Health check endpoint
/users	GET	Retrieve user list
/cases	GET	Retrieve legal cases
/cases	POST	Create new case
/auth/login	POST	User authentication

The `/health` endpoint is particularly important for CI/CD deployment validation, as it confirms that the application is running successfully after deployment.

(Note: APIs listed for design representation.)

3. Data Design

3.1 Data Model

The Legal Application follows a relational data model structure. The core entities include:

- Users
- Legal Cases
- Documents
- Roles and Permissions

Each entity is designed to maintain referential integrity and enforce role-based access control. The application uses ORM-based interaction with the database to maintain abstraction and scalability.

Although the CI/CD pipeline does not directly manipulate the database structure, it ensures that updated application versions are deployed safely without affecting existing data.

The application accesses data using:

- ORM-based queries
- Secure environment variables for database credentials
- Connection pooling for optimized performance

Database credentials are not hardcoded. They are configured securely using environment variables to maintain security best practices.

3.3 Data Retention Policies

Data retention policies ensure that system artifacts and logs are managed efficiently.

- Application logs are rotated regularly.
- Docker containers do not permanently store data.
- Amazon ECR uses lifecycle policies to remove older unused images.
- Database backups (future enhancement) ensure long-term data safety.

3.4 Data Migration

Database schema updates are handled using migration tools. The CI/CD pipeline ensures:

- New versions are deployed without breaking schema.
- Backward compatibility is maintained.

deployment failure.

- Rollback is possible in case of

---

4. Interfaces

The system interacts with both internal and external interfaces.

External Interfaces

- GitHub API (source control integration)
- AWS ECR API (image push/pull)
- AWS EC2 SSH interface (deployment)
- REST APIs accessed by end users

Internal Interfaces

- Application-to-database communication
- GitHub Actions to AWS authentication
- Docker engine interaction

These interfaces ensure seamless communication between system components.

---

5. State and Session Management

The application follows a stateless architecture.

- No session data is stored in memory.
- Authentication is handled using token-based mechanisms (e.g., JWT).
- Containers can be restarted without affecting user sessions.
- Stateless design improves scalability and fault tolerance.

This design aligns with modern cloud-native deployment practices.

---

6. Caching

Currently, the system implements minimal caching. However, caching considerations include:

- Docker layer caching in CI pipeline for faster builds.
- API response caching (future enhancement).
- Redis-based caching for performance optimization (future enhancement).

Caching improves performance, reduces load, and enhances scalability in high-traffic environments.

---

7. Non-Functional Requirements

7.1 Security Aspects

Security is a primary concern in the system design. The following measures are implemented:

- GitHub Secrets for storing credentials.
- IAM roles with least privilege access.
- No hardcoded credentials in repository.
- SSH key-based EC2 access.
- Protected branches in GitHub.
- Encrypted communication between services.

These measures ensure confidentiality, integrity, and controlled access.

7.2 Performance Aspects

The system is designed to deliver optimized performance.

- Lightweight Docker image for faster startup.
- Parallel execution of CI jobs.
- Cached dependencies to reduce build time.
- Optimized Python environment.
- Scalable cloud infrastructure ready for horizontal expansion.

These aspects ensure faster deployments and responsive application performance.

---

8. References

1. GitHub Actions Documentation
2. Docker Official Documentation
3. Amazon ECR Documentation
4. Amazon EC2 Documentation
5. FastAPI Documentation
6. DevOps Best Practices Guide