PROJECT REPORT ON

"IaC Provisioning for Finance System"

Course Name: DevOps Foundation

Institute: Medi-caps University - Datagami Skill-Based Course

Sr No.	Student Name	Enrolment Number
01	AYUSH SAHU	EN22IT301027
02	MAHIMA SANTORE	EN22IT301053
03	HITAKSHI SONI	EN22IT301039
04	BHUMIKA ATALSIYA	EN24CA5030035
05	HARSH PATEL	EN24CA5030059

Group Name: Group 05D12

Project Number: DO-43

Industry Mentor: Mr. Vaibhav

University Mentor: Prof. Akshay Saxena

Academic Year: 2025-2026

Acknowledgement

The successful completion of this project, **"IaC Provisioning for Finance System (DO-43)"**, would not have been possible without the support and guidance of several individuals and institutions. We would like to express our deepest gratitude to everyone who contributed to this journey.

First and foremost, we are immensely grateful to our industry mentor, Mr. Vaibhav sir, for his invaluable technical guidance, constant encouragement, and for sharing his expertise in DevOps methodologies, specifically in Terraform and Docker. His insights into Infrastructure as Code (IaC) helped us bridge the gap between theoretical knowledge and industry standards.

We also extend our sincere thanks to our university mentor, **Akshay Saxena sir**, for his continuous support, academic supervision, and for providing us with the necessary resources to carry out this project within the university framework.

A special thanks to **Medicaps University** and the **Datagami** team for designing the **DevOps Foundation** course, which provided us with a platform to work on real-world automation challenges in the finance domain. The hands-on experience with cloud provisioning and CI/CD pipelines has been a significant learning milestone for all of us.

Lastly, we would like to thank our team members for their coordination and hard work, and our parents for their constant motivation and belief in our dreams.

AYUSH SAHU

MAHIMA SANTORE

HITAKSHI SONI

BHUMIKA ATALSIYA

HARSH PATEL

Faculty of Engineering

Medicaps University

Abstract

The project titled **"IaC Provisioning for Finance System" (Project DO-43)** addresses the critical need for automation, scalability, and security in modern financial infrastructure. In the traditional approach, manual configuration of servers often leads to "Environmental Drift," where development, testing, and production environments become inconsistent, causing deployment failures. This project eliminates such challenges by treating infrastructure as software through **Infrastructure as Code (IaC)**.

The core objective is to establish a **100% Repeatable and Scalable Setup** using a robust DevOps toolchain. The system leverages **Terraform** for automated provisioning of AWS resources like VPCs, Subnets, and EC2 instances in a declarative manner. To ensure application isolation and portability, the entire finance application is containerized using **Docker**.

A strictly automated **CI/CD pipeline** is implemented via **GitHub Actions**, which triggers a build-and-deploy sequence whenever code is pushed to the repository. Security is a primary focus, integrated through **JWT (JSON Web Tokens)** for stateless authentication and **AWS Security Groups** for network isolation. By bridging the gap between manual orchestration and modern automation, this system provides a secure, high-availability framework optimized for professional financial data visualization and processing.

Keywords: Infrastructure as Code (IaC), Terraform, Docker, AWS EC2, GitHub Actions, CI/CD, Finance System Automation.

Table of content

Chapter-1: Problem Statement & Objectives

1.1 Problem Statement

In the traditional financial sector, infrastructure deployment is mostly handled manually, which creates serious challenges in consistency, security, and reliability. One major issue is Environmental Drift, where local development environments differ from production cloud environments such as Amazon Web Services. Differences in software versions, security patches, and configurations often lead to application failures when systems are deployed, especially in high-risk financial operations.

Manual provisioning of network components like VPCs and Security Groups increases the chances of human error. A small misconfiguration, such as leaving ports open, can expose sensitive financial data and create major security vulnerabilities. Additionally, without Infrastructure as Code (IaC), recovering from server failures takes significant time because infrastructure must be rebuilt manually. In financial systems that require near 100% uptime and secure transaction handling, such delays are unacceptable.

Overall, the absence of IaC leads to inconsistency, security risks, slow recovery, and operational inefficiency, making manual infrastructure management unsuitable for modern financial systems.

1.2 Project Objectives

The primary objective of this project is to implement Infrastructure as Code (IaC) by defining and provisioning complete cloud infrastructure on Amazon Web Services using Terraform. By writing infrastructure configurations in code, the system ensures 100% repeatability, consistency across environments, and elimination of manual provisioning errors. Any infrastructure can be recreated at any time with the same configuration, reducing environmental drift and improving reliability.

Another key objective is to adopt the concept of Immutable Infrastructure by packaging the finance application inside Docker containers. This guarantees that the application runs identically across development, testing, and production environments. Containers bundle the application along with its dependencies, removing compatibility issues and ensuring predictable behavior regardless of the host system.

The project also aims to implement Automated Configuration Management using Ansible. Ansible will handle software installation, server configuration, and security hardening automatically after infrastructure provisioning. This eliminates manual "snowflake" servers—systems that are individually configured and difficult to replicate—ensuring standardized and secure setups across all environments.

Finally, the project seeks to integrate a fully automated CI/CD pipeline using GitHub Actions. The pipeline will automatically trigger on code pushes, build and test the application, and deploy updates safely and efficiently. This ensures rapid delivery of financial system updates while maintaining high reliability, security, and operational stability.

1.3 Scope of the Project

The scope of this project includes the design and deployment of a secure three-tier architecture consisting of a frontend dashboard layer, a backend business logic layer built using Django, and a fully automated infrastructure layer. The architecture is structured to ensure separation of concerns, enhanced security, scalability, and efficient handling of financial transactions and analytical operations.

The project covers the complete lifecycle of the system, starting from source code management on GitHub and extending to automated infrastructure provisioning on Amazon Web Services. Infrastructure components include

the creation and configuration of Virtual Private Clouds (VPCs), subnets, routing mechanisms, security groups, and

Amazon EC2 instances. All resources are provisioned using Infrastructure as Code principles to ensure repeatability, consistency, and secure deployment.

Additionally, the backend system is optimized for secure financial data processing and mathematical analysis using NumPy. The scope emphasizes automation, security hardening, containerization, and CI/CD integration to support reliable and high-performance financial operations in a cloud-based environment.

Chapter 2: Proposed Solution

2.1 Key Features

The system provides Automated Cloud Provisioning, enabling one-click deployment of cloud resources on Amazon Web Services using Terraform. This ensures that infrastructure such as VPCs, subnets, and virtual servers can be created consistently and repeatedly without manual intervention, reducing errors and deployment time.

It supports Containerized Microservices by packaging application components using Docker. Containerization isolates services and their dependencies, preventing conflicts between libraries or runtime versions, and ensuring that the application behaves identically across development and production environments.

The project implements Configuration Management through Ansible, which automates software installation, server configuration, and security hardening. This eliminates manually configured "snowflake" servers and ensures standardized, secure infrastructure setups.

For scalability and secure session handling, the system uses Stateless Scaling with JSON Web Tokens (JWT). JWT-based authentication allows secure user session management without storing session data on the server, making horizontal scaling efficient and reliable.

Finally, the solution integrates Automated CI/CD using GitHub Actions. The pipeline automatically builds, tests, and deploys updates whenever code changes are pushed, enabling fast, secure, and continuous delivery of financial system improvements.

2.2 Overall Architecture / Workflow

The overall architecture of the system is designed as a fully automated, secure, and scalable pipeline that

connects development, infrastructure provisioning, and production deployment in a seamless flow. It follows a structured multi-layer approach to ensure reliability, consistency, and high availability for financial operations.

Developer Layer:
Developers write application code, infrastructure definitions, and configuration scripts on their local machines. The backend logic is developed using Django and related Python tools, while infrastructure is defined using Terraform and configuration scripts using Ansible. Once the code is tested locally, it is pushed to a centralized repository hosted on GitHub. This repository acts as the single source of truth for both application and infrastructure code, ensuring version control, collaboration, and traceability.

CI/CD Automation Layer:
When new code is pushed to the repository, GitHub Actions automatically triggers a predefined workflow. This pipeline performs multiple automated steps such as code validation, dependency installation, application build, and Docker image creation using Docker. It also validates Terraform scripts to ensure infrastructure configurations are correct before deployment. Automated testing ensures that only stable and secure updates proceed to the deployment stage, reducing human intervention and minimizing risk.

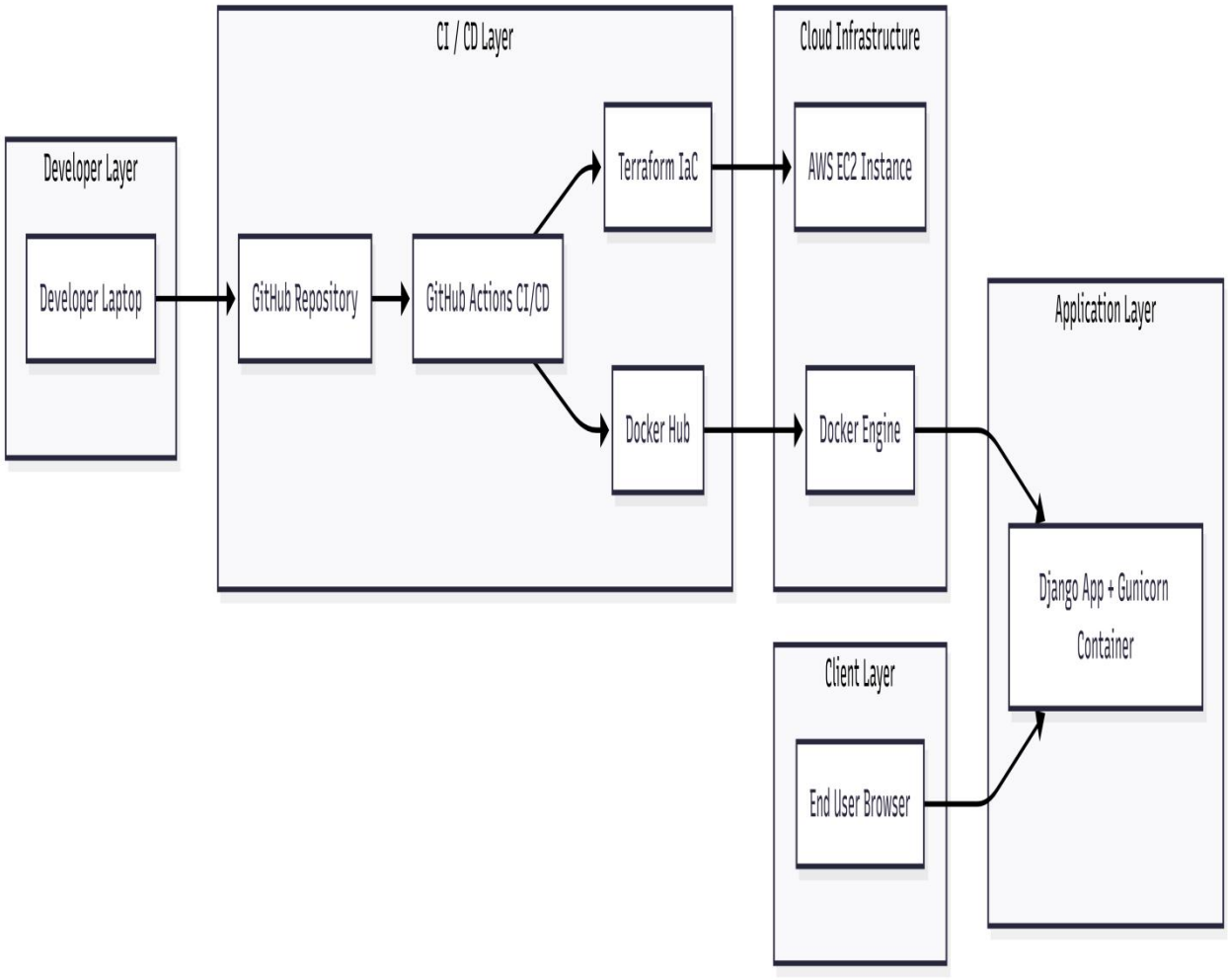
Infrastructure Provisioning Layer:
After successful validation, Terraform provisions the required infrastructure on Amazon Web Services. This includes creating a secure Virtual Private Cloud, subnets (public and private), route tables, internet gateways, and Security Groups. Virtual servers are launched using Amazon EC2, and storage volumes are attached as needed. Since the infrastructure is defined as code, it can be recreated consistently at any time, ensuring disaster recovery and eliminating environmental drift.

Configuration & Deployment Layer:
Once the EC2 instances are created, Ansible automatically connects to them to install required software such as Docker and security updates. The EC2 server then pulls the latest Docker image built during the CI/CD process and deploys the application. The backend runs using Django served by Gunicorn, ensuring efficient handling of concurrent financial requests. Environment variables and secrets are securely managed to protect sensitive financial data.

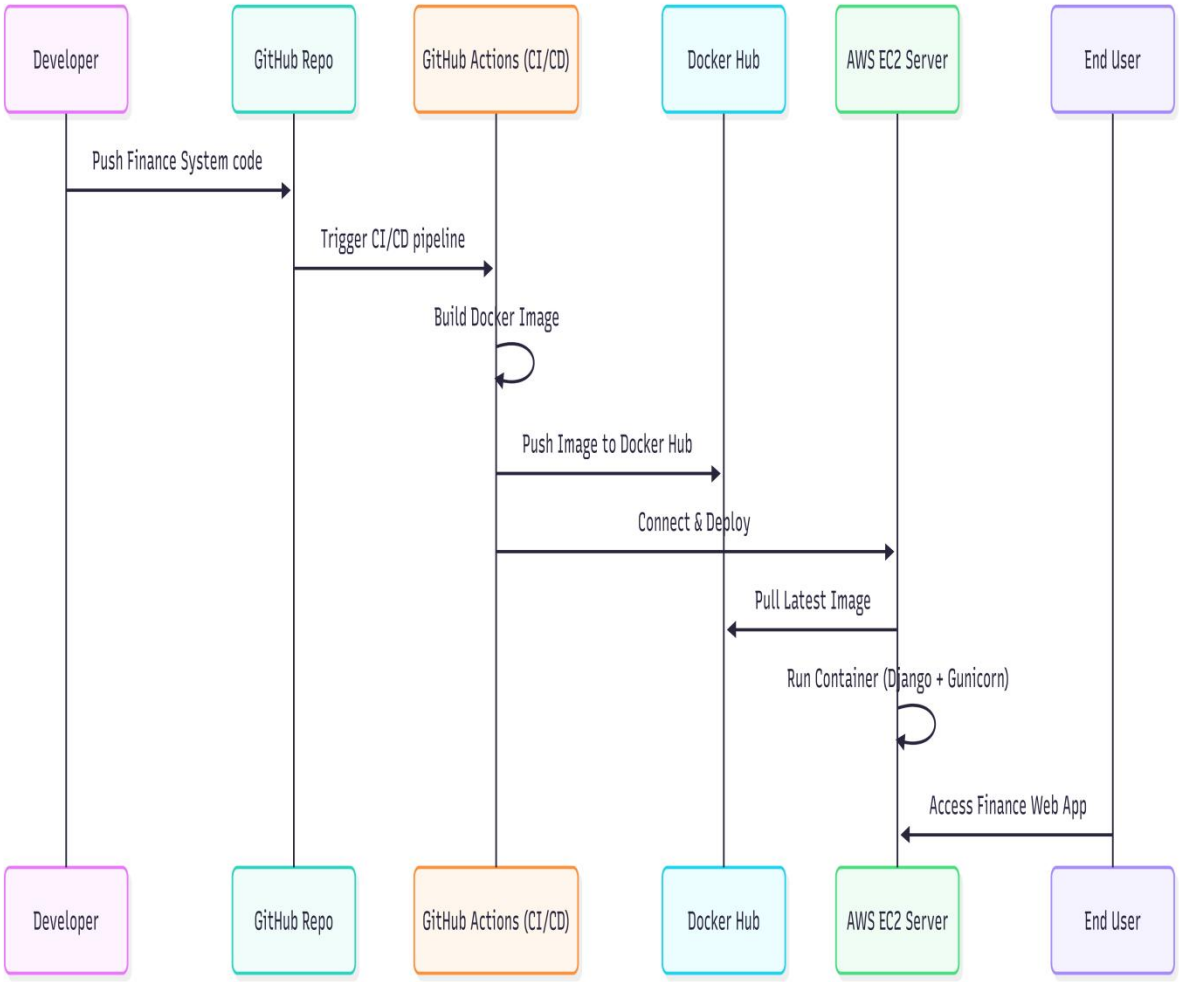
Application & User Access Layer:
After deployment, the frontend dashboard communicates with the backend APIs over secure HTTP/HTTPS protocols. The system is designed to support stateless authentication (such as JWT), enabling horizontal scaling if traffic increases. Users can securely access financial dashboards, perform transactions, and process analytical operations without affecting system stability.

Overall, this architecture ensures a complete automation cycle—from code commit to production deployment—while maintaining high security, scalability, fault tolerance, and operational efficiency. It eliminates manual configuration errors and provides a reliable infrastructure foundation for high-stakes financial applications.

HIGH LEVEL DIAGRAM:



DEPLOYEMENT DIAGRAM:



The Deployment Flow diagram provides a temporal view of how data and commands move through the system in real-time. It outlines the specific sequence of events required to transition a code update into a live production feature:

Step 1: Code Integration: The Developer pushes the Finance System source code to the GitHub Repo. This

serves as the single source of truth for both application logic and infrastructure definitions.

Step 2: Pipeline Trigger: The GitHub Repo sends a webhook signal to GitHub Actions, which automatically initiates the build process.

Step 3: Containerization: GitHub Actions executes a "Build Docker Image" task. This bundles the Python backend, NumPy libraries, and Django framework into a portable image.

Step 4: Registry Update: The newly built image is pushed to Docker Hub with a unique version tag, ensuring a version-controlled history of all deployments.

Step 5: Infrastructure Connection: GitHub Actions communicates with the AWS EC2 Server to signal a deployment update.

Step 6: Image Pull & Execution: The AWS EC2 Server pulls the latest image from Docker Hub. The existing container is stopped, and the new Django + Gunicorn container is started.

Step 7: Final Access: Once the container is healthy, the End User can access the updated Finance Web App through their browser without manual intervention from the development team.

2.3 Tools & Technologies Used

Frontend:
The user interface is developed using HTML5, CSS3, and JavaScript to create a responsive and interactive dashboard for financial data visualization and user interaction.

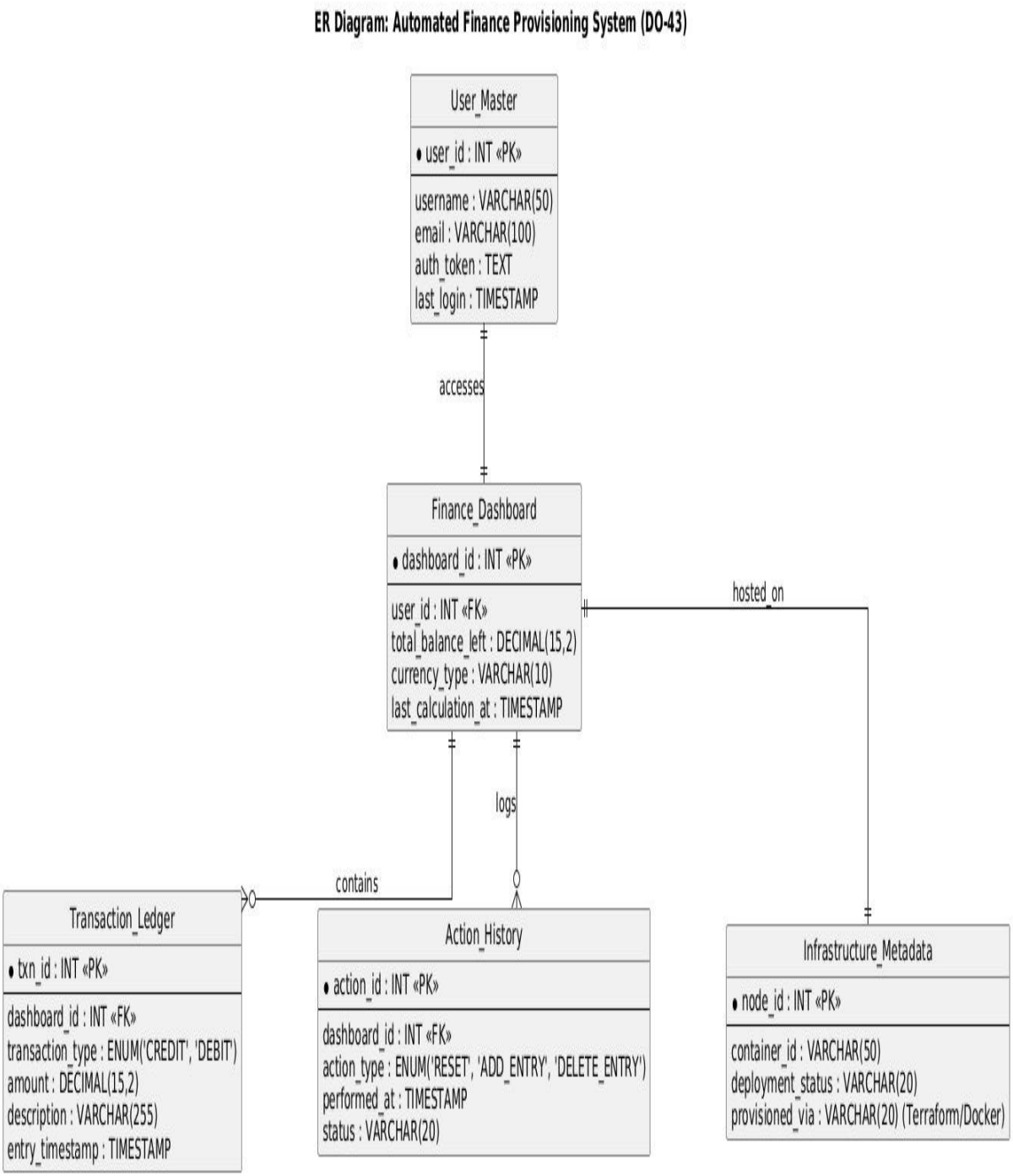
Backend:
The server-side logic is implemented using Python with frameworks such as Django and FastAPI for handling APIs and business logic. The application is deployed using Gunicorn as the WSGI application server, while NumPy is used for efficient financial calculations and mathematical analysis.

DevOps / Infrastructure as Code (IaC):
Infrastructure provisioning is handled using Terraform. Configuration management and server automation are performed using Ansible. Application containerization is achieved with Docker, and container orchestration and scaling are managed using Kubernetes (K8s).

CI/CD & Version Control:
Source code management is done using Git and hosted on GitHub. Continuous Integration and Continuous Deployment are automated using GitHub Actions to ensure smooth, reliable, and automated updates.

Cloud Platform:
The infrastructure is deployed on Amazon Web Services using services such as Amazon EC2 for compute resources, Amazon VPC for secure networking, Amazon EBS for storage, and AWS Identity and Access Management (IAM) for access control and security management.

Entity Relationship Diagram (ER Diagram)



The Entity-Relationship (ER) Diagram of the Automated Finance Provisioning System (Project DO-43) represents a structured, normalized, and audit-ready database design. The schema is designed according to Third Normal Form (3NF) principles to eliminate redundancy, maintain data

integrity, and ensure scalability for high-volume financial operations. The model integrates user management, financial tracking, infrastructure metadata, and auditing mechanisms into a unified relational structure.

1. User_Master Entity

The User_Master table serves as the core authentication and identity management entity of the system.

Primary Key:

user_id (INT, PK)

Attributes:

username – Stores unique login names.

email – Used for communication and identity verification.

auth_token – Stores authentication tokens for secure session handling (e.g., JWT).

last_login – Tracks user activity for monitoring and auditing.

Role in Architecture:

This entity controls access to the financial dashboard and ensures that only authenticated users interact with financial data. It forms a one-to-many relationship with the Finance_Dashboard table, meaning one user can access or own multiple dashboards if required.

2. Finance_Dashboard Entity

The Finance_Dashboard entity represents the main operational interface where financial data is aggregated and displayed.

Primary Key:

dashboard_id (INT, PK)

Foreign Key:

user_id (INT, FK → User_Master)

Attributes:

total_balance_left – Stores computed financial balance.

currency_type – Defines transaction currency (e.g., INR, USD).

last_calculation_at – Timestamp of the most recent financial computation.

Functional Importance:

This table acts as the central financial aggregation unit. It maintains a real-time snapshot of a user's financial state and links transactional and action-based logs. Every financial transaction and system action is associated with a dashboard for traceability.

3. Transaction_Ledger Entity The Transaction_Ledger table records all monetary activities within the system.

Primary Key:

txn_id (INT, PK)

Foreign Key:

dashboard_id (INT, FK → Finance_Dashboard)

Attributes:

transaction_type – ENUM ('CREDIT', 'DEBIT')

amount – Stores financial value with precision (DECIMAL 15,2).

description – Provides contextual explanation of transaction.

entry_timestamp – Records the exact time of entry.

Significance:

This entity ensures financial transparency and audit readiness. Every credit and debit is logged with timestamp and description, forming a complete ledger. Since it references the dashboard, it supports a one-to-many relationship where one dashboard contains multiple transactions.

This structure ensures:

Accurate balance calculation

Historical tracking

Compliance with financial auditing standards

4. Action_History Entity

The Action_History table logs system-level and user-level operations performed within the dashboard.

Primary Key:

action_id (INT, PK)

Foreign Key:

dashboard_id (INT, FK → Finance_Dashboard)

Attributes:

action_type – ENUM ('RESET', 'ADD_ENTRY', 'DELETE_ENTRY')

performed_at – Timestamp of action execution

status – Tracks execution outcome (SUCCESS/FAILED)

Purpose:

This entity ensures operational accountability. While Transaction_Ledger captures monetary activity, Action_History captures behavioral and system modifications. It strengthens compliance and debugging capability by logging every operational change.

It creates a one-to-many relationship: One dashboard logs multiple actions.

5. Infrastructure_Metadata Entity

The Infrastructure_Metadata entity bridges application data with DevOps automation and cloud provisioning details.

Primary Key:

node_id (INT, PK)

Attributes:

container_id – Identifies Docker container instance

deployment_status – Indicates running, stopped, or failed

provisioned_via – Specifies provisioning tool (Terraform/Docker)

Strategic Importance:

This table integrates Infrastructure as Code (IaC) principles directly into the database model. It tracks where and how the finance application is deployed. This is crucial for:

•

Audit compliance

Infrastructure traceability

DevOps monitoring

Disaster recovery validation

It represents the "hosted_on" relationship from Finance_Dashboard to infrastructure nodes, meaning dashboards are deployed on specific provisioned cloud resources.

Relationship Summary

1. User_Master → Finance_Dashboard
   One-to-Many relationship.
   A single user can manage multiple dashboards.
2. Finance_Dashboard → Transaction_Ledger
   One-to-Many relationship.
   A dashboard contains multiple financial transactions.
3. Finance_Dashboard → Action_History
   One-to-Many relationship.
   A dashboard logs multiple operational actions.
4. Finance_Dashboard → Infrastructure_Metadata
   One-to-One or Many-to-One (depending on scaling model).
   Indicates which infrastructure node hosts the dashboard instance.

Normalization and Data Integrity

The schema follows 3NF principles:

No repeating groups

No partial dependencies

No transitive dependencies

Each entity has a single responsibility:

Authentication → User_Master

Financial aggregation → Finance_Dashboard

Monetary transactions → Transaction_Ledger

Operational logs → Action_History

Infrastructure tracking → Infrastructure_Metadata

This ensures:

Reduced redundancy

Improved scalability

Strong referential integrity via foreign keys

Simplified auditing and compliance reporting

Security & Audit Readiness

The ER model is designed specifically for financial-grade applications:

Every transaction is timestamped.

Every action is logged.

User access is tracked.

Infrastructure state is recorded.

This enables:

Financial reconciliation

Regulatory auditing

Disaster recovery traceability

Infrastructure change monitoring

Overall Architectural Impact

The ER Diagram is not just a database design; it reflects the entire philosophy of the automated finance provisioning system:

Integration of DevOps metadata with business logic

Full traceability from user interaction to infrastructure node

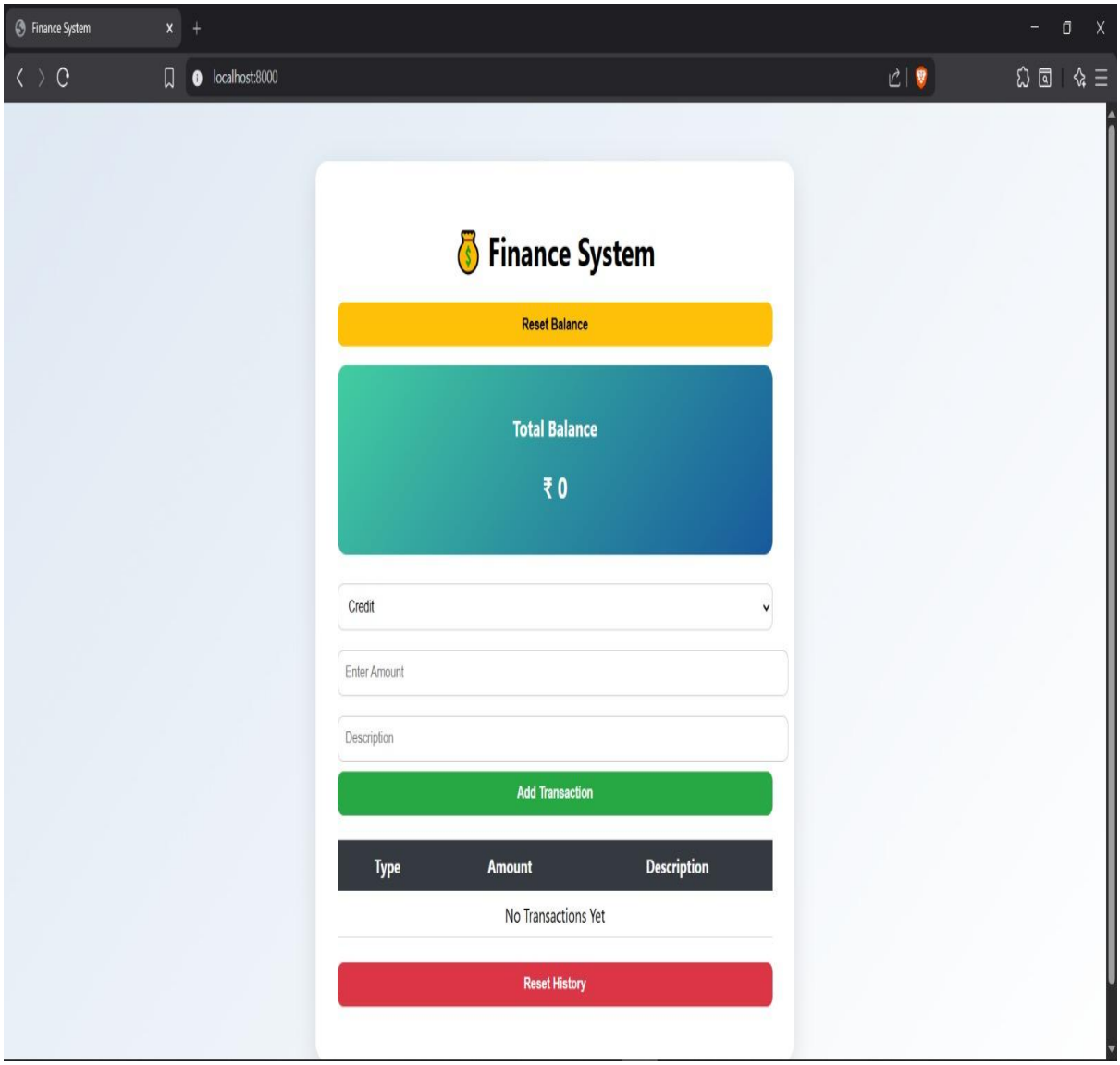
Strong separation of concerns

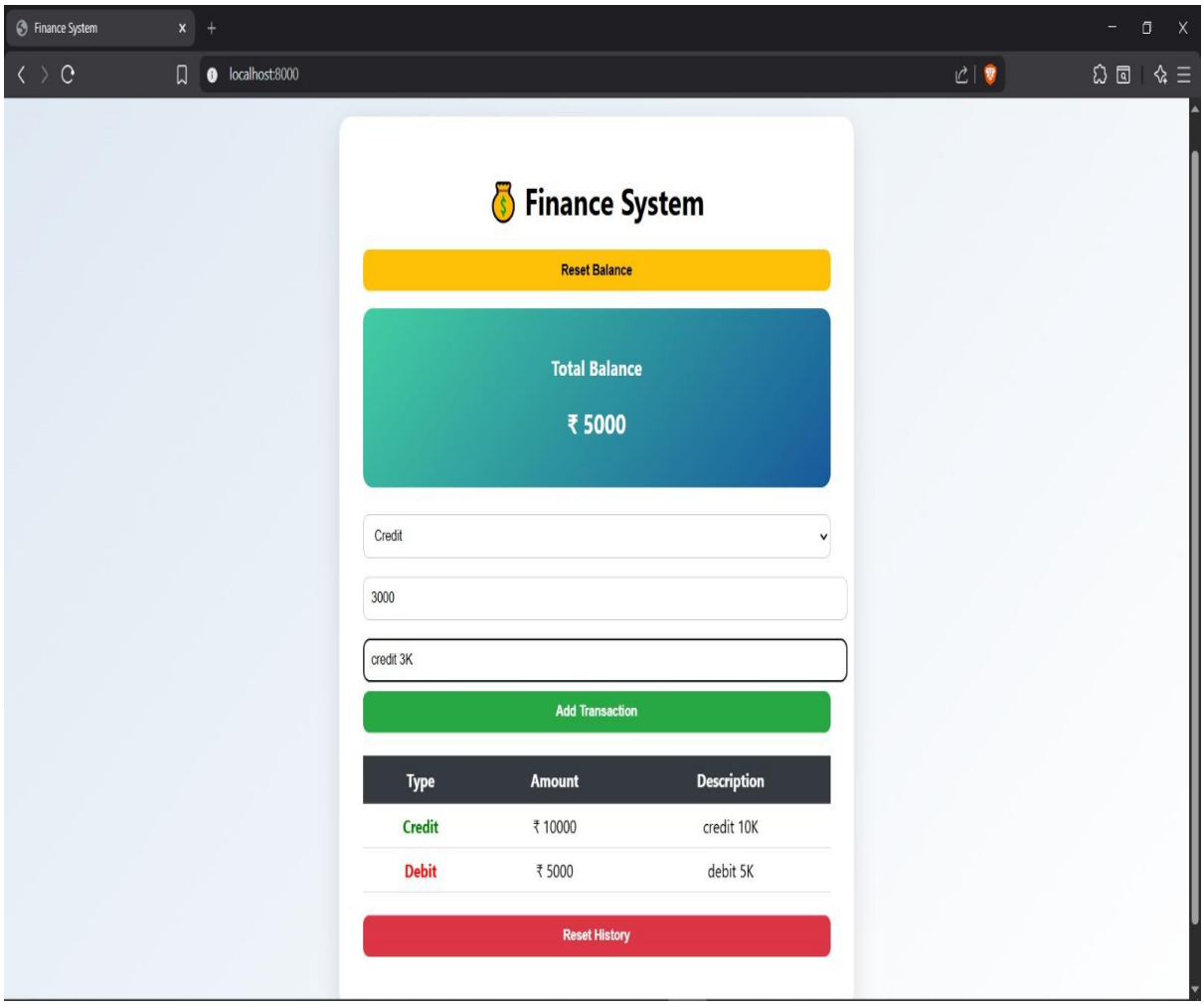
Scalable multi-user financial environment

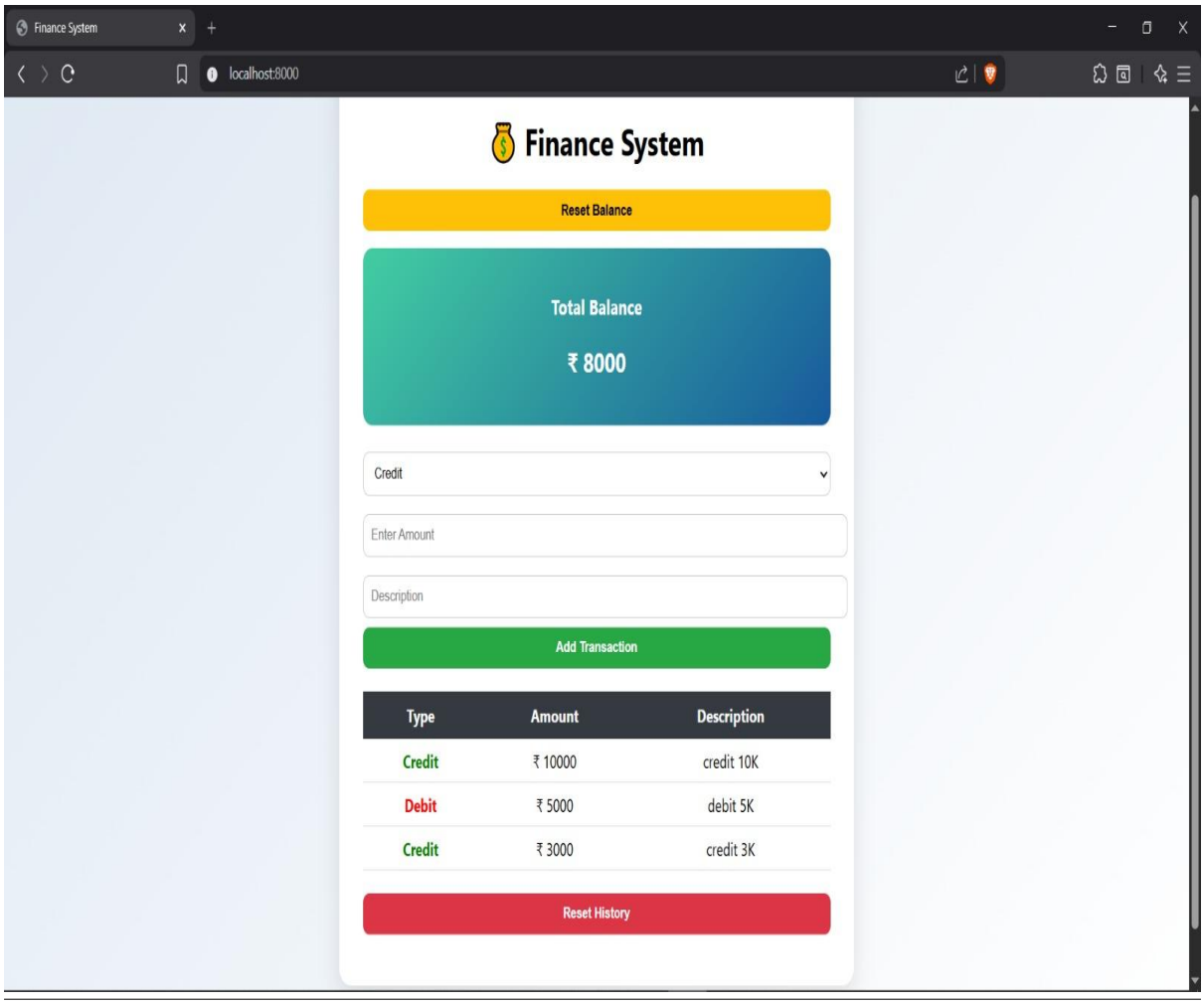
By combining transactional logging, action auditing, and infrastructure metadata within a normalized relational schema, the system achieves high reliability, compliance readiness, and operational transparency—making it suitable for modern cloud-based financial applications.

Chapter-3: Result and Output

3.1 Screenshots / Outputs :




Finance System
localhost:8000
Finance System
Reset Balance
Total Balance
₹ 5000
Credit
3000
credit 3K
Add Transaction
Type
Amount
Description
Credit
₹ 10000
credit 10K
Debit
₹ 5000
debit 5K
Reset History


Finance System
localhost:8000
Finance System
Reset Balance
Total Balance
₹ 8000
Credit
Enter Amount
Description
Add Transaction
Type
Amount
Description
Credit
₹ 10000
credit 10K
Debit
₹ 5000
debit 5K
Credit
₹ 3000
credit 3K
Reset History

Infrastructure as Code (IaC) provisioning for a finance system refers to the automated creation, configuration, and management of cloud infrastructure using machine-readable definition files instead of manual setup. In high-stakes financial environments where accuracy, security, and uptime are critical, IaC ensures that infrastructure is consistent, repeatable, and secure across all stages of development and production.

In this project, infrastructure is provisioned on Amazon Web Services using Terraform. All cloud resources such as Virtual Private Clouds (VPCs), subnets, route tables, internet gateways, Security Groups, and Amazon EC2 instances are defined in Terraform configuration files. These files describe the desired state of the infrastructure. When executed, Terraform automatically creates and configures the resources exactly as defined, eliminating manual errors and configuration drift.

For a finance system, this approach offers several major advantages. First, it guarantees environment consistency. The same Terraform code can be used to create development, testing, and production environments that are identical in configuration. This prevents issues caused by mismatched software versions or network settings. Second, it enhances security by codifying firewall rules, access controls, and IAM policies directly in the infrastructure scripts. Security configurations are no longer dependent on manual setup, reducing the risk of open ports or misconfigured permissions.

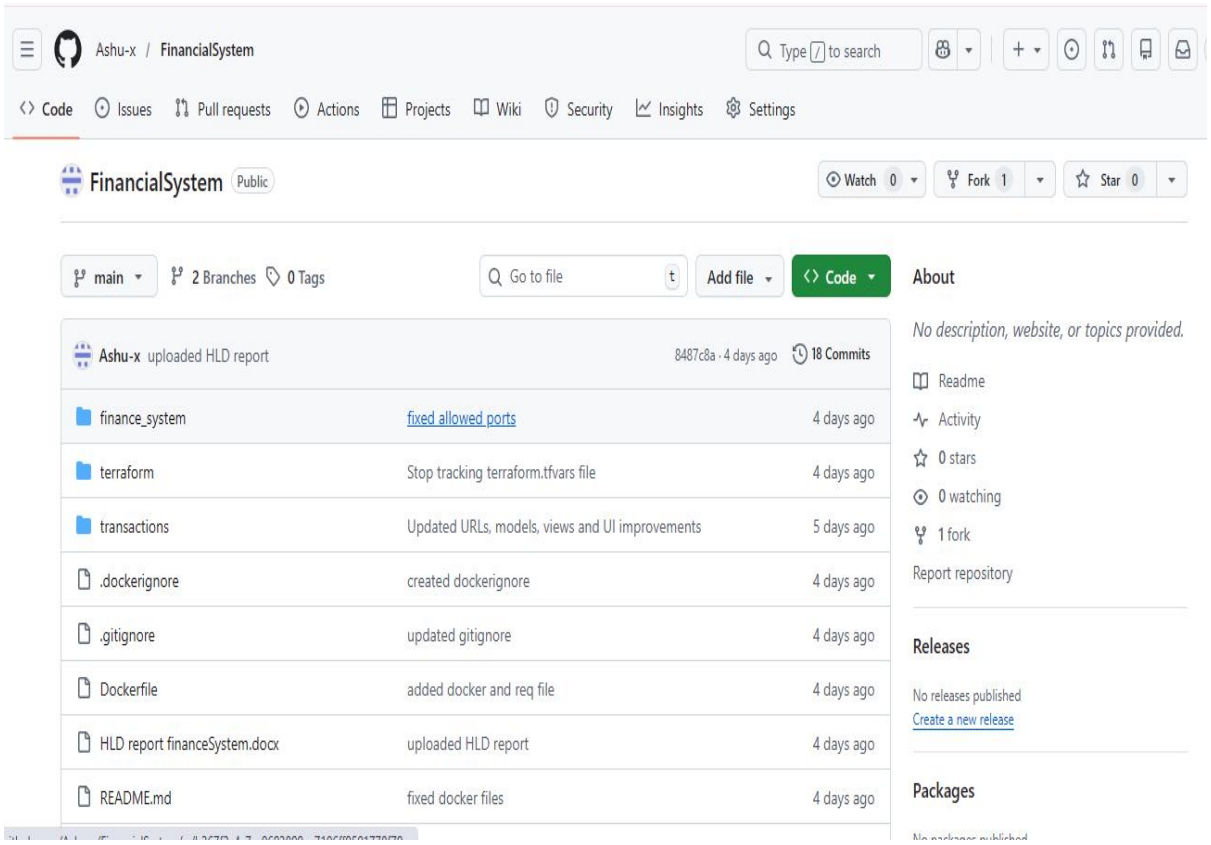
IaC also improves disaster recovery and scalability. If a server fails, the entire infrastructure can be recreated within minutes by re-running the Terraform scripts. New environments can be provisioned instantly to handle increased financial transaction loads. This is essential for systems that require high availability and near-zero downtime.

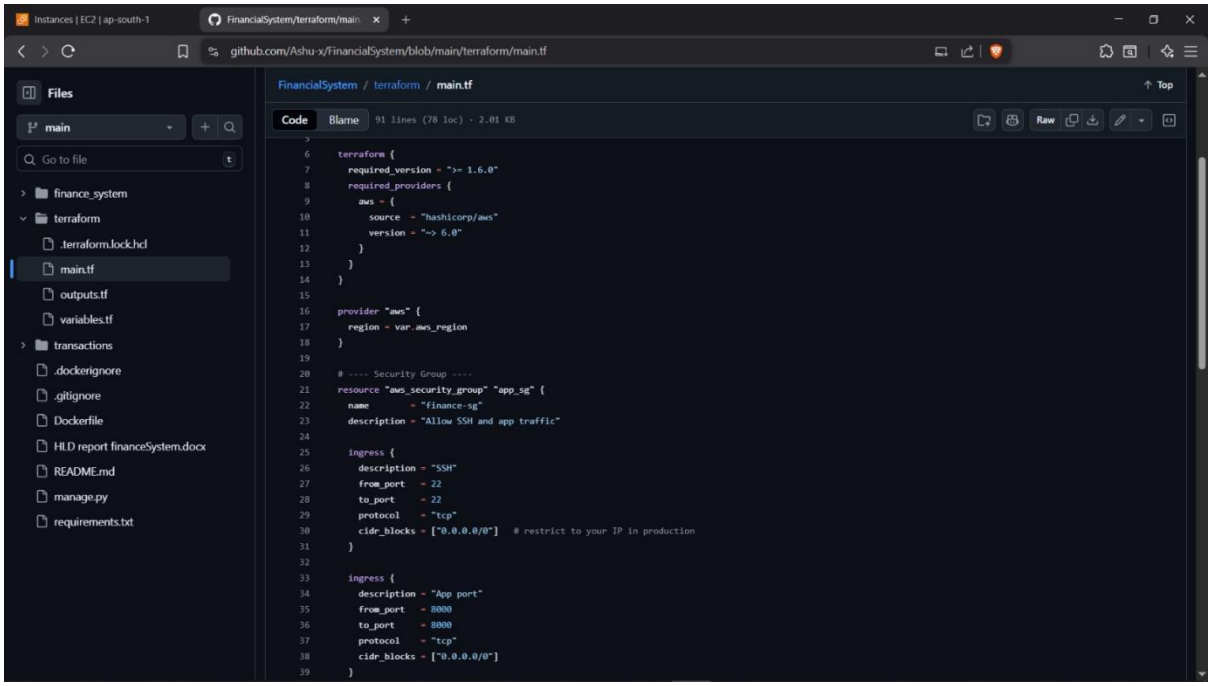
Additionally, IaC integrates seamlessly with CI/CD pipelines such as GitHub Actions. Infrastructure changes

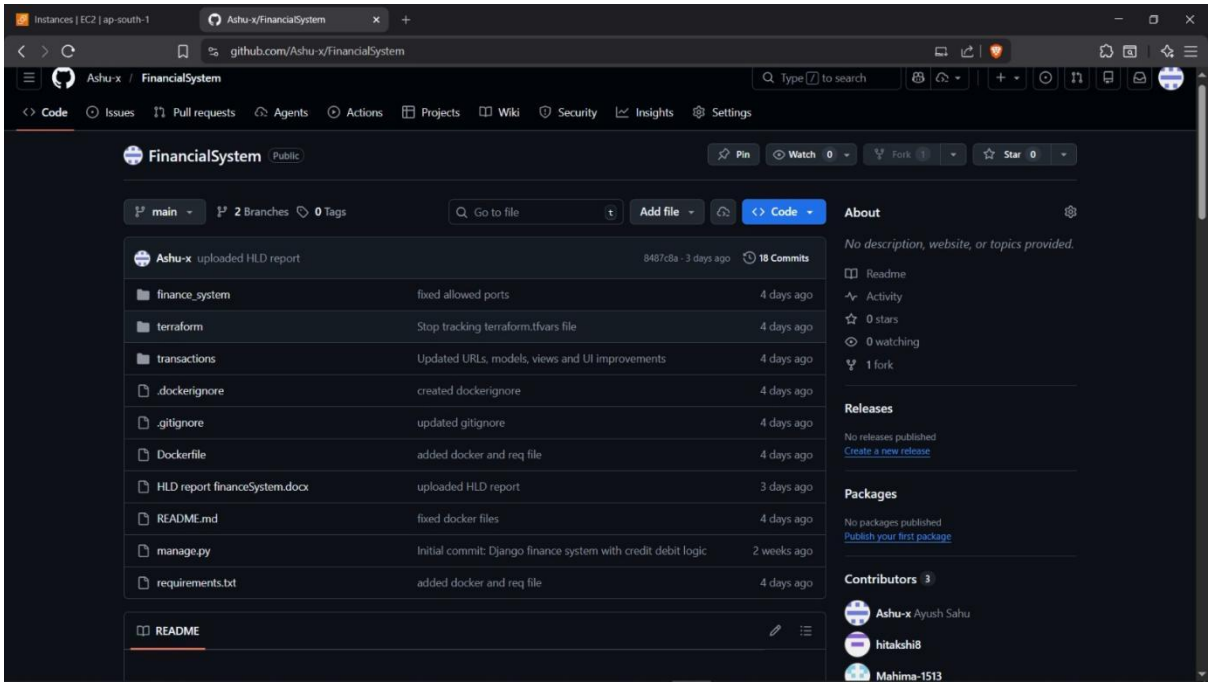
can be version-controlled in GitHub, reviewed, and automatically validated before deployment. This ensures controlled, auditable, and secure infrastructure updates.

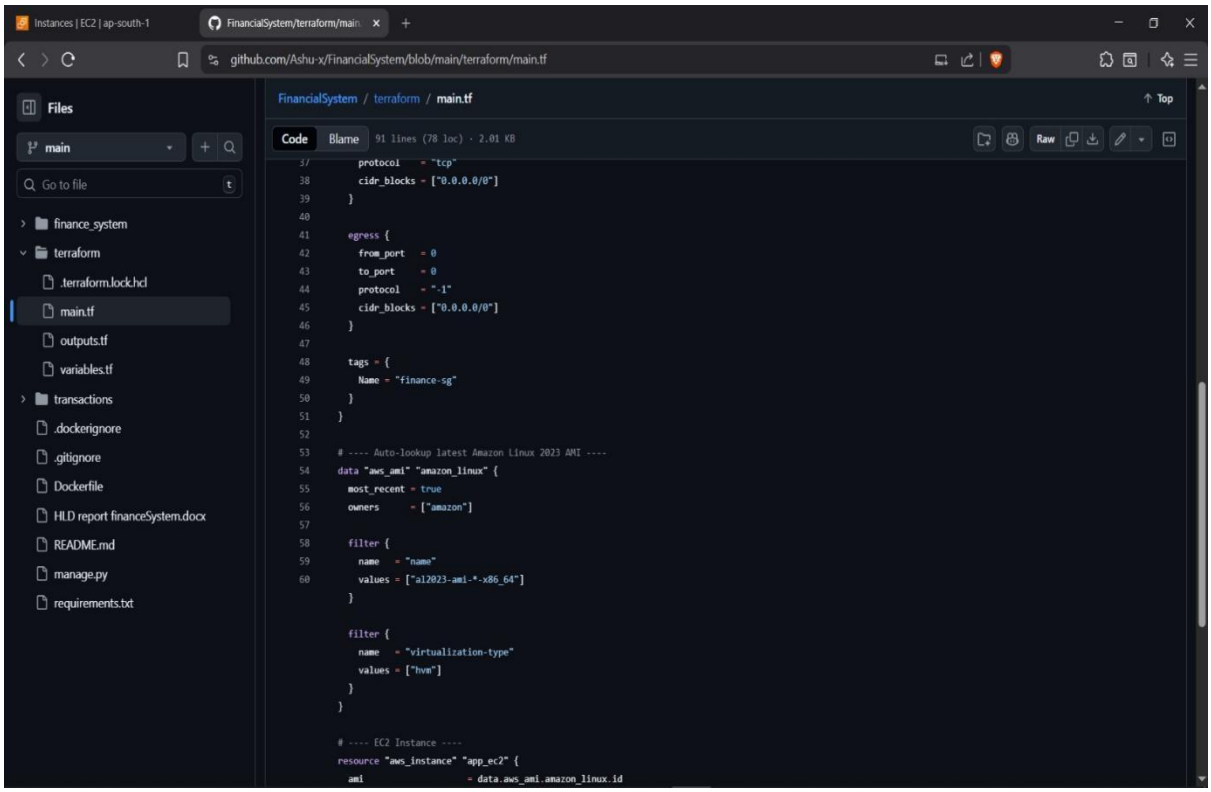
Overall, IaC provisioning transforms the finance system from a manually managed setup into a fully automated, secure, scalable, and resilient cloud environment. It reduces operational risks, speeds up deployments, and provides a strong foundation for reliable financial data processing and transaction management.

3.2 Models

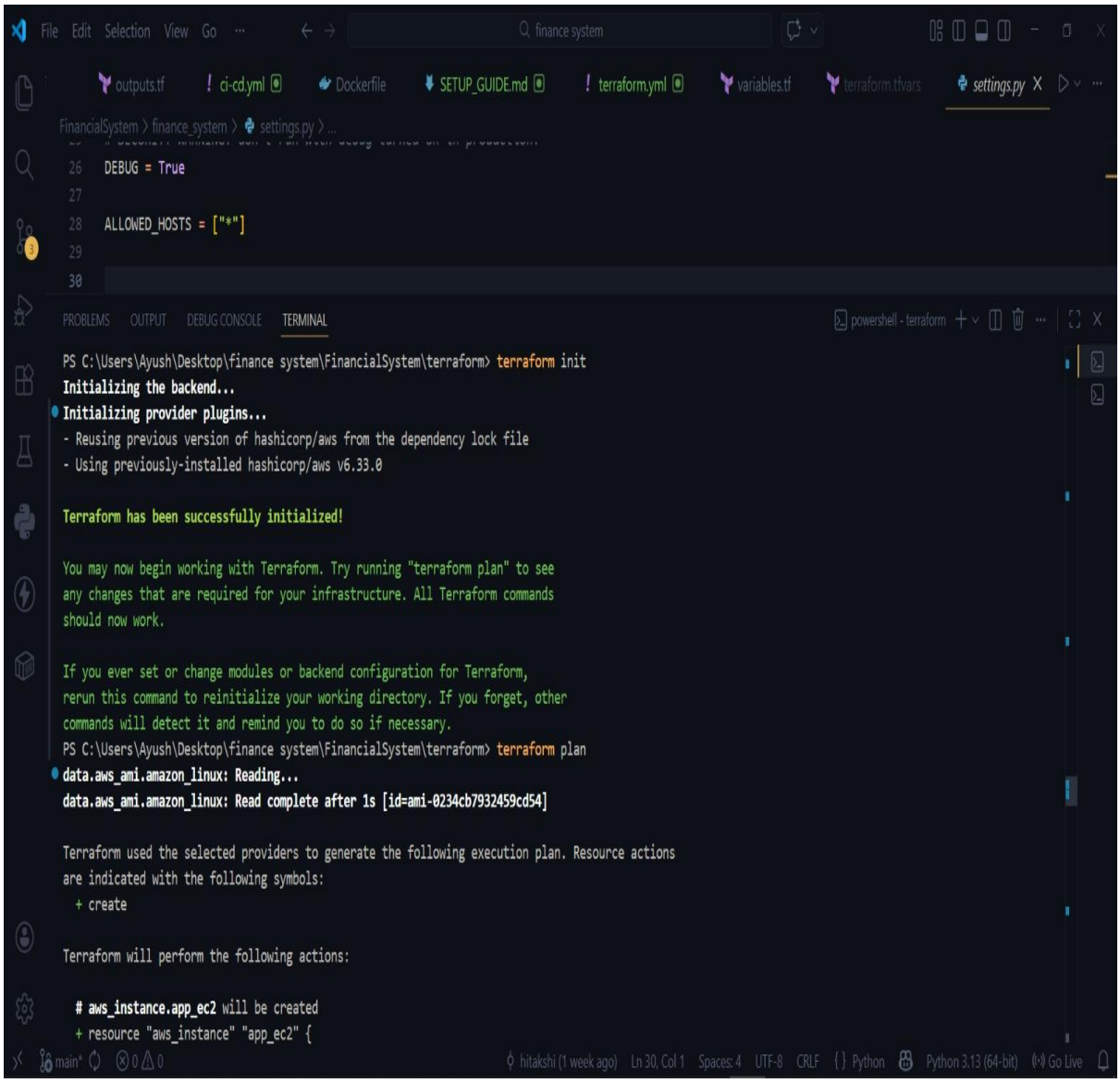

Ashu-x / FinancialSystem
Type / to search
Code
Issues
Pull requests
Actions
Projects
Wiki
Security
Insights
Settings
FinancialSystem
Public
Watch 0
Fork 1
Star 0
main
2 Branches
0 Tags
Go to file
Add file
Code
About
No description, website, or topics provided.
Ashu-x uploaded HLD report
8487c8a · 4 days ago
18 Commits
finance_system
fixed allowed ports
4 days ago
terraform
Stop tracking terraform.tfvars file
4 days ago
transactions
Updated URLs, models, views and UI improvements
5 days ago
.dockerignore
created dockerignore
4 days ago
.gitignore
updated gitignore
4 days ago
Dockerfile
added docker and req file
4 days ago
HLD report financeSystem.docx
uploaded HLD report
4 days ago
README.md
fixed docker files
4 days ago
Readme
Activity
0 stars
0 watching
1 fork
Report repository
Releases
No releases published
Create a new release
Packages

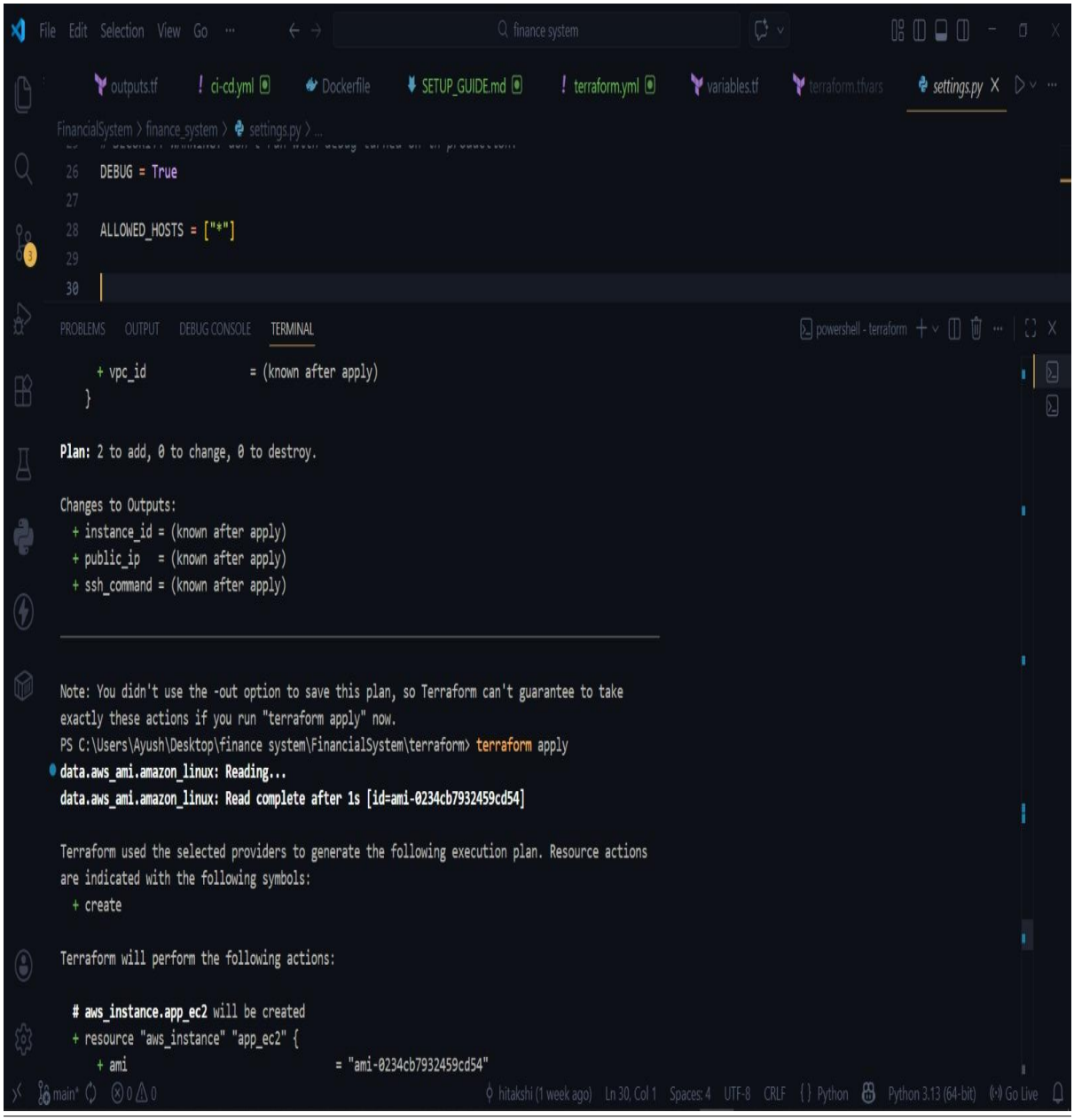

github.com/Ashu-x/FinancialSystem/blob/main/terraform/main.tf
Files
main
Go to file
finance_system
terraform
.terraform.lock.hcl
main.tf
outputs.tf
variables.tf
transactions
.dockerignore
.gitignore
Dockerfile
HLD report financeSystem.docx
README.md
manage.py
requirements.txt
FinancialSystem / terraform / main.tf
Code
Blame
91 lines (78 loc) · 2.01 KB
Raw
terraform {
required_version = ">= 1.6.0"
required_providers {
aws = {
source = "hashicorp/aws"
version = "~> 6.0"
}
}
}
provider "aws" {
region = var.aws_region
}
# ---- Security Group ----
resource "aws_security_group" "app_sg" {
name = "finance-sg"
description = "Allow SSH and app traffic"
ingress {
description = "SSH"
from_port = 22
to_port = 22
protocol = "tcp"
cidr_blocks = ["0.0.0.0/0"] # restrict to your IP in production
}
ingress {
description = "App port"
from_port = 8000
to_port = 8000
protocol = "tcp"
cidr_blocks = ["0.0.0.0/0"]
}

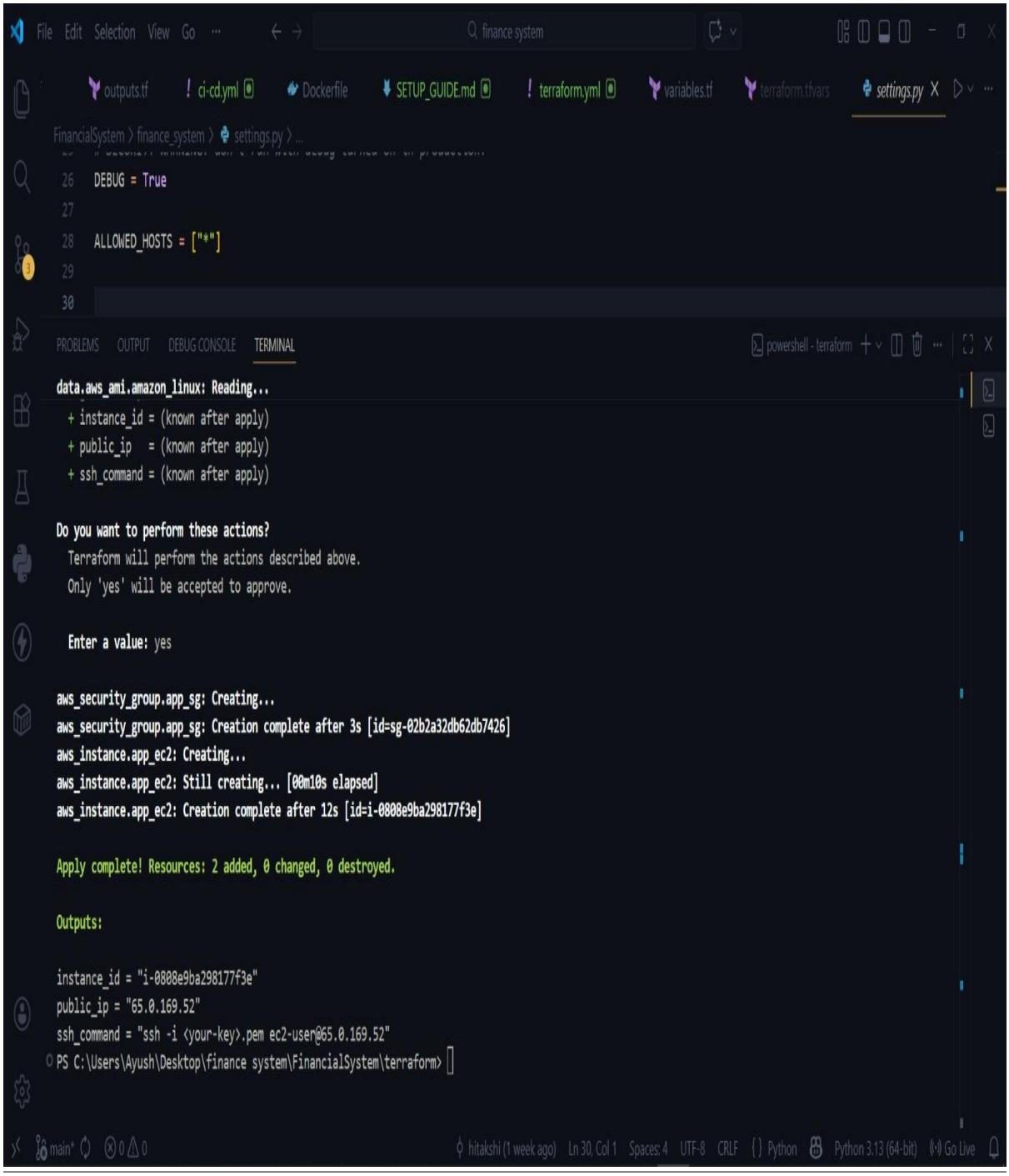

Instances | EC2 | ap-south-1
Ashu-x/FinancialSystem
github.com/Ashu-x/FinancialSystem
Ashu-x / FinancialSystem
Code Issues Pull requests Agents Actions Projects Wiki Security Insights Settings
FinancialSystem Public
Pin Watch 0 Fork 1 Star 0
main 2 Branches 0 Tags Go to file Add file Code
Ashu-x uploaded HLD report 8487c8a · 3 days ago 18 Commits
finance_system fixed allowed ports 4 days ago
terraform Stop tracking terraform.tfvars file 4 days ago
transactions Updated URLs, models, views and UI improvements 4 days ago
.dockerignore created dockerignore 4 days ago
.gitignore updated gitignore 4 days ago
Dockerfile added docker and req file 4 days ago
HLD report financeSystem.docx uploaded HLD report 3 days ago
README.md fixed docker files 4 days ago
manage.py Initial commit: Django finance system with credit debit logic 2 weeks ago
requirements.txt added docker and req file 4 days ago
README
About
No description, website, or topics provided.
Readme
Activity
0 stars
0 watching
1 fork
Releases
No releases published
Create a new release
Packages
No packages published
Publish your first package
Contributors 3
Ashu-x Ayush Sahu
hitakshi8
Mahima-1513


Instances | EC2 | ap-south-1
FinancialSystem/terraform/main
github.com/Ashu-x/FinancialSystem/blob/main/terraform/main.tf
Files
main
Go to file
finance_system
terraform
.terraform.lock.hcl
main.tf
outputs.tf
variables.tf
transactions
.dockerignore
.gitignore
Dockerfile
HLD report financeSystem.docx
README.md
manage.py
requirements.txt
FinancialSystem / terraform / main.tf
Top
Code Blame 91 lines (78 loc) · 2.01 KB
Raw
protocol = "tcp"
cidr_blocks = ["0.0.0.0/0"]
}
egress {
from_port = 0
to_port = 0
protocol = "-1"
cidr_blocks = ["0.0.0.0/0"]
}
tags = {
Name = "finance-sg"
}
}
# ---- Auto-lookup latest Amazon Linux 2023 AMI ----
data "aws_ami" "amazon_linux" {
most_recent = true
owners = ["amazon"]
filter {
name = "name"
values = ["al2023-ami-*-x86_64"]
}
filter {
name = "virtualization-type"
values = ["hvm"]
}
}
# ---- EC2 Instance ----
resource "aws_instance" "app_ec2" {
ami = data.aws_ami.amazon_linux.id

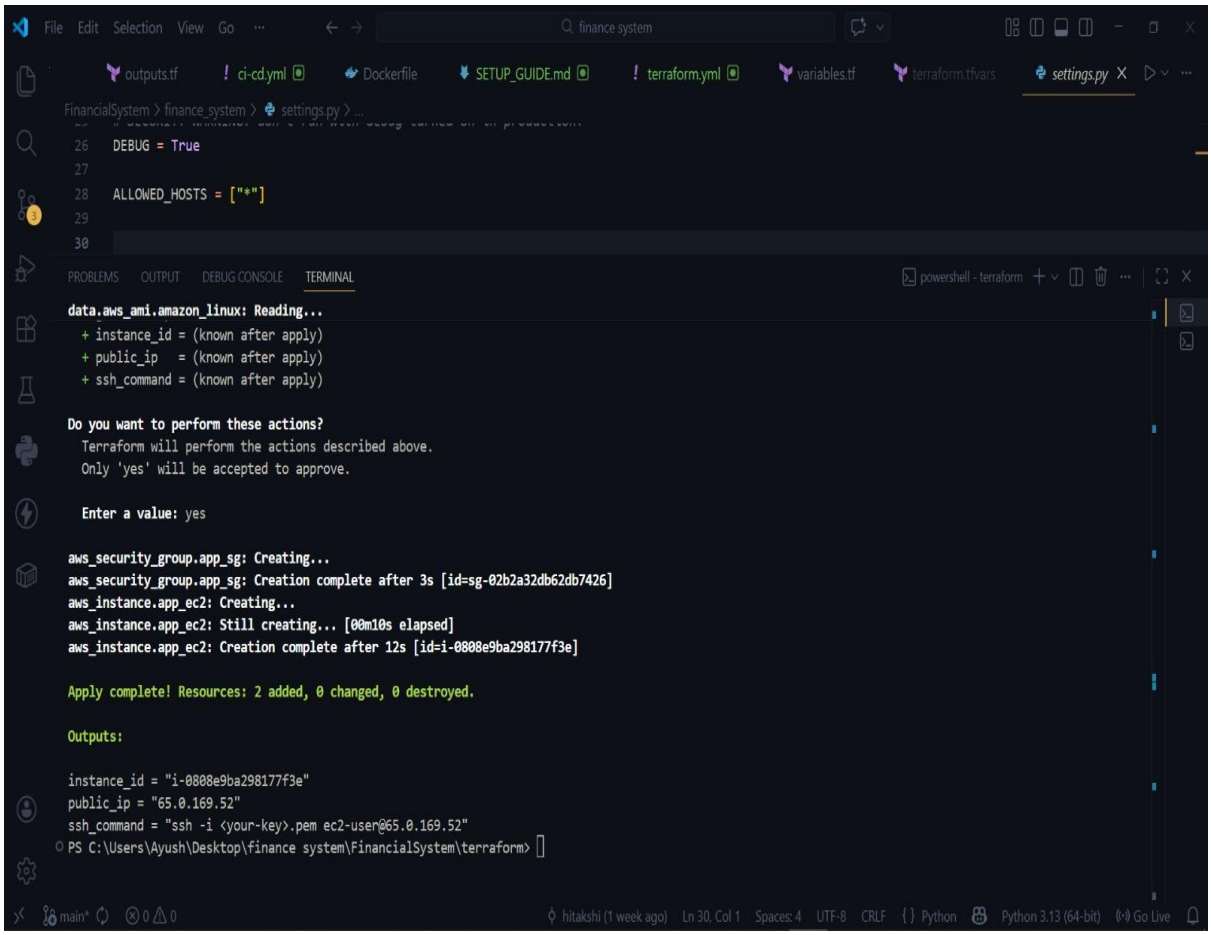
System Execution:




File Edit Selection View Go
finance system
outputs.tf
ci-cd.yml
Dockerfile
SETUP_GUIDE.md
terraform.yml
variables.tf
terraform.tfvars
settings.py
FinancialSystem > finance_system > settings.py > ...
26 DEBUG = True
27
28 ALLOWED_HOSTS = ["*"]
29
30
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
powershell - terraform
+ vpc_id = (known after apply)
}
Plan: 2 to add, 0 to change, 0 to destroy.
Changes to Outputs:
+ instance_id = (known after apply)
+ public_ip = (known after apply)
+ ssh_command = (known after apply)
Note: You didn't use the -out option to save this plan, so Terraform can't guarantee to take exactly these actions if you run "terraform apply" now.
PS C:\Users\Ayush\Desktop\finance system\FinancialSystem\terraform> terraform apply
data.aws_ami.amazon_linux: Reading...
data.aws_ami.amazon_linux: Read complete after 1s [id=ami-0234cb7932459cd54]
Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following symbols:
+ create
Terraform will perform the following actions:
# aws_instance.app_ec2 will be created
+ resource "aws_instance" "app_ec2" {
+ ami = "ami-0234cb7932459cd54"
main*
hitakshi (1 week ago)
Ln 30, Col 1
Spaces: 4
UTF-8
CRLF
Python
Python 3.13 (64-bit)
Go Live


File Edit Selection View Go
finance system
outputs.tf
ci-cd.yml
Dockerfile
SETUP_GUIDE.md
terraform.yml
variables.tf
terraform.tfvars
settings.py
FinancialSystem > finance_system > settings.py > ...
26 DEBUG = True
27
28 ALLOWED_HOSTS = ["*"]
29
30
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
powershell - terraform
data.aws_ami.amazon_linux: Reading...
+ instance_id = (known after apply)
+ public_ip = (known after apply)
+ ssh_command = (known after apply)
Do you want to perform these actions?
Terraform will perform the actions described above.
Only 'yes' will be accepted to approve.
Enter a value: yes
aws_security_group.app_sg: Creating...
aws_security_group.app_sg: Creation complete after 3s [id=sg-02b2a32db62db7426]
aws_instance.app_ec2: Creating...
aws_instance.app_ec2: Still creating... [00m10s elapsed]
aws_instance.app_ec2: Creation complete after 12s [id=i-0808e9ba298177f3e]
Apply complete! Resources: 2 added, 0 changed, 0 destroyed.
Outputs:
instance_id = "i-0808e9ba298177f3e"
public_ip = "65.0.169.52"
ssh_command = "ssh -i <your-key>.pem ec2-user@65.0.169.52"
PS C:\Users\Ayush\Desktop\finance system\FinancialSystem\terraform>
main*
hitakshi (1 week ago) Ln 30, Col 1 Spaces: 4 UTF-8 CRLF Python Python 3.13 (64-bit) Go Live

System Output:



3.3 Key Outcomes

The implementation of the Automated Finance Provisioning System produced strong technical, operational, and architectural outcomes that demonstrate the effectiveness of Infrastructure as Code and DevOps automation in a financial environment.

The project successfully eliminated environmental drift by defining the entire infrastructure using Terraform and deploying it consistently on Amazon Web Services. Development, staging, and production environments now share identical configurations, ensuring predictable behavior and reducing deployment failures.

A fully automated end-to-end CI/CD pipeline was implemented using GitHub Actions integrated with GitHub. Every code push triggers automated validation, container build, infrastructure checks, and deployment, enabling continuous delivery with minimal human intervention. This significantly reduces release time while maintaining reliability.

The adoption of Docker enabled immutable infrastructure practices. The finance application, including the Django backend and supporting libraries, runs inside portable containers. This guarantees runtime consistency across all hosts and simplifies scaling and rollback procedures.

The system achieved rapid disaster recovery and reproducibility. In case of server failure, infrastructure can be recreated instantly using Terraform scripts, and application containers can be redeployed from versioned images. Recovery time is reduced from hours to minutes, meeting high-availability expectations of financial systems.

Security posture improved significantly through automated network provisioning, structured Security Group rules, and controlled access policies. By integrating infrastructure definitions with code reviews, security misconfigurations such as open ports or excessive permissions are minimized.

The database design ensured full audit traceability. Every transaction, dashboard action, and infrastructure deployment event is logged and timestamped. This supports compliance requirements, financial reconciliation, and regulatory audits, making the system audit-ready.

The architecture demonstrated horizontal scalability. Since the system uses stateless application containers and structured infrastructure, additional compute instances can be provisioned dynamically to handle increased user traffic or transaction loads without affecting performance.

Operational efficiency improved through automation of repetitive tasks such as server setup, dependency installation, container deployment, and configuration management. This reduced manual workload and minimized human error.

The project also strengthened collaboration between development and operations teams by integrating version control, automated workflows, and infrastructure provisioning into a unified DevOps pipeline. This alignment improves transparency, accountability, and deployment confidence.

Overall, the system successfully transformed a traditionally manual financial deployment model into a modern, automated, secure, scalable, and resilient cloud-based architecture. It demonstrates real-world application of IaC, containerization, CI/CD, and cloud-native design principles in the financial domain.

3.4 Challenges faced:

The development and deployment of the Automated Finance Provisioning System involved multiple technical, architectural, security, and operational challenges. Since the project integrates Infrastructure as Code (IaC), CI/CD automation, containerization, and cloud deployment, each layer introduced its own complexities. Below is a comprehensive discussion of the major challenges encountered during the implementation.

1. Environmental Drift and Configuration Consistency

One of the primary challenges was maintaining consistency across development, staging, and production environments. Even minor differences in Python versions, dependency packages, or system libraries caused unexpected runtime errors. Before containerization, local machines behaved differently from cloud servers provisioned on Amazon Web Services.

Although Docker eventually resolved many compatibility issues, initial container builds failed due to missing system dependencies and incorrect base images. Ensuring that the Docker image worked identically across different machines required repeated testing and debugging.

2. Complexity in Terraform Infrastructure Design

Designing infrastructure using Terraform was challenging, particularly when defining secure and scalable network architectures. Creating a properly structured Virtual Private Cloud (VPC) required careful planning of:

Public and private subnets

Route tables

Internet Gateways

NAT configurations

Misconfigured Security Group rules initially caused connectivity failures between EC2 instances and the internet. Additionally, Terraform state management posed difficulties. Managing remote state files securely and preventing accidental state corruption required careful configuration.

Understanding Terraform's dependency graph and resource ordering also took time. Improper references between resources resulted in provisioning errors that required restructuring configuration files.

3. IAM and Security Permissions Issues

Managing permissions through AWS Identity and Access Management was another major challenge. Assigning overly restrictive permissions caused pipeline failures, while overly permissive policies introduced security risks.

Balancing the principle of least privilege with functional requirements required iterative adjustments. Ensuring that the CI/CD pipeline could provision infrastructure without exposing sensitive credentials was particularly complex.

Securely handling AWS access keys in GitHub Actions workflows required encrypted secrets and proper environment variable management.

4. CI/CD Pipeline Failures and Debugging

Setting up a fully automated pipeline introduced challenges related to workflow configuration and debugging. Errors such as:

Incorrect YAML syntax

Missing dependencies

Incorrect environment variables

Docker build failures caused pipeline interruptions.

Since CI/CD runs in isolated virtual environments, debugging failures required analyzing logs carefully. Unlike local development, direct troubleshooting was not possible, making root-cause identification more difficult.

Ensuring proper synchronization between Terraform provisioning and Docker deployment stages required precise workflow orchestration.

5. Containerization and Application Deployment Issues

Packaging the application using Docker required careful configuration of:

Base image selection

Python dependency installation

Gunicorn server configuration

Environment variables

Incorrect port exposure or misconfigured Gunicorn settings prevented the Django application from being accessible externally.

Additionally, handling persistent storage for logs and database files inside containers required proper volume mapping. Without correct configuration, container restarts resulted in data loss during initial testing phases.

6. Networking and Connectivity Problems

Network configuration was one of the most technically challenging aspects. Common issues included:

EC2 instance not reachable via SSH

Application not accessible via public IP

Misconfigured inbound/outbound rules

Firewall restrictions

Even a single incorrect port configuration in Security Groups blocked access to the deployed finance application.

Ensuring secure access while keeping necessary ports open required careful rule definition and repeated validation.

7. Database Normalization and Schema Design

Designing the ER model in Third Normal Form (3NF) required detailed analysis of entity relationships. Avoiding redundancy while maintaining performance was challenging.

Balancing normalization with query efficiency required careful structuring of:

- User entity
- Finance dashboard
- Transaction ledger
- Action history
- Infrastructure metadata

Maintaining referential integrity through foreign keys also required proper constraint definitions to prevent orphan records.

8. Audit and Compliance Considerations

Since the system is finance-oriented, audit readiness was critical. Designing a logging mechanism that captured:

- User actions
- Financial transactions
- Infrastructure changes

Deployment logs without affecting system performance was complex.

Ensuring that logs were stored persistently using Amazon EBS while maintaining security controls required additional configuration.

9. Disaster Recovery and State Management

Ensuring high availability and disaster recovery introduced additional complexity. Managing Terraform state files and ensuring reproducibility required strict version control and backup mechanisms.

Handling partial infrastructure failures was difficult. If Terraform provisioning failed mid-execution, some resources were created while others were not, requiring manual cleanup and state reconciliation.

10. Scalability Planning

Designing the system for horizontal scalability required stateless application architecture and careful load distribution planning.

Ensuring that Docker containers could scale without session persistence issues required adopting token-based authentication mechanisms. Improper session handling initially caused user authentication failures during scaling tests.

11. Learning Curve and Integration Complexity

The integration of multiple technologies—Terraform, Docker, Django, CI/CD automation, AWS networking—created a steep learning curve. Each tool had its own syntax, configuration style, and operational model.

Understanding how these tools interact in a single automated pipeline required continuous experimentation and debugging.

12. Time Management and Iterative Debugging

Due to the interconnected nature of DevOps workflows, a small misconfiguration in one layer affected the entire pipeline. This increased troubleshooting time.

Repeated cycles of:

Code modification

Pipeline execution

Log analysis

Infrastructure cleanup

were required before achieving a stable deployment.

Chapter- 4: Conclusion

4.1 Conclusion :

The development and implementation of the Automated Finance Provisioning System using Infrastructure as Code (IaC) represents a significant transformation from traditional manual infrastructure management to a fully automated, secure, and scalable cloud-native architecture. This project successfully demonstrates how modern DevOps principles can be applied to high-stakes financial systems where reliability, security, auditability, and uptime are critical requirements.

At its core, the project addressed the major problem of environmental drift and inconsistent infrastructure configurations. By defining cloud resources programmatically using Terraform and deploying them on Amazon Web Services, the system achieved complete reproducibility and consistency across environments. Infrastructure that once required hours of manual configuration can now be provisioned in minutes with predictable outcomes. This not only reduces operational overhead but also significantly minimizes the risk of human error.

The integration of a CI/CD pipeline using GitHub Actions transformed the development lifecycle into an automated and continuous process. Every code push triggers automated validation, containerization, testing, and deployment. This ensures that updates to the finance application are delivered rapidly while maintaining stability and security. Automation at this level enhances collaboration between development and operations teams, aligning them under a unified DevOps workflow.

Containerization using Docker played a crucial role in ensuring runtime consistency. The finance application, built with Django and powered by supporting libraries such as NumPy, runs within isolated containers. This eliminates dependency conflicts, simplifies scaling, and

ensures that the application behaves identically across development, testing, and production environments.

From a security perspective, the system was designed with strict access control, structured networking, and defined security group rules. By codifying infrastructure policies, the project minimized risks associated with open ports, misconfigured permissions, and unauthorized access. Logging mechanisms and audit trails were integrated at both application and infrastructure levels, ensuring that every financial transaction and system action is traceable. This makes the system compliant-ready and suitable for regulatory auditing.

The database design followed Third Normal Form (3NF), ensuring logical separation of entities such as users, financial dashboards, transaction ledgers, action history, and infrastructure metadata. This structured approach enhances data integrity, scalability, and maintainability. By logging both financial transactions and operational events, the system provides full transparency and accountability.

One of the most impactful achievements of the project is the ability to achieve rapid disaster recovery. In traditional systems, server failure could lead to extended downtime. In contrast, this architecture allows infrastructure to be recreated instantly by reapplying Terraform scripts, and application services can be restored by redeploying Docker images. This capability is critical for financial applications that demand near-zero downtime and continuous availability.

Scalability was another key outcome. The stateless application architecture and container-based deployment model allow horizontal scaling when transaction loads increase. New compute instances can be provisioned dynamically without affecting existing services. This ensures that the system remains responsive and stable even under high traffic conditions.

Beyond technical implementation, this project provided deep insights into cloud architecture design, security best

practices, CI/CD orchestration, and DevOps integration. It demonstrated that automation is not merely a convenience but a necessity for modern financial systems. By eliminating manual configuration, improving traceability, and enforcing structured processes, the project established a strong foundation for long-term operational efficiency.

The journey also highlighted the importance of planning, testing, and iterative refinement. Integrating multiple technologies—cloud infrastructure, containerization, automation pipelines, and database modeling—required careful coordination. Overcoming these complexities strengthened the overall system and ensured that the final solution was robust and production-ready.

In conclusion, the IaC-based Automated Finance Provisioning System successfully transformed a traditionally manual deployment approach into a modern, automated, secure, and scalable cloud architecture. It demonstrates how Infrastructure as Code, containerization, and CI/CD automation can collectively build a resilient financial system capable of meeting high performance, compliance, and availability standards. The project not only fulfills its technical objectives but also establishes a practical framework for deploying secure financial applications in real-world cloud environments.

4.2 Future Scope & Enhancements :

The current implementation of the Automated Finance Provisioning System establishes a strong foundation based on Infrastructure as Code (IaC), containerization, CI/CD automation, and cloud deployment. However, as financial systems continuously evolve with growing data volumes, regulatory demands, and security requirements, there are multiple areas where the system can be enhanced and expanded. The following section outlines detailed future scope and potential improvements that can transform the project into an enterprise-grade, production-ready financial platform.

1. Advanced Cloud Architecture Enhancements

Although the current system provisions infrastructure using Terraform on Amazon Web Services, future enhancements can include multi-region and high-availability architectures.

Multi-AZ Deployment

Deploying infrastructure across multiple Availability Zones (AZs) will improve fault tolerance. If one data center fails, traffic can automatically shift to another zone without service disruption.

Load Balancing

Integrating Application Load Balancers can distribute traffic across multiple EC2 instances, ensuring high availability and improved performance under heavy transaction loads.

Auto Scaling Groups

Auto Scaling can dynamically increase or decrease compute resources based on CPU utilization or traffic thresholds. This ensures cost optimization while maintaining performance.

Multi-Region Disaster Recovery

Deploying infrastructure across multiple AWS regions can provide geographic redundancy and faster recovery in case of regional outages.

2. Migration to Kubernetes-Based Orchestration

Currently, Docker containers are deployed on EC2 instances. A future enhancement would involve full orchestration using Kubernetes.

Benefits include:

- Automatic container scaling
- Self-healing (automatic restart of failed containers)
- Rolling updates with zero downtime
- Advanced resource allocation
- Service discovery and networking

Migrating to managed services like Amazon EKS would further simplify Kubernetes cluster management.

3. Enhanced Security & Compliance Features

Financial systems require advanced security controls beyond basic firewall and IAM policies.

Zero Trust Architecture

Implementing Zero Trust principles ensures that no user or service is trusted by default. Every access request is verified continuously.

Encryption Improvements

- Encrypting all data at rest using AWS-managed key
- Enforcing HTTPS with SSL/TLS certificates
- Enabling encryption for database storage volumes

Advanced Monitoring & Threat Detection

Integrating AWS GuardDuty and CloudTrail for real-time threat detection and auditing.

Role-Based Access Control (RBAC)

Expanding access control with granular user roles such as Auditor, Finance Analyst, Operations Admin, etc.

4. Observability and Monitoring Improvements

Future enhancements should include advanced monitoring, logging, and alerting mechanisms.

Centralized Logging

Using AWS CloudWatch or ELK stack for centralized log aggregation.

Application Performance Monitoring (APM)

Integrating tools like Prometheus and Grafana for performance metrics visualization.

Real-Time Alerts

Setting up automated alerts for:

- High CPU usage
- Failed deployment
- Unauthorized access attempts
- Abnormal transaction patterns

This will enable proactive incident management.

5. Database Scalability & Optimization

The current database follows 3NF normalization, but future improvements can enhance performance and scalability.

Database Replication

Implementing read replicas for load distribution and fault tolerance.

Managed Database Service

Migrating to Amazon RDS for automated backups, patching, and scaling.

Data Archiving Strategy

Older transaction logs can be archived to cold storage to improve query performance.

Sharding Strategy

For very large financial datasets, database sharding can distribute data across multiple nodes.

6. Advanced Financial Analytics Integration

The system currently supports mathematical analysis using NumPy. Future scope includes:

AI-Based Financial Forecasting

Integrating machine learning models for:

- Expense prediction
- Fraud detection
- Risk analysis
- Investment forecasting

Real-Time Analytics Dashboard

Using data visualization libraries to provide interactive graphs and financial insights.

Big Data Processing

Integration with data lakes and analytics services for processing high-volume transaction data.

7. Microservices Architecture Expansion

Currently, the application is containerized but may run as a monolithic backend.

Future improvements include:

- Splitting authentication service
- Separating transaction processing service
- Independent reporting service
- Notification and alert service

Each microservice can scale independently, improving performance and maintainability.

8. CI/CD Pipeline Enhancements

The CI/CD workflow using GitHub Actions can be enhanced with:

Automated Security Scanning

- Container vulnerability scanning
- Static code analysis
- Dependency vulnerability checks

Blue-Green Deployment

Allowing seamless upgrades without downtime.

Canary Releases

Gradually rolling out updates to a subset of users before full deployment.

Infrastructure Testing

Using Terraform plan validation and automated infrastructure testing frameworks.

9. API Gateway & Service Layer Enhancements

Introducing an API Gateway can:

- Provide centralized request routing
- Implement rate limiting
- Enable request logging
- Improve API security

This ensures better control over backend services.

10. Cost Optimization Strategies

Future enhancements should include cost management features:

- Resource tagging for cost tracking
- Auto shutdown of idle instances
- Right-sizing compute resources

Using Spot Instances for non-critical workloads

Cost dashboards can help monitor financial usage of cloud resources.