PROJECT REPORT

Scalable PHP and Flask Web Application Deployment on AWS EKS with Multi-IaC Strategy and Full DevOps Lifecycle

JUNE 26, 2025

UST GLOBAL
India

Project Overview

This project delivers a comprehensive deployment of a highly available and disaster-resilient multi-tier web application on Amazon Web Services (AWS). It comprises a PHP-based frontend, a Flask/Python backend, and a MySQL database, all containerized and deployed within Amazon Elastic Kubernetes Service (EKS).

The application architecture is distributed across two AWS regions: **us-east-1** (primary) and **us-west-1** (disaster recovery). Infrastructure provisioning is automated through Terraform (for the primary region) and AWS CloudFormation (for the DR region). A complete CI/CD pipeline enables consistent builds, testing, and deployments, while integrated monitoring and automated DNS failover ensure high availability and rapid recovery in case of regional failure.

Application Stack

The application is built using a multi-tier architecture and utilizes modern, cloud-native technologies. The table below summarizes the core components of the stack:

Tier	**Technology**
Frontend	PHP (HTML/CSS rendered)
Backend	Flask (Python REST APIs)
Database	Amazon RDS MySQL (Multi-AZ)
Containerization	Docker
Orchestration	Amazon EKS (Kubernetes)
Load Balancing	Application Load Balancer (ALB)
Networking	VPC with public/private subnets, NAT Gateways, Route Tables

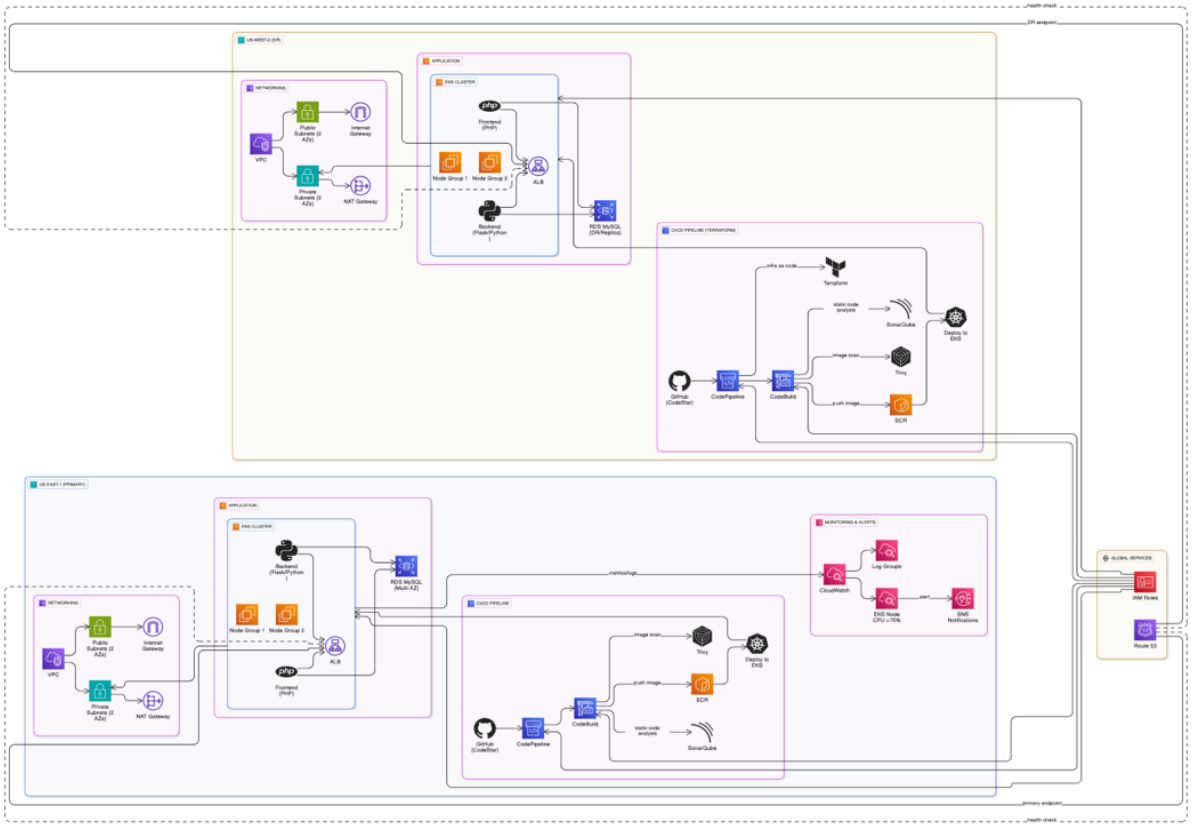
Project Overview & Architecture

Core Concepts

- **Multi-Tier Application:** A classic three-tier architecture separates the user interface (Frontend), business logic (Backend), and data storage (Database) for scalability and maintainability.
- **Infrastructure as Code (IaC):** The entire infrastructure is defined as code. The primary region (us-east-1) is deployed using **Terraform**, and the disaster recovery region (us-west-1) is deployed using **AWS CloudFormation**. This ensures consistent, repeatable, and version-controlled environments.
- **CI/CD Automation:** AWS CodePipeline automates the entire release process. Code changes pushed to GitHub trigger automated builds, security scans, and deployments to the EKS cluster.
- **Disaster Recovery (DR):** An active-passive DR strategy is implemented using Amazon Route 53's DNS failover. If the primary region becomes unhealthy, traffic is automatically rerouted to the standby region with minimal downtime.
- **Comprehensive Monitoring:** Amazon CloudWatch, including Container Insights, provides deep visibility into the application and infrastructure performance. Alarms are configured with Amazon SNS to notify the operations team of issues like high CPU usage or pipeline failures.

Architecture Diagram

The architecture consists of two mirrored regional deployments managed by a global Route 53 layer.



Common Architecture and AWS Resources Across Regions

To support high availability and fault tolerance, the application infrastructure is mirrored across two AWS regions: us-east-1 (primary) and us-west-1 (disaster recovery). The following AWS services and resources are consistently provisioned in both regions to ensure functional parity and seamless failover support.

Networking Layer

Resource	Description
VPC (Virtual Private Cloud)	Custom VPC created for the application in each region with a /16 CIDR block (e.g., 10.0.0.0/16), providing private IP addressing for all AWS resources.
Subnets (4 total)	Each VPC is configured with subnets across 2 Availability Zones for high availability: 2 Public Subnets for the ALB and NAT Gateways, and 2 Private Subnets for the EKS worker nodes and RDS database.
Internet Gateway (IGW)	Attached to the VPC to allow internet access for resources in the public subnets, such as the Application Load Balancer.
NAT Gateways	Placed in each AZ's public subnet, allowing resources in the private subnets (like EKS nodes) to securely access the internet for updates and pulling container images.
Route Tables	Custom route tables direct traffic within the VPC; the public route table forwards traffic to the IGW, while private route tables forward outbound internet traffic through the NAT gateways.

Compute Layer: Amazon EKS

Resource	Description
Amazon EKS Cluster	A fully managed Kubernetes control plane named three-tier-cluster is deployed in each region to orchestrate the application's containers and deployments.
EKS Node Group (Managed)	An auto-scaled group of t3.medium EC2 instances is managed by EKS to serve as the worker nodes for the Kubernetes cluster, running across private subnets for high availability.
Security Groups for EKS	Granular security groups control traffic to and from the EKS nodes, allowing inbound traffic from the ALB, outbound traffic to the RDS instance on port 3306, and necessary internal cluster communication.
Amazon ECR	Private Elastic Container Registry repositories are created in each region to store the frontend and backend Docker images securely.

IAM (Identity & Access Management) Roles

Role	Usage & Permissions
EKS Cluster Role	Grants permissions for EKS control plane to interact with other AWS services (e.g., ELB, Auto Scaling, etc.). - AmazonEKSClusterPolicy, AmazonEKSServicePolicy
EKS Node IAM Role	Attached to the EC2 worker nodes to allow access to Amazon ECR and other AWS resources. - AmazonEC2ContainerRegistryReadOnly, AmazonEKSWorkerNodePolicy, AmazonEKS_CNI_Policy, CloudWatchAgentServerPolicy
CodeBuild Role	Executes pipeline build jobs. Has permissions to: - Build/push Docker images - Pull Secrets - Update EKS (via kubectl) - Needs inline policy for eks:DescribeCluster, eks:UpdateConfigMap - Needs AmazonEC2ContainerRegistryPowerUser
CodePipeline Role	Coordinates actions between GitHub (source), CodeBuild (build), and EKS (deploy). Includes permissions for: - codecommit:*, codebuild:*, eks:*, s3:*
CloudFormation Execution Role	Created during stack deployment. Allows CFN to create/modify AWS resources.

Prerequisites

Before you begin, ensure you have the following:

- An **AWS Account** with administrative privileges.
- **AWS CLI** installed and configured on your local machine.
- **Terraform** installed.
- **kubectl** (Kubernetes command-line tool) installed.
- **Docker Desktop** installed and running.
- A **GitHub Account** to host the application and IaC repositories.
- A registered domain name in **Amazon Route 53**.

Step-by-Step Implementation

This guide will walk you through setting up the primary region, building the application CI/CD pipeline, provisioning the DR region, and configuring failover and monitoring.

Part 1: Setting Up the GitHub Repositories

First, structure your code in GitHub. Based on the project, you will need at least three repositories:

1. **Terraform-Configuration-Files:** Contains the Terraform code responsible for provisioning the **primary infrastructure in** us-east-1. This includes VPCs, subnets, compute resources, and other foundational AWS infrastructure.

2. **PHP-Flask-application:** Stores the application codebase, including:

    - PHP-based frontend
    - Flask-powered Python backend
    - Kubernetes manifests for deployment
    - buildspec.yml for the CI/CD pipeline This repository is linked to the infrastructure created via Terraform.

3. **CloudFormation-Template:** Hosts CloudFormation templates for setting up a **disaster recovery environment in** us-west-1. This includes infrastructure similar to the primary region and has **additional logic to automatically set up an application pipeline** (e.g., CodePipeline, CodeBuild).

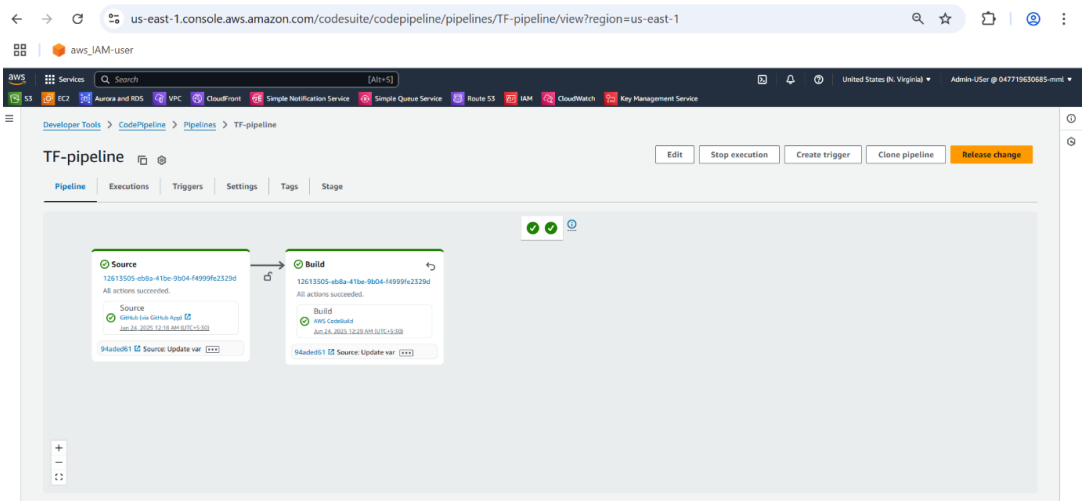
4. **PHP-Flask-application-CF:** Similar to the PHP-Flask-application repo, this holds the app code and deployment configuration, but tailored for the **CloudFormation-managed DR region**. It mirrors the code structure with minor tweaks to align with the CF-based infrastructure setup.

Part 2: Provisioning the Primary Region (us-east-1) with Terraform

We will use a CI/CD pipeline to deploy our Terraform code, ensuring an automated and repeatable process.

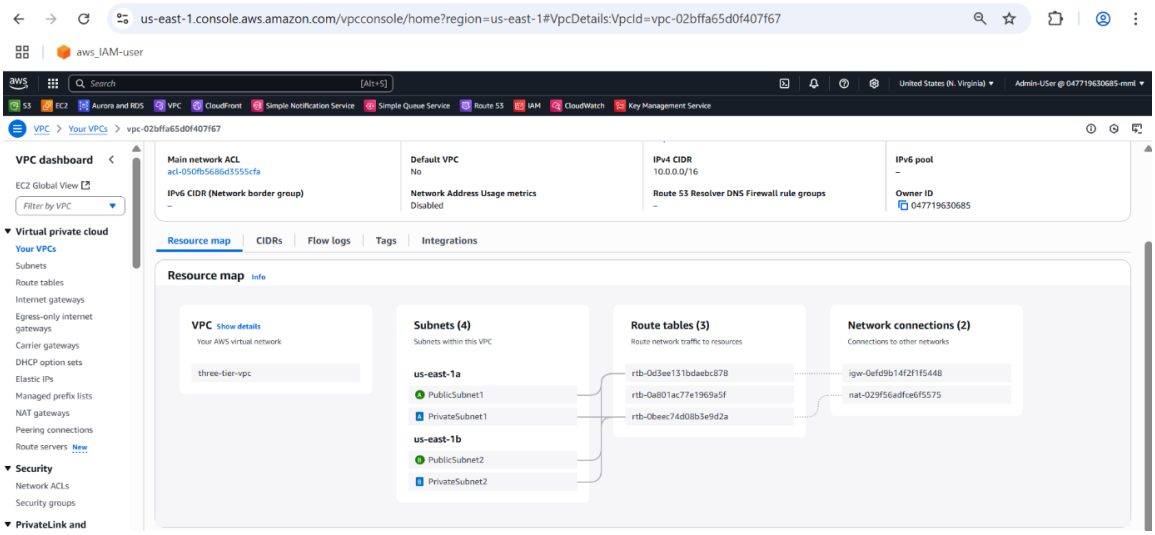
Step 1: Terraform Infrastructure Pipeline

- **CI/CD Tool**: AWS CodePipeline (Region: us-east-1)
- **Pipeline Name**: TF-pipeline
- **Source Stage**:
    - Source Provider: GitHub (Version 2)
    - Repository: Terraform-Configuration-Files
    - Branch: main
- **Build Stage**:
    - Provider: AWS CodeBuild
    - This CodeBuild project is responsible for reading the Terraform configuration files and applying them.
    - It uses a buildspec.yml file

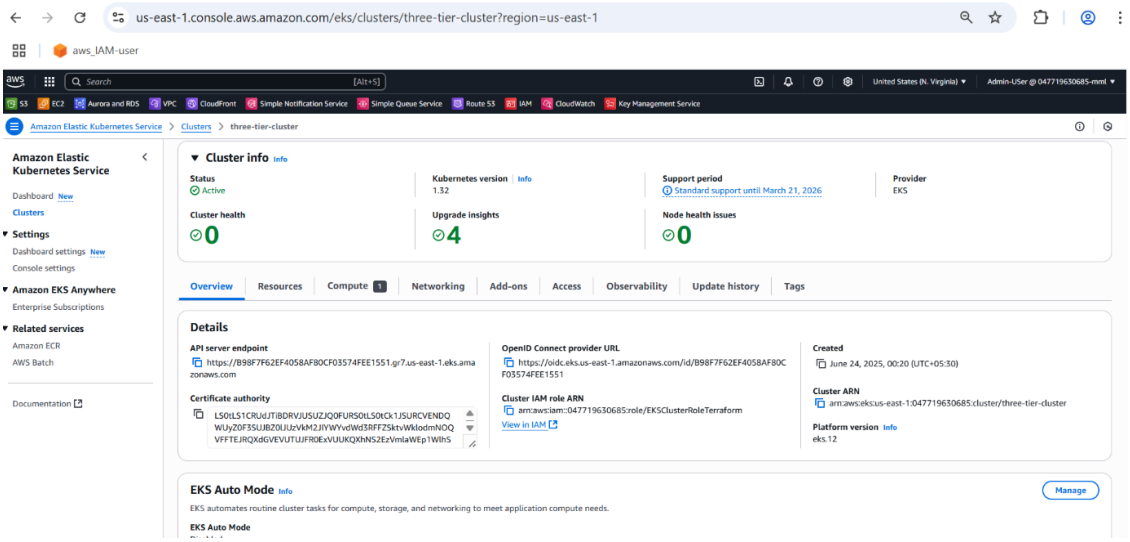


Once this pipeline runs, it provisions the following infrastructure in the us-east-1 region:

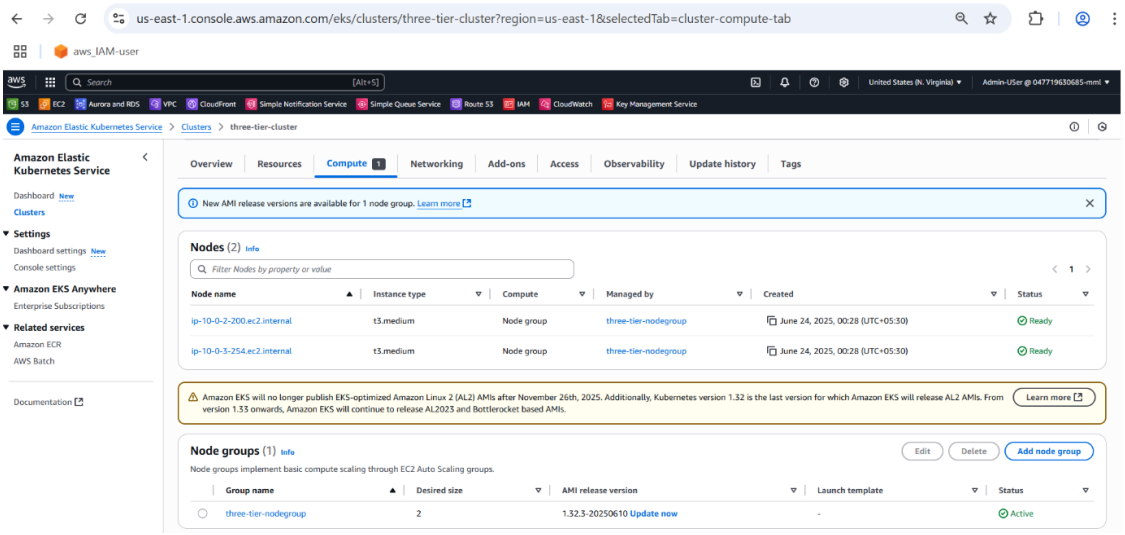
- A Virtual Private Cloud (VPC) with two public and two private subnets across two Availability Zones



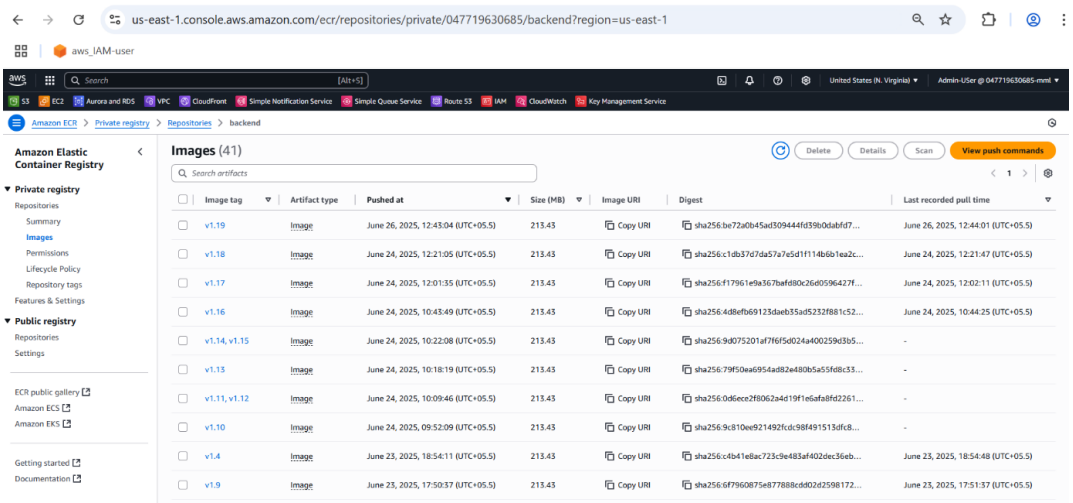
- An Amazon EKS cluster named three-tier-cluster

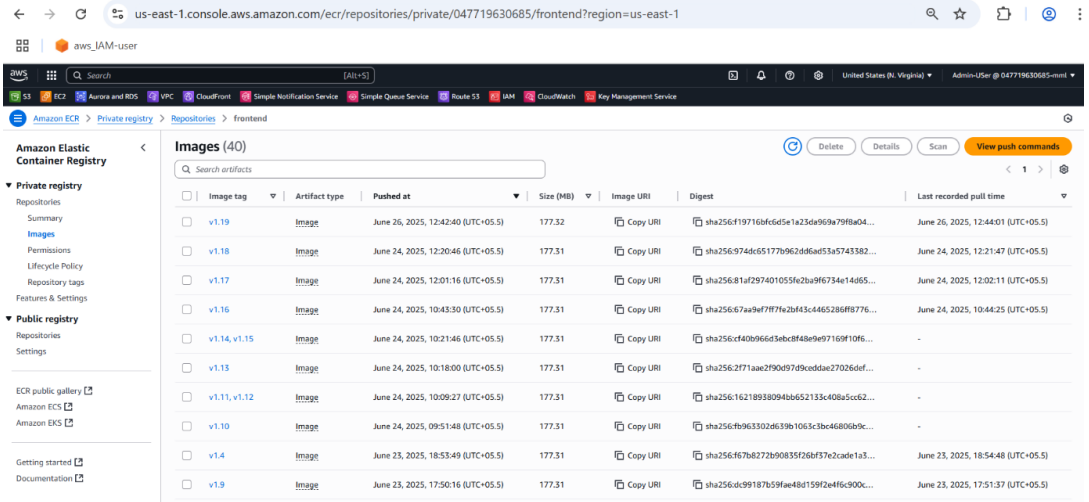


- An EKS node group to host application pods

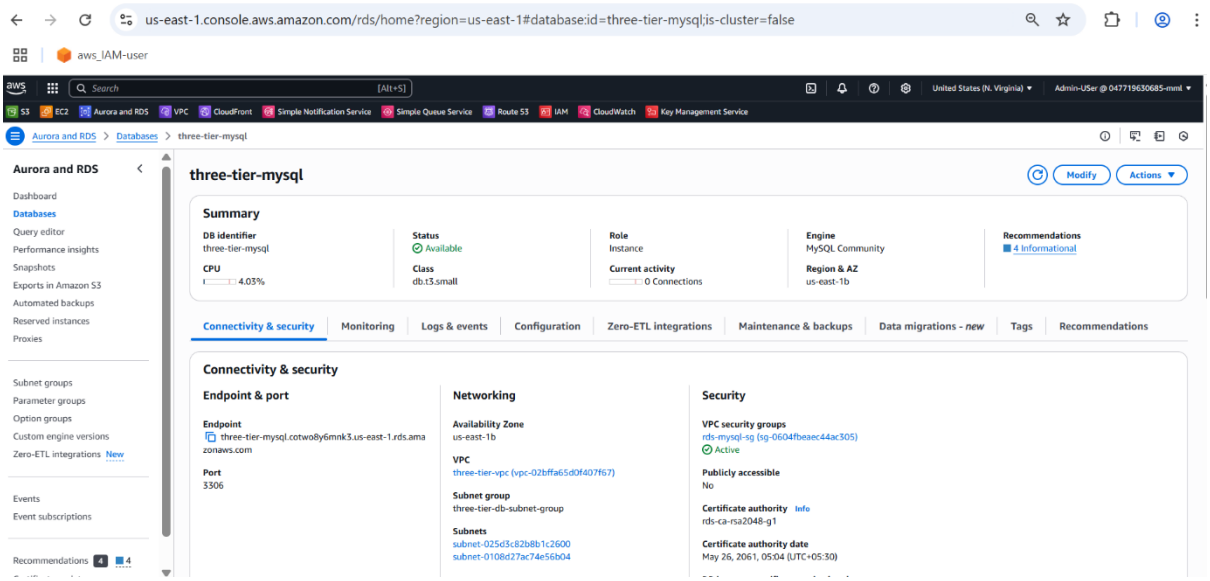


- An ECR Repositories for the frontend and backend images





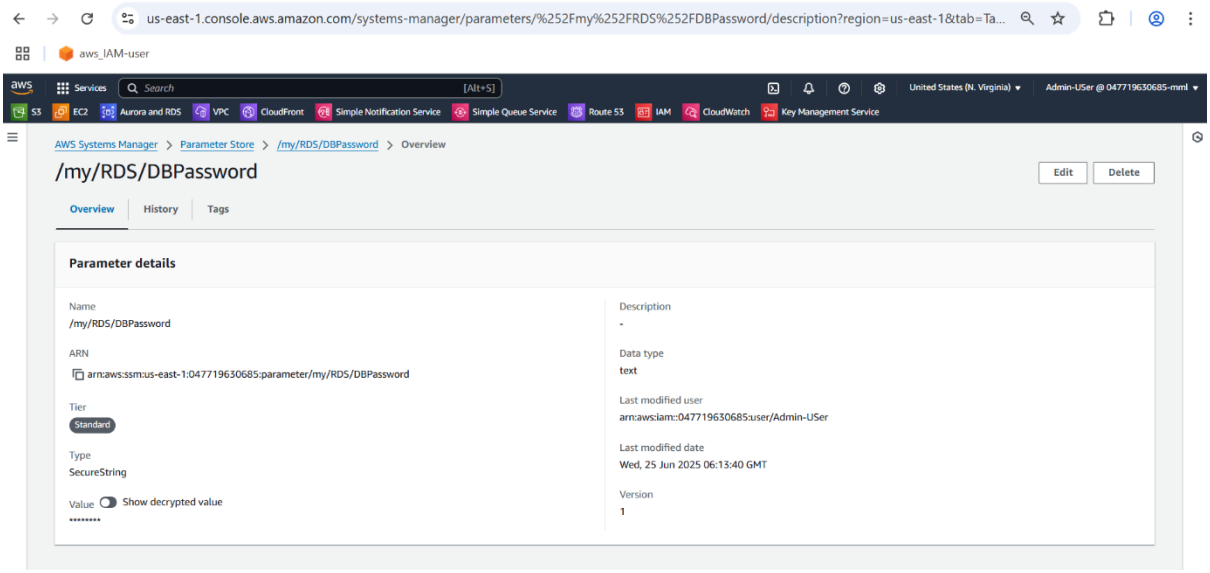
- An Amazon RDS MySQL database instance named three-tier-mysql



Step 2: Securely Store the Database Password

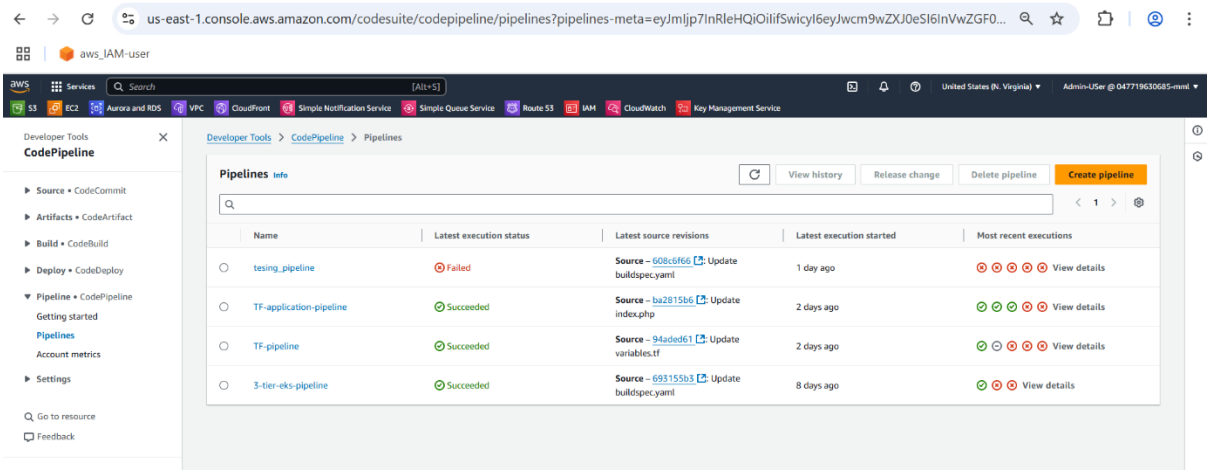
Hardcoding secrets is a security risk. Use AWS Systems Manager Parameter Store.

1. Navigate to **Systems Manager** > **Parameter Store** in us-east-1.
2. Create a parameter with the name /my/RDS/DBPassword and select the SecureString type.
3. Store your RDS master password as the value. Your application will fetch this parameter at runtime.



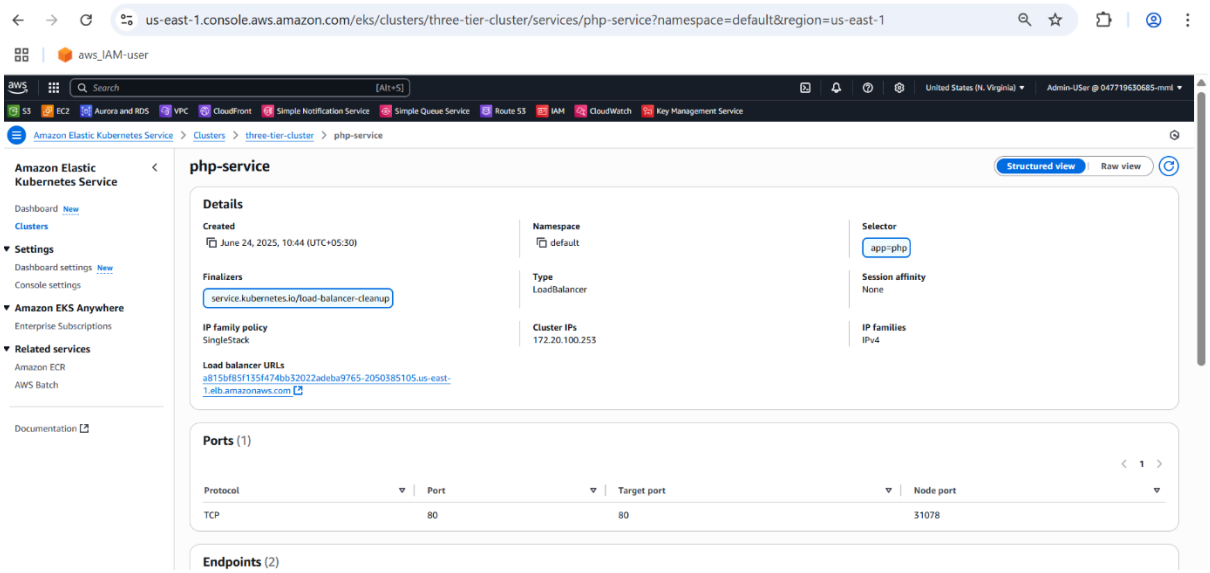
Step 3: Create the Main Application Pipeline

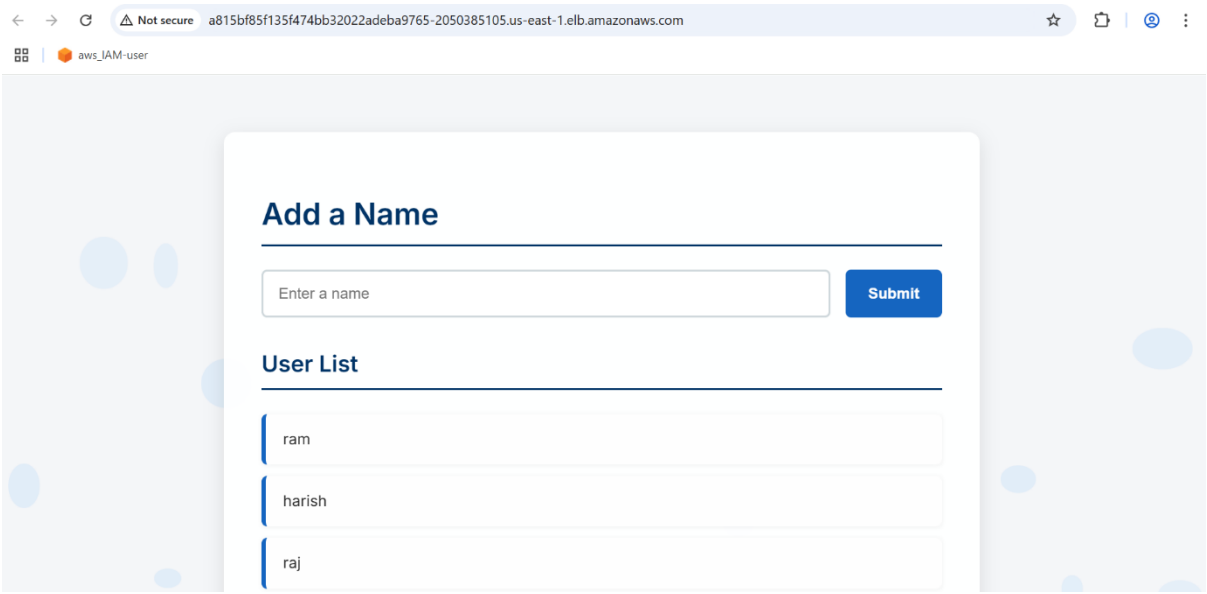
1. In **CodePipeline**, create a new pipeline (e.g., three-tier-eks-app-pipeline).
2. **Source Stage:** Connect to your php-flask-application-CF GitHub repository.
3. **Build Stage:**
    - Create a new **CodeBuild** project. This is the core of your CI/CD process.
    - The buildspec.yml file in your application repository should perform the following actions:
        1. **Code & Image Scanning:** Integrate static code analysis with **SonarQube** and vulnerability scanning with **Trivy**. The build should fail if critical vulnerabilities are found.
        2. **Build Docker Images:** Build the frontend and backend Docker images.
        3. **Push to ECR:** Tag the images and push them to the ECR repositories created in the previous step.
        4. **Deploy to EKS:** Use kubectl commands to apply your Kubernetes manifests (deployment.yaml, service.yaml) to the three-tier-cluster. This will trigger a rolling update of your application pods.
4. **Deploy Stage (Optional - Direct EKS Deployment):**
    - Alternatively, CodePipeline offers a direct "Deploy to EKS" action which can be used after the build stage.



Step 4: Verify Deployment in EKS

1. Navigate to your **EKS cluster** > **Resources** > **Services**.
2. You should see the services defined in your manifests, such as php-service (which creates a public-facing Load Balancer) and app-service for backend communication.
3. Access the Load Balancer's DNS to see your live application.



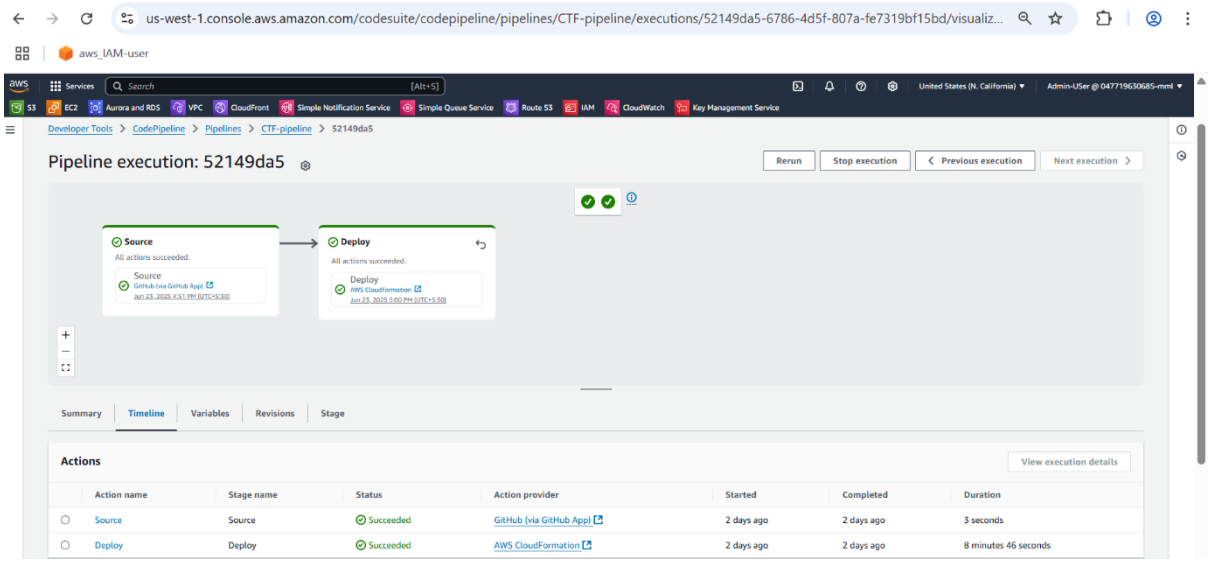


Part 3: Provisioning the DR Region (us-west-1) with CloudFormation

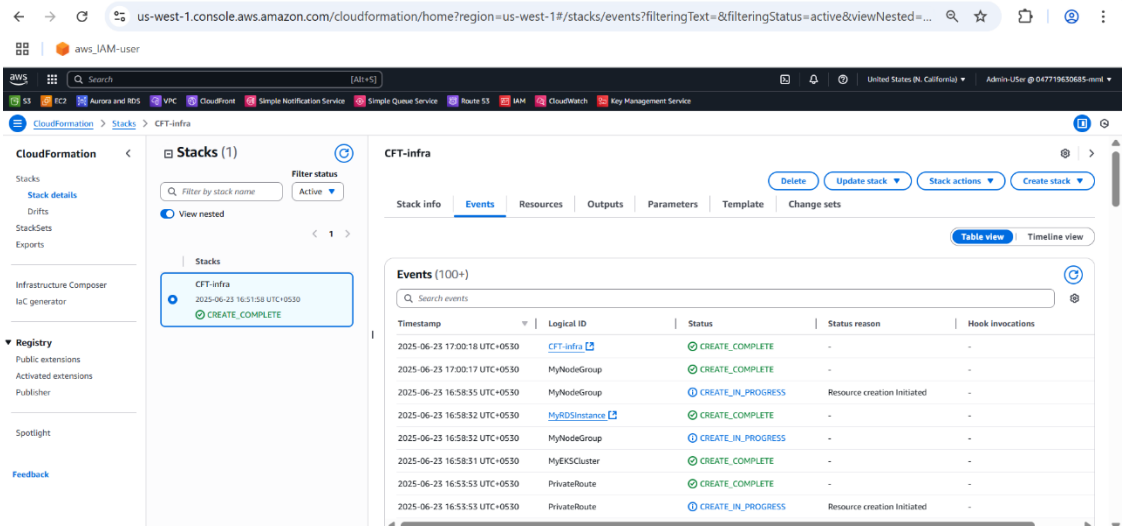
For disaster recovery, we will provision a mirrored infrastructure stack in us-west-1 using CloudFormation.

Step 1: Create the CloudFormation Infrastructure Pipeline

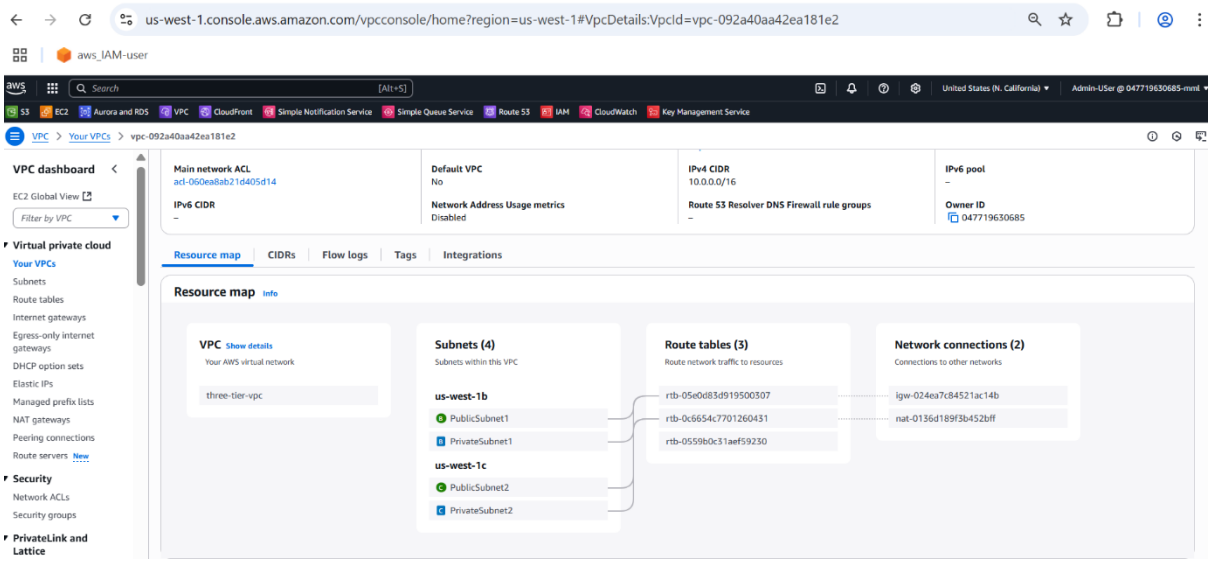
1. In the us-west-1 region, create a new **CodePipeline** named CTF-pipeline.



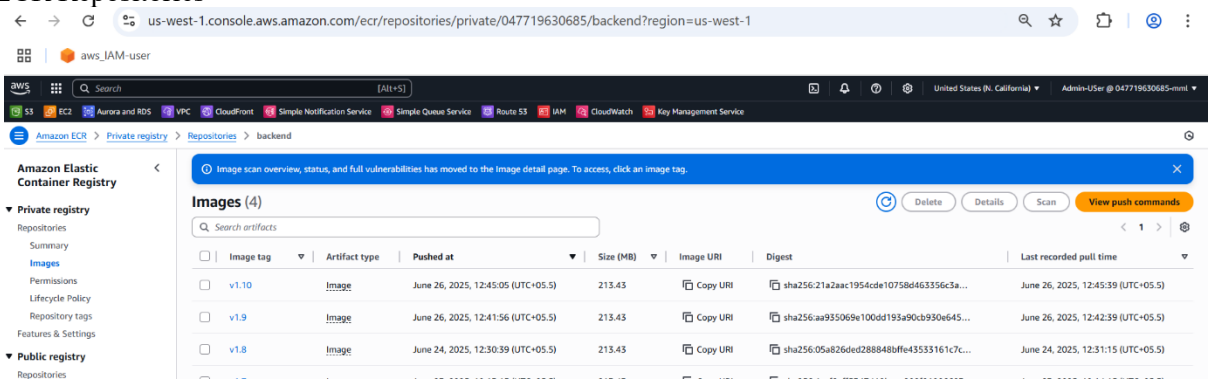
2. **Source Stage:** Connect it to your CloudFormation-Template GitHub repository.
3. **Deploy Stage:**
   - Choose **AWS CloudFormation** as the action provider.
   - Select the "Create or update a stack" action mode.
   - Provide a stack name, like CFT-infra.
   - Specify your template file (CFT.yml) and parameter file (parameters.json).

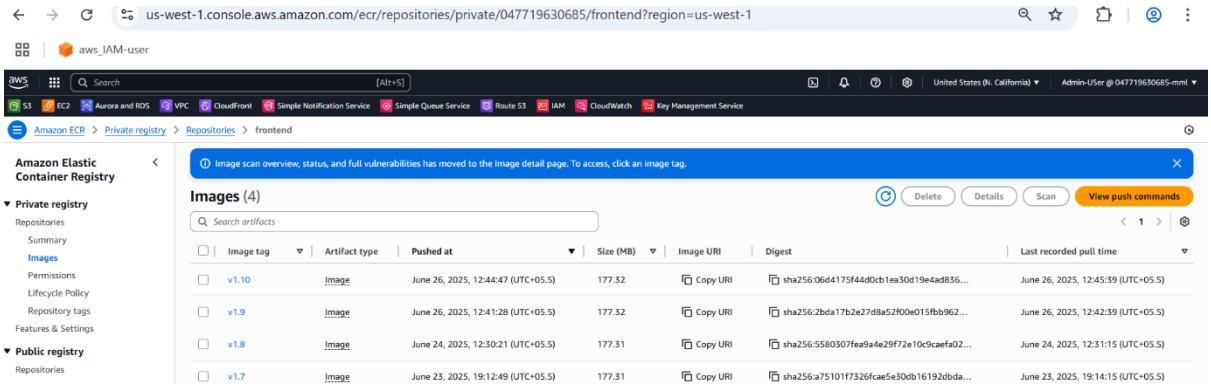


4. **Run the pipeline**: This will create a parallel set of resources (VPC, EKS, RDS) in the us-west-1 region.

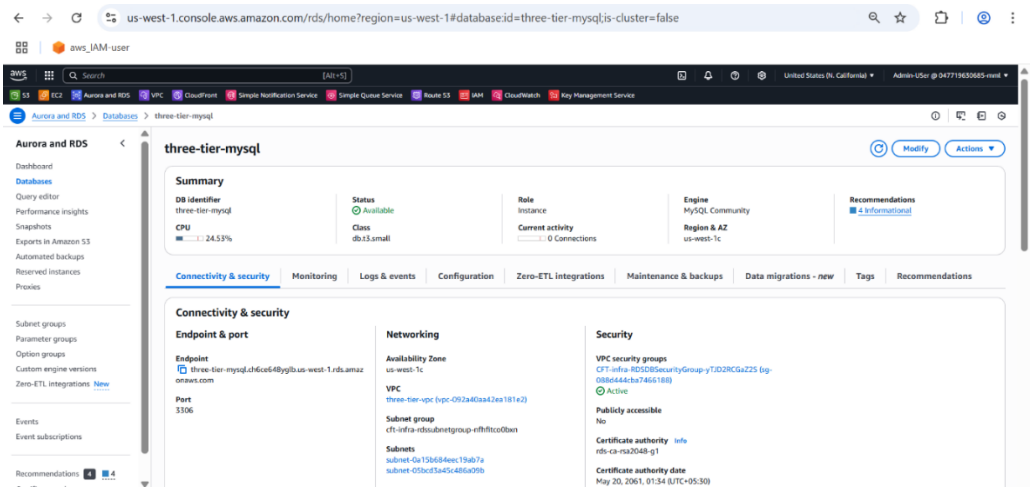


5. ECR Repositories

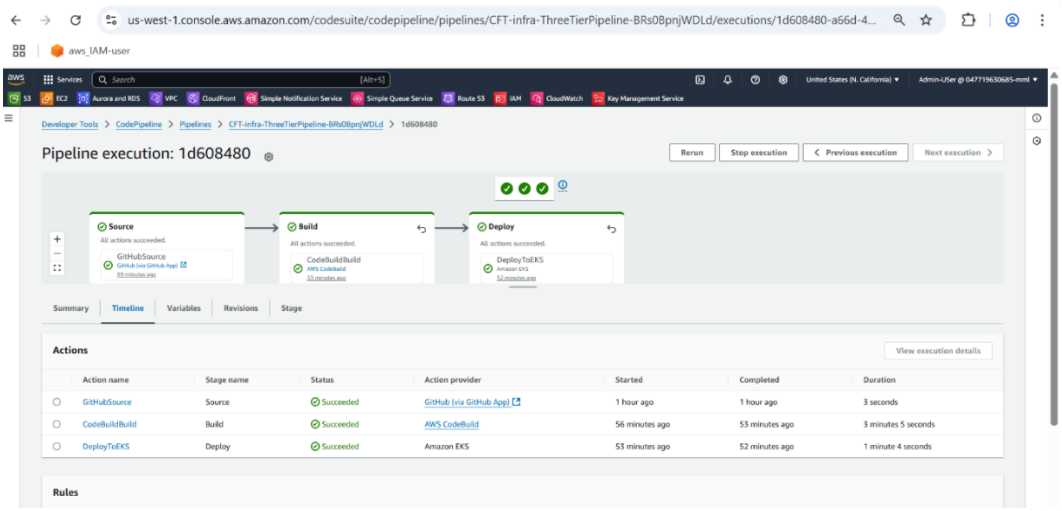




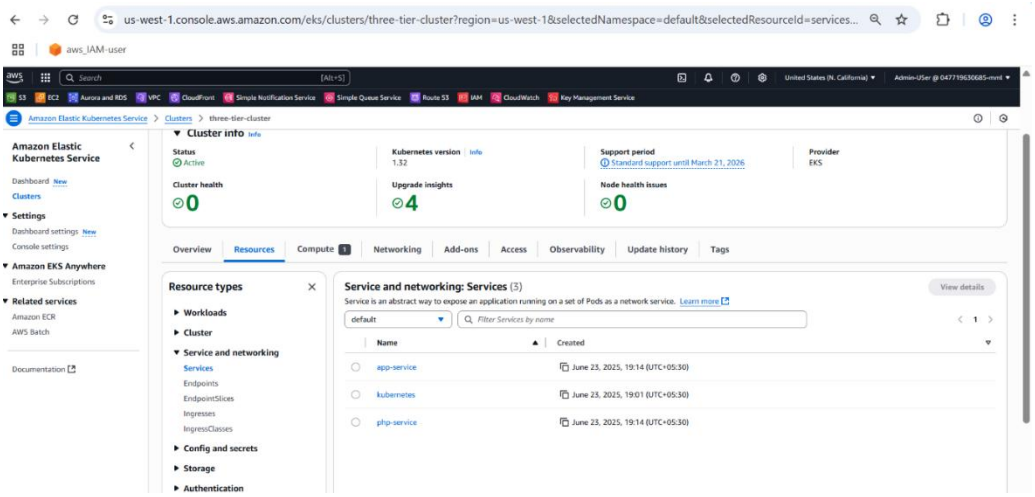
- An Amazon RDS MySQL database instance named three-tier-mysql



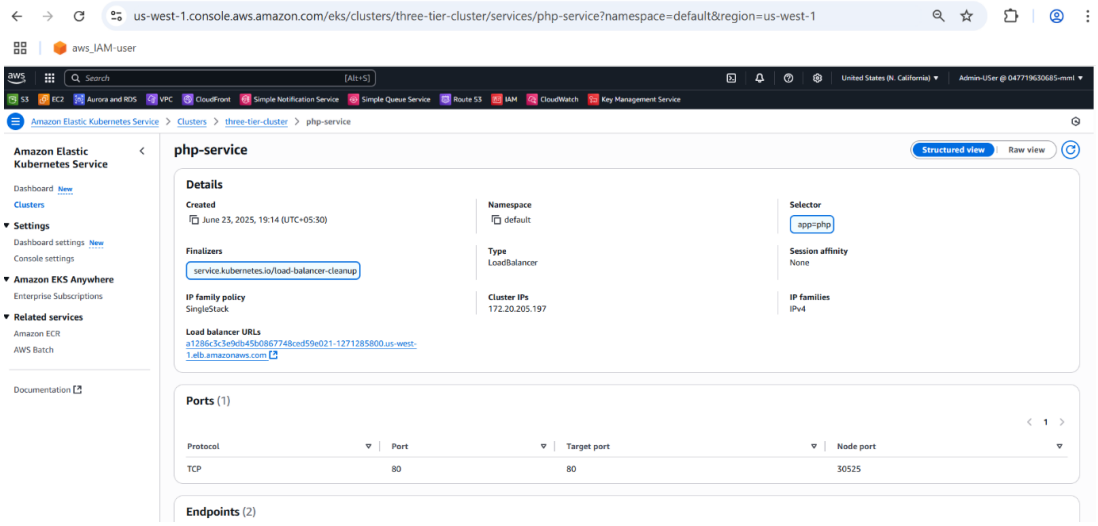
- Application pipeline



- EKS Cluster



- Load balancer service of PHP(frontend).



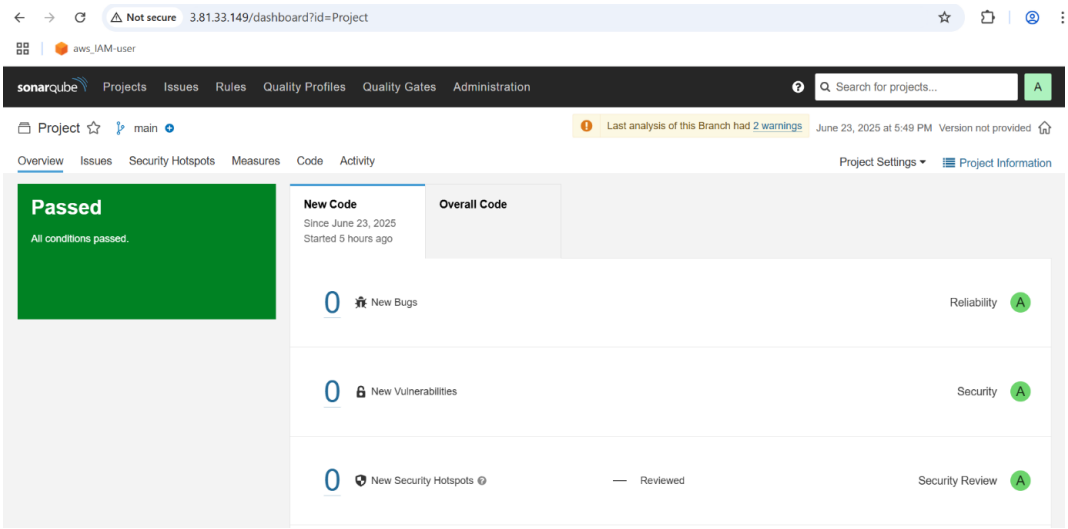
Integrating Security Scanning (SonarQube & Trivy)

The project utilizes both SonarQube and Trivy for security scanning as part of its automated CI/CD pipeline. These tools are integrated into the AWS CodeBuild stage to ensure that code quality and container security are checked automatically before any deployment occurs.

SonarQube: Static Code Analysis

SonarQube is used as a static code analysis tool to inspect the PHP and Flask source code for quality issues, bugs, and security vulnerabilities before the application is even built.

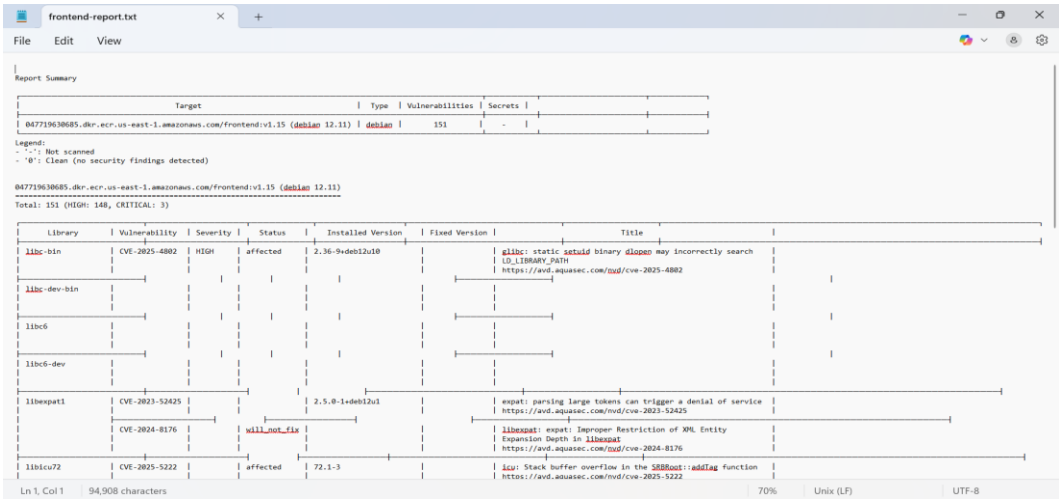
- **Purpose:** It helps identify potential issues like code smells, security hotspots, and maintainability problems directly in the source code.
- **Integration:** It is run as a step during the CodeBuild process within the CI/CD pipeline.
- **Project Results:** The provided SonarQube dashboard screenshot for the "main" project shows that the code **Passed** the Quality Gate check, indicating it meets the configured quality standards. The scan reported **0 New Bugs** and **0 New Vulnerabilities**.



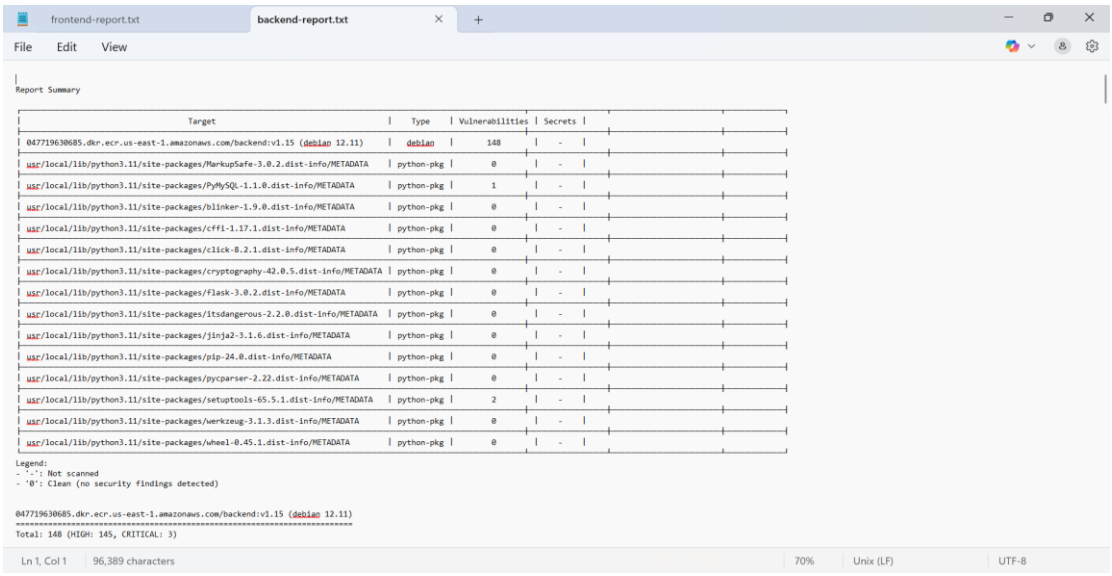
Trivy: Container Vulnerability Scanning

Trivy is used to scan the Docker images after they are built to detect known vulnerabilities within their layers and packages.

- **Purpose:** It ensures that the container images being pushed to the Amazon Elastic Container Registry (ECR) are free from known security threats.
- **Integration:** This scan is performed within the CodeBuild step after the docker build command and before the docker push command.
- **Project Results:** The scan reports provided in the screenshots show the following findings for the version v1.15 images:
    - **Frontend Image (frontend:v1.15):** A total of 151 vulnerabilities were found, with **3 rated as CRITICAL**.



    - **Backend Image (backend:v1.15):** A total of 148 vulnerabilities were found, with **3 rated as CRITICAL**.

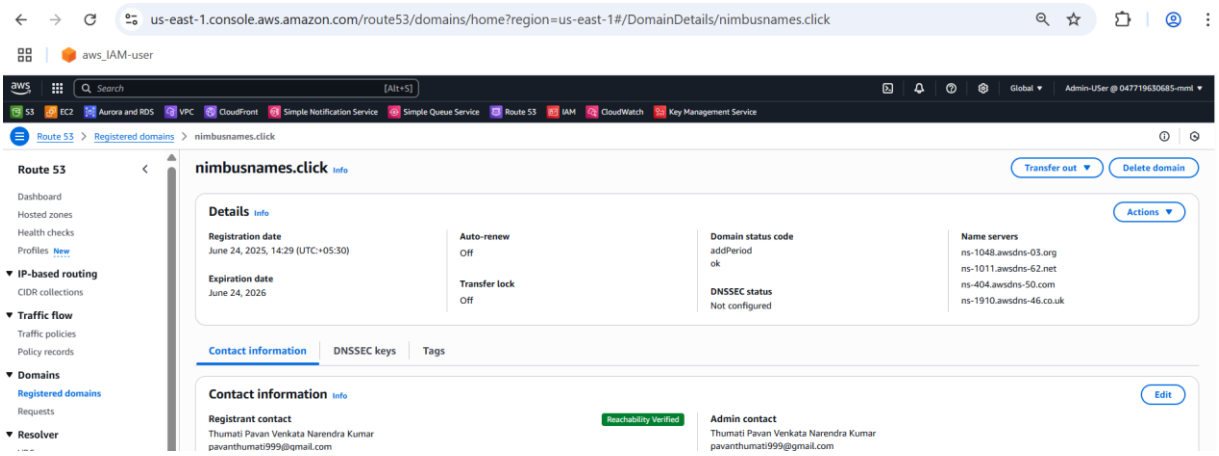


These reports provide a detailed list of vulnerabilities, allowing developers to address critical security issues by updating base images or libraries.

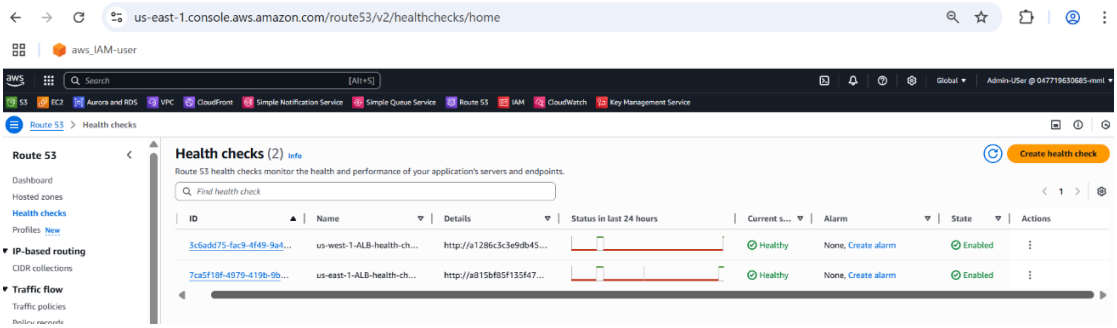
Part 5: Configuring Global Failover & Monitoring

Step 1: Configure Route 53 DNS Failover

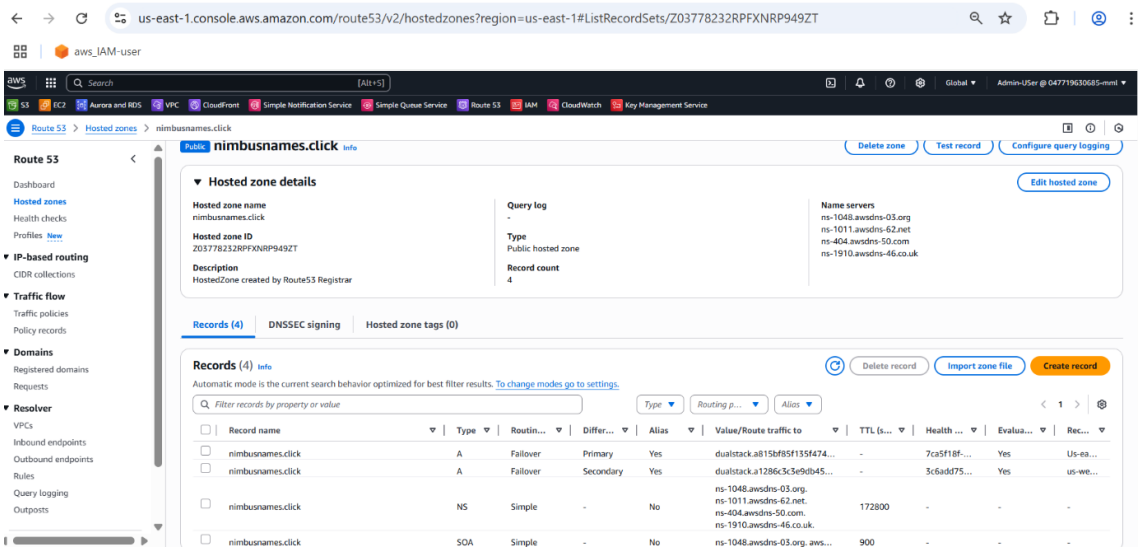
1. Navigate to **Route 53** in the AWS Console (it is a global service).

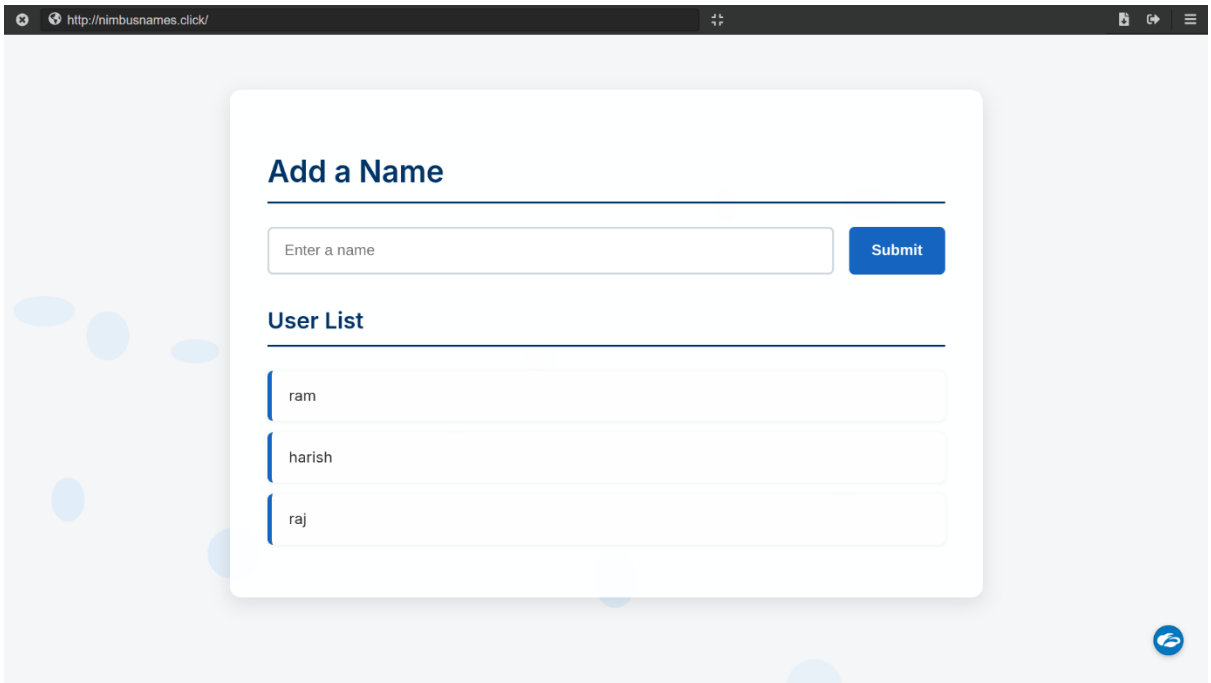


2. **Health Checks:**
   - Create two health checks, one for the public endpoint (Load Balancer URL) of the application in us-east-1 and another for the endpoint in us-west-1.



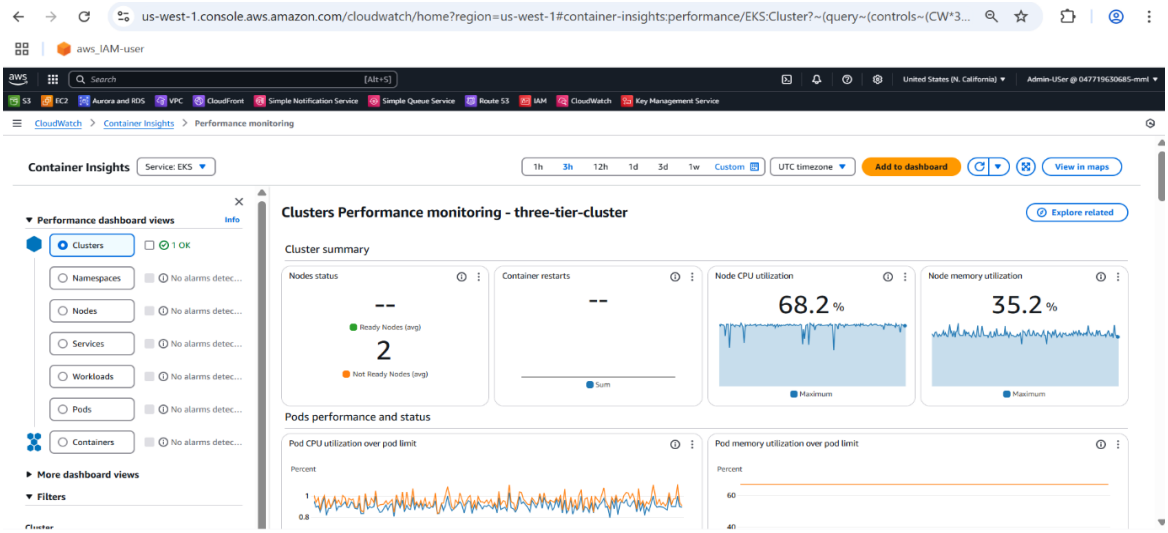
3. **Hosted Zone Records:**
    - Go to your domain's Hosted Zone.
    - Create two A records for your domain (e.g., nimbusnames.click).
    - **Primary Record (us-east-1):**
        - Set **Routing policy** to **Failover**.
        - Set **Failover record type** to **Primary**.
        - Point the value to the us-east-1 Load Balancer.
        - Associate it with the us-east-1 health check.
    - **Secondary Record (us-west-1):**
        - Set **Routing policy** to **Failover**.
        - Set **Failover record type** to **Secondary**.
        - Point the value to the us-west-1 Load Balancer.
        - Associate it with the us-west-1 health check.

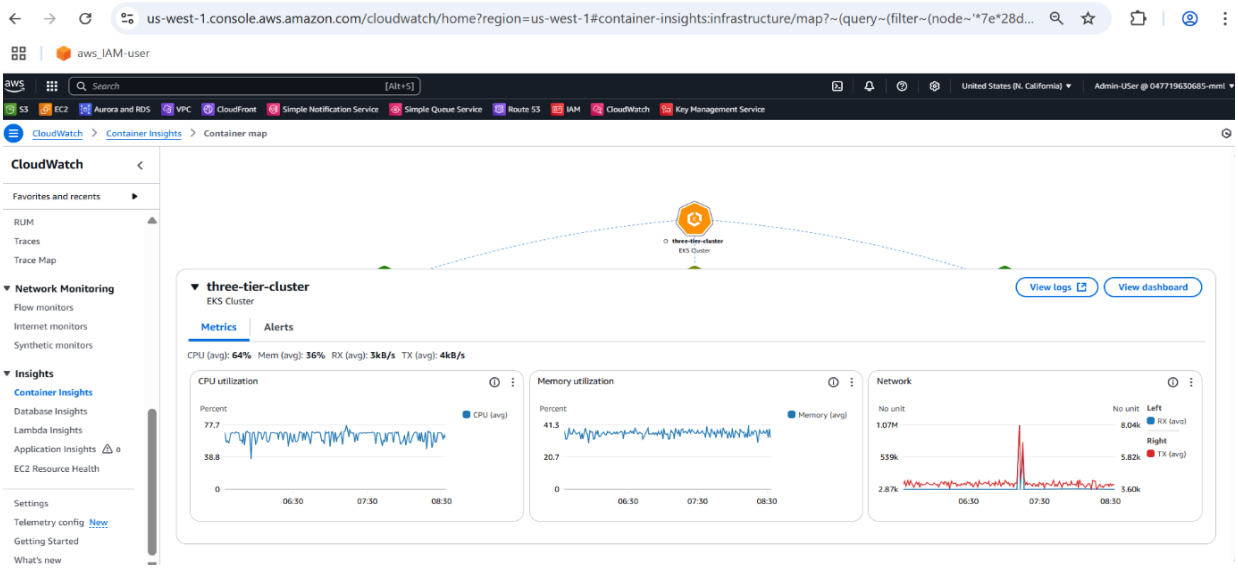


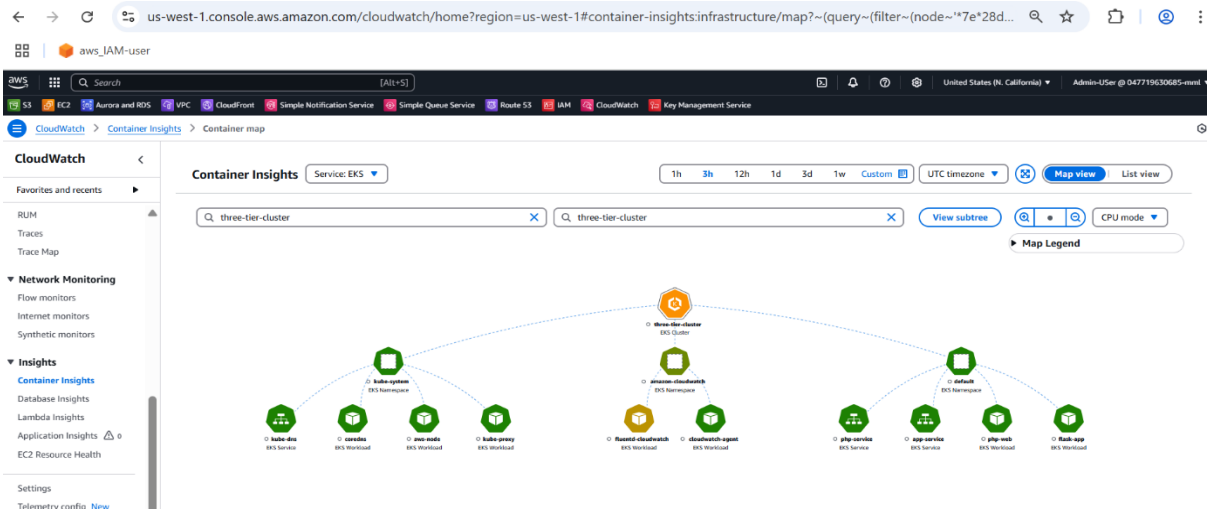


Step 2: Configure CloudWatch Monitoring and Alarms

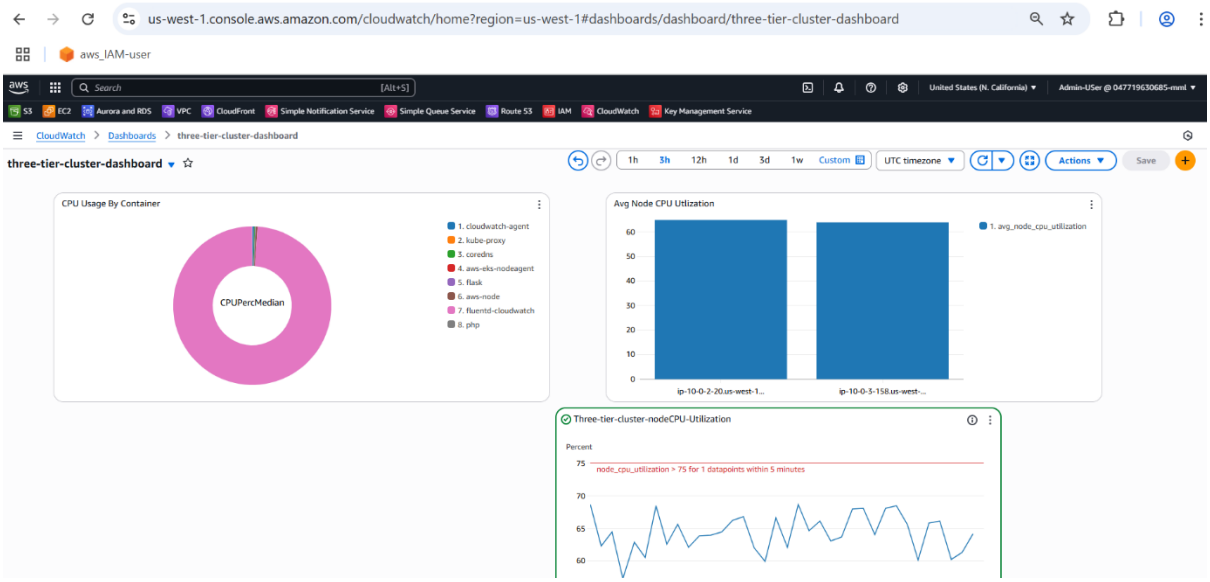
1. **Enable Container Insights:** For both EKS clusters, enable Container Insights to get detailed performance metrics on pods, nodes, and containers.



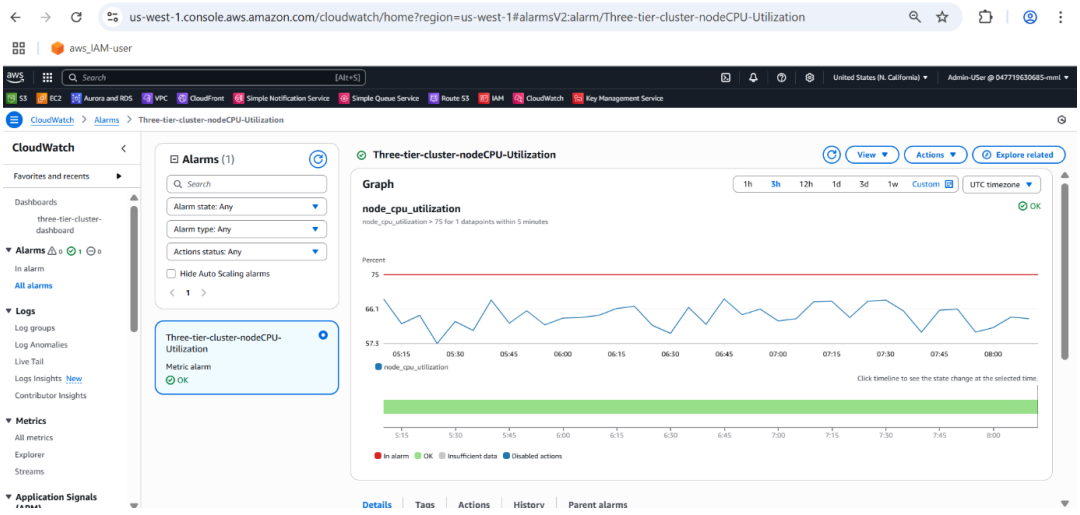


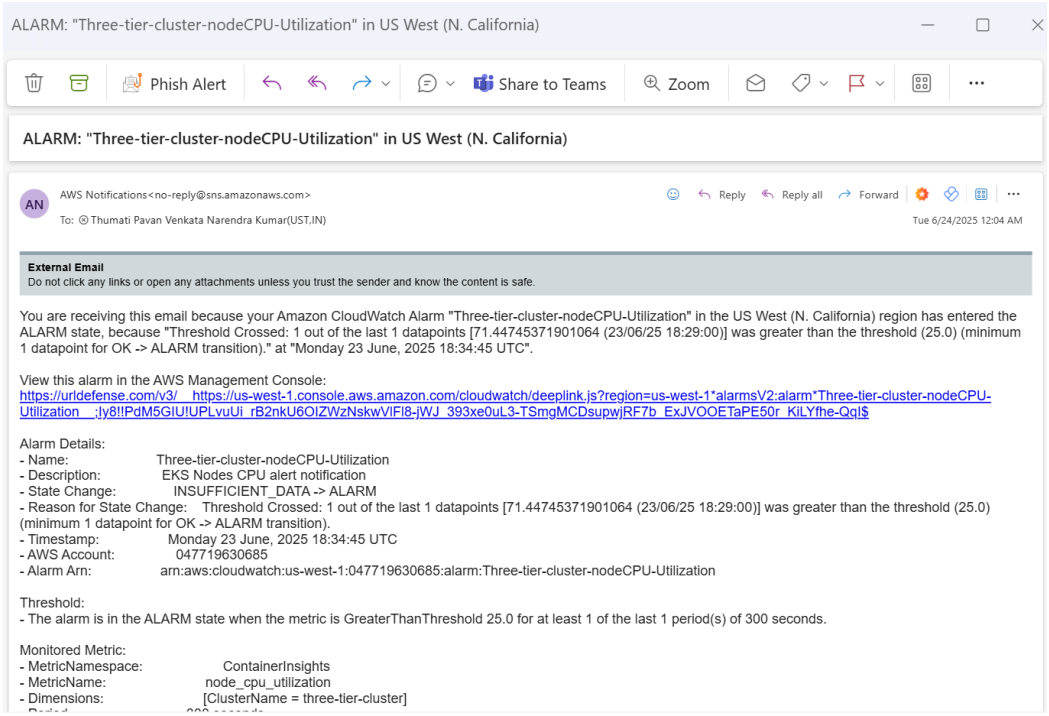


2. **Create a CloudWatch Dashboard:** Create a centralized dashboard to monitor key metrics like Node CPU Utilization, Memory Utilization, and Pod performance.

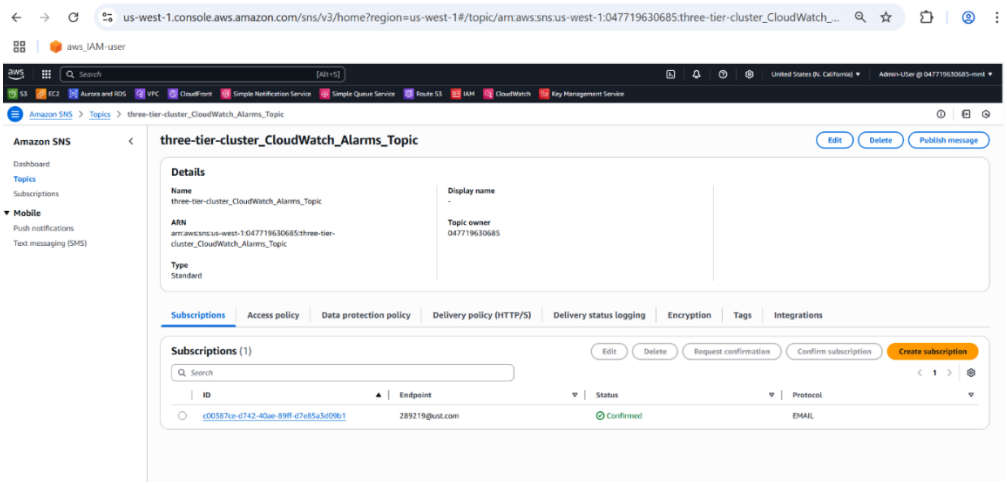


3. **Set Up Alarms:**
   - In **CloudWatch** > **Alarms** (in the us-west-1 region for this example), create an alarm.
   - Select the node_cpu_utilization metric from the ContainerInsights namespace for your cluster.
   - Set a threshold, for example, > 75% for 5 minutes.



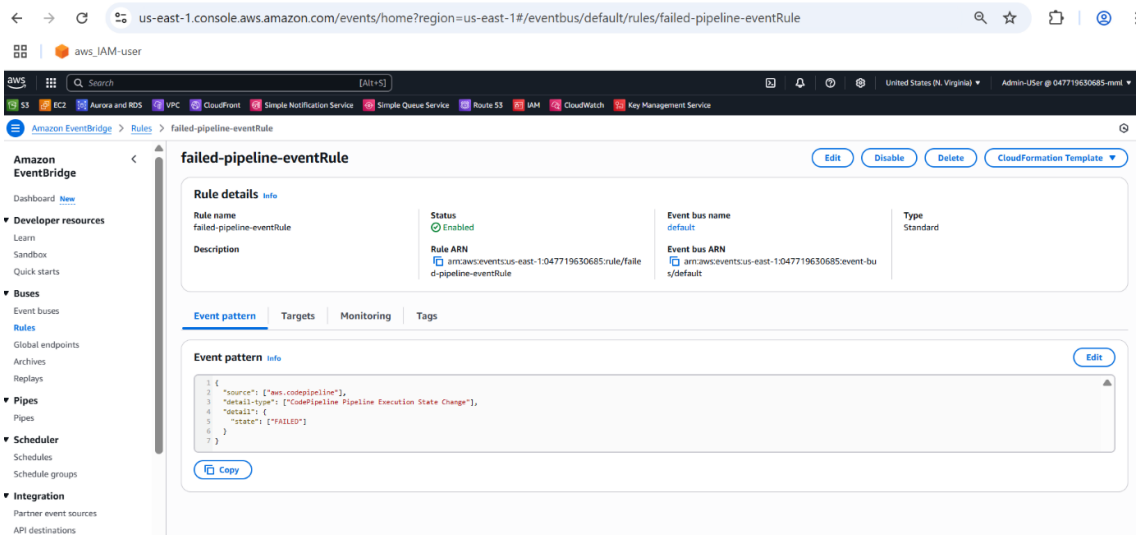


4. **Configure SNS Notifications:**
   - Create an **SNS Topic** (e.g., three-tier-cluster-CloudWatch_Alarms_Topic).
   - Create an email subscription to this topic and confirm it.
   - Configure the CloudWatch alarm's action to publish a notification to this SNS topic whenever it enters the ALARM state.



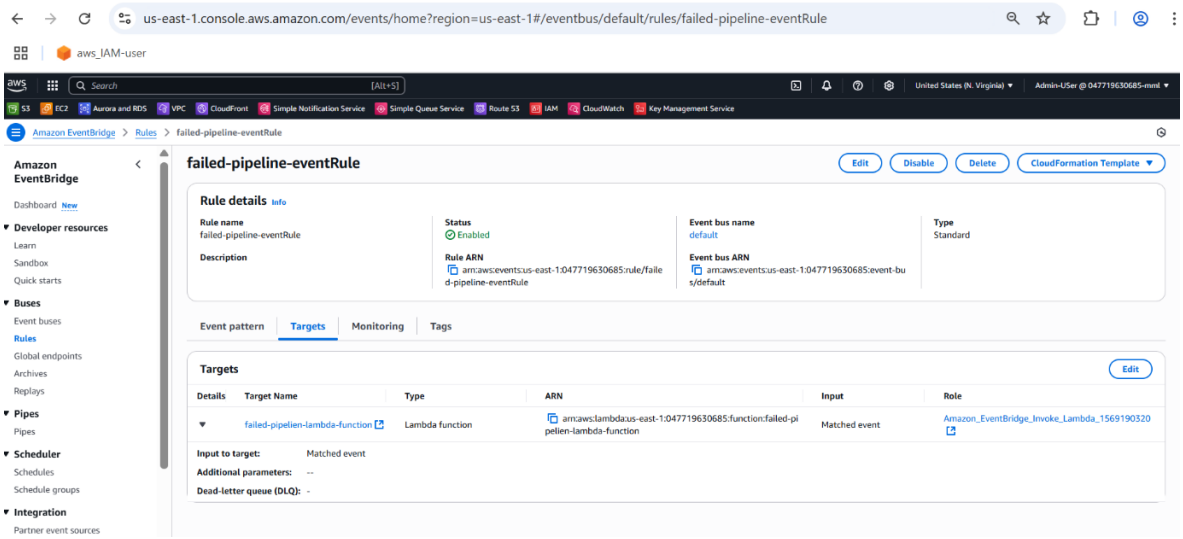
Step 3: Set Up Pipeline Failure Notifications

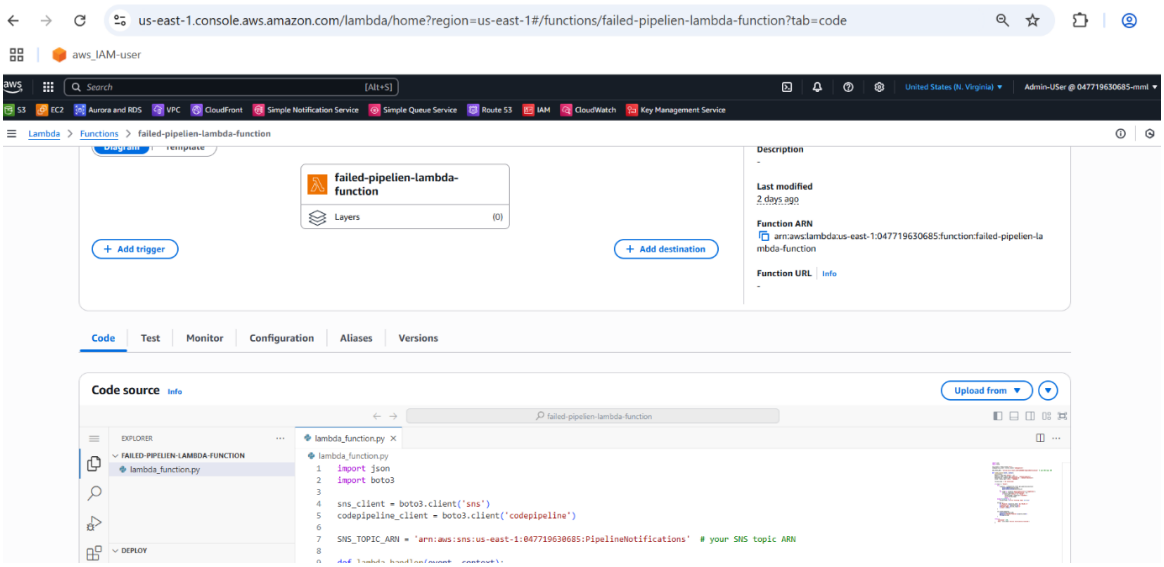
1. Navigate to **Amazon EventBridge** and create a new rule.
2. Define an event pattern to listen for



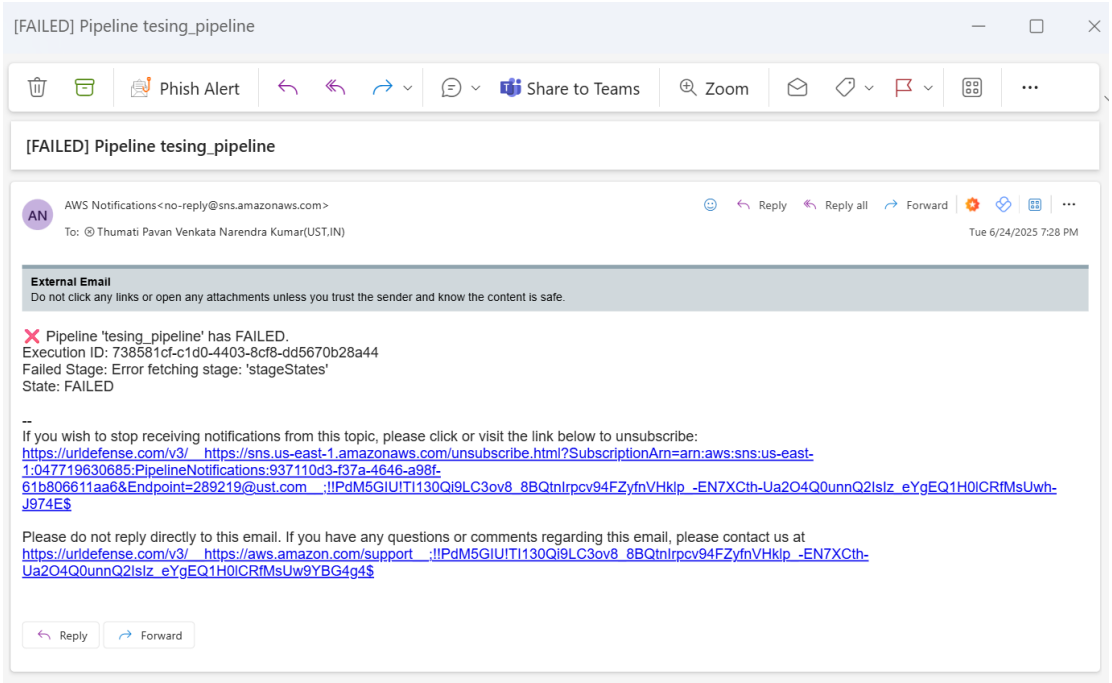
CodePipeline Pipeline Execution State Change events where the state is FAILED.

3. Set the target as a new **Lambda function**. This function will parse the event and send a formatted message to a dedicated SNS topic (e.g., PipelineNotifications).





4. Subscribe your email to the PipelineNotifications SNS topic to receive alerts when a pipeline fails.



Conclusion and Next Steps

You have successfully deployed a robust, multi-region, three-tier application on AWS. This project showcases best practices in IaC, CI/CD automation, container orchestration, and disaster recovery.

Future Enhancements

- **Implement Auto-Scaling:** Configure Horizontal Pod Autoscaler (HPA) in EKS and EC2 Auto Scaling for the node groups based on the metrics you are now monitoring.
- **Automate Database Failover:** Enhance the DR strategy by automating the promotion of the RDS read replica in the DR region upon failover.
- **Cost Optimization:** Implement cost-saving measures such as using EC2 Spot Instances for worker nodes or Reserved Instances for RDS.
- **Expand Monitoring:** Expand the detailed monitoring coverage from the DR region (us-west-1) to the primary region (us-east-1) for a complete operational view.

Final Summary

This project successfully demonstrates the design and implementation of an enterprise-grade, highly available, and resilient web application on the AWS cloud. By leveraging a multi-region architecture with automated failover, the application ensures maximum uptime and business continuity. The use of a dual Infrastructure as Code strategy, with Terraform for the primary region and CloudFormation for the disaster recovery region, showcases flexibility in tooling while maintaining consistency and automation.

Furthermore, the integration of a full DevOps lifecycle via AWS developer tools, including a CI/CD pipeline with embedded security scanning (SonarQube, Trivy), ensures that deployments are rapid, reliable, and secure. The project is rounded out by a comprehensive monitoring and alerting system, providing the necessary observability to maintain operational excellence. Ultimately, this capstone serves as a practical blueprint for building and managing modern, scalable cloud-native applications.

References

- AWS EKS Documentation
- Terraform AWS Provider
- Amazon RDS Cross-Region Replication
- AWS CI/CD Best Practices