Project - High Level Design

on

Real Estate System – DevOps Enabled Deployment using IaC

Course Name:

Institution Name: Medicaps University – Datagami Skill Based Course

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

Sr no	Student Name	Enrolment Number
1	Samruddhi Muley	EN22CS304056
2	Payal Singh	EN22CS301701
3	Aarti Bamanke	EN22EE301001
4	Akshat Joshi	EN22CS304006
5	Punit Pawar	EN22CS303038
6	Shubhi Dubey	EN22CS304061

Group Name: Group 01D11

Project Number: DO-27

Industry Mentor Name: Mr. Vaibhav

University Mentor Name: Prof. Shyam Patel

Academic Year: 2026

Table of Contents

1. Introduction

The Real Estate DevOps System is a containerized web application built using Flask that enables users to browse property listings and view detailed property information. The system is designed following DevOps principles with automated infrastructure provisioning using Infrastructure as Code (IaC) tools such as Terraform, containerization using Docker, and orchestration using Kubernetes.

The primary objective of this system is to demonstrate automated, scalable, and repeatable infrastructure deployment while maintaining application portability and reliability.

1.1 Scope of the Document

This document defines architectural boundaries, deployment strategy, and operational workflows. It outlines high-level interactions between system components without detailing internal implementation logic.

It also serves as a baseline document for future enhancements, including database scaling, security improvements, and monitoring integration.

This document provides a high-level design of:

- Application architecture
- Infrastructure architecture
- Deployment workflow
- Data design
- API structure
- Security and performance considerations
- DevOps automation process

This document does not include low-level code implementation details.

1.2 Intended Audience

The document assumes that readers have basic knowledge of:

- Cloud computing concepts
- DevOps practices
- Containerization technologies
- Web application architecture

This document is intended for:

- DevOps Engineers
- Cloud Architects

- Backend Developers
- System Administrators
- Technical Reviewers
- Project Evaluators

1.3 System Overview

The system follows a modular design approach where infrastructure, application, and deployment layers are logically separated. This separation allows independent scaling, maintenance, and upgrades without affecting other layers.

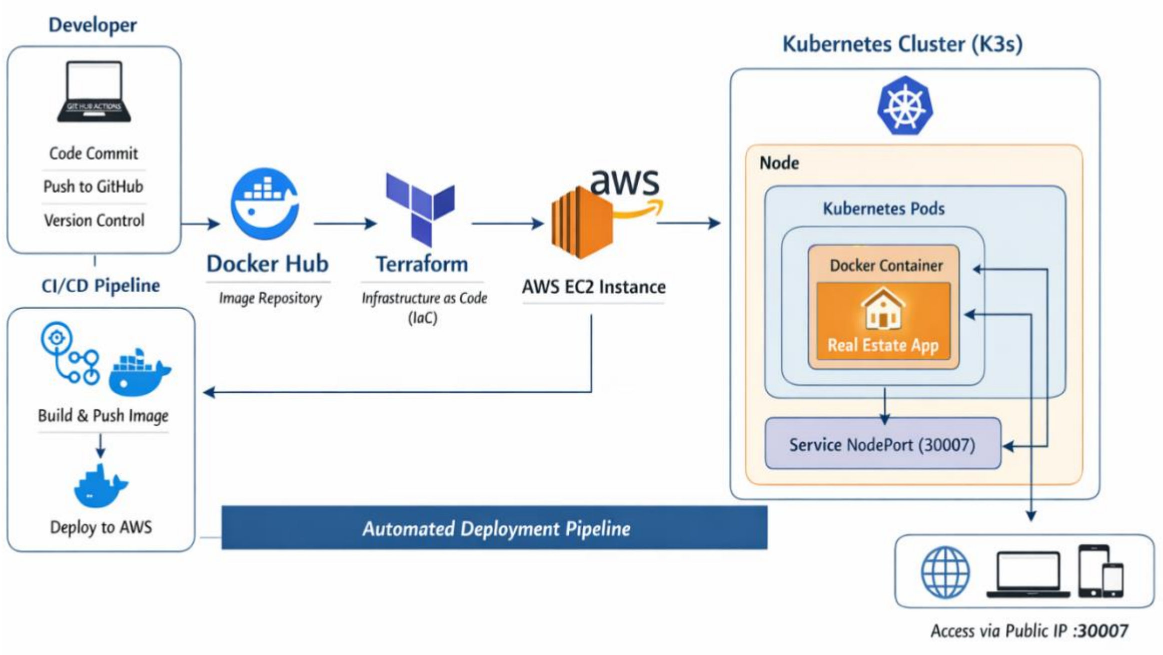
The architecture is cloud-ready and can be extended to multi-region deployments.

The system consists of:

- A Flask-based backend application
- HTML/CSS/Jinja2 frontend
- SQLite database
- Docker containerization
- Infrastructure provisioning using Terraform
- CI/CD pipeline using GitHub Actions or Jenkins
- Kubernetes for container orchestration (planned/implemented)

The system enables automated infrastructure setup and application deployment without manual configuration.

High Level Architecture - Real Estate System Deployment



2. System Design

2.1 Application Design

The application follows a layered architecture:

Presentation Layer-

HTML, CSS frontend

Flask templates

Application Layer-

Flask backend logic

API endpoints

Request handling

Infrastructure Layer-

Docker container

Kubernetes deployment

AWS EC2 hosting

The application follows a request-response lifecycle:

1. Client sends HTTP request
2. Flask route handles request
3. Database queried via ORM
4. Template renders response
5. Response returned to client

```
FRONTEND LAYER
├── HTML5 (Structure)
├── CSS3 (Styling - Gradient design, animations)
├── Jinja2 (Templating engine)
└── Unsplash (Property images)
```

```
BACKEND LAYER
├── Python 3.11 (Programming language)
├── Flask 3.0.0 (Web framework)
└── Werkzeug (WSGI utility library)
```

```
CONTAINERIZATION LAYER
├── Docker (Container platform)
├── Docker Compose (Multi-container orchestration)
└── Docker Hub (Container registry)
```

```
CONTAINERIZATION LAYER
├── Docker (Container platform)
├── Docker Compose (Multi-container orchestration)
└── Docker Hub (Container registry)
```

```
CI/CD LAYER
├── GitHub Actions (CI/CD platform)
├── pytest 7.4.3 (Testing framework)
└── YAML (Pipeline configuration)
```

```
VERSION CONTROL LAYER
├── Git (Version control system)
├── GitHub (Repository hosting)
└── Branching strategy (main, feature/*, bugfix/*)
```

```
INFRASTRUCTURE LAYER (Planned)
├── Terraform (IaC tool)
├── AWS (Cloud provider)
├── EC2 (Compute)
├── VPC (Networking)
├── Security Groups (Firewall)
└── ELB (Load balancing - optional)
├── Kubernetes (Container orchestration)
└── Ansible (Configuration management)
```

2.2 Process Flow

The deployment pipeline ensures zero manual intervention after code push. Automated testing and image building reduce the risk of broken deployments.

The runtime process ensures stateless behaviour to support horizontal scaling in Kubernetes environments.

User Flow:

1. User opens browser

2. Sends HTTP request to EC2 Public IP

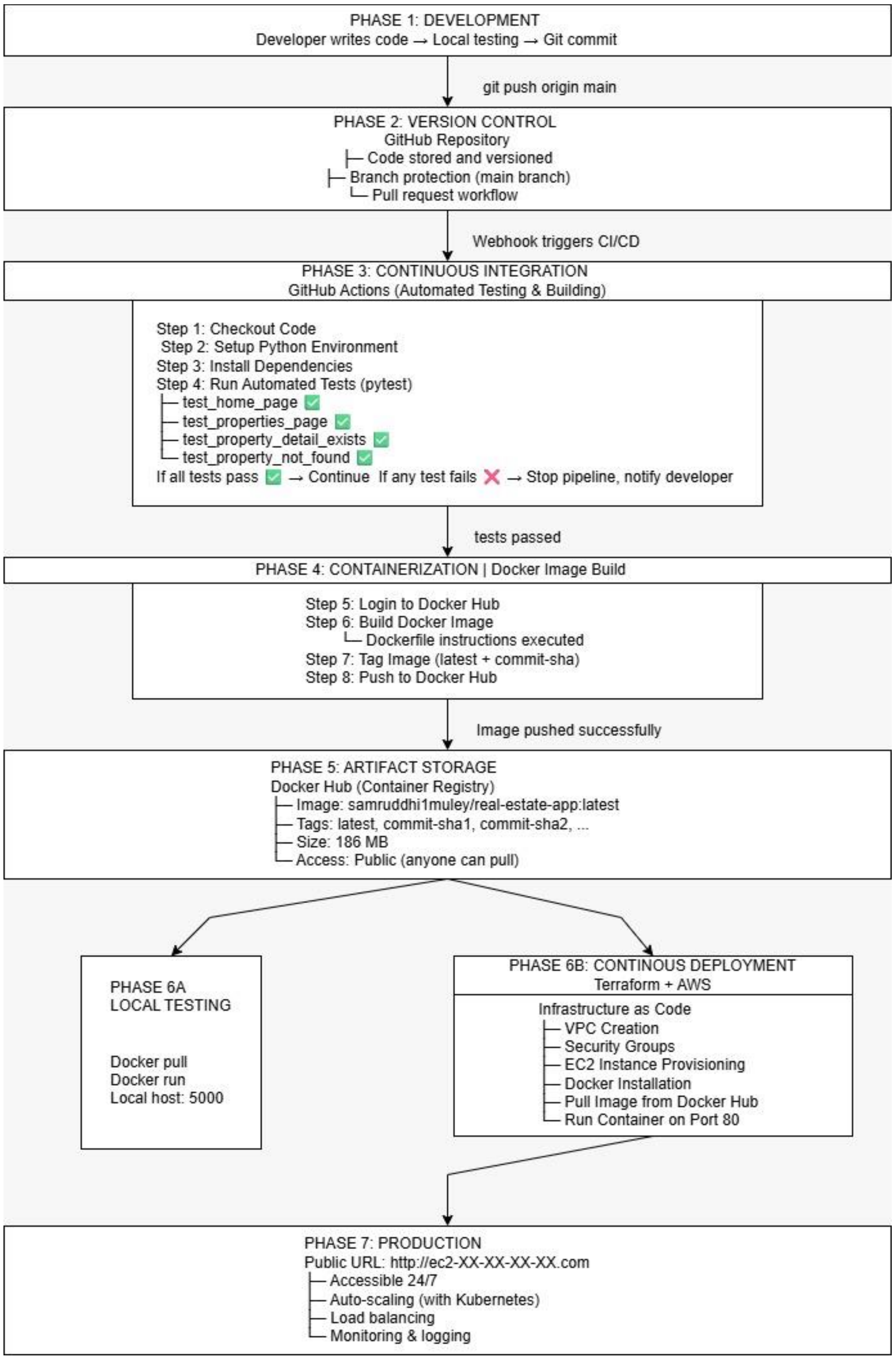
3. Kubernetes service forwards request to container

4. Flask application processes request

5. Response displayed in browser

CI/CD Flow:

1. Code pushed to Git repository
2. Pipeline triggers automatically
3. Docker image built
4. Image pushed to registry
5. Terraform provisions infrastructure
6. Kubernetes deploys updated container


PHASE 1: DEVELOPMENT
Developer writes code → Local testing → Git commit
git push origin main
PHASE 2: VERSION CONTROL
GitHub Repository
├ Code stored and versioned
├ Branch protection (main branch)
└ Pull request workflow
Webhook triggers CI/CD
PHASE 3: CONTINUOUS INTEGRATION
GitHub Actions (Automated Testing & Building)
Step 1: Checkout Code
Step 2: Setup Python Environment
Step 3: Install Dependencies
Step 4: Run Automated Tests (pytest)
├ test_home_page ✅
├ test_properties_page ✅
├ test_property_detail_exists ✅
└ test_property_not_found ✅
If all tests pass ✅ → Continue If any test fails ❌ → Stop pipeline, notify developer
tests passed
PHASE 4: CONTAINERIZATION | Docker Image Build
Step 5: Login to Docker Hub
Step 6: Build Docker Image
└ Dockerfile instructions executed
Step 7: Tag Image (latest + commit-sha)
Step 8: Push to Docker Hub
Image pushed successfully
PHASE 5: ARTIFACT STORAGE
Docker Hub (Container Registry)
├ Image: samruddhi1muley/real-estate-app:latest
├ Tags: latest, commit-sha1, commit-sha2, ...
├ Size: 186 MB
└ Access: Public (anyone can pull)
PHASE 6A
LOCAL TESTING
Docker pull
Docker run
Local host: 5000
PHASE 6B: CONTINOUS DEPLOYMENT
Terraform + AWS
Infrastructure as Code
├ VPC Creation
├ Security Groups
├ EC2 Instance Provisioning
├ Docker Installation
├ Pull Image from Docker Hub
└ Run Container on Port 80
PHASE 7: PRODUCTION
Public URL: http://ec2-XX-XX-XX-XX.com
├ Accessible 24/7
├ Auto-scaling (with Kubernetes)
├ Load balancing
└ Monitoring & logging

Complete devops workflow

2.3 Information Flow

Information flow is unidirectional in request-response format, ensuring clarity in data processing.

User Browser

↓

AWS EC2 Public IP

↓

Kubernetes Service

↓

Kubernetes Pod

↓

Docker Container

↓

Flask Application

↓

HTML Response

2.4 Components Design

Major Components:

- **GitHub Repository**

  Stores source code

  Stores Terraform configuration
- **GitHub Actions**

  CI/CD pipeline

  Builds Docker image

  Pushes image to Docker Hub
- **Docker**

  Containerizes application
-

- **Docker Hub**

  Stores Docker image

- **Terraform**

  Provisions infrastructure

  Creates EC2 instance

  Configures networking

- **AWS EC2**

  Hosts Kubernetes cluster

- **Kubernetes (K3s)**

  Orchestrates containers

  Manages pods and services

2.5 Key Design Considerations

- Infrastructure automation using Terraform
- Container portability using Docker
- Scalability using Kubernetes
- Automated deployment using CI/CD
- Cloud-based hosting for high availability
- Security via controlled acce

2.6 API Catalogue

All APIs are designed to follow RESTful standards and proper HTTP methods.

Future enhancements may include:

- POST APIs for adding properties
- PUT APIs for updating properties
- DELETE APIs for administrative control
- Authentication endpoints

API versioning strategy can be introduced for backward compatibility.

Endpoint	Method	Description
/	GET	Home page
/properties	GET	List all properties
/property/<id>	GET	View property details
/static/*	GET	Serve static assets

All APIs follow RESTful design principles and return HTML responses.

3. Data Design

3.1 Data Model

- Data includes:
- Property ID
- Property Name
- Location
- Price
- Description

Data is stored in SQLite database.

3.2 Data Access Mechanism

Data is accessed using:

- Flask backend logic
- SQL queries
- ORM abstraction layer where applicable

Database interactions are handled securely through the backend application.

3.3 Data Retention Policies

- Data is stored on the EC2 instance.
- Backups can be created using AWS snapshots.
- Data is retained until manually deleted or infrastructure is destroyed.
- Future implementations may include automated backup strategies.

3.4 Data Migration

Terraform enables recreation of infrastructure without affecting application source code.

Database migration can be handled using scripts or migration tools during upgrades.

4. Interfaces

External Interfaces:

- Web Browser
- AWS EC2
- Docker Hub
- GitHub

Internal Interfaces:

- Kubernetes Service to Pod
- Pod to Docker Container
- Container to Flask Application

5. State and Session Management

The application follows a stateless architecture.

Each request is processed independently.

No persistent session storage is currently implemented.

Kubernetes ensures container restart and high availability without affecting system stability.

6. Caching

Currently, no caching mechanism is implemented.

Future improvements may include:

- Redis-based application caching
- CDN caching for static assets

7. Non-Functional Requirements

7.1 Security Aspects

- AWS Security Groups restrict external access.
- SSH access is controlled via key pairs.
- Docker provides container-level isolation.
- Kubernetes ensures pod-level isolation.
- IAM roles control access to AWS resources.

7.2 Performance Aspects

- Kubernetes enables horizontal scalability.
- Docker ensures fast application startup.
- Terraform enables quick infrastructure provisioning.
- Cloud deployment ensures better availability and reliability.

8. References

- AWS Documentation
- Terraform Documentation
- Docker Documentation
- Kubernetes Documentation
- Flask Documentation
- GitHub Actions Documentation