CHAPTER 1

INTRODUCTION

In the era of digital transformation, cloud computing has emerged as a vital backbone for deploying scalable and resilient IT infrastructure. Amazon Web Services (AWS), being a leading cloud service provider, offers a wide range of services to manage virtualized resources. However, manually managing these resources through the AWS Management Console or CLI often becomes complex and inefficient, especially in large-scale environments.

To overcome these challenges, this project introduces the Smart Cloud Management Portal, a centralized web-based platform developed to automate and streamline AWS resource management. The portal is designed using Infrastructure as Code (IaC) principles through Terraform, allowing infrastructure to be provisioned, updated, and version-controlled in a consistent and reliable manner.

- AWS Lambda: Serverless compute functions are used to perform actions such as attaching EBS volumes, restarting EC2 instances, and scaling Auto Scaling Groups.
- API Gateway: Provides RESTful endpoints to trigger Lambda functions remotely.
- Amazon Cognito: Enables secure user authentication, role-based access control, and password recovery via SMS.
- Amazon S3: Hosts the static web interface for the portal, making it accessible through a browser.
- CloudWatch & SNS: Ensure real-time monitoring and notifications for cloud events and alarms.

By combining these components, the portal offers an intuitive interface for cloud administrators and DevOps professionals to manage and automate cloud infrastructure tasks efficiently. This approach not only minimizes manual operations but also enhances security, scalability, and maintainability in cloud environments.

User authentication and role-based access control are managed via AWS Cognito, allowing developers to securely access their specific group dashboards while administrators retain full control through a dedicated admin interface. Each group page shows only relevant instances, and action buttons appear dynamically when resources exceed 70% utilization, enabling intelligent scaling with minimal user effort. This platform not only reduces operational complexity but also empowers teams with real-time decision-making capabilities and scalable automation. The project introduces smart auto-scaling policies using CloudWatch and Lambda to maintain performance and reduce costs, while also enabling teams to visualize system health instantly through live graphs and actionable insights. With built-in CI/CD pipelines powered by Jenkins and modular design via Terraform and Ansible, this solution is robust, extensible, and production-ready for enterprise cloud management. This project combines the power of automation, monitoring, and visualization to deliver a secure, scalable, and responsive cloud management solution tailored for modern DevOps and infrastructure teams.

1.1 SYSTEM OVERVIEW

The Smart Cloud Infrastructure Management Portal is a fully integrated, web-based solution designed to simplify AWS infrastructure management for both administrators and operators. The system provides a centralized platform that enables users to perform key cloud operations such as restarting EC2 instances, scaling Auto Scaling Groups (ASGs), and attaching EBS volumes—all through a user-friendly graphical interface. The portal leverages a serverless architecture using AWS Lambda to execute backend logic, with API Gateway acting as the entry point for all external requests. The frontend is hosted as a static website on Amazon S3, developed using HTML, CSS, and JavaScript. Real-time system monitoring is achieved via AWS CloudWatch, with visual dashboards built using Chart.js for CPU, network, and disk metrics.

User authentication and authorization are securely managed through Amazon Cognito, which supports user signup, login, role-based access control, and SMS-based password recovery. The system infrastructure is fully automated and reproducible using Terraform, which manages the provisioning of all AWS services. IAM roles and policies ensure that each component has only the necessary permissions to perform its function. CloudWatch Alarms are configured to monitor thresholds and trigger SNS notifications to alert administrators in case of critical incidents. This modular and scalable system not only improves visibility and control over cloud resources but also enhances operational efficiency by reducing manual tasks. The architecture is designed to be extensible, allowing future integration of services like CI/CD pipelines, AI-based optimization, or multi-cloud support.

1.1.1 PURPOSE OF THE SYSTEM

The primary purpose of the system is to simplify the deployment, monitoring, and management of cloud infrastructure by providing:

A. A web-based control interface for EC2, EBS, and Auto Scaling resources.

B. Serverless automation of common administrative tasks via Lambda functions.

C. Secure, role-based user access with authentication and password recovery.

1.1.2 KEY FUNCTION COMPONENTS

a) Infrastructure as Code (IaC)

- Uses **Terraform** to provision and manage all AWS resources.
- Ensures reproducibility and traceability of infrastructure changes.

b) Lambda Functions

- Three Python-based serverless functions:
- Attach volume: Attaches a specified EBS volume to an EC2 instance.
- Restart instance: Restarts a selected EC2 instance.
- Scale asg: Modifies the desired capacity of an Auto Scaling Group.
- Each function is triggered through HTTP endpoints via API Gateway

c) API Gateway Integration

- Acts as the interface layer between the frontend and backend.
- Provides secure and scalable endpoints for Lambda invocation.

d) Cognito User Management

- Implements authentication using **AWS Cognito User Pools**.
- Supports user roles (admin, user) and SMS-based password reset.
- A default admin user is provisioned at deployment for immediate access.

e) Frontend Web Portal

- A static web application hosted on **Amazon S3**.
- Interacts with API Gateway and Cognito for operations and authentication.

f) Monitoring and Notifications

- **CloudWatch Alarms** monitor EC2 metrics (e.g., CPU utilization).
- **SNS Topics** send alerts and optionally invoke corrective actions.

1.1.3 REQUIREMENT GATHERING

Requirement gathering is a critical phase in the software development lifecycle, where the functional and non-functional expectations of the system are identified, documented, and analyzed. For the Smart Cloud Management Portal, both technical and user-centered requirements were collected through research, practical needs assessment, and experience in cloud operations.

1.1.4 SOURCE OF REQUIREMENTS

Discussions With DevOps Professionals.

- Analysis of common AWS management tasks.
- Study of existing infrastructure automation tools.
- Review of AWS service documentation.

1.1.5 FUNCTIONAL REQUIRMENTS

ID	Requirement Description
1	User should be able to log in and authenticate via Cognito.
2	Admin users should have additional privileges (e.g., restart instances, scale ASG).
3	User should be able to attach EBS volumes to EC2 instances via UI/API.
4	System should allow restarting of EC2 instances using a Lambda function.
5	User should be able to scale the Auto Scaling Group up or down.

ID	Requirement Description
6	System should expose all Lambda functions through REST APIs.
7	Web UI should interact with AWS services via secure API Gateway calls.
8	SMS-based password recovery should be available via Cognito.
9	Admin user should be provisioned automatically during deployment.
10	CloudWatch alarms should notify users or trigger automation if thresholds are exceeded.

1.1.6 NON-FUNCTIONAL REQUIREMENTS

ID	Requirement Description
1	The system must ensure secure communication through HTTPS.
2	Authentication tokens must be securely stored and expired appropriately.
3	The portal should be accessible 24/7 with minimal downtime.
4	The infrastructure should be scalable and cost-effective.
5	All actions must be logged for auditing purposes (via CloudTrail or Lambda logs).
6	The deployment process must be fully automated using Terraform.
7	The system must follow least privilege principles using IAM roles.

1.1.7 PROBLEM STATEMENT

In modern cloud-based infrastructures, especially those hosted on Amazon Web Services (AWS), administrators and DevOps engineers are often required to manage a wide variety of services such as EC2 instances, EBS volumes, Auto Scaling Groups, and user authentication systems. These tasks typically involve manual configurations through the AWS Management Console or command-line tools, which can be time-consuming, error-prone, and inefficient, particularly in dynamic or large-scale environments.Furthermore, many organizations lack a centralized interface to manage their cloud operations, leading to fragmented workflows, lack of visibility, and inconsistent configurations. Security concerns also arise when there is no standardized way to enforce user authentication, authorization, and access control across the infrastructure.

There is a clear **need for an automated, secure, and centralized platform** that allows users to:

- Perform day-to-day cloud operations such as restarting instances, attaching volumes, or scaling compute resources without direct console access.
- Authenticate securely using modern identity services with role-based access.
- Monitor infrastructure health and receive alerts or trigger corrective actions automatically.
- Deploy infrastructure through reusable, version-controlled code.

The Smart Cloud Management Portal is proposed to address these challenges by providing a unified solution that integrates AWS services like Lambda, API Gateway, Cognito, EC2, and CloudWatch through a web-based interface. The system aims to reduce operational overhead, improve infrastructure consistency, and enhance security through automation and Infrastructure as Code (IaC) using Terraform.

1.1.8 PURPOSE AND NEED

Purpose:

The primary purpose of the Smart Cloud Management Portal is to offer a centralized, secure, and automated platform for managing AWS cloud infrastructure efficiently. It streamlines operations by integrating essential AWS services and automation tools, allowing users to perform critical cloud tasks with ease and minimal manual effort.

This portal is designed to empower DevOps engineers and cloud administrators to:

- Execute essential operations like restarting EC2 instances, attaching EBS volumes, and managing Auto Scaling Groups.
- Ensure secure access through Amazon Cognito, implementing role-based permissions to control user activities.
- Monitor infrastructure health and respond to events using CloudWatch alarms and SNS notifications.
- Provide a clean and intuitive user experience via a static web interface hosted on Amazon S3 Overall, the system enhances operational efficiency, reduces the risk of human error, and ensures scalability and maintainability for modern cloud-based environments.

Need:

With the rapid growth of cloud adoption, organizations are increasingly encountering several recurring challenges in managing cloud infrastructure effectively:

- Manual Operations: Frequent manual interventions increase the likelihood of misconfigurations, delays, and system downtime.
- Lack of Centralization: Managing various AWS services through separate consoles or custom scripts leads to inefficiencies and a fragmented user experience.

- Security Risks: Inconsistent authentication methods and lack of standardized access control mechanisms can compromise system security.
- No Automation: The absence of automated workflows delays troubleshooting, scaling, and recovery processes during infrastructure failures.
- Inconsistent Infrastructure: Without Infrastructure as Code (IaC), environments become difficult to reproduce and manage, resulting in potential configuration drift.

1.2 AIM OF THE PROJECT

The aim of the Smart Cloud Management Portal project is to design and implement a centralized, secure, and automated platform that streamlines the management of AWS cloud infrastructure through a user-friendly web-based interface. The project addresses the growing need for simplified and scalable cloud operations in modern DevOps environments.

This system is developed to empower DevOps engineers, system administrators, and cloud architects with a comprehensive set of tools to manage cloud infrastructure efficiently and securely. It eliminates the need for manual intervention by automating key processes, ensuring reliability and consistency in operations.

The core objectives include:

- Efficiently managing AWS services such as **EC2 instances**, **EBS volumes**, **VPC**, and **Auto Scaling Groups** with a centralized interface.
- Automating cloud operations like **instance reboot**, **volume attachment**, and **resource scaling** using **AWS Lambda functions** triggered through **API Gateway**.
- Securing access via **Amazon Cognito** by providing **authentication**, **role-based access control**, and **multi-factor authentication (MFA)** support.
- Utilizing **Terraform** to provision and manage infrastructure as code, allowing for reproducibility, scalability, and version control across multiple environments.
- Enabling **real-time monitoring and alerting** through **CloudWatch metrics and alarms**, with proactive notifications using **SNS and Lambda-based remediation**.
- Offering an interactive and responsive **frontend portal hosted on S3 and served via CloudFront**, allowing users to perform operations and visualize status effortlessly.
- Facilitating **audit logging** and tracking of user actions for improved governance and troubleshooting.

By fulfilling these goals, the project aims to reduce manual overhead, minimize downtime, ensure faster deployment cycles, and promote best practices in cloud infrastructure management. It lays a solid foundation for future expansion, including mobile support, AI-driven scaling policies, and multi-cloud integration.

CHAPTER 2

LITERATURE SURVEY

The literature survey focuses on examining the current landscape of cloud infrastructure management, including the technologies, tools, and practices employed in the industry. It aims to identify the strengths and weaknesses of existing solutions while understanding their impact on cloud operations. By reviewing related research, the survey highlights the evolution of cloud management systems, their capabilities, and challenges. The goal is to pinpoint the gaps in current solutions, particularly in areas like automation, scalability, and cost optimization. Furthermore, the survey aims to evaluate industry-standard tools and explore how the Smart Cloud Management Portal can address these limitations. Through this analysis, the survey sets the foundation for proposing an innovative solution to improve cloud resource management.

2.1 AWS CLOUD SERVICE

Amazon Web Services (AWS) provides a comprehensive suite of cloud computing services including compute (EC2), storage (EBS, S3), monitoring (CloudWatch), automation (Lambda), and identity management (Cognito, IAM). These services are highly scalable, reliable, and secure, making them ideal for modern application development. AWS enables developers to build and deploy infrastructure quickly and efficiently. However, as environments scale, managing these services manually via the AWS Console or CLI becomes increasingly complex. This calls for automation and orchestration tools like **Terraform and Ansible** to simplify operations. Integrating these services forms the foundation of intelligent, automated cloud management solutions.

2.2 INFRASTRUCTURE AS CODE

Terraform, developed by HashiCorp, is a powerful Infrastructure as Code (IaC) tool that allows users to define, provision, and manage cloud infrastructure using a declarative configuration language. It brings automation, repeatability, and consistency to infrastructure management. By codifying infrastructure, teams can version-control their environments just like application code. Studies and industry use cases have shown that IaC significantly reduces deployment time and minimizes human error. In this project, Terraform is used to provision AWS resources such as EC2 instances, IAM roles, CloudWatch alarms, and networking components. It serves as the backbone for creating scalable and maintainable cloud infrastructure.

2.3 SERVERLESS COMPUTING WITH AWS LAMBDA

Serverless architecture enables developers to run backend code without the need to manage or provision servers. **AWS Lambda** is a leading serverless service that automatically handles compute resources, scaling, and availability. It allows functions to be triggered by events, making it highly efficient for automation. Industry research confirms that serverless functions

are ideal for tasks like auto-scaling, system backups, monitoring alerts, and event-driven workflows. In this project, Lambda plays a key role in responding to CloudWatch alarms

2.4 CLOUD MONITORING AND ALERTS

Amazon CloudWatch provides deep insights into the performance and health of AWS resources, enabling real-time monitoring of metrics such as CPU utilization, memory, and disk usage. It helps users track system performance, detect anomalies, and trigger automated actions based on predefined thresholds. SNS (Simple Notification Service), on the other hand, complements CloudWatch by offering scalable messaging and alerting capabilities. When critical infrastructure events are detected, SNS can send notifications to administrators, systems, or other services to ensure timely responses. Together, these tools are essential for maintaining high availability, providing real-time alerts, and automating actions that enhance the reliability of cloud environments in enterprise settings.

2.5 AUTHENTICATION WITH AMOZON COGNITO

AWS Cognito is a robust identity and access management service that provides user authentication, authorization, and user directory management. It supports user pools, federated identities, and multi-factor authentication (MFA), enabling secure access to cloud applications. Cognito simplifies the integration of sign-up and sign-in features with minimal backend configuration. In this project, it enforces role-based access control, directing users to group-specific dashboards based on their roles. It securely integrates with API Gateway and Lambda to protect backend resources. Overall, Cognito enhances security while reducing the complexity of managing custom authentication logic.

2.6 RELATED WORK AND GAPS

While many organizations rely on individual AWS services for specific tasks such as automation, monitoring, identity management, and infrastructure control, there is often a lack of a unified platform that brings these capabilities together in one centralized system. Existing dashboards or management portals tend to have limitations, such as being rigid and non-customizable, which reduces their effectiveness in adapting to unique organizational needs. Additionally, some platforms require costly licenses, which can be a barrier for smaller enterprises. As a result, businesses struggle to maintain an efficient and seamless workflow across various cloud services. This gap highlights the need for a more flexible and customizable solution that can integrate these functionalities under a single portal, optimizing management and resource allocation.

CHAPTER 3

SYSTEM ANALYSIS

3.1 EXISTING SYSTEM

In traditional cloud environments, AWS resources are typically managed through manual or semi-automated means using the AWS Management Console, Command Line Interface (CLI), or third-party tools. This setup works for small-scale applications but becomes increasingly inefficient and error-prone as cloud infrastructure scales.

3.2 MANUAL MANAGEMENT

Tasks such as starting/stopping EC2 instances, attaching EBS volumes, and scaling Auto Scaling Groups are often performed manually by administrators. These actions require navigation through the AWS Console or scripting through CLI, which introduces several challenges:

- Time-consuming processes
- High risk of human error
- Lack of consistency and standardization
- Delayed response to system alerts

3.2.1 LIMITED AUTOMATION

While automation tools like AWS CloudFormation or basic shell scripts exist, they often lack integration with real-time triggers, secure access control, and user-level granularity. Additionally, they are not intuitive for non-technical users.

3.2.2 SECURITY AND ACCESS CONTROL

Access is typically managed using IAM roles and policies, which, though powerful, are not easily manageable for multiple users or dynamic roles. Multi-factor authentication and role-based access control are not always enforced or available in custom-built tools.

3.2.3 MONITORING AND NOTIFICATION

CloudWatch and SNS are available for monitoring, but configuring them manually for each metric and resource is inefficient. Alert responses are often delayed or require manual intervention, increasing system downtime risk.

3.2.4 LACK OF CENTRALIZED INTERFACE

There is no unified dashboard that integrates all necessary functions such as:

- User login
- Instance control
- Volume attachment
- Scaling ASGs
- Viewing monitoring alerts

As a result, administrators have to toggle between multiple AWS services or consoles, leading to operational complexity.

3.3 LIMITATION OF THE EXISTING SYSTEM

The existing system of managing AWS cloud infrastructure through the AWS Console, CLI, or basic scripts presents several significant limitations that affect efficiency, scalability, and security. These limitations are outlined below

1. Manual Operations

Most tasks such as starting/stopping instances, attaching volumes, or modifying Auto Scaling configurations are done manually. This is time-consuming and increases the risk of human error.

2. No Centralized Interface

There is no unified dashboard or portal that brings together all cloud operations. Users must navigate across multiple AWS services and interfaces to perform different tasks.

3. Limited Automation

While some automation is possible via scripts or scheduled events, it is not robust or user-friendly. Real-time, event-driven automation is lacking, and maintenance of these scripts can become complex.

4. Weak User Access Control

Managing users through IAM is powerful but not user-friendly, especially when dealing with multiple users or needing role-based access. There is no support for self-service signup, role assignment, or password recovery.

5. Scalability Issues

Manual methods do not scale efficiently with the growth of infrastructure. As the number of instances or users grows, the effort to manage them increases exponentially.

6.Lack Monitoring Integration

Monitoring tools like CloudWatch are available but must be configured individually for each resource. There's no centralized view or auto-remediation tied to real-time alerts.

7. Poor User Experience

Non-technical users find it difficult to interact with AWS services due to the complexity of the console and CLI tools. There's no simplified frontend tailored for general usage.

8. No Infrastucture As Code

Infrastructure changes made manually are not version-controlled or repeatable. This results in inconsistencies, configuration drift, and lack of documentation.

3.4 PORPOSED SYSTEM

The proposed system, **Smart Cloud Management Portal**, aims to provide a centralized, secure, and intelligent interface for managing cloud infrastructure on AWS. It addresses the limitations of manual cloud operations by introducing automation, scalability, and user access control through a web-based dashboard.

The system is designed to simplify key AWS operations such as launching, monitoring, and scaling EC2 instances, attaching or detaching EBS volumes, and managing user sessions. It leverages modern cloud-native technologies including **Terraform** for Infrastructure as Code (IaC), **AWS Lambda** for serverless computing, **Amazon API Gateway** for secure API endpoints, and **Amazon Cognito** for user authentication and role-based access control.

The portal's frontend is hosted on **Amazon S3** and provides a responsive user interface, allowing both administrators and standard users to perform actions based on their roles. Real-time monitoring is achieved through **CloudWatch integration**, enabling the system to visualize key metrics like CPU utilization, memory usage, and instance status.

By automating infrastructure management and reducing human intervention, the proposed system enhances operational efficiency, minimizes errors, and promotes cost-effective resource utilization.

3.4.1 ADVANTAGE OF THE PROPOSED SYSTEM

The **Smart Cloud Management Portal** offers several advantages over traditional, manual methods of managing cloud infrastructure:

1. **Automation and Efficiency** By using Terraform and serverless AWS services, the system automates routine cloud management tasks, reducing the need for manual intervention and accelerating deployment times.
2. **User-Friendly Interface** user The web-based dashboard provides an intuitive and accessible way to manage cloud resources, even for users with limited technical knowledge.
3. **Role-Based Access control Integration** with Amazon Cognito enables secure authentication and authorization, allowing different access levels for administrators and standard users.
4. **Real-Time Monitoring** CloudWatch integration provides real-time visibility into system performance, helping users track resource usage and detect anomalies promptly.
5. **Scalability and Flexibility** The system supports auto-scaling of EC2 instances based on load, ensuring high availability and performance under varying traffic conditions.
6. **Infrastructure as code** Terraform ensures that infrastructure changes are version-controlled, auditable, and easily replicable across environments.
7. **Cost optimization** By enabling better resource management and automation, the system reduces operational costs and avoids unnecessary resource consumption.
8. **Improved Security** Serverless components and managed services reduce the attack surface, while encrypted communications and user authentication enhance overall security.

3.5 REQUIREMENT SPECIFICATION

The requirement specification defines the essential hardware, software, and functional requirements for developing and deploying the **Smart Cloud Management Portal**. These specifications ensure that the system operates efficiently and securely in a cloud-native environment.

3.5.1 SOFTWARE REQUIREMENTS

The development and deployment of the Smart Cloud Management Portal require specific software tools and services to ensure functionality, scalability, and security. Below is a list of all necessary software components:

1. Operating System

- Windows 10 / 11, macOS, or any Linux distribution (Ubuntu preferred)

2. Development Tools

- Visual Studio Code – Source code editor for frontend and backend development
- Terraform CLI – For writing and applying Infrastructure as Code (IaC)
- Node.js & npm – Required for frontend development and package management
- Git – Version control for collaborative development and code management

3. Web Technologies

- HTML, CSS, JavaScript – Core technologies for frontend UI
- React.js (or similar framework) – For building a responsive and dynamic frontend

4. Cloud Services (AWS)

- Amazon EC2 – Compute resource management
- Amazon S3 – Hosting the frontend
- Amazon Lambda – Backend logic using serverless functions
- Amazon API Gateway – Exposing REST APIs securely
- Amazon Cognito – User authentication and access control
- Amazon CloudWatch – Monitoring and logging
- AWS IAM – Role and permission management
- AWS Auto Scaling – Automatic scaling of compute resources
- Amazon EBS – Block storage management

5. Browsers

- Google Chrome, Mozilla Firefox, Microsoft Edge (latest versions recommended)

6. Miscellaneous

- Postman or any REST client – For API testing
- AWS CLI (optional) – For command-line AWS resource management

3.5.2 HARDWARE REQUIREMENTS

1. Development Machine (Local System)

Component	Minimum Requirement	Recommended Requirement
Processor	Intel Core i3 or equivalent	Intel Core i5/i7 or equivalent
RAM	4 GB	8 GB or higher
Storage	50 GB free disk space	100 GB SSD
Display	13" display, 1366×768 resolution	15"+ display, Full HD (1920×1080)
Internet	Stable broadband connection	High-speed internet (10 Mbps+)

2. Cloud Infrastructure (AWS Hosted Components)

Since the project is cloud-based, most of the heavy lifting is offloaded to AWS. The system primarily utilizes:

- **EC2 Instances** (t2.micro or higher for development/testing)
- **S3 Buckets** for frontend hosting
- **Lambda Functions** for backend logic
- **CloudWatch** for monitoring
- **Cognito** for authentication

This cloud-centric architecture reduces the dependency on powerful local hardware, enabling development and management from lightweight systems.

3.6 LANGUAGE SPECIFICATION

3.6.1 PROGRAM LANGUAGE SPECIFICATION

Language	Purpose	Justification
JavaScript	Fronted development using React.js or plain JS	Enables dynamic and interactive user interfaces; widely supported across browsers.
HTML/CSS	Web page structure and styling	Essential for designing responsive and accessible web interfaces.
Python	Backend logic within AWS Lambda functions	Lightweight, fast, and offers excellent AWS SDK (boto3) support.
HCL (Terraform)	Infrastructure as Code (IaC) for AWS provisioning	Declarative syntax ideal for defining and managing cloud infrastructure.
JSON/YAML	Configuration for APIs, IAM policies, and deployment templates	Standard formats for cloud configurations and API integration.

3.7 ALGORITHM DESCRIPTION

The **Smart Cloud Management Portal** relies on a set of well-defined algorithms and logical flows to manage user roles, interact with AWS services, and automate cloud infrastructure operations.

Below are the key algorithms used within the system:

1. User Authentication and Authorization Algorithm (Using Amazon Cognito)

Step 1: User submits login credentials (username/email and password).
Step 2: Credentials are verified via Amazon Cognito.
Step 3: If valid, Cognito returns an authentication token (ID, Access, and Refresh tokens).
Step 4: Based on the token's user group (Admin/User), the portal enables or restricts access to certain features.
Step 5: If invalid, return an error and prompt for re-authentication.

2. EC2 Instance Management Algorithm

Step 1: User selects an action (Start, Stop, Reboot) for a specific EC2 instance.
Step 2: A request is sent to the backend (AWS Lambda via API Gateway).

Step 3: Lambda function receives the request and uses **boto3** to interact with EC2.
Step 4: The instance state is modified based on the user's action.
Step 5: The updated instance state is returned and displayed on the frontend.

3. Auto Scaling Configuration Algorithm

Step 1: Admin sets scaling policies (e.g., scale out if CPU > 70%, scale in if CPU < 30%).
Step 2: CloudWatch monitors metrics in real-time.
Step 3: If thresholds are breached, CloudWatch triggers a scaling policy via Auto Scaling Group.
Step 4: EC2 instances are added or removed automatically based on demand.

4. EBS Volume Attachment/Detachment Algorithm

Step 1: Admin selects a volume and an EC2 instance from the dashboard.
Step 2: Lambda function uses **boto3** to attach/detach the selected EBS volume.
Step 3: AWS performs the requested operation.
Step 4: The updated volume status is shown on the frontend.

5. Real-Time Monitoring Algorithm

Step 1: CloudWatch collects metrics like CPU, memory, and disk usage.
Step 2: A scheduled Lambda function fetches the latest metrics at defined intervals.
Step 3: Data is returned to the frontend via API Gateway.
Step 4: Graphs and charts are updated in real time using libraries like Chart.js or Recharts.

CHAPTER- 4

SYSTEM DESIGN

4.1 ARCHITECTURE DIAGRAM

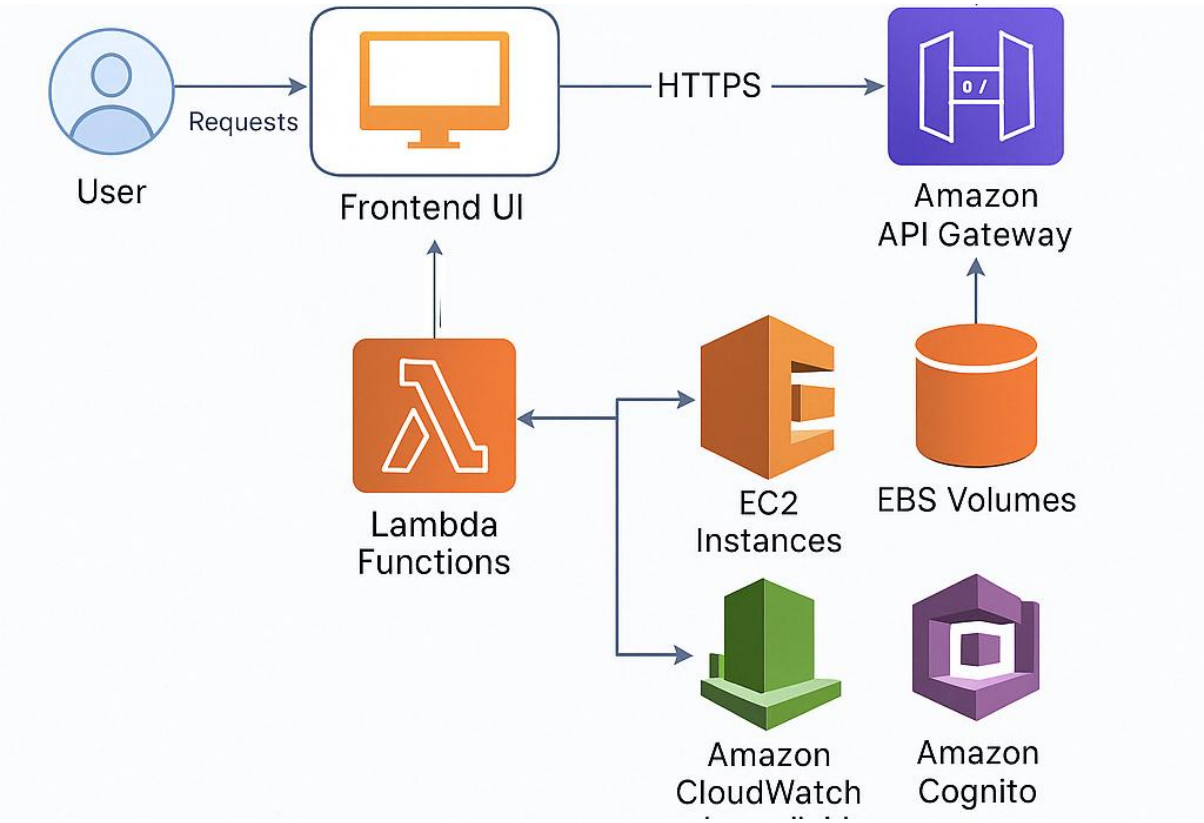

Fig 4.1 Architecture diagram

The system design of the **Smart Cloud Management Portal** defines the architectural layout, components, modules, and data flow necessary for building a secure, scalable, and efficient cloud management platform. It is designed to ensure seamless interaction between the frontend, backend services, and AWS cloud infrastructure.

4.2 USE CASE DIAGRAM

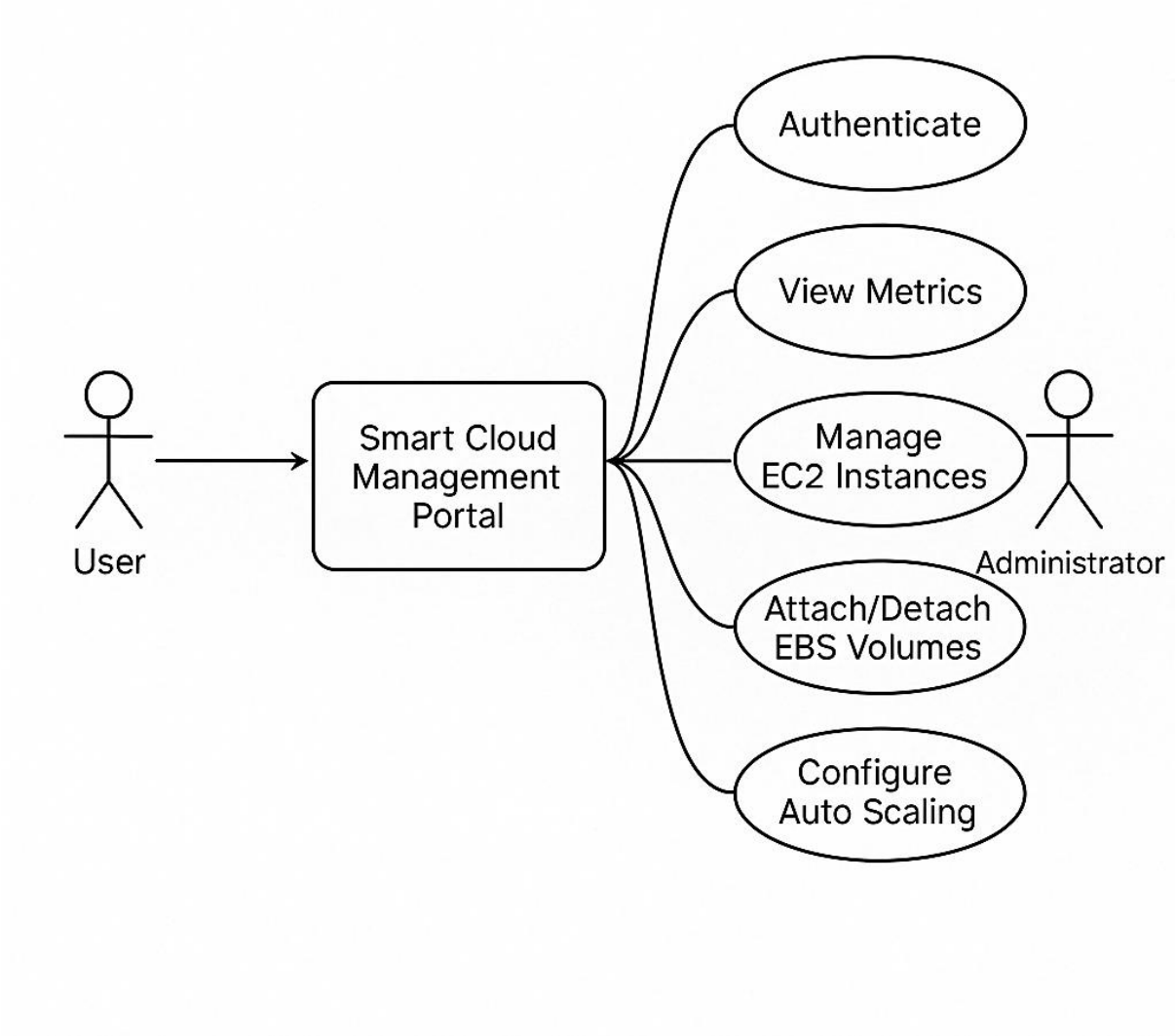

Fig 4.2 Use Case Diagram

The **Use Case Diagram** for the *Smart Cloud Management Portal* illustrates the interactions between users and the system's functionalities. It includes two primary actors: **Admin** and **Standard User**. The Admin has extended privileges such as launching EC2 instances, attaching/detaching EBS volumes, configuring auto-scaling policies, and managing user roles. The Standard User, on the other hand, can log in, view instance status, and perform limited operations like starting or stopping assigned EC2 instances. This diagram helps visualize the overall system behavior from a user interaction perspective, ensuring clarity in role-based access and feature allocation.

4.3 ACTIVITY DIAGRAM

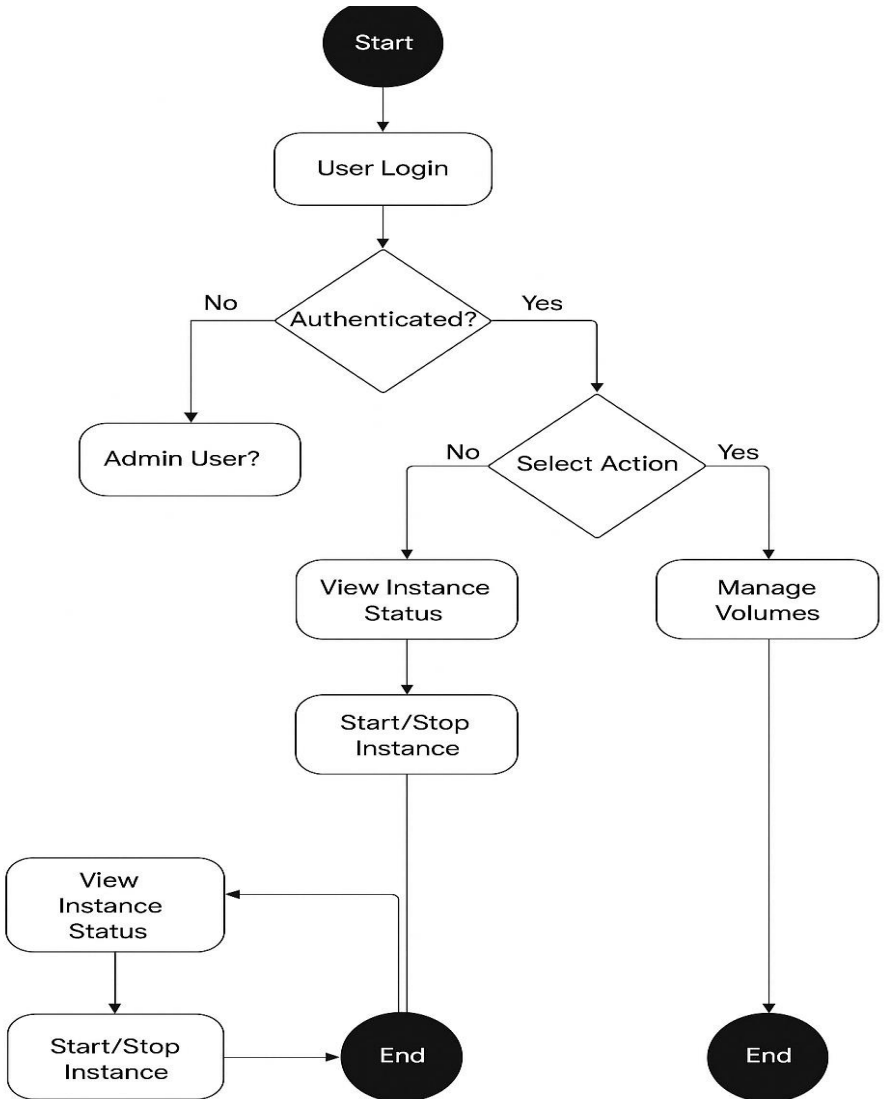

Fig 4.3 Activity Diagram

The **Activity Diagram** illustrates the workflow of operations within the *Smart Cloud Management Portal*, focusing on user interaction and system response. It visualizes the dynamic behavior of the system through a flow of activities, decision points, and concurrent processes.

4.4 SEQUENCE DIAGRAM

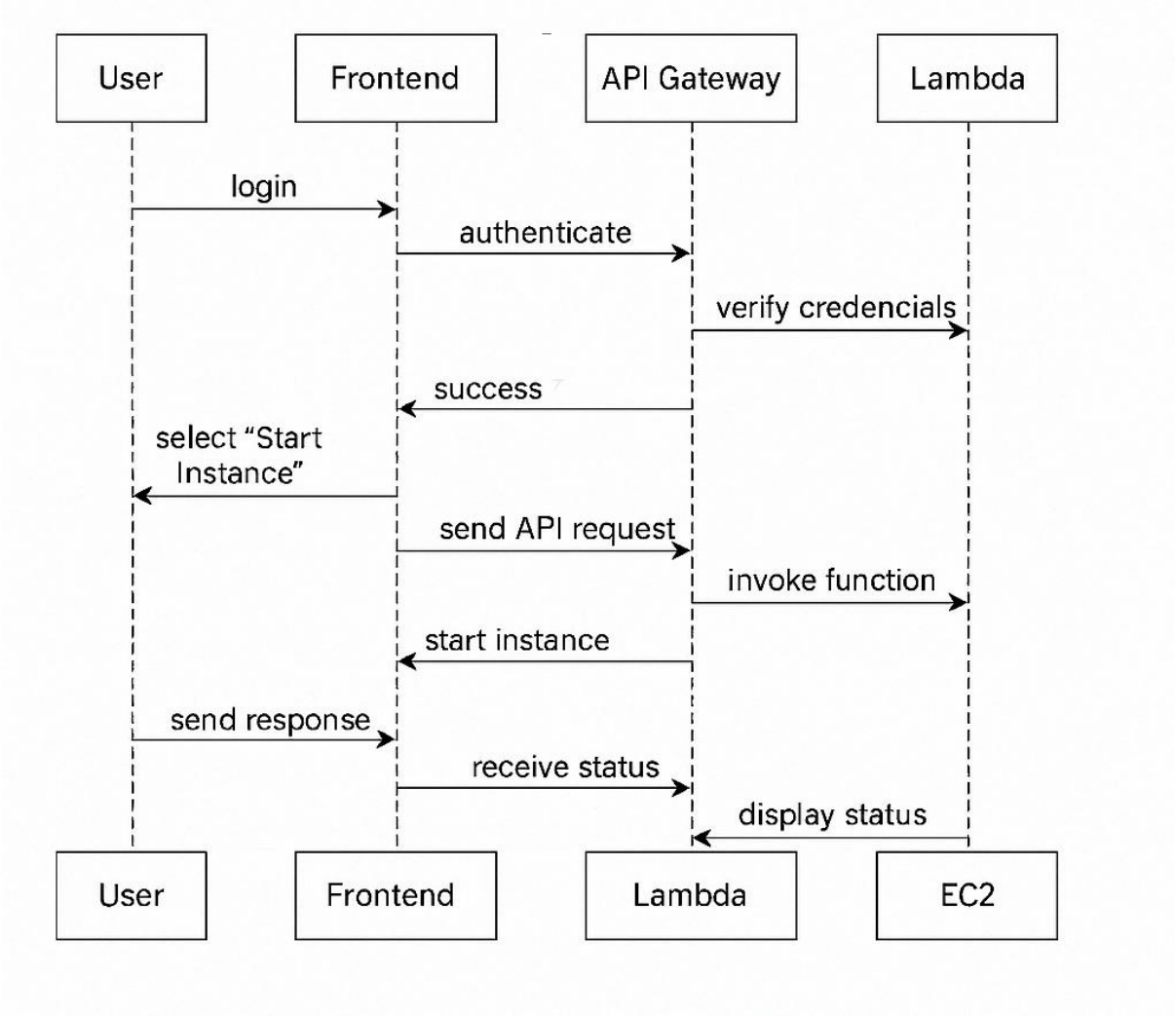

Fig 4.4 Sequnce Diagram

The **Sequence Diagram** for the *Smart Cloud Management Portal* showcases the chronological interaction between system components during a cloud operation, such as starting an EC2 instance. It begins with user authentication through Amazon Cognito, ensuring secure access. Upon successful login, the user initiates an action, which the frontend passes through the API Gateway to an AWS Lambda function. The Lambda function processes the request using AWS SDK (boto3) and interacts with EC2 to perform the required task. Once the instance is started, the response is returned to the frontend and displayed to the user. This diagram helps visualize the system's event-driven architecture and demonstrates the efficient orchestration of cloud-native services.

4.5 COMPONENT DIAGRAM

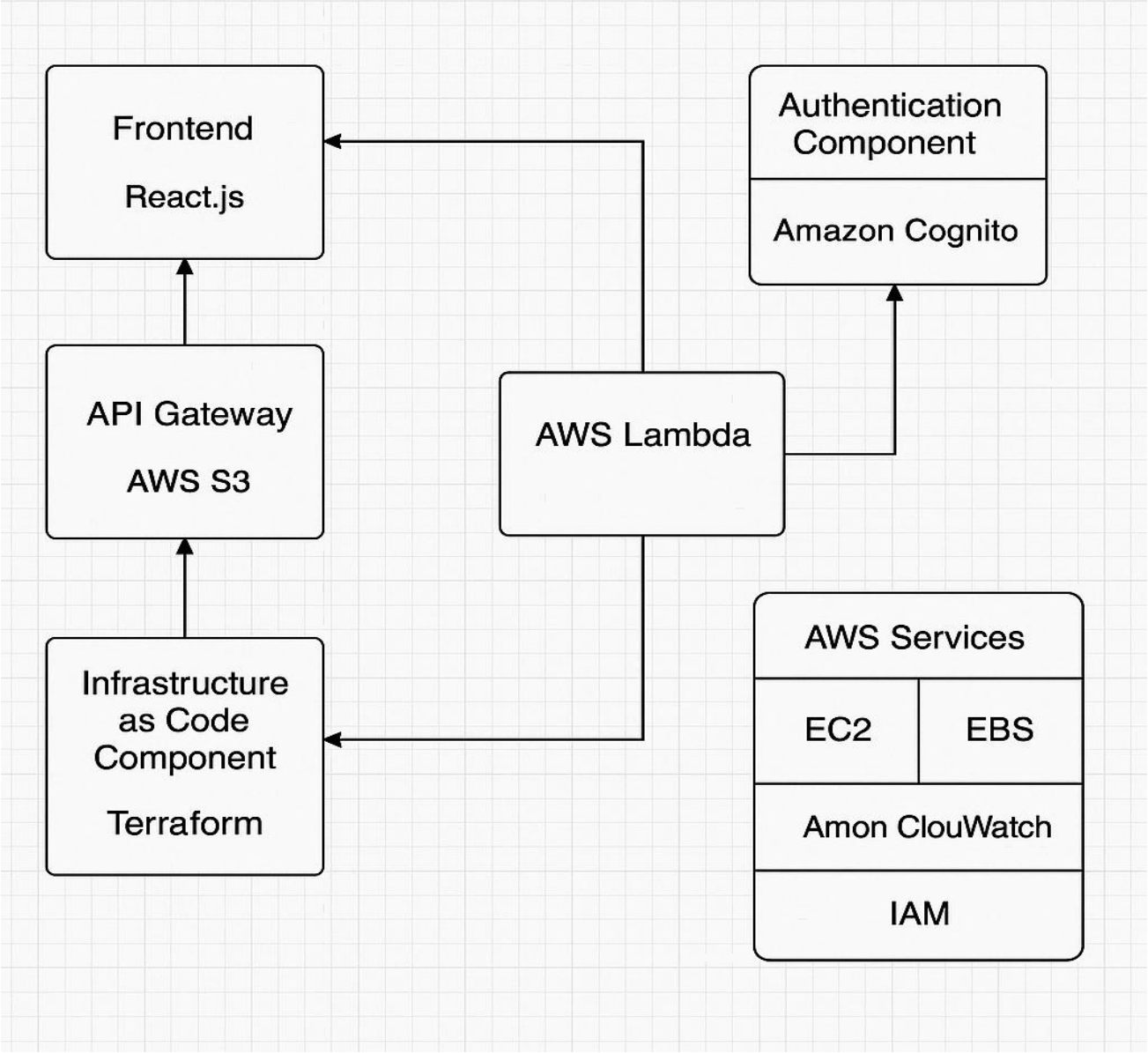

Fig 4.5 Component Diagram

The **Component Diagram** of the *Smart Cloud Management Portal* outlines the major modules and their interactions within the system. The frontend component, developed with React.js and hosted on AWS S3, communicates with the backend through Amazon API Gateway..

CHAPTER 5

SYSTEM IMPLEMENTATION

The implementation of the Smart Cloud Management Portal centers on developing a full-stack, cloud-native application that leverages AWS services, serverless architecture, and modern web technologies. This approach ensures scalability, reliability, and cost-efficiency while maintaining seamless integration with cloud infrastructure. The system is modular by design, with each component functioning independently to perform specific tasks such as monitoring, automation, or identity management. These components communicate securely through APIs, allowing for flexible interactions and easy maintenance. By combining serverless functions, API Gateway, Lambda, DynamoDB, and frontend frameworks, the portal delivers a responsive and centralized solution for cloud resource management.

.5.1 FRONTEND IMPLEMENTATION

The frontend of the Smart Cloud Management Portal is developed using React.js, HTML, CSS, and JavaScript, offering a modern and interactive user experience. This web interface is hosted on Amazon S3 and distributed via Amazon CloudFront, ensuring low-latency access and global availability. It acts as the primary point of interaction for users, including both Admins and Standard Users.

Users can securely log in using integrated authentication mechanisms, view real-time EC2 instance statuses, and perform key operations such as starting, stopping, or rebooting instances. The interface also includes features to monitor cloud resource usage like CPU, memory, and disk, and provides access to system logs for troubleshooting.

The frontend communicates securely with backend APIs to fetch data and execute actions. The design prioritizes responsiveness and cross-platform compatibility, ensuring a seamless experience on desktops, tablets, and mobile devices. Components are modular and reusable, enabling scalability and easy future enhancements.

Visualizations such as charts and tables are used to present data clearly, improving usability and decision-making. The UI maintains consistency with AWS design principles, creating a familiar environment for cloud users.

5.2 BACKEND IMPLEMENTATION

The backend of the Smart Cloud Management Portal is built using AWS Lambda functions written in Python, providing a lightweight, scalable, and cost-effective serverless architecture. These functions are designed to handle specific tasks such as starting, stopping, or rebooting EC2 instances, as well as managing volumes and monitoring services. Communication between the frontend and backend is handled securely through Amazon API Gateway, which routes HTTP requests to the appropriate Lambda function.

Each function executes in a stateless environment, ensuring rapid scaling and independent processing of requests without maintaining server infrastructure. This serverless model allows for reduced operational costs, as resources are only consumed when the function is triggered (pay-per-use model). The architecture supports high availability and can handle fluctuating workloads efficiently.

The modular design of the backend ensures that new functionalities can be easily added without affecting the existing system. Logging and monitoring are integrated through CloudWatch to track performance and troubleshoot issues. Overall, the backend acts as the core logic layer, enabling secure, reliable, and automated cloud operations through API-driven workflows.

5.3 AUTHENTICATION IMPLEMENTATION

The Smart Cloud Management Portal leverages **Amazon Cognito** to manage user authentication and authorization securely and efficiently. Cognito allows users to register, log in, and manage their accounts with robust password policies and multi-factor authentication options. It supports token-based session management using JWT (JSON Web Tokens), ensuring secure and stateless user sessions across the portal.

Role-based access control is enforced, distinguishing between **Admin** and **Standard Users**, with each role granted different permissions based on their access level. For instance, only Admins can perform critical infrastructure actions, while Standard Users may have limited visibility or control. Cognito seamlessly integrates with **Amazon API Gateway**, enabling secure authorization checks on every API request by validating JWT tokens.

This setup not only enhances security but also simplifies user management by offloading identity and access control to a scalable, fully managed AWS service. By using Cognito, the portal ensures that only authenticated and authorized users can access sensitive features, maintaining both security and operational integrity.

5.4 INFRASTRUCTURE AUTOMATION

Tool Used: Terraform

Terraform is a powerful Infrastructure as Code (IaC) tool used to automate the provisioning and management of cloud resources in the Smart Cloud Management Portal. With Terraform, all infrastructure components are defined in code, making the environment predictable and easy to manage.

Key resources provisioned include:

- EC2 instances for compute
- EBS volumes for persistent storage
- Security groups to manage network access

- IAM roles and policies for secure permission control

This code-driven approach allows for version-controlled, consistent deployments that can be reproduced or modified with ease. It also simplifies infrastructure updates and makes it possible to deploy identical setups across multiple environments, improving reliability and operational efficiency.

5.5 MONITORING AND LOOGING

Service Used: Amazon CloudWatch

Amazon CloudWatch is used to enable real-time monitoring and centralized logging across the Smart Cloud Management Portal. All **Lambda functions** and **EC2 instances** are fully integrated with CloudWatch to capture logs and track key performance metrics.

It monitors system metrics such as **CPU utilization**, **disk I/O**, **network performance**, and **memory usage**, helping to detect anomalies and performance bottlenecks. Logs from backend operations and API activities are automatically recorded for audit and debugging purposes.

CloudWatch Alarms are configured to trigger actions when critical thresholds are breached—for example, initiating auto-scaling or recovery processes when CPU usage exceeds 80%. This monitoring setup enhances **system visibility**, supports proactive issue resolution, and contributes to overall **infrastructure optimization and reliability**.

5.6 DEPLOYMENT WORKFLOW

The deployment and integration of the Smart Cloud Management Portal follow a structured and automated DevOps pipeline to ensure consistency, scalability, and security.

- **Code Development:** Frontend and backend logic are developed locally using modern development tools and frameworks, ensuring modularity and maintainability.
- **Infrastructure Provisioning:** All AWS resources such as EC2, IAM, Lambda, and S3 are provisioned using Terraform, enabling Infrastructure as Code (IaC) for reproducibility and version control.
- **Frontend Deployment:** The React-based frontend is built locally and uploaded to an Amazon S3 bucket, then distributed globally via CloudFront for fast, secure access.
- **Backend Deployment:** AWS Lambda functions and API Gateway routes are deployed through Terraform or directly via the AWS Console for serverless execution.
- **Integration:** The frontend is connected to backend APIs securely using API Gateway endpoints, with JWT tokens from Amazon Cognito used for authorization.
- **Testing:** The complete system is tested to ensure end-to-end functionality, secure login, role-based access control, and reliable interaction between all components.

CHAPTER 6

CONCLUSION AND FUTURE ENHANCEMENT

6.1 CONCLUSION

The *Smart Cloud Management Portal* provides a secure, scalable, and user-friendly interface for managing AWS cloud infrastructure. By leveraging Amazon Web Services like EC2, Lambda, API Gateway, Cognito, and CloudWatch, the system enables both administrative and user-level access for launching, monitoring, and managing cloud resources. The implementation using serverless architecture ensures cost-effectiveness and high availability. Automation through Terraform further simplifies deployment and resource management. Overall, the project successfully demonstrates the integration of cloud-native technologies into a centralized platform that enhances productivity and operational efficiency for cloud users.

6.2 FUTURE ENHANCEMENT

To enhance the Smart Cloud Management Portal, several features can be added in the future:

1. Multi-Cloud Support to integrate platforms like Microsoft Azure and Google Cloud for broader cloud management.
2. Billing and Cost Analytics through integration with AWS Cost Explorer for cost tracking and budget reporting.
3. Mobile Application for managing cloud resources from mobile devices.
4. Auto-Scaling Optimization using AI/ML to predict and adjust scaling policies based on historical trends.
5. Notification System for alerts via email/SMS regarding instance status or resource utilization.

These enhancements aim to improve scalability, user experience, and security**.**

REFERENCES

[1] A. Koschel, S. Klassen, K. Jdiya, M. Schaaf and I. Astrova, "Cloud Computing: Serverless," *2021 12th International Conference on Information, Intelligence, Systems & Applications (IISA)*, Chania Crete, Greece, 2021, pp. 1-7, doi: 10.1109/IISA52424.2021.9555534.

[2] A. P. Rajan, "A review on serverless architectures - function as a service (FaaS) in cloud computing," TELKOMNIKA (Telecommunication Computing Electronics and Control), vol. 18, no. 1, pp. 47–54, 2020, doi: 10.12928/telkomnika.v18i1.12169.

[3] A. K. Jain, "Overview of Serverless Architecture," International Journal of Engineering Research & Technology (IJERT), vol. 11, no. 9, pp. 1–4, Sep. 2022, doi: 10.17577/IJERTV11IS090057.

[4] A. S. George and A. S. H. George, "Serverless Computing: the Next Stage in Cloud Computing's Evolution and an Empowerment of a New Generation of Developers," *International Journal of All Research Education and Scientific Methods (IJARESM)*, vol. 9, no. 4, pp. 21–25, Apr. 2021.

[5] B. R. Cherukuri, "Serverless computing: Exploring serverless architecture and its applications," *International Research Journal of Multidisciplinary Science & Technology*, vol. 7, no. 4, pp. 1–5, May 2024.

[6] H. Shafiei, A. Khonsari, and P. Mousavi, "Serverless computing: a survey of opportunities, challenges, and applications," *ACM Computing Surveys*, vol. 54, no. 11s, pp. 1–32, 2022.

[7] M. Ghorbian and M. Ghobaei-Arani, "Serverless Computing: Architecture, Concepts, and Applications," *arXiv preprint arXiv:2501.09831*.

[8] M. Kumar, "Serverless Architectures Review, Future Trend and the Solutions to Open Problems," *American Journal of Software Engineering*, vol. 6, no. 1, pp. 1–10, 2019, doi: 10.12691/ajse-6-1-1.

[9] P. Muralidhara and V. Jan**ardhan**, "Serverless Computing: Evaluating Performance, Scalability, and Cost-Effectiveness for Modern Applications," *International Journal of Engineering and Computer Science*, vol. 5, no. 8, pp. 17810–17834, Aug. 2016, doi: 10.18535/ijecs/v5i8.64.

[10] R. Sahay, "Serverless Computing," in *Microsoft Azure Architect Technologies Study Companion*, Apress, Berkeley, CA, 2020, pp. 433–496, doi: 10.1007/978-1-4842-6200-9_13.

[11] R. A. P. Rajan, "Serverless Architecture - A Revolution in Cloud Computing," 2018 Tenth International Conference on Advanced Computing (ICoAC), Chennai, India, 2018, pp. 88-93, doi: 10.1109/ICoAC44903.2018.8939081.

[12] S. Brenner and R. Kapitza, "Trust more, serverless," in *Proc. 12th ACM Int. Conf. on Systems and Storage (SYSTOR '19)*, Haifa, Israel, Jun. 2019, pp. 33–43, doi: 10.1145/3319647.3325825.

[13] S. R. Goniwada, "Serverless Architecture," in Cloud Native Architecture and Design, Apress, Berkeley, CA, 2022, pp. 295–324, doi: 10.1007/978-1-4842-7226-8_7.

[14] T. Lynn, P. Rosati, A. Lejeune, and V. Emeakaroha, "A Preliminary Review of Enterprise Serverless Cloud Computing (Function-as-a-Service) Platforms," 2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), Hong Kong, China, 2017, pp. 162-169, doi: 10.1109/CloudCom.2017.15.

[15] U. Shankar and V. K., "Serverless architecture: the future of computation," *International Journal of Current Advanced Research*, vol. 7, no. 12, pp. 16400–16403, Dec. 2018, doi: 10.24327/ijcar.2018.12612.2223.

[16] U. S. M. and V. K., "Serverless architecture the future of computation," *International Journal of Current Advanced Research*, vol. 7, no. 12, pp. 16400–16403, Dec. 2018, doi: 10.24327/ijcar.2018.12612.2223.

[17] V. Vudayagiri, "Demystifying Serverless Architecture for Scalable Web Applications," *International Journal of Scientific Research in Computer Science, Engineering and Information Technology*, vol. 10, no. 6, pp. 254–263, Nov.–Dec. 2024, doi: 10.32628/CSEIT24106176.

[18] M. Abdullahi et al., "Detecting cybersecurity attacks in Internet of Things using artificial intelligence methods: A systematic literature review," *Electronics*, vol. 11, no. 2, p. 198, 2022.

[19] M. Ghorbian and M. Ghobaei-Arani, "Serverless Computing: Architecture, Concepts, and Applications," arXiv preprint arXiv:2501.09831

[20] S. Spillner, M. Al-Ameen, and D. Boruta, "Serverless Literature Dataset," *Zenodo*, 2019, doi: 10.5281/zenodo.1175423.

APPENDIX A

CODING

MAIN.TF

```
provider "aws" {
  region = var.aws_region
}

resource "random_id" "bucket_id" {
  byte_length = 4
}

data "aws_ami" "amazon_linux" {
  most_recent = true
  owners      = ["amazon"]
  filter {
    name   = "name"
    values = ["amzn2-ami-hvm-*-x86_64-gp2"]
  }
}

resource "aws_vpc" "main" {
  cidr_block = "10.0.0.0/16"
}

resource "aws_subnet" "main" {
  vpc_id                  = aws_vpc.main.id
  cidr_block              = "10.0.1.0/24"
  availability_zone       = "us-east-1a"
  map_public_ip_on_launch = true
}

resource "aws_security_group" "ec2_sg" {
  name   = "ec2_sg"
  vpc_id = aws_vpc.main.id

  ingress {
    from_port   = 22
    to_port     = 22
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  ingress {
    from_port   = 80
    to_port     = 80
```

```
    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"]
  }
}

resource "aws_instance" "project" {
  ami                    = data.aws_ami.amazon_linux.id
  instance_type          = var.instance_type
  subnet_id              = aws_subnet.main.id
  vpc_security_group_ids = [aws_security_group.ec2_sg.id]
  key_name               = var.key_pair

  tags = {
    Name = "ProjectEC2"
  }
}

resource "aws_ebs_volume" "extra_storage" {
  size              = 10
  availability_zone = aws_instance.project.availability_zone

  lifecycle {
    ignore_changes = [size]
  }
}

resource "aws_volume_attachment" "ebs_att" {
  device_name = "/dev/sdh"
  volume_id   = aws_ebs_volume.extra_storage.id
  instance_id = aws_instance.project.id
}

resource "aws_iam_role" "lambda_exec_role" {
  name = "lambda_exec_role"

  assume_role_policy = jsonencode({
    Version = "2012-10-17",
    Statement = [{
      Action = "sts:AssumeRole",
      Effect = "Allow",
      Principal = {
        Service = "lambda.amazonaws.com"
      }
```

```
    }]
  })
}

resource "aws_iam_policy" "lambda_ec2_permissions" {
  name        = "LambdaEC2Permissions"
  description = "Allow Lambda to manage EC2 instances"

  policy = jsonencode({
    Version = "2012-10-17",
    Statement = [{
      Effect = "Allow",
      Action = [
        "ec2:RunInstances",
        "ec2:CreateTags",
        "ec2:DescribeInstances",
        "ec2:DescribeImages",
        "ec2:DescribeSecurityGroups",
        "ec2:DescribeSubnets",
        "ec2:DescribeVpcs",
        "ec2:AttachVolume",
        "ec2:RebootInstances",
        "ec2:ModifyVolume",
        "ec2:CreateSnapshot",
        "ec2:DescribeSnapshots",
        "ec2:StopInstances",
        "ec2:StartInstances",
        "ec2:ModifyInstanceAttribute",
        "iam:PassRole"
      ]
      Resource = "*"
    }]
  })
}

resource "aws_iam_policy_attachment" "lambda_ec2_attach" {
  name       = "lambda-ec2-access"
  roles      = [aws_iam_role.lambda_exec_role.name]
  policy_arn = aws_iam_policy.lambda_ec2_permissions.arn
}

resource "aws_iam_policy_attachment" "lambda_basic_attach" {
  name       = "lambda-basic-exec"
  roles      = [aws_iam_role.lambda_exec_role.name]
  policy_arn = "arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole"
}

locals {
  lambda_functions = {
    reset = {
```

```
    handler = "reset.lambda_handler"
   }
   extend = {
    handler = "extend.lambda_handler"
   }
   stop = {
         handler = "stop.lambda_handler"
   scale = {
    handler = "scale.lambda_handler"
   }

  }
}
}

resource "aws_lambda_function" "functions" {
  for_each = local.lambda_functions

  function_name     = each.key
  filename          = "${path.module}/lambda/${each.key}.zip"
  source_code_hash  = filebase64sha256("${path.module}/lambda/${each.key}.zip")
  handler           = each.value.handler
  runtime           = "python3.8"
  role              = aws_iam_role.lambda_exec_role.arn

  timeout = 60
  environment {
   variables = {
    INSTANCE_ID = aws_instance.project.id
    AMI_ID      = data.aws_ami.amazon_linux.id
    KEY_NAME    = var.key_pair
    SUBNET      = aws_subnet.main.id
    SG_ID       = aws_security_group.ec2_sg.id
    VOLUME_ID   = aws_ebs_volume.extra_storage.id
   }
  }
}

resource "aws_lambda_permission" "allow_api_gateway" {
  for_each = aws_lambda_function.functions

  statement_id  = "AllowExecutionFromAPIGateway-${each.key}"
  action        = "lambda:InvokeFunction"
  function_name = each.value.function_name
  principal     = "apigateway.amazonaws.com"
  source_arn    = "${aws_api_gateway_rest_api.main.execution_arn}/*/*"
}

resource "aws_api_gateway_rest_api" "main" {
  name = "dashboard-api"
```

```
}

resource "aws_api_gateway_resource" "lambda_resource" {
  for_each    = aws_lambda_function.functions
  rest_api_id = aws_api_gateway_rest_api.main.id
  parent_id   = aws_api_gateway_rest_api.main.root_resource_id
  path_part   = each.key
}

resource "aws_api_gateway_method" "lambda_post" {
  for_each      = aws_api_gateway_resource.lambda_resource
  rest_api_id   = each.value.rest_api_id
  resource_id   = each.value.id
  http_method   = "POST"
  authorization = "NONE"
}

resource "aws_api_gateway_integration" "lambda_integration" {
  for_each              = aws_api_gateway_method.lambda_post
  rest_api_id           = each.value.rest_api_id
  resource_id           = each.value.resource_id
  http_method           = each.value.http_method
  integration_http_method = "POST"
  type                  = "AWS_PROXY"
  uri                   = "arn:aws:apigateway:${var.aws_region}:lambda:path/2015-03-
31/functions/${aws_lambda_function.functions[each.key].arn}/invocations"
}

resource "aws_api_gateway_deployment" "deploy" {
  depends_on  = [aws_api_gateway_integration.lambda_integration]
  rest_api_id = aws_api_gateway_rest_api.main.id
}

resource "aws_api_gateway_stage" "prod" {
  deployment_id = aws_api_gateway_deployment.deploy.id
  rest_api_id   = aws_api_gateway_rest_api.main.id
  stage_name    = "prod"
}

resource "aws_cloudwatch_metric_alarm" "alarms" {
  for_each = {
    cpu     = { metric = "CPUUtilization", ns = "AWS/EC2", threshold = 70, desc = "CPU
usage > 70%" }
    memory  = { metric = "mem_used_percent", ns = "CWAgent", threshold = 70, desc =
"Memory usage > 70%" }
    disk    = { metric = "disk_used_percent", ns = "CWAgent", threshold = 70, desc = "Disk
usage > 70%" }
    network = { metric = "NetworkIn", ns = "AWS/EC2", threshold = 104857600, desc =
"Network In > 100MB" }
  }
```

```
  alarm_name          = "${each.key}-alarm"
  comparison_operator = "GreaterThanThreshold"
  evaluation_periods  = 1
  metric_name         = each.value.metric
  namespace           = each.value.ns
  period              = 60
  statistic           = "Average"
  threshold           = each.value.threshold
  alarm_actions       = [aws_sns_topic.alerts.arn]
  dimensions = {
    InstanceId = aws_instance.project.id
  }
  alarm_description = each.value.desc
}

resource "aws_sns_topic" "alerts" {
  name = "dashboard-alerts"
}

resource "aws_sns_topic_subscription" "sms" {
  topic_arn = aws_sns_topic.alerts.arn
  protocol  = "sms"
  endpoint  = "+919087827374"
}

resource "aws_cognito_user_pool" "user_pool" {
  name = "dashboard-users"
}

resource "aws_cognito_user_pool_client" "user_pool_client" {
  name            = "web-client"
  user_pool_id    = aws_cognito_user_pool.user_pool.id
  generate_secret = false
}

resource "aws_cognito_user_group" "admin_group" {
  user_pool_id = aws_cognito_user_pool.user_pool.id
  name         = "admin"
  description  = "Control node users"
  precedence   = 1
}

resource "aws_cognito_user" "dashboard_user" {
  user_pool_id         = aws_cognito_user_pool.user_pool.id
  username             = "admin"
  force_alias_creation = false
  message_action       = "SUPPRESS"
  attributes = {
    email        = "kishoresuzil1005@gmail.com"
    phone_number = "+919087827374"
```

```
  }
}

resource "null_resource" "admin_group_add" {
  provisioner "local-exec" {
    command = <<EOT
      aws cognito-idp admin-add-user-to-group \
        --user-pool-id ${aws_cognito_user_pool.user_pool.id} \
        --username ${aws_cognito_user.dashboard_user.username} \
        --group-name ${aws_cognito_user_group.admin_group.name}
    EOT
  }
  triggers = {
    user = aws_cognito_user.dashboard_user.id
  }
}

resource "aws_s3_bucket" "dashboard" {
  bucket        = "dashboard-bucket-${random_id.bucket_id.hex}"
  force_destroy = true
}

resource "aws_s3_bucket_public_access_block" "dashboard_bucket_block" {
  bucket = aws_s3_bucket.dashboard.id

  block_public_acls       = false
  block_public_policy     = false
  ignore_public_acls      = false
  restrict_public_buckets = false
}

resource "aws_s3_bucket_website_configuration" "dashboard" {
  bucket = aws_s3_bucket.dashboard.id

  index_document {
    suffix = "index.html"
  }
}

resource "aws_s3_bucket_policy" "dashboard_public" {
  bucket = aws_s3_bucket.dashboard.id

  policy = jsonencode({
    Version = "2012-10-17",
    Statement = [{
      Effect    = "Allow",
      Principal = "*",
      Action    = "s3:GetObject",
      Resource  = "${aws_s3_bucket.dashboard.arn}/*"
    }]
```

```
  })

  depends_on = [aws_s3_bucket_public_access_block.dashboard_bucket_block]
}

resource "aws_s3_object" "dashboard_files" {
  for_each = fileset("${path.module}/project-dashboard", "**")
  bucket       = aws_s3_bucket.dashboard.id
  key          = each.key
  source       = "${path.module}/project-dashboard/${each.key}"
  content_type = lookup({
    html = "text/html"
    css  = "text/css"
    js   = "application/javascript"
  }, split(".", each.key)[length(split(".", each.key)) - 1], "text/plain")
}
```

VARIABLES.TF

```
variable "instance_type" {
  description = "EC2 instance type"
  type        = string
  default     = "t2.micro"
}

variable "key_pair" {
  description = "EC2 Key pair name"
  type        = string
  default     = "project"
}
variable "aws_region" {
  default = "us-east-1"
}
```

OUTPUTS.TF

```
output "cognito_user_pool_id" {
  description = "Cognito user pool ID"
  value       = aws_cognito_user_pool.user_pool.id
}

output "cognito_user_pool_client_id" {
  description = "Cognito user pool client ID"
  value       = aws_cognito_user_pool_client.user_pool_client.id
}

output "dashboard_url" {
  description = "S3 website URL for the dashboard"
```

```
  value       = aws_s3_bucket_website_configuration.dashboard.website_endpoint
}

output "lambda_function_names" {
  description = "Deployed Lambda function names"
  value       = { for name, fn in aws_lambda_function.functions : name => fn.function_name }
}

output "cpu_alarm_id" {
  description = "CloudWatch Alarm ID for CPU"
  value       = aws_cloudwatch_metric_alarm.alarms["cpu"].id
}
```

APPENDIX B

SCREENSHOTS

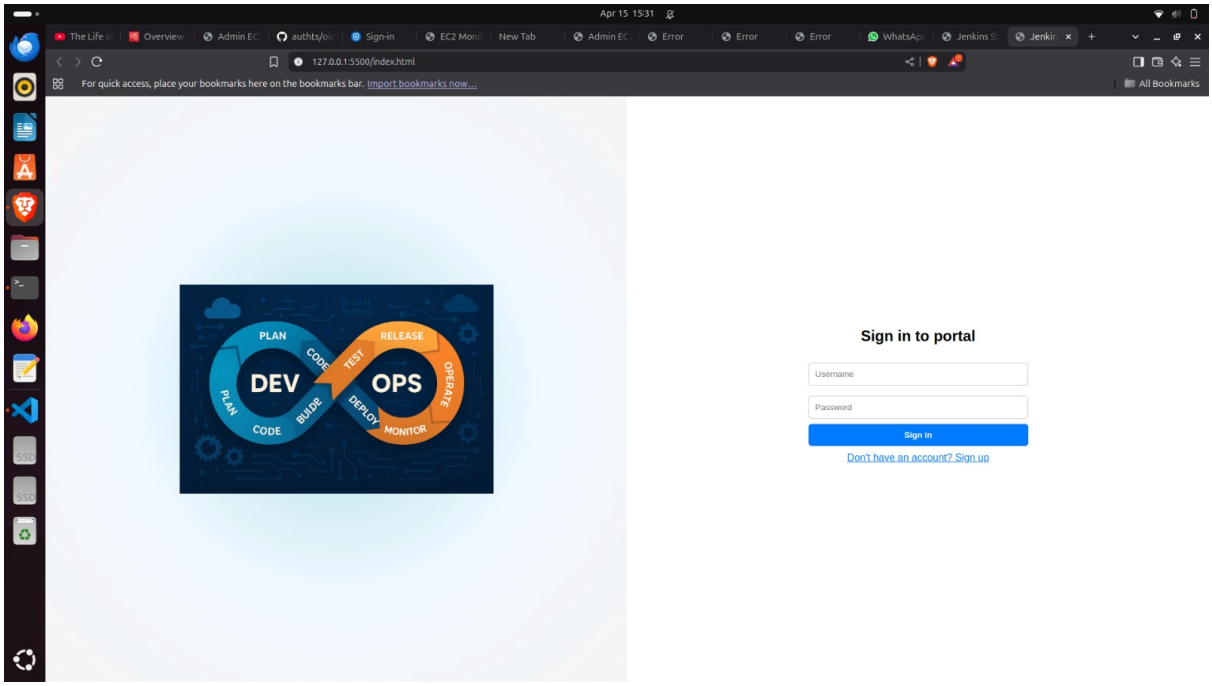

Fig B.1 Sign Page

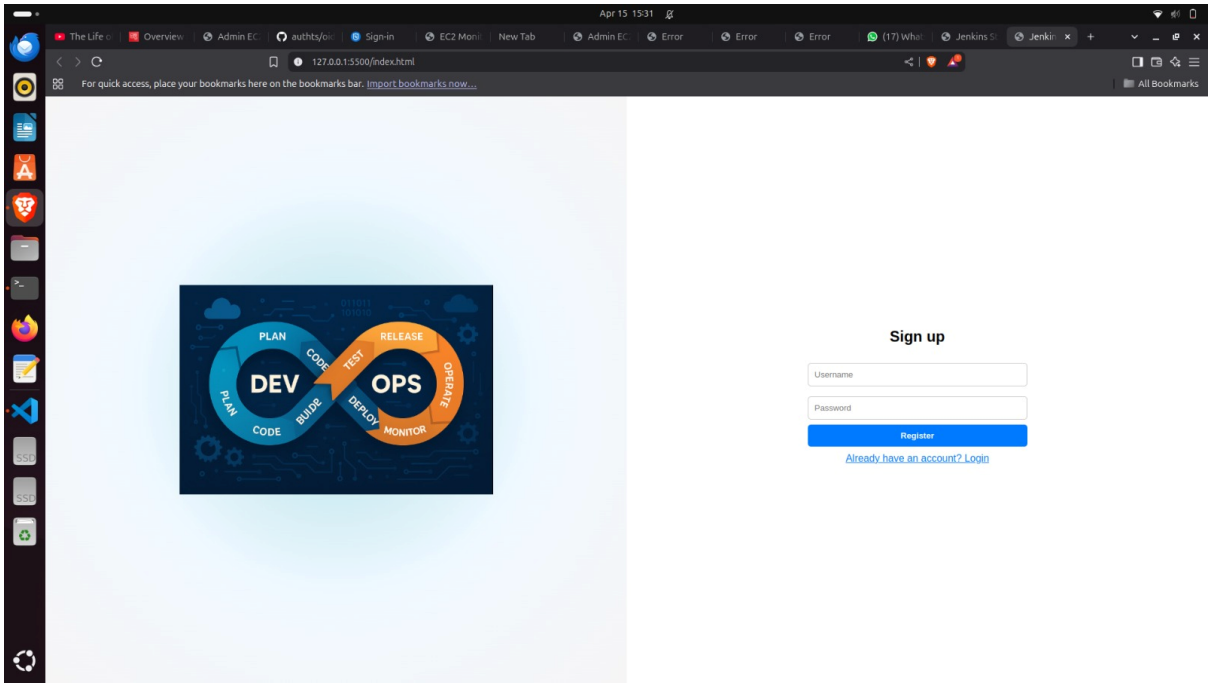

Fig B.2 Sign up page

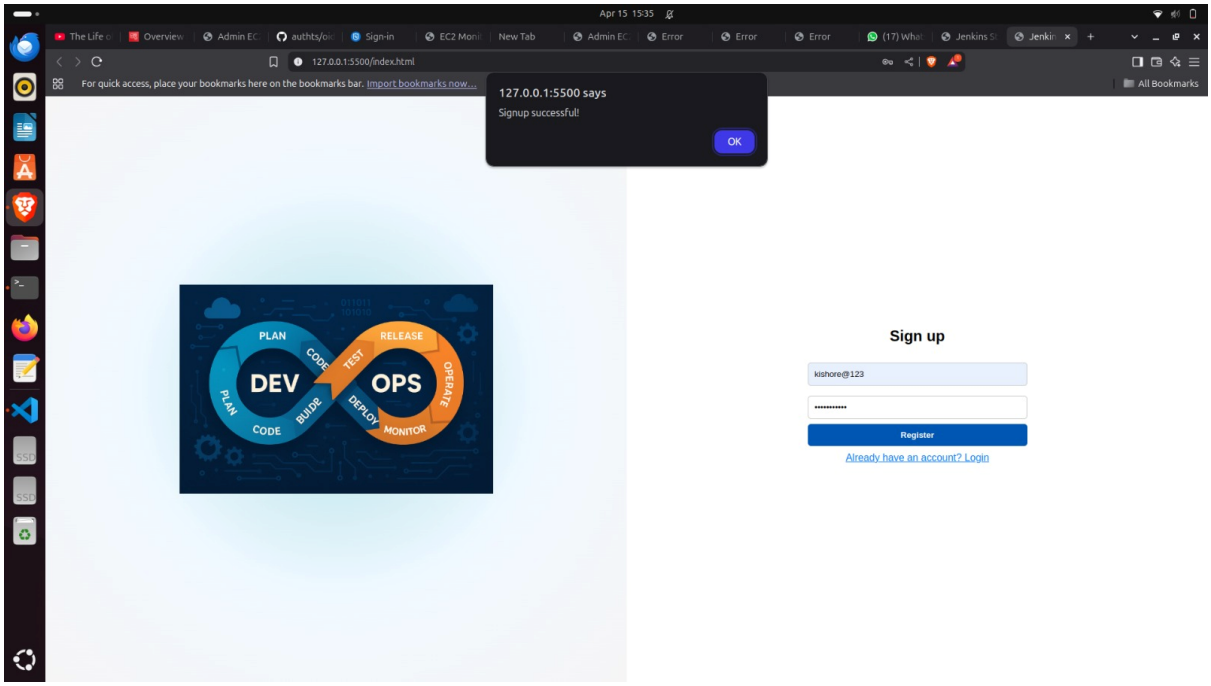

Fig B.3 Sign up page

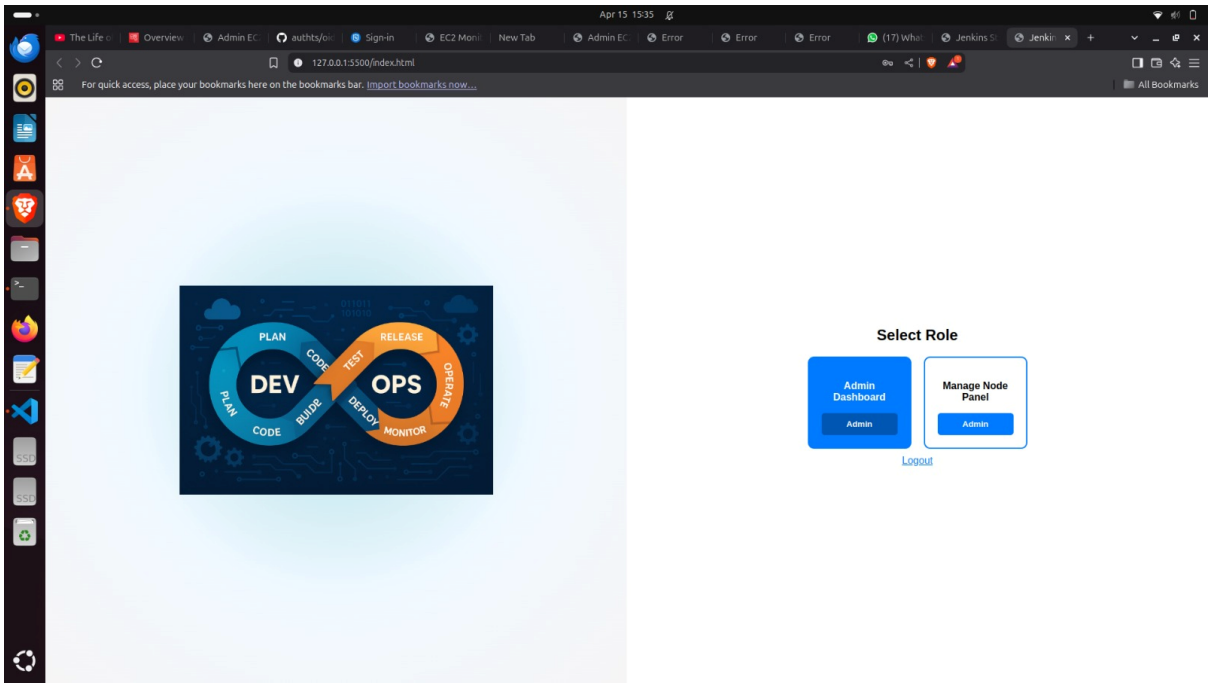

Fig B.4 Select Role

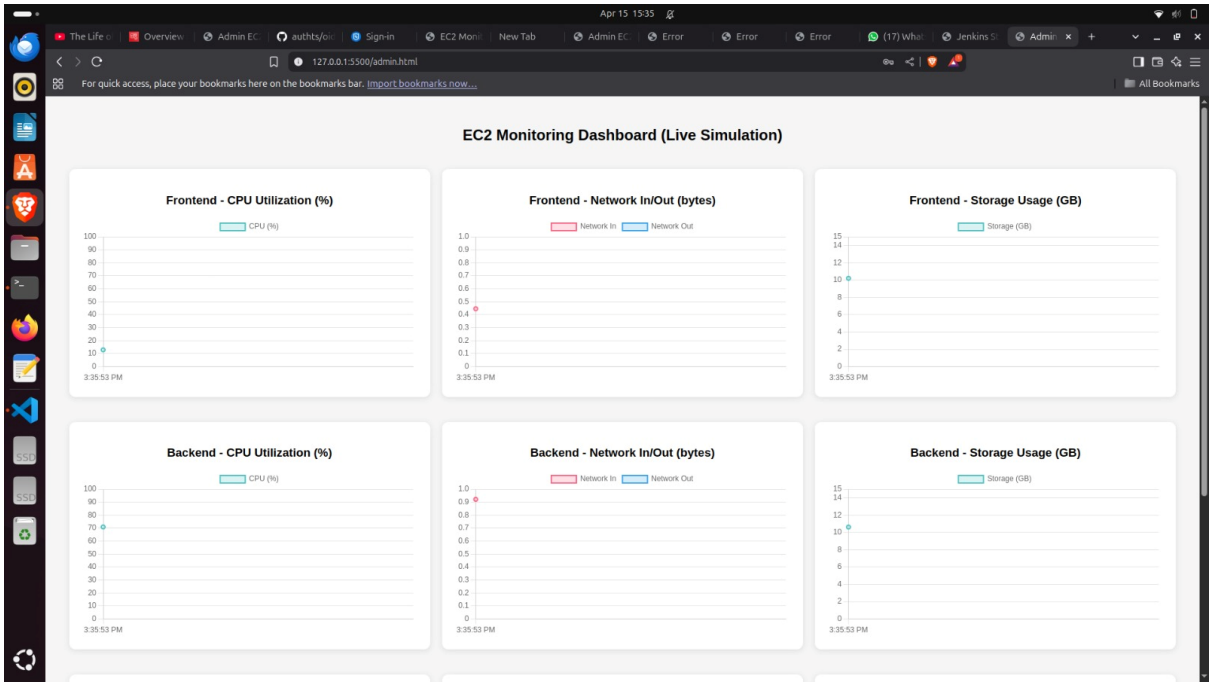

Fig B.5 Admin page

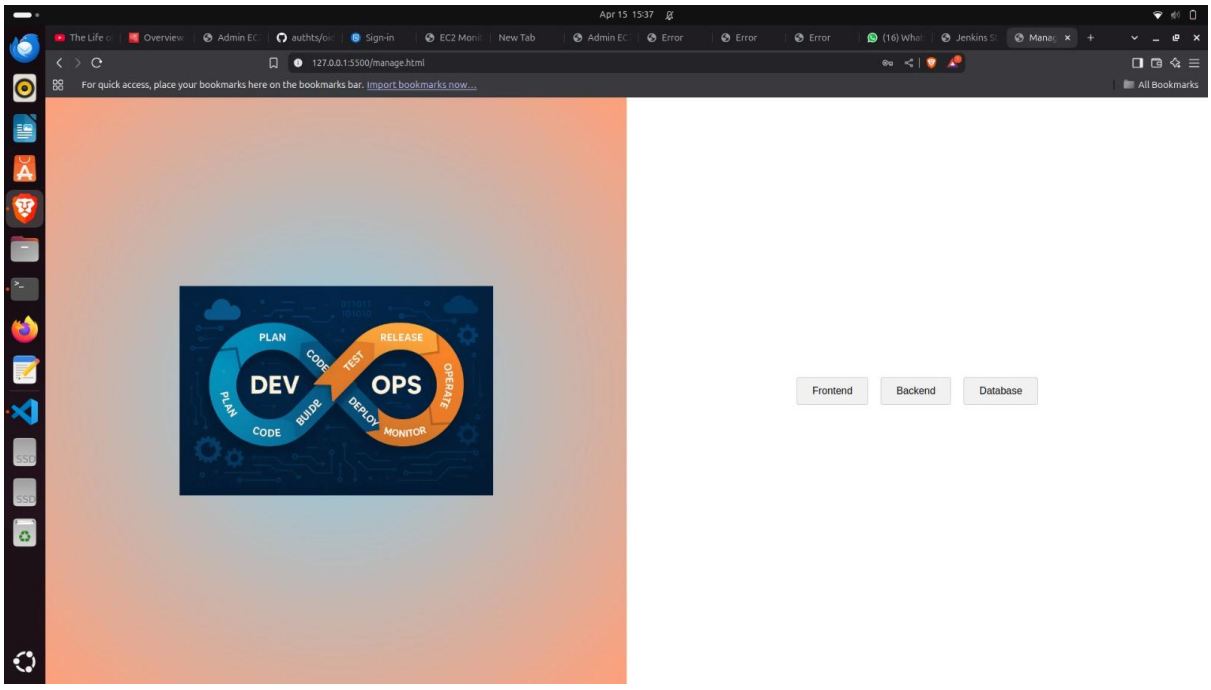

Fig B.6 Manage page

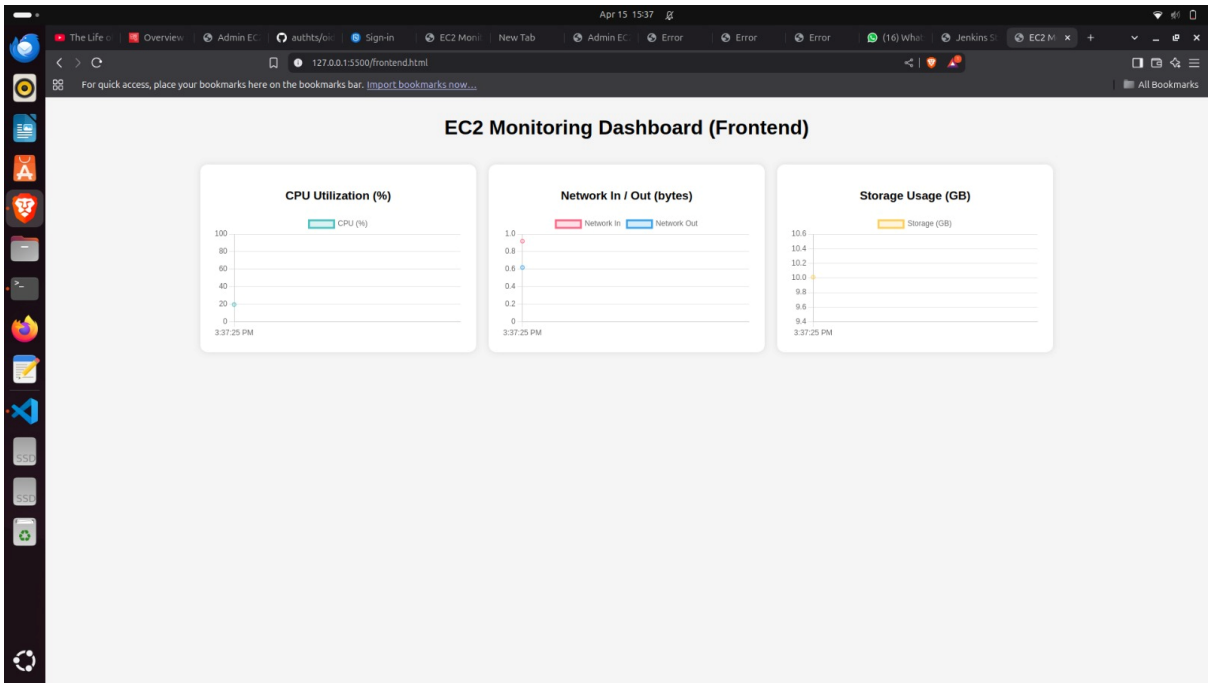

Fig B.7 Frontend page

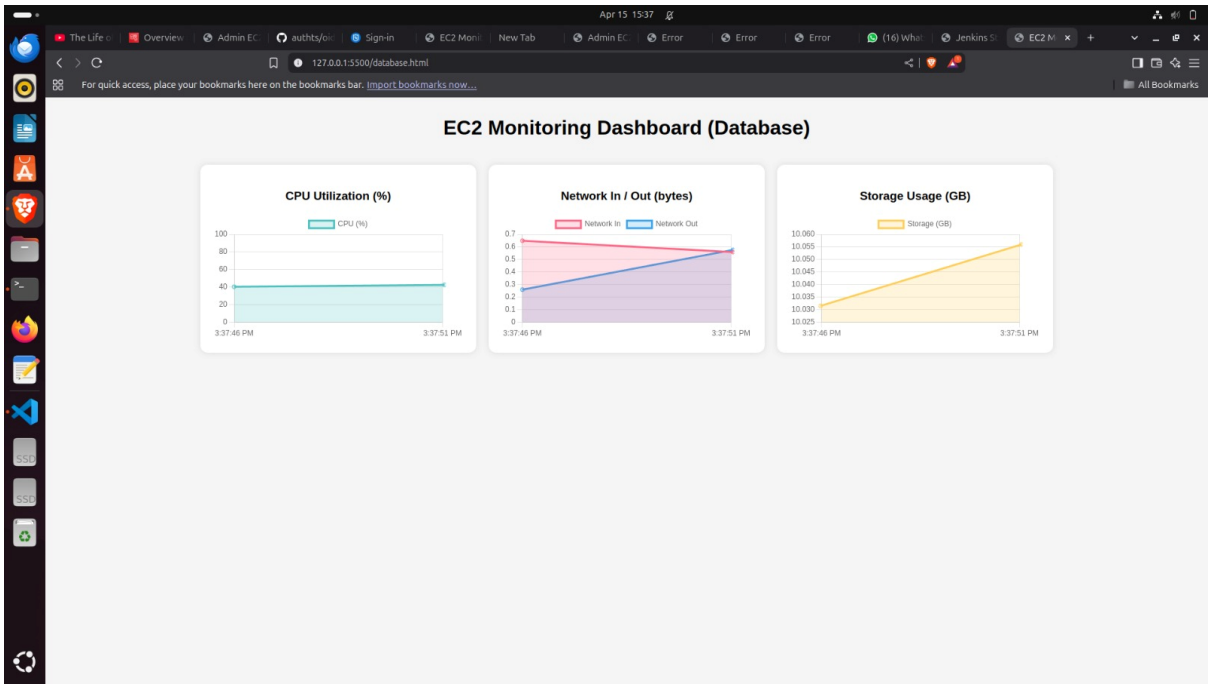

Fig B.8 Database page

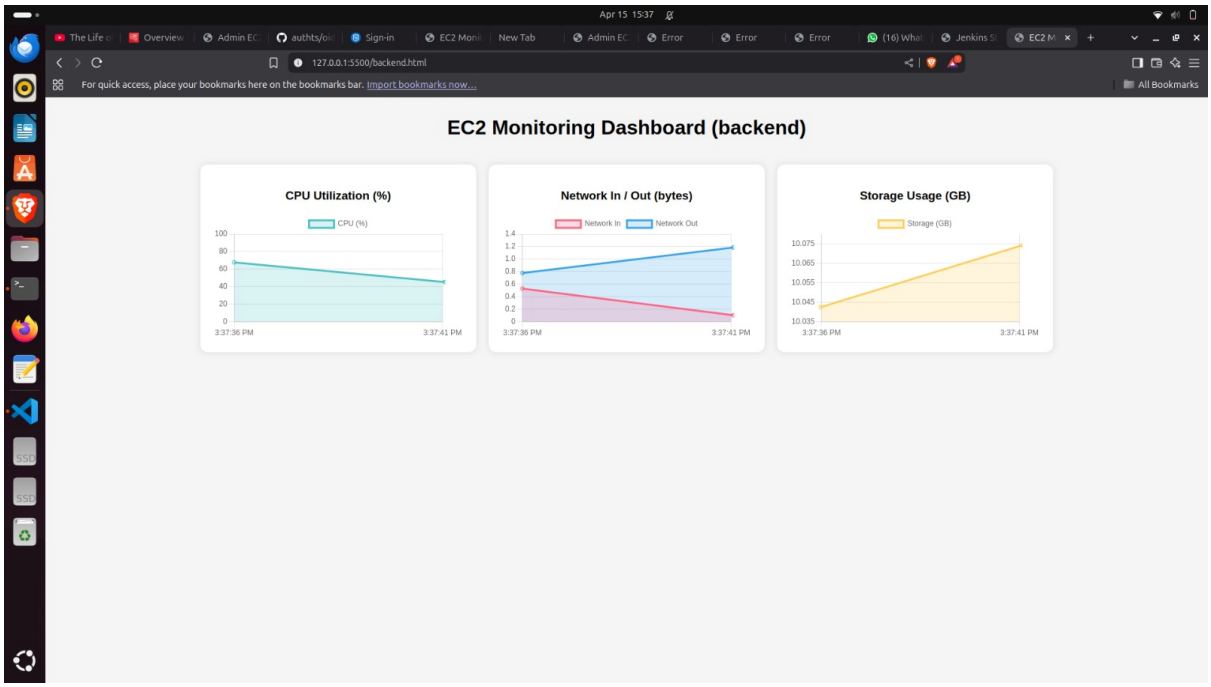

Fig B.9 Backend Page

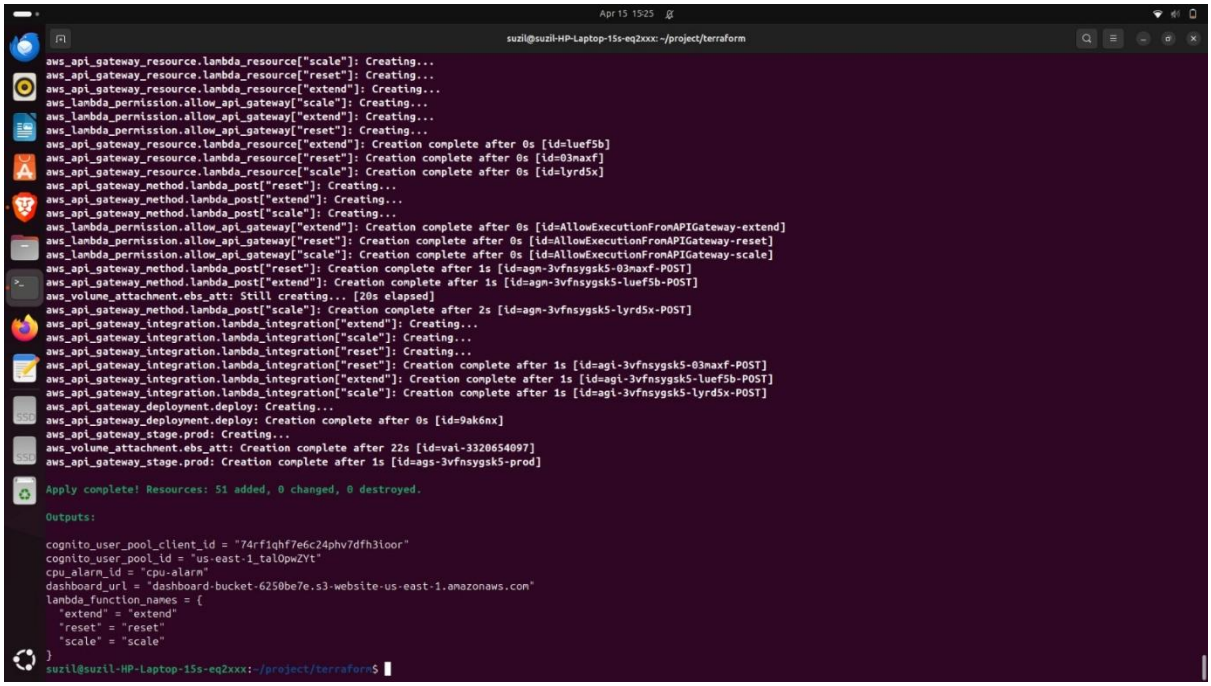

Fig B.10 Terraform output

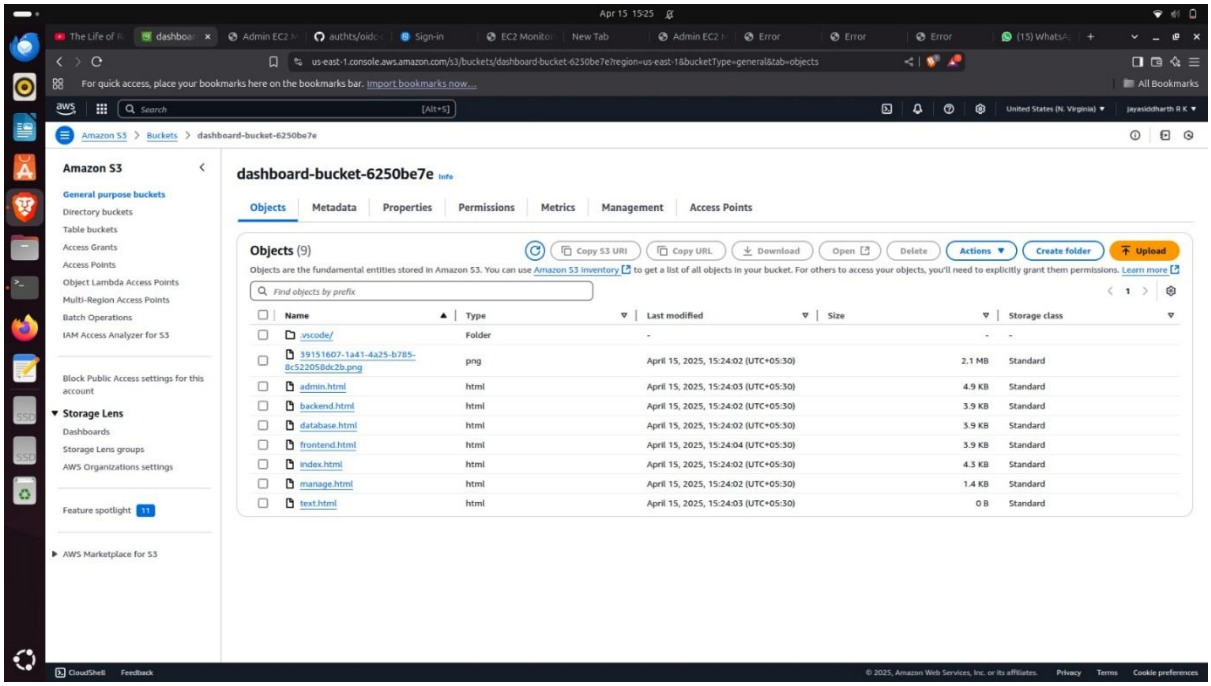

Fig B.11 S3 bucket

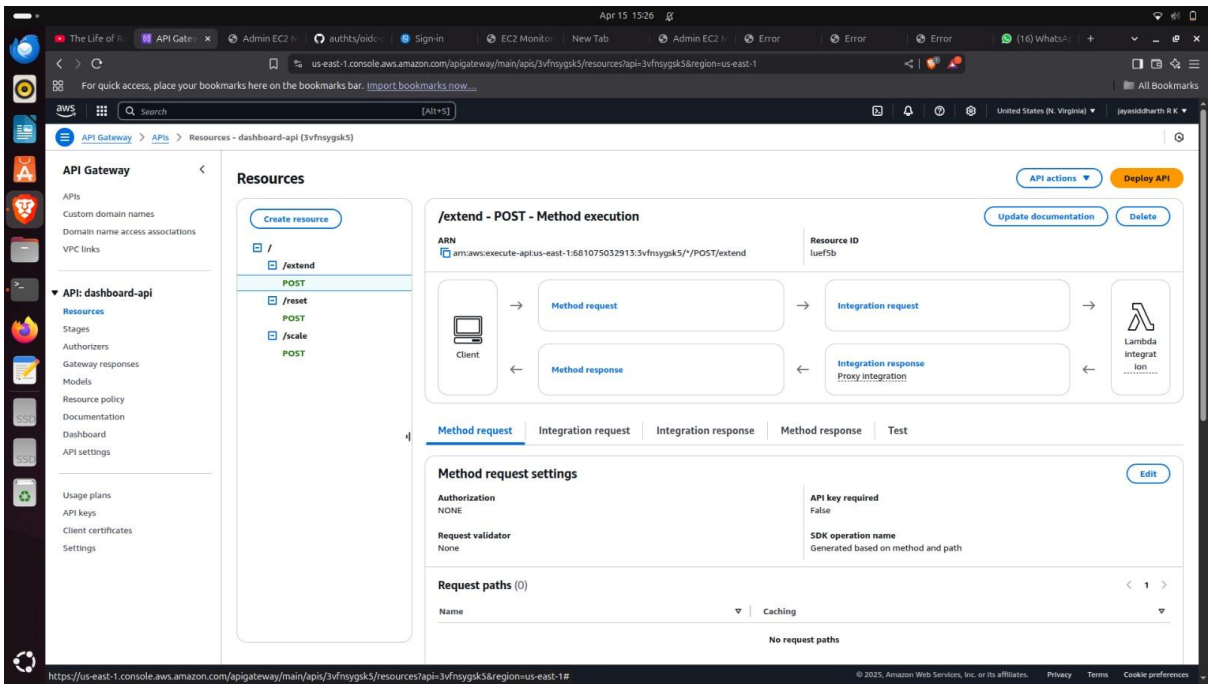

Fig B.12 Api gateway

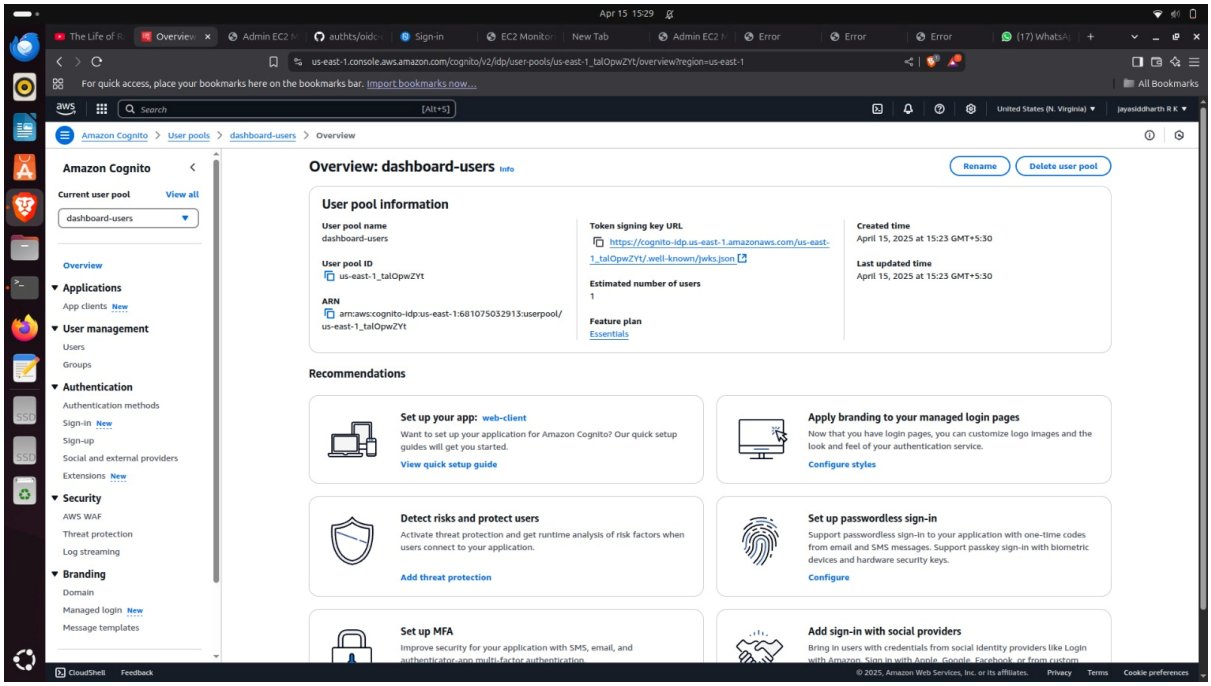

Fig B.13 Amazon cognito