Project - High Level Design
on
IaC Provisioning for Finance System

Course Name: DevOps Foundation

Institution Name: Medicaps University-Datagami Skill Based Course

Sr.no	Student Name	Enrolment Number
1.	Tarang Purohit	EN22CS3011033
2.	Suhani Maheshwari	EN22CS301988
3.	Smita Kumawat	EN22CS301961
4.	Sneha Agrawal	EN22CS301963
5.	Somya Neema	EN22CS301973

Group Name: Group 03D9
Project Number: DO-03
Industry Mentor Name: Mr. Vaibhav Sir
University Mentor Name: Dr. Ritesh Joshi
Academic Year: 2025-26

Table of Contents

1. Introduction

In modern software development, automation, cloud computing, and infrastructure management play a critical role in delivering scalable and reliable applications. Traditional manual server configuration is time-consuming, error-prone, and difficult to maintain. To overcome these limitations, Infrastructure as Code (IaC) and containerization technologies are widely adopted.

The project titled **"Infrastructure as Code (IaC) Provisioning for Finance System"** integrates cloud computing, DevOps practices, and web application development into a single automated deployment solution.

The system consists of:

- A Django-based Finance Web Application
- Docker-based containerized environment
- Terraform-based infrastructure provisioning
- AWS-based cloud hosting
- GitHub Actions-based CI/CD automation

The goal of this project is to demonstrate how a web application can be automatically deployed in a cloud environment using modern DevOps tools.

1.1Scope of the Document

This High-Level Design (HLD) document describes:

- Overall architecture of the Finance System
- Application structure and modules
- Infrastructure design using Terraform
- Containerization approach using Docker
- CI/CD workflow using GitHub Actions
- Security and performance considerations

This document focuses only on implemented components. Advanced enterprise-level systems such as load balancers, distributed databases, microservices, and auto-scaling are not included since they are beyond the academic scope of this project.

1.2 Intended Audience

This document is intended for:

- DevOps Engineers
  Responsible for designing, implementing, and maintaining automated infrastructure pipelines.
- Cloud Architects
  Designing scalable, secure, and resilient cloud-based financial systems.
- Developers
  Understanding deployment workflows and environment setup processes.
- System Administrators
  Managing servers, monitoring deployments, and maintaining system availability.
- Academic Evaluators / Mentors
  Assessing the design, architecture, and technical implementation of the project.

1.3 System Overview

The Finance System is a web-based application developed using the Django framework in Python. The system allows users to:

- Register and authenticate
- Add income records
- Add expense records
- Categorize expenses
- View dashboard summary

The application is:

- Containerized using Docker
- Hosted on AWS EC2
- Infrastructure provisioned using Terraform
- Deployed automatically using GitHub Actions

The system is publicly accessible using the EC2 Public IP address over HTTP (Port 80).

2. System Design

System Design defines how different components of the application interact with each other to deliver the required functionality. It includes application architecture, infrastructure architecture, deployment flow, and component structure.

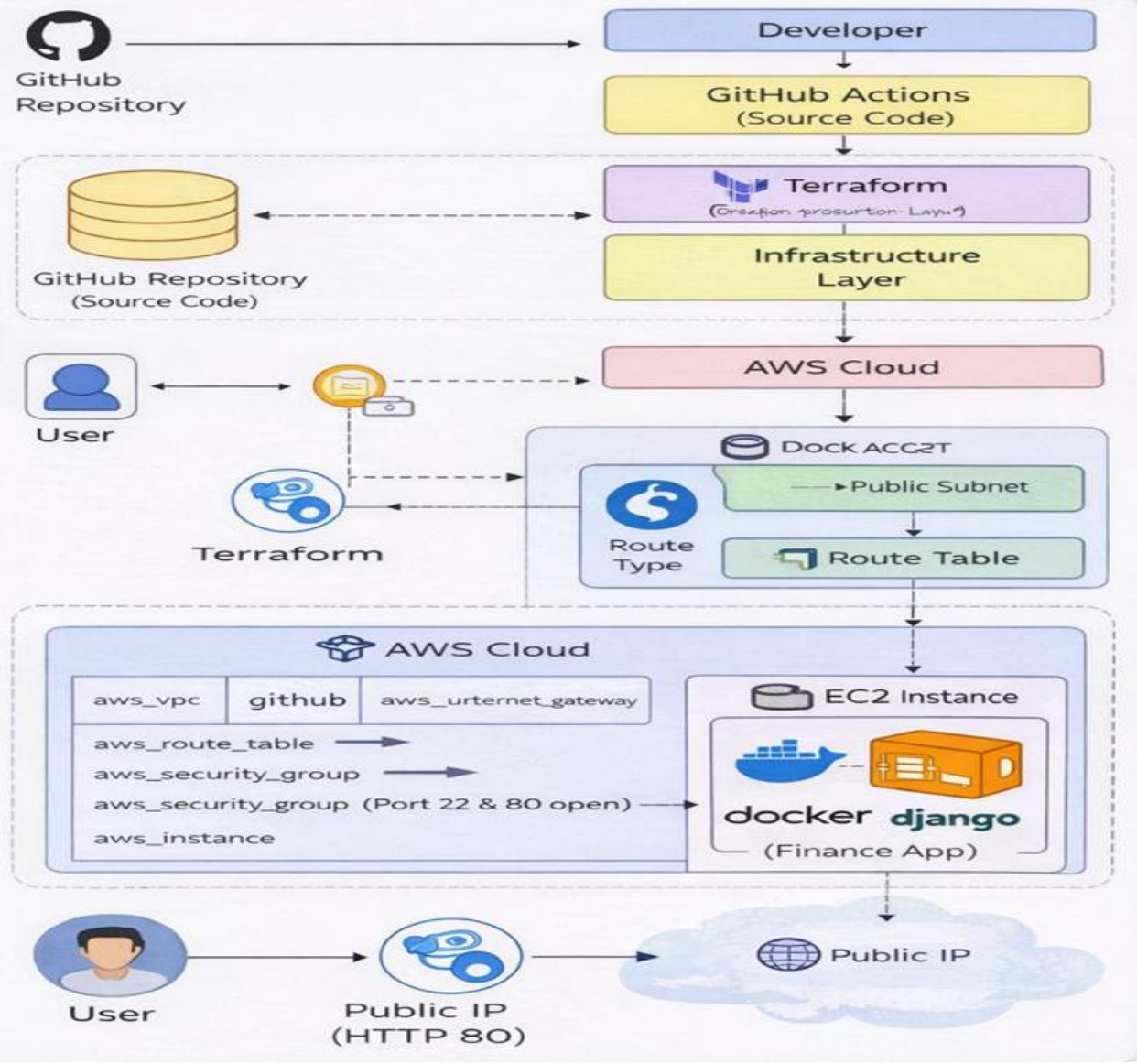

Fig. 1 System Architecture

2.1 Application Design

Application Design describes how the web application is structured internally.

1. Architectural Pattern

The application follows the **Model-View-Template (MVT)** architecture of Django.

- **Model**: Defines database structure and relationships.
- **View**: Contains business logic and request handling.
- **Template**: Handles user interface using HTML and CSS.

This separation improves maintainability and modularity.

2. Functional Modules

The application consists of the following modules:

1. Authentication Module

Handles:

- User registration
- User login
- User logout

Django's built-in authentication system is used

2. Expense Module

Allows users to:

- Add expense records
- Assign categories
- View expense history

3. Income Module

Allows users to:

- Add income records
- View income history

4. Dashboard Module

Displays:

- Summary of income
- Summary of expenses
- Financial overview

3. Technology Stack

Layer	Technology Used
Backend	Django(Python)
Frontend	HTML,CSS
Database	SQLite
Containerization	Docker
Infrastructure	Terraform
Cloud Platform	AWS EC2
CI/CD	GitHub Actions

2. Process Flow

Process Flow describes how the system moves from development stage to deployment stage.

2.2.1 Development Flow

1. The developer writes code locally.
2. Code is pushed to GitHub.

3. GitHub Actions pipeline is triggered.
4. Docker image is built automatically.
5. Docker image is pushed to DockerHub.
6. Terraform provisions AWS infrastructure.
7. EC2 installs Docker using user_data.
8. EC2 pulls the latest Docker image.
9. Docker container runs the application.
10. Application becomes accessible via public IP.
11. This automated pipeline reduces manual errors and ensures consistency.

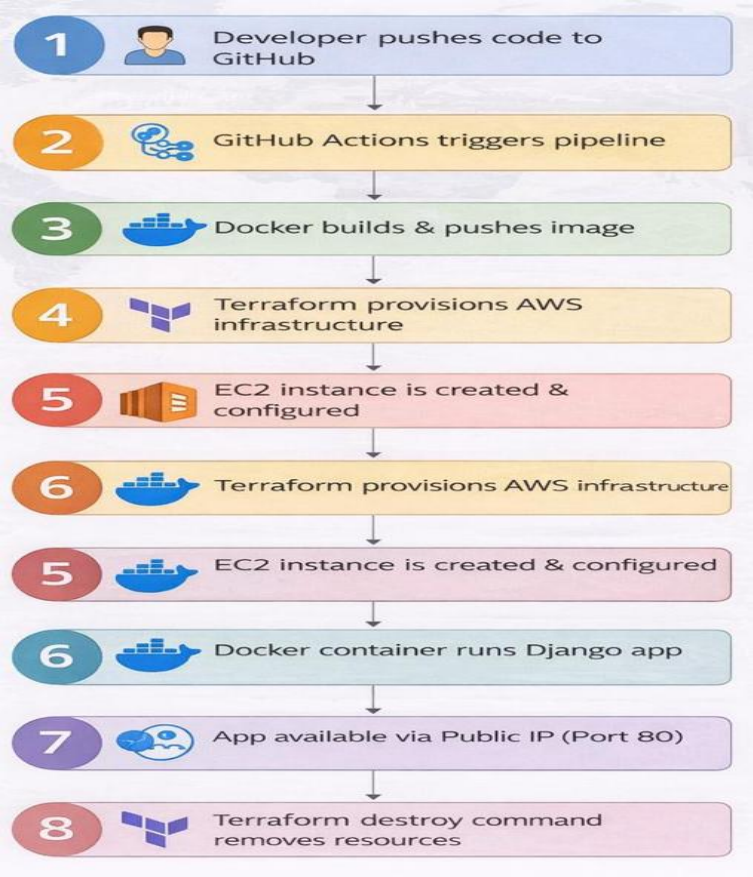

Fig. 2

2.3 Information Flow

Information Flow describes how user requests travel through the system.

Request Flow:

User

- Internet
- AWS Internet Gateway
- EC2 Public IP (Port 80)
- Docker Container (Port 8000 internal)
- Django Application
- SQLite Database

Response travels back in reverse order.

2.4 Component Design

Component Design defines individual system elements.

Fig. 3

2.4.1 Infrastructure Components (Terraform Managed)

The infrastructure is defined using Terraform configuration files. Resources created:

- VPC
- Public Subnet
- Internet Gateway
- Route Table
- Route Table Association
- Security Group
- EC2 Instance

Infrastructure can be created using: **terraform apply**

And destroyed using: **terraform destroy**

2.4.2 Application Components

- Django Project
- Expense & Income Models
- Authentication System
- SQLite Database

2.4.3 Container Components

- Dockerfile
- Docker Image
- Running Docker Container
  Port Mapping: Container Port 8000→ EC2 Port 80

2.4.3 CI/CD Components

- GitHub Repository
- GitHub Actions workflow file (deploy.yml)
- DockerHub integration
- AWS credentials stored as GitHub Secrets This enables fully automated deployment.

2.5 Key Design Considerations

1. Infrastructure as Code

   Infrastructure is defined in code format, making deployment repeatable and consistent.

2. Automation

   CI/CD eliminates manual deployment steps.

3. Containerization

   Docker ensures same environment across local and cloud systems.

4. Cost Optimization

   t2.micro / t3.micro instances reduce AWS cost.

5. Simplicity

   SQLite is used instead of RDS for academic simplicity.

6. Security

   Only required ports (80, 22) are open.

2.6 API Catalogue

The system is not REST-based but provides web endpoints handled by Django views.

End point	Method	Description
/	GET	Home Page
/login	GET/POST	User login
/register	GET/POST	User registration
/add-expense	GET/POST	Add expense
/add-income	GET/POST	Add income

3. DATA DESIGN

Data Design defines how data is structured and managed.

3.1 Data Model

1. Entities:

    1. User (Django default model)

    2. Expense

        - Amount
        - Description
        - Category
        - Date
        - User (Foreign Key)

3. Income
    - Amount
    - Description
    - Date
    - User (Foreign Key)

4. Category
    - Name
    - User (Foreign Key)

Relationships are managed using Django ORM.

3.2 Data Access Mechanism

The project uses Django ORM for:

- Insert operations
- Update operations
- Delete operations
- Select queries

No raw SQL queries are used.

3.3 Data Retention Policies

- Data stored in SQLite database inside container
- If EC2 instance is destroyed, data is lost
- No backup strategy implemented (academic scope)

3.4 Data Migration

Database schema changes are managed using:

- python manage.py makemigrations
- python manage.py migrate

These commands ensure database consistency.

4. INTERFACES

Interfaces define how the system interacts externally and internally.

External Interfaces

- GitHub (Source Code Hosting)
- DockerHub (Image Storage)
- AWS EC2 (Cloud Hosting)
- S3 (Optional backend storage)

Internal Interfaces

- Django Views ↔ Django ORM
- Docker Container ↔ EC2 OS

5. STATE AND SESSION MANAGEMENT

The application uses the Django session framework.

- Session stored in database
- Login maintained using session cookies
- Password hashing handled securely

6. CACHING

No external caching system like Redis is implemented.
Application relies on:

- Django default request-response cycle
- Browser-level caching

Suitable for low-traffic academic usage.

7. NON-FUNCTIONAL REQUIREMENTS

7.1 Security Aspects

- Security Groups restrict ports
- SSH access via key pair
- Credentials stored as GitHub Secrets
- ALLOWED_HOSTS configured
- Password hashing enabled

7.2 Performance Aspects

- Lightweight EC2 instance
- Docker optimized runtime
- Suitable for small-scale usage

8. REFERENCES

- Django Documentation
- Terraform Documentation
- AWS EC2 Documentation
- Docker Documentation
- GitHub Actions Documentation