**Capstone Project Documentation**

**ONLINE BANKING SYSTEM using Cloud-Native Disaster Recovery and Infrastructure Automation using AWS**

**Author: Srikar Padakanti**

**UID: 289227**

**Table Of Contents**

**Application Stack**

**Part 1: Project Initialization and Prerequisite Setup**

* Objective Definition
* Required Tools and Access
* Source Code Preparation

**Part 2: Primary Region Infrastructure Deployment (us-east-1)**

* Review CloudFormation Template
* Execute CloudFormation Stack Creation
* Verify EKS Cluster Connectivity

**Part 3: Secondary Region Infrastructure Deployment (us-west-2)**

* Prepare Terraform Configuration
* Execute Terraform Deployment
* Verify Secondary EKS Cluster

**Part 4: CI/CD Pipeline Configuration and Automation**

* Create ECR Repository
* Create Kubernetes Manifests
* Configure the buildspec.yml
* Build the AWS CodePipeline
* Static Code Analysis & Container Security (SonarQube + Trivy)
* Grant Permissions to CodeBuild Role

**Part 5: Route 53 Global DNS Failover**

* Create Hosted Zone
* Configure DNS Failover Records
* Health Check

**Part 6: Monitoring and Alerting (CloudWatch + SNS)**

* EKS Container Insights Dashboard
* CloudWatch Alarms
* SNS Setup

**Part 7: Security Best Practices**

* IAM and Role Isolation
* Secrets Management
* Network Security

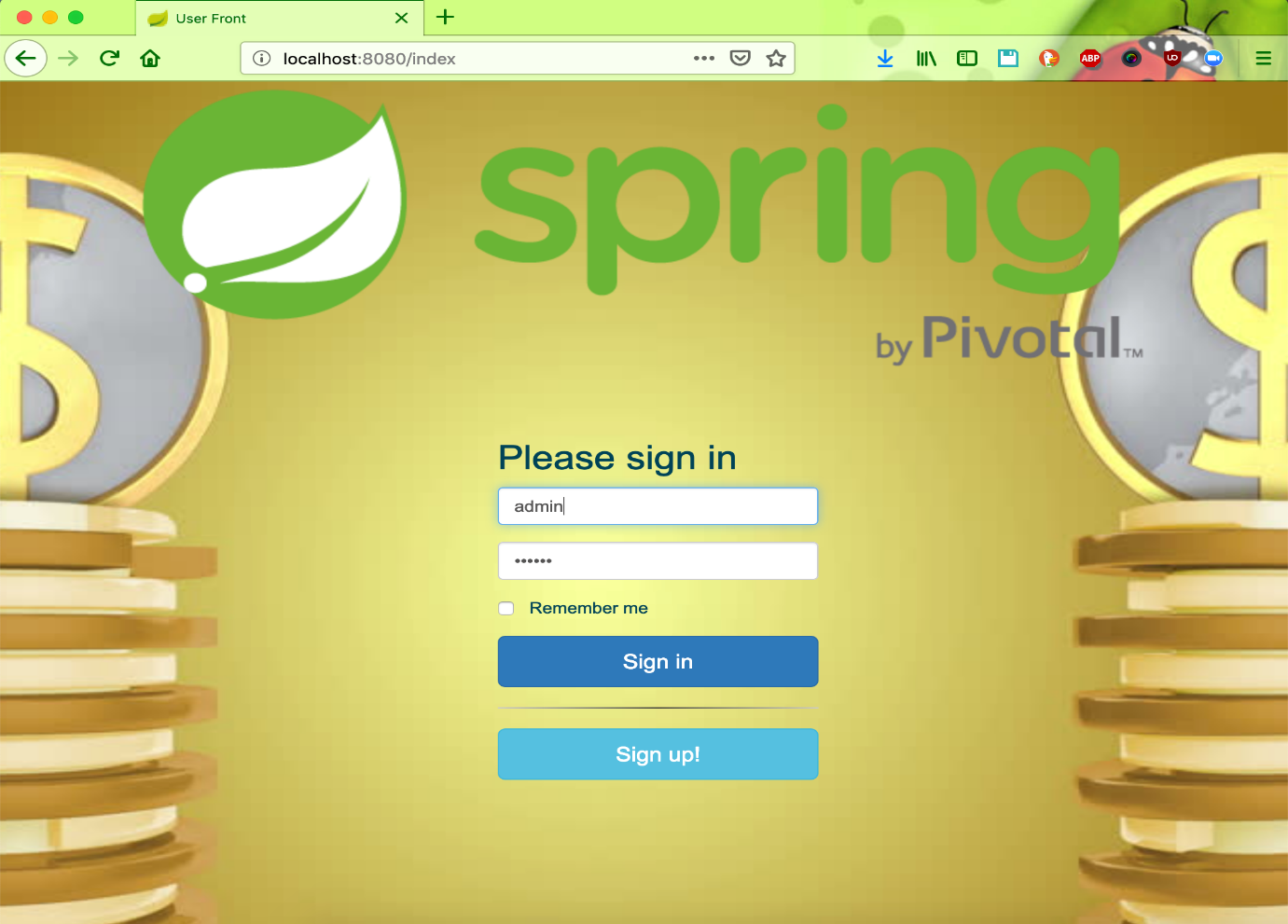
**AWS Capstone Project Documentation: Online Banking Application**

The document provides a comprehensive overview of the CI/CD pipeline, Dockerization, and Kubernetes deployment strategy for Spring Boot-based Online Banking application. It details the architecture, build processes, security scanning, deployment considerations, advanced AWS service integrations, and key project outcomes.

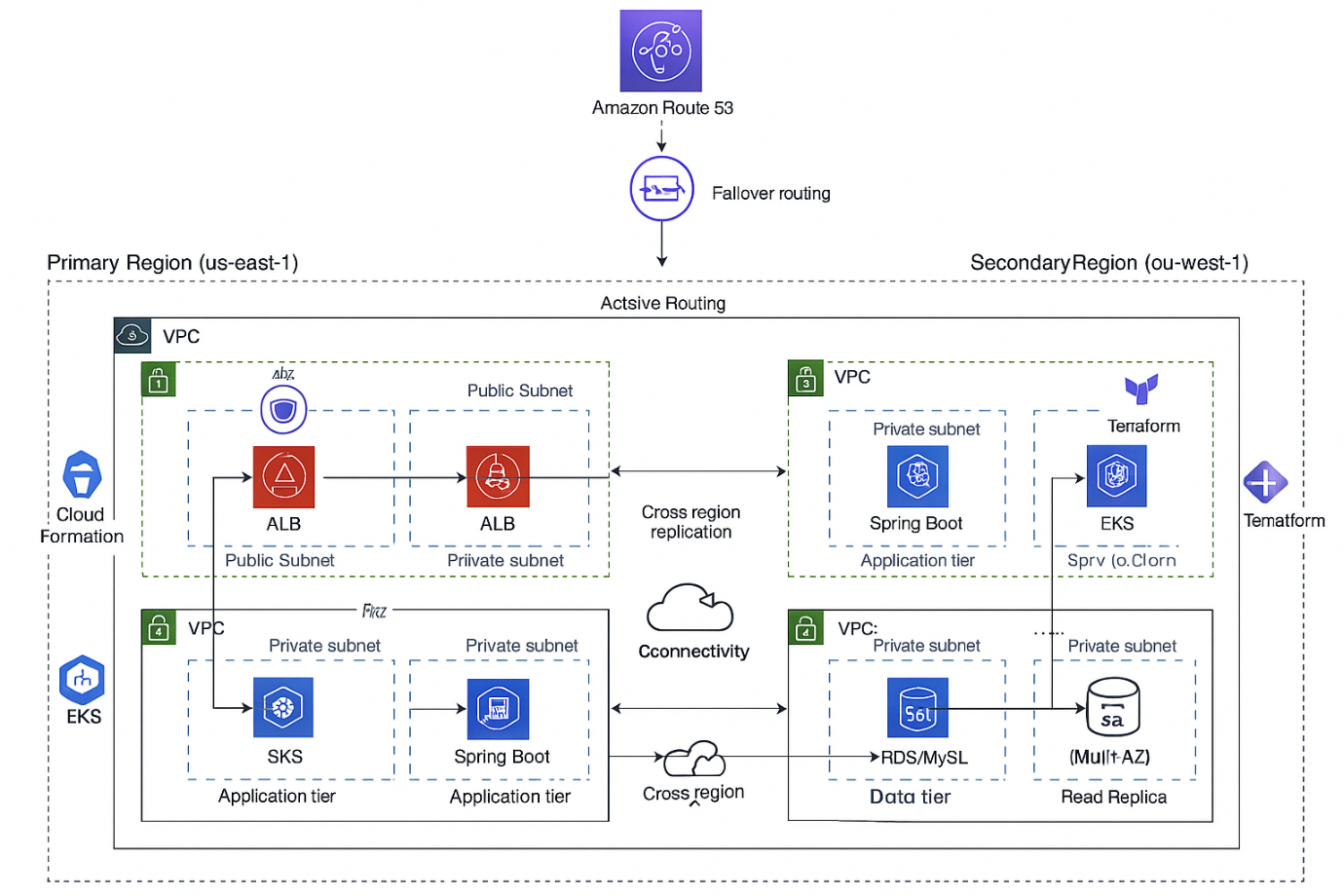
**1. PROJECT OVERVIEW**

The "Online Banking" application is a robust, web-based service developed using the **Spring Boot** framework (Java). It is designed to provide core banking functionalities, including user authentication, account management, and transaction processing, as evidenced by the login page and database schema. The application leverages **Maven** as its build automation tool and interacts with a **MySQL** database for data persistence. Key dependencies like mysql-connector-j for database connectivity and flyway-mysql for database migrations indicate a strong emphasis on controlled schema evolution and reliable database management.

The project embraces cloud-native principles by being **containerized with Docker** and slated for deployment on **Kubernetes clusters**. The entire lifecycle, from code commit to deployment-ready artifacts, is automated through a sophisticated **AWS CodeBuild CI/CD pipeline**, supporting both single-region and multi-region deployments.



**2. PROJECT ARCHITECTURE**



**2.1 Comprehensive Architecture Overview**

The solution employs a modern, scalable, and resilient cloud-native architecture, leveraging several key AWS services and open-source technologies. The infrastructure is designed for high availability, disaster recovery, and operational efficiency.

* **Application Layer**: The core of the system, a Spring Boot application packaged as a self-contained JAR, handling business logic and API endpoints.
* **Database Layer**: A **MySQL database**, provisioned as a managed service, specifically **Amazon RDS (Relational Database Service)**. RDS simplifies database administration, offering scalability, high availability (Multi-AZ), automated backups, and robust security.
* **Containerization (Docker)**: The application is packaged into lightweight, portable Docker images, ensuring consistent execution across different environments.
* **Container Registry (Amazon ECR)**: Docker images are securely stored and managed in Amazon Elastic Container Registry. ECR integrates seamlessly with AWS CodeBuild and Amazon EKS, acting as the centralized repository for all application images.
* **Container Orchestration (Kubernetes)**: The application is deployed and managed across multiple **Amazon EKS (Elastic Kubernetes Service)** clusters. Kubernetes provides robust features for service discovery, load balancing, auto-scaling, self-healing, and declarative management of containerized workloads.
* **CI/CD Pipeline (AWS CodePipeline & CodeBuild)**: Orchestrates the entire software release process, from source code changes to deployment. AWS CodeBuild acts as a critical component within these pipelines for building, scanning, and pushing container images, and preparing deployment manifests.
* **Global Traffic Management (Amazon Route 53)**: Provides intelligent routing capabilities across multiple regions, enabling multi-region ALB targets, failover policies, and custom domain DNS for high availability and disaster recovery.
* **Monitoring & Observability (Amazon CloudWatch & Grafana)**: Centralized services for collecting logs, metrics, and setting up alarms to ensure real-time insights into application and infrastructure health.
* **Event-Driven Automation (Amazon EventBridge)**: Enables real-time alerts and automated responses based on system events.
* **Data Protection (AWS Backup)**: Centralized service for automating backup and recovery of AWS resources, including RDS databases.

**INFRASTRUCTURE COMPOSER**



**Detailed Infrastructure Components**

The project leverages a comprehensive set of AWS components, often provisioned as Standard Component units:

* **ECR Repository**: For storing application Docker images.
* **MyNodeGroup**: Represents the EC2 worker nodes for the EKS cluster, managed by an EksNodeGroupRole.
* **MyEKSCluster**: The Amazon EKS Kubernetes cluster, managed by an EksClusterRole.
* **S3 Bucket (ArtifactBucket)**: Used for storing build artifacts and potentially other pipeline-related data.
* **CodeBuildProject**: The AWS CodeBuild project definition, managed by a CodeBuildServiceRole.
* **CodePipeline**: Specifically, ThreeTierPipeline and CodePipelineDeployRole, indicating multi-stage CI/CD pipelines.
* **BackupPlan**, **BackupSelection**, **BackupVault**: Components of AWS Backup for managing data protection strategies.
* **MyRDSInstance**: The Amazon RDS MySQL database instance, protected by RDSDBSecurityGroup and residing in RDSSubnetGroup.
* **VPC Components**: Includes the VPC itself, PrivateRouteTable, PublicRouteTable, GatewayAttachment, InternetGateway, NatGateway, and various PublicSubnet and PrivateSubnet definitions, indicating a robust networking setup.

**3. CI/CD Pipeline with AWS Code Build (buildspec.yml)**

The buildspec.yml file acts as the blueprint for your Continuous Integration and Continuous Delivery (CI/CD) workflow within AWS CodeBuild. It dictates a structured process for building, testing, securing, and preparing your application for deployment. The project employs **two separate end-to-end DevOps pipelines**, one orchestrated by **CloudFormation in us-east-1** and another by **Terraform in us-west-1**. This highlights a strong multi-region deployment strategy.

**3.1. Environment Variables**

The env section centralizes critical parameters, making the buildspec.yml reusable and adaptable:

* DB\_USERNAME, DB\_PASSWORD: Credentials for database access. **For production environments, these sensitive values should be managed securely using AWS Secrets Manager or similar secrets management solutions and injected at runtime.**
* CLUSTER\_NAME: Identifies the target Kubernetes cluster for deployment.
* AWS\_REGION: Specifies the AWS region where services (ECR, RDS, CodeBuild) are provisioned (e.g., us-east-1 or us-west-1 depending on the pipeline).
* AWS\_ACCOUNT\_ID: Your unique AWS account identifier, crucial for ECR image URIs.
* IMAGE\_NAME: The chosen name for your Docker application image (e.g., online-banking).

**3.2. Install Phase**

This phase is dedicated to setting up the necessary tools and dependencies within the CodeBuild environment before the actual build process begins:

* yum update -y: Ensures all pre-installed packages on the CodeBuild agent are up to date, reducing potential security vulnerabilities and compatibility issues.
* yum install -y unzip wget python3-pip gnupg tar: Installs a set of fundamental utilities required for various tasks, including downloading files (wget), decompressing archives (unzip, tar), and managing Python packages (python3-pip). gnupg is often needed for verifying downloaded software signatures.
* pip3 install --upgrade awscli: Upgrades the AWS Command Line Interface (CLI) to its latest version, ensuring compatibility with current AWS APIs and services.
* **kubectl Installation**:
  + curl -LO "https://dl.k8s.io/release/v1.29.0/bin/linux/amd64/kubectl": Downloads the kubectl binary, essential for interacting with Kubernetes clusters, allowing the pipeline to prepare Kubernetes deployment manifests.
  + chmod +x kubectl && mv kubectl /usr/local/bin/: Makes the downloaded kubectl executable and moves it to a directory within the system's PATH.
* **Trivy Installation**:
  + echo Installing Trivy...: Provides a clear log message.
  + curl -sfL https://raw.githubusercontent.com/aquasecurity/trivy/main/contrib/install.sh | sh -s -- -b /usr/local/bin: This command executes the official Trivy installation script. It is the most reliable way to install Trivy as it automatically fetches the latest stable version and handles platform-specific installation details, preventing issues like 404 Not Found errors from outdated download URLs. This ensures your security scans are always using the most current vulnerability definitions.

**3.3. pre\_build Phase**

This phase handles crucial setup and configuration before the main build and scanning operations:

* echo Logging in to Amazon ECR...: Informative message.
* aws ecr get-login-password --region $AWS\_REGION | docker login --username AWS --password-stdin ${AWS\_ACCOUNT\_ID}.dkr.ecr.${AWS\_REGION}.amazonaws.com: This command securely authenticates the Docker daemon with your specified Amazon ECR repository. It retrieves a temporary authentication token from ECR and pipes it to the docker login command, granting permission to push and pull images.
* **RDS Endpoint Retrieval**:
  + export RDS\_ENDPOINT=$(aws rds describe-db-instances --query "DBInstances[?DBName=='mydb'].Endpoint.Address" --output text --region $AWS\_REGION): Dynamically retrieves the endpoint address of your Amazon RDS database instance named mydb. This is vital for your application to connect to the database.
  + echo "RDS\_ENDPOINT=$RDS\_ENDPOINT": Prints the retrieved RDS endpoint to the build logs for verification. The presentation (Image 5) shows an RDS endpoint example: mysqldatabase.c92sws0uib11.us-west-1.rds.amazonaws.com:3306.
* export IMAGE\_URI="${AWS\_ACCOUNT\_ID}.dkr.ecr.${AWS\_REGION}.amazonaws.com/${IMAGE\_NAME}:latest": Constructs the complete URI for your Docker image in ECR. This URI will be used for tagging the image and for configuring Kubernetes deployment manifests.

**3.4. build Phase**

This is the core phase where your application is compiled, containerized, and subjected to a security scan:

* echo Building Docker image...: Log message.
* docker build -t $IMAGE\_NAME .: Builds the Docker image for your online banking application using the Dockerfile located in the current directory (.). The -t $IMAGE\_NAME flag tags the image with a local name (online-banking).
* docker tag $IMAGE\_NAME:latest $IMAGE\_URI: Creates an additional tag for the locally built image, aligning it with the full ECR URI. This is the tag used for pushing to the remote ECR repository.
* echo Scanning image with Trivy...: Log message.
* mkdir -p trivy-reports: Creates a directory to store the Trivy scan report.
* trivy image --severity HIGH,CRITICAL --exit-code 0 --format table -o trivy-reports/report.txt $IMAGE\_NAME:latest || true: Executes the Trivy vulnerability scan:
  + --severity HIGH,CRITICAL: Instructs Trivy to report only vulnerabilities classified as "High" or "Critical" severity. This focuses on the most impactful issues.
  + --exit-code 0: **Important for build continuity.** This ensures that the CodeBuild step itself will **not fail** even if Trivy discovers vulnerabilities. This allows the pipeline to continue, providing the scan report as an artifact for later review. For a stricter DevSecOps pipeline, this value could be set to 1 to enforce a security gate, failing the build if high/critical vulnerabilities are found.
  + --format table: Formats the scan results in a readable table format in the console output.
  + -o trivy-reports/report.txt: Saves the full scan report to a file within the trivy-reports directory, which will be collected as an artifact.
  + || true: This ensures that even if Trivy were configured to exit with a non-zero code (e.g., if --exit-code 1 was used and vulnerabilities were found), the shell command itself would still succeed, preventing premature build failure at this specific step. This acts as a safety net in conjunction with --exit-code 0.

**3.5. post\_build Phase**

This phase handles the final steps after the Docker image is built and scanned, primarily pushing to ECR and preparing Kubernetes deployment files:

* echo Pushing Docker image to ECR...: Log message.
* docker push $IMAGE\_URI: Uploads the newly built and tagged Docker image to your Amazon ECR repository, making it available for deployment.
* echo Preparing Kubernetes manifests...: Log message.
* mkdir -p k8s-out: Creates a directory (k8s-out) where the final, deployment-ready Kubernetes manifests will be stored.
* sed "s|IMAGE\_PLACEHOLDER|$IMAGE\_URI|g" k8s/deployment.yaml > k8s-out/deployment.yaml: This command uses sed (stream editor) to perform a find-and-replace operation. It substitutes the placeholder IMAGE\_PLACEHOLDER within your k8s/deployment.yaml file with the actual ECR image URI ($IMAGE\_URI). This dynamic update is essential so that your Kubernetes deployment pulls the correct, newly built application image. The modified content is then redirected to a new file in k8s-out.
* cp k8s/service.yaml k8s-out/service.yaml: Copies your Kubernetes Service manifest directly to the k8s-out directory.
* echo Build and Scan Complete: Final log message for the phase.

**3.6. artifacts Section**

This section explicitly tells CodeBuild which files generated during the build should be preserved and made available after the build completes. These artifacts are crucial for subsequent stages in a broader CI/CD pipeline.

* base-directory: .: Specifies that artifact collection should start from the root of the CodeBuild project's output directory.
* files: Lists the specific paths of the files to be collected:
  + k8s-out/deployment.yaml: The dynamically updated Kubernetes Deployment manifest.
  + k8s-out/service.yaml: The Kubernetes Service manifest.
  + trivy-reports/report.txt: The detailed vulnerability scan report generated by Trivy.

These artifacts can then be downloaded, reviewed, or used by tools like AWS CodeDeploy, or directly by kubectl to deploy your application to the EKS cluster.

**4. Dockerization (Dockerfile)**

The Dockerfile defines the instructions for building your application's Docker image. It employs a **multi-stage build** pattern, which is a best practice for creating smaller, more secure, and production-ready images. The application specifies Java 21 in its pom.xml, and the Dockerfile reflects this.

**4.1. Stage 1: builder (Build Environment)**

This stage is responsible for compiling your Java application and packaging it into an executable JAR. It includes all necessary build tools and dependencies, which are discarded in the final image.

* FROM maven:3.8.5-openjdk-21 AS builder: This line sets the base image for the build stage. It uses a pre-built Docker image that includes Maven 3.8.5 and OpenJDK 21, providing the complete environment needed to compile your Spring Boot application.
* WORKDIR /app: Defines /app as the working directory within this stage of the container. All subsequent commands (like COPY, RUN) will execute relative to this directory.
* COPY maven-settings.xml /usr/share/maven/conf/settings.xml: This is a critical step for robust dependency management. It copies a custom settings.xml file (which you must provide alongside your Dockerfile) into Maven's configuration directory. This settings.xml is configured to use a public Maven mirror (e.g., Aliyun Maven), effectively bypassing potential 429 Too Many Requests errors from Maven Central during dependency downloads, ensuring reliable builds.
* COPY pom.xml .: Copies your project's pom.xml file into the current working directory (/app) of the container. This file is essential for Maven to understand your project structure and dependencies.
* COPY src ./src: Copies your application's source code (the src directory) into the container.
* RUN mvn clean package -DskipTests: This command executes the Maven build process:
  + clean: Cleans the target directory, removing any previous build artifacts.
  + package: Compiles the source code, runs tests (unless skipped), and packages the compiled code into a distributable format, which for a Spring Boot application is typically an executable JAR file (e.g., target/online-banking-0.0.1-SNAPSHOT.jar).
  + -DskipTests: Skips running unit and integration tests during the build. This speeds up the Docker image creation, especially in CI/CD environments where tests might be run in a separate, dedicated phase.

**4.2. Stage 2: Final Image (Runtime Environment)**

This stage creates a lean, production-ready image containing only your application's executable and its minimal runtime dependencies.

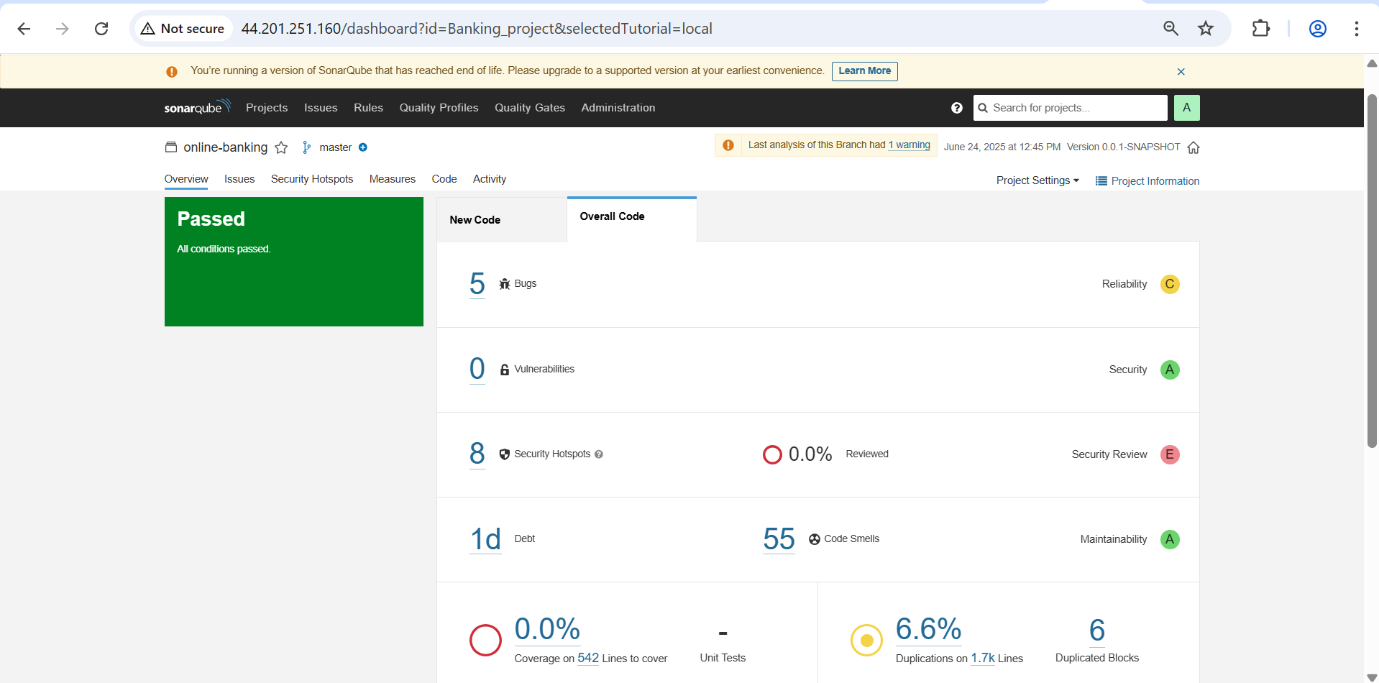
* FROM public.ecr.aws/docker/library/openjdk:21-slim: This sets the base image for the final, runtime stage. public.ecr.aws/docker/library/openjdk:21-slim provides a minimal OpenJDK 21 runtime environment. Using AWS's public ECR for official Docker images is beneficial as it is highly available and avoids potential rate limits associated with Docker Hub. A "slim" image reduces the attack surface and overall image size.
* WORKDIR /app: Sets the working directory for the running application inside the final container.
* COPY --from=builder /app/target/online-banking-0.0.1-SNAPSHOT.jar app.jar: This is the essence of multi-stage builds. It copies only the compiled application JAR (online-banking-0.0.1-SNAPSHOT.jar) from the builder stage's /app/target directory to the final image's /app directory, renaming it to app.jar for convenience. This excludes all the build tools and source code, resulting in a significantly smaller and more secure final image.
* EXPOSE 8080: Declares that the container listens on TCP port 8080 at runtime. This is standard for Spring Boot applications and helps with network configuration and firewall rules.
* ENTRYPOINT ["java", "-jar", "app.jar"]: Specifies the command that will be executed when a container based on this image starts. It runs your Spring Boot application's executable JAR.

**5. Vulnerability Scanning with SonarQube & Trivy**

The project implements a layered approach to code quality and security scanning, incorporating both static code analysis (SonarQube) and container image vulnerability scanning (Trivy).

**5.1. SonarQube (Code Quality & Static Analysis)**

* **Purpose**: SonarQube performs static code analysis to detect bugs, vulnerabilities, and "code smells" (maintainability issues) directly in the source code. It ensures that the codebase adheres to predefined quality standards and security rules.
* **Integration**: Integrated early in the CI/CD pipeline, ideally before or during the build phase (though not explicitly shown in buildspec.yml, the presentation confirms its use).
* **Benefits**: Improves code maintainability, reduces technical debt, ensures adherence to coding standards, and proactively identifies potential issues. The provided screenshot (Image 9) shows "New Code" passing with 0 vulnerabilities, indicating good code quality.



**5.2. Trivy (Container Image Vulnerability Scanning)**

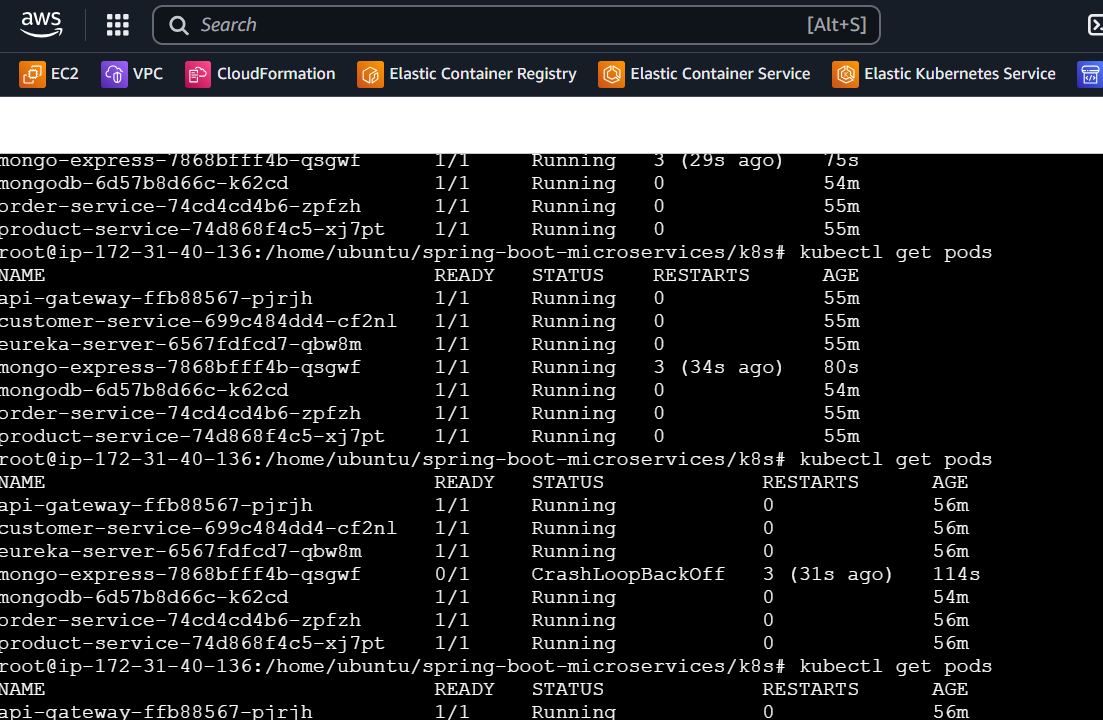
* **Purpose**: Trivy is a comprehensive and easy-to-use open-source vulnerability scanner specifically for container images, file systems, and Git repositories. It identifies known security vulnerabilities in the image's operating system packages and application dependencies.
* **Integration**: As detailed in the build phase of buildspec.yml, Trivy scans the Docker image immediately after it's built and tagged, before it's pushed to ECR.
* **Benefits**:
  + **Early Detection**: Identifies vulnerabilities early in the development lifecycle, making them cheaper and easier to fix.
  + **Automated Security Check**: Automates the security scanning process, ensuring consistent checks without manual intervention.
  + **Comprehensive Coverage**: Scans various layers of your image, including the base OS and programming language-specific dependencies.
  + **Artifact Generation**: The scan report (trivy-reports/report.txt) provides actionable insights into discovered vulnerabilities, including their severity, package affected, and potential remedies.
* **Integration in Pipeline**: By setting --exit-code 0 in the buildspec.yml, the pipeline collects the scan report but doesn't halt the build if vulnerabilities are found. While this ensures continuous delivery, for production-grade security, it's highly recommended to configure --exit-code 1 (and remove || true) to enforce a "fail-fast" policy, stopping the build if critical or high-severity vulnerabilities are detected.

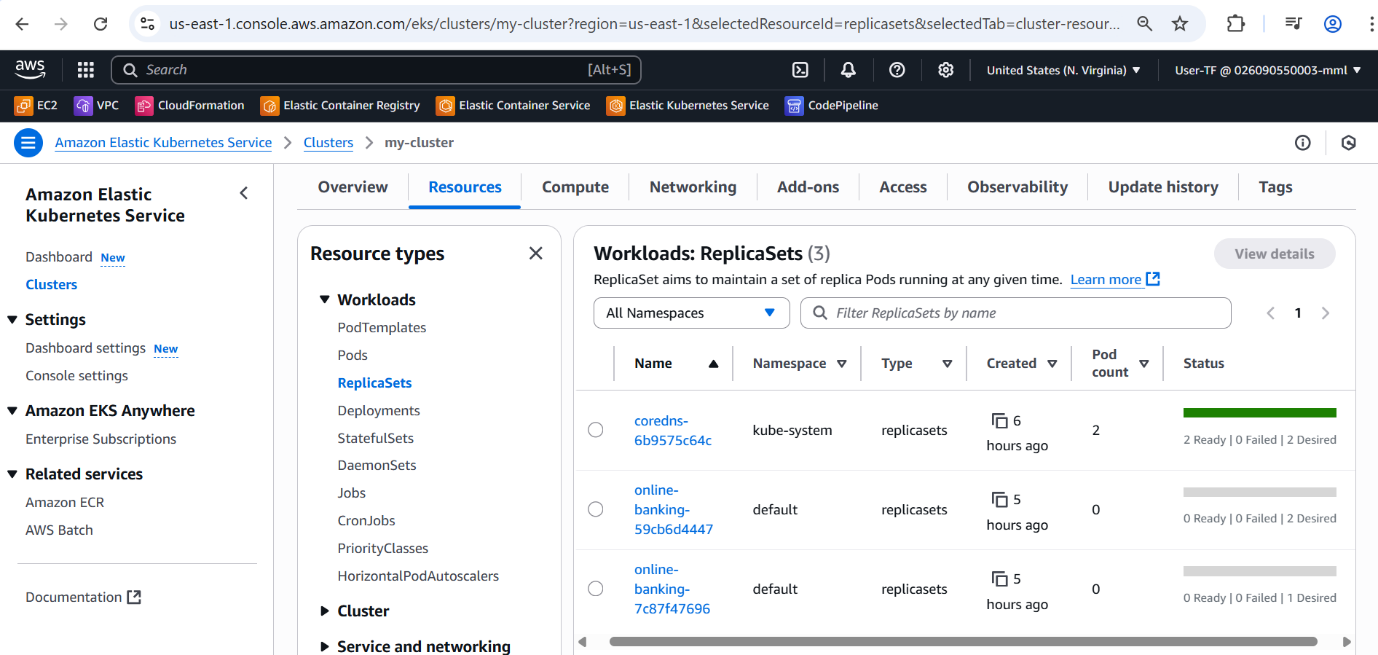


**6. Kubernetes Deployment (k8s/deployment.yaml, k8s/service.yaml)**

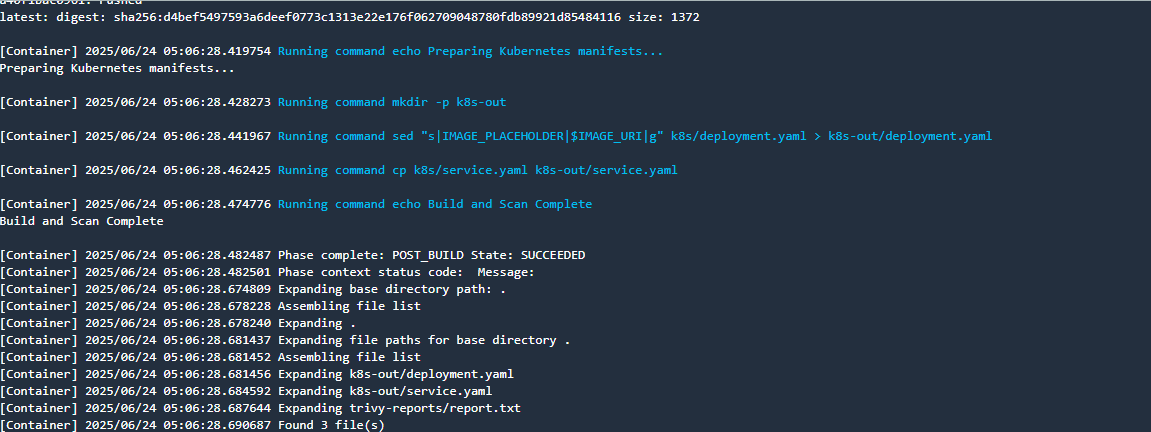
The Kubernetes manifests are the declarative definitions of your application's desired state within the Kubernetes cluster.

* **k8s/deployment.yaml**: This manifest defines a Kubernetes **Deployment** resource. A Deployment ensures that a specified number of replicas of your application's pod (a running instance of your Docker container) are always running.
  + It specifies the Docker image to use (which is dynamically updated by sed with the ECR URI during the CodeBuild post\_build phase).
  + It defines important parameters like replica count, container ports, environment variables (which can include the RDS\_ENDPOINT), resource requests and limits (CPU/Memory), and health probes.
* **k8s/service.yaml**: This manifest defines a Kubernetes **Service** resource. A Service provides a stable network endpoint for your application pods. It abstracts away the dynamic IP addresses of individual pods, allowing other services or external users to reliably access your application.
  + It defines how traffic should be routed to your application's pods, including port mappings and selector labels. Common types include ClusterIP (internal only), NodePort (exposes on each node's IP), and LoadBalancer (integrates with cloud provider's load balancer, e.g., AWS ELB/ALB).



****

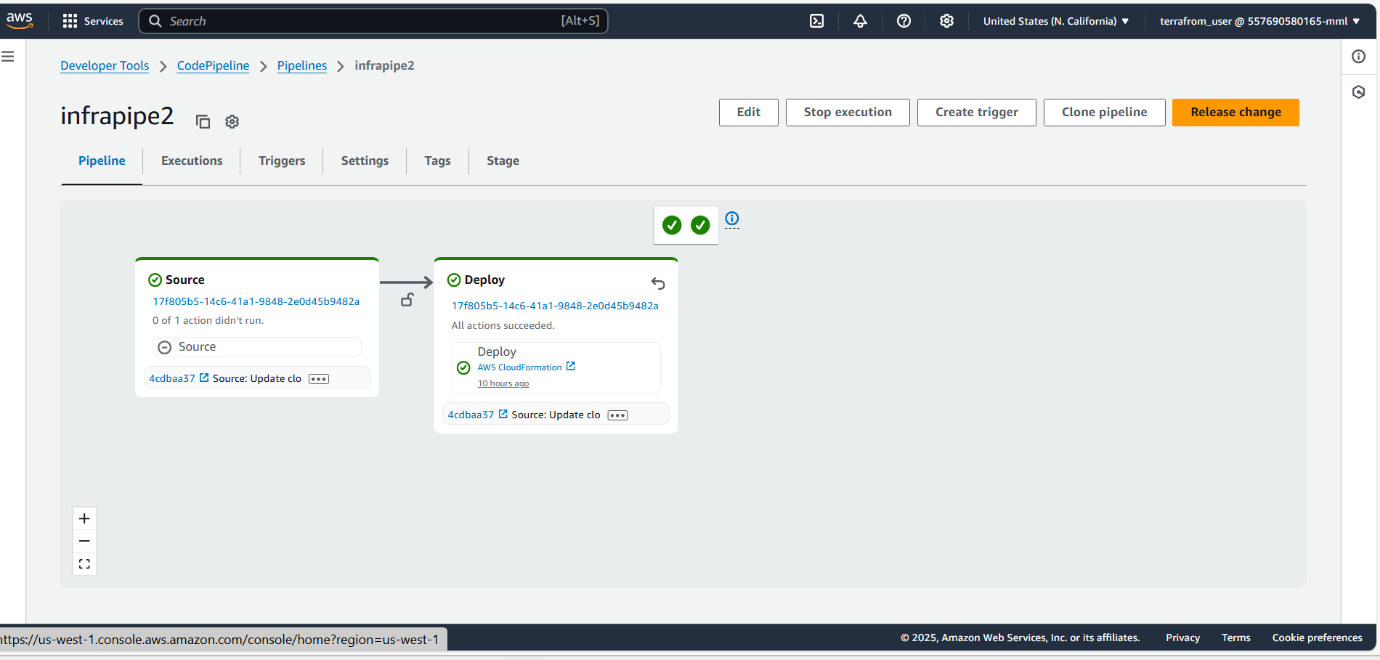
These manifests are generated as build artifacts, ready to be applied to your EKS cluster using **kubectl apply -f k8s-out/.**

****

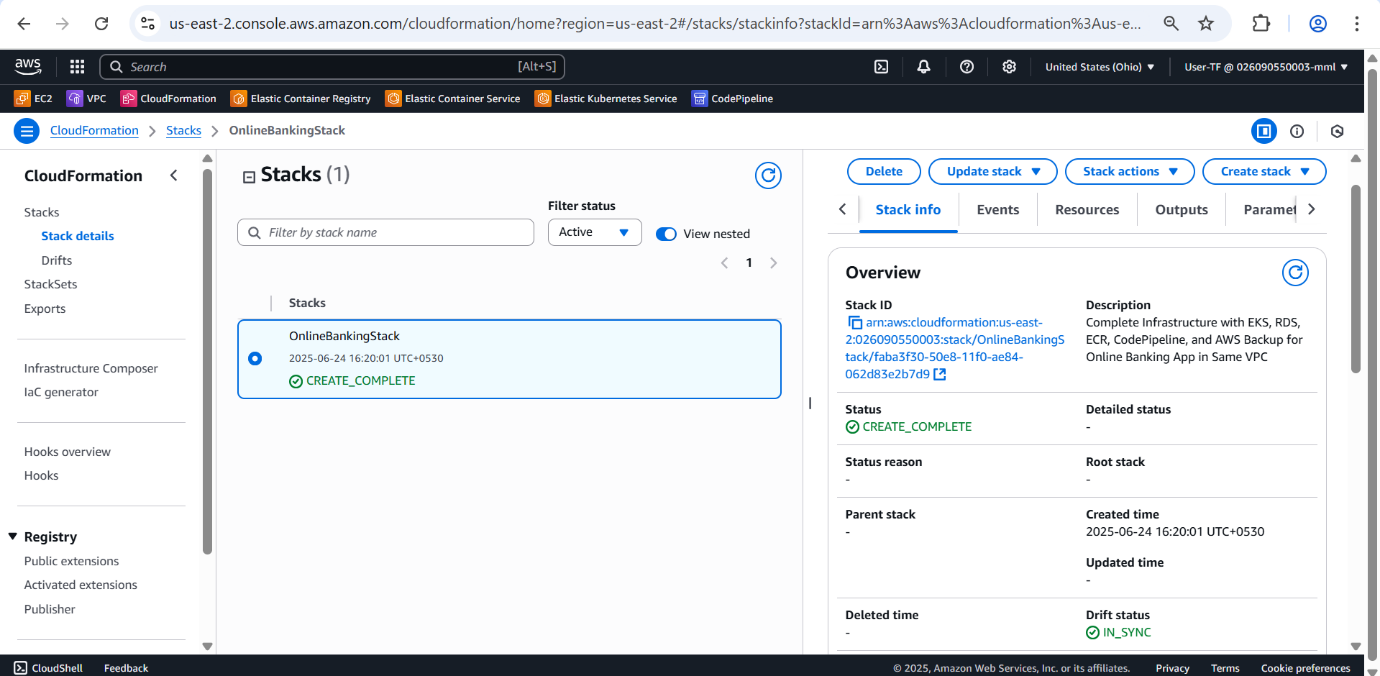
**7. AWS Infra Resource Creation**

**7.1 Using Cloud Formation**

**7.1.1 Cloud Formation Infra resources created using CodePipeline**

****

**7.1.2 After Running the pipeline stack gets created and next Resources with automatic pipeline**

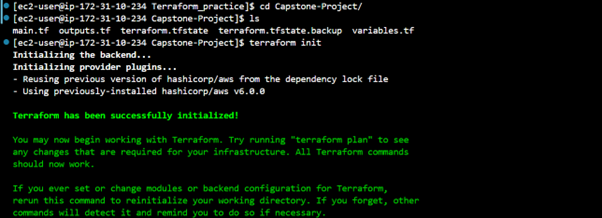


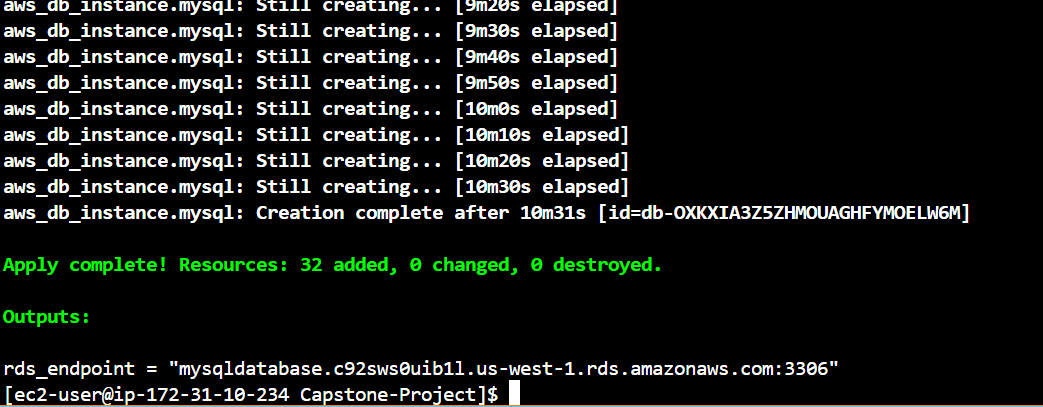
**7.2 Using Terraform**

Terraform is used to programmatically provision and manage your AWS infrastructure, particularly in the us-west-1 region, as indicated by the "End-to-End DevOps Pipeline Using Terraform pipeline created in us-east-1 region"

In essence, creating resources using Terraform involves:

1. **Defining Infrastructure as Code:** main.tf configuration files that declaratively describe the desired state of your infrastructure (e.g., VPCs, subnets, RDS instances, EKS clusters).
2. **Planning:** Terraform generates an execution plan that outlines what it will do to achieve the desired state, showing additions, changes, and destructions.
3. **Applying:** You apply this plan and Terraform automatically provisions or modifies the resources in your AWS account according to the configuration.

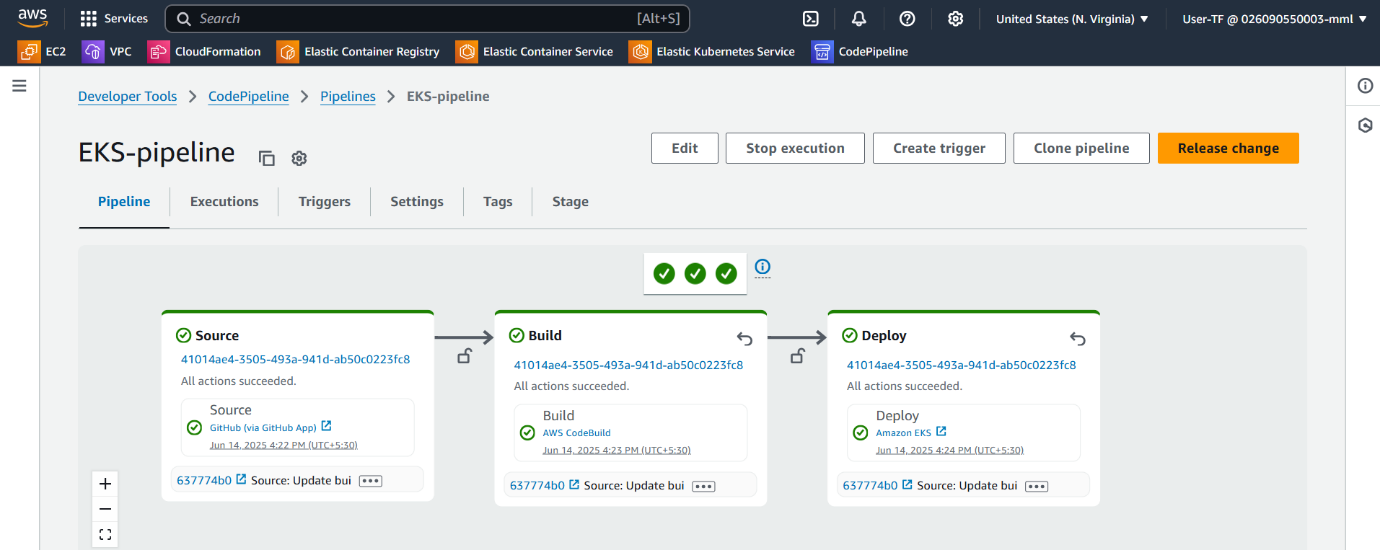




**8. Application running PIPELINES**

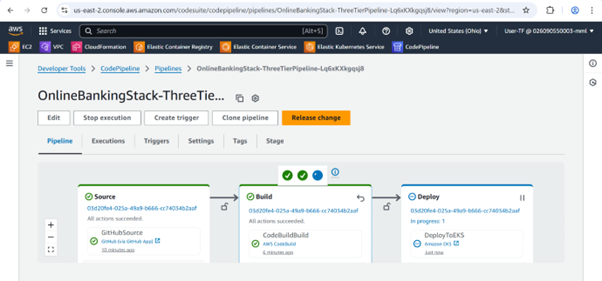
**8.1 Application running pipeline Automation Pipeline Cloud Formation**

**Region 1:** In this region, the entire foundational infrastructure for the application (VPC, Subnets, EKS Cluster, RDS, CI/CD pipeline components) is provisioned using AWS CloudFormation. An independent application pipeline (e.g., OnlineBankingStack-ThreeTierPipeline) builds, scans, and deploys the Online Banking application to the EKS cluster within this CloudFormation-managed environment.



**8.2 Application running pipeline in Terraform Resources Creation**

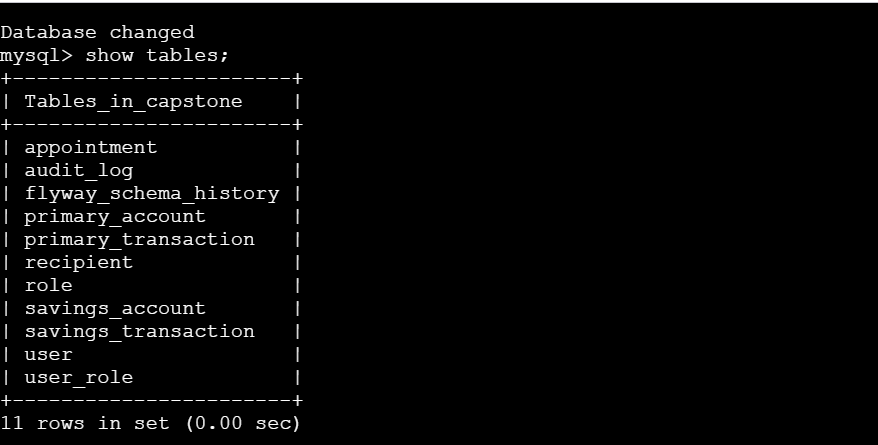
**Region 2:** Similarly, in this second region, a complete duplicate of the application's infrastructure is provisioned using Terraform. A separate application pipeline (e.g., EKS-pipeline) then handles the build, scan, and deployment of the Online Banking application to the Terraform-managed EKS cluster in this region.

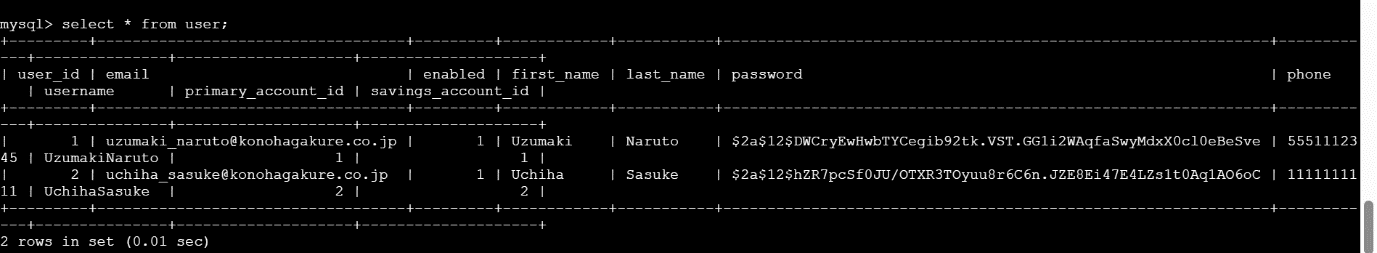


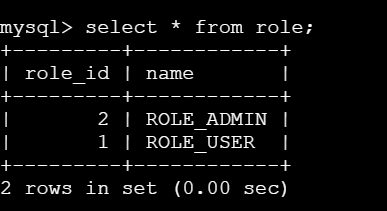
**9. Database Schema and Management**

The project utilizes a MySQL database for data persistence, with schema management handled by Flyway.

* **Database Schema**: The presentation (Images 20-23) reveals the database schema, including tables for:
  + appointment
  + audit\_log
  + flyway\_schema\_history (managed by Flyway)
  + primary\_account
  + primary\_transaction
  + recipient
  + role
  + savings\_account
  + savings\_transaction
  + user
  + user\_role
* **Sample Data**: The database output screenshots show sample data, including ROLE\_ADMIN and ROLE\_USER in the role table, and user entries like "Naruto" and "Sasuke". This confirms a foundational user management system.
* **Flyway Migrations**: The inclusion of flyway-mysql indicates that database schema changes are managed through version-controlled migration scripts, ensuring consistent database states across environments.

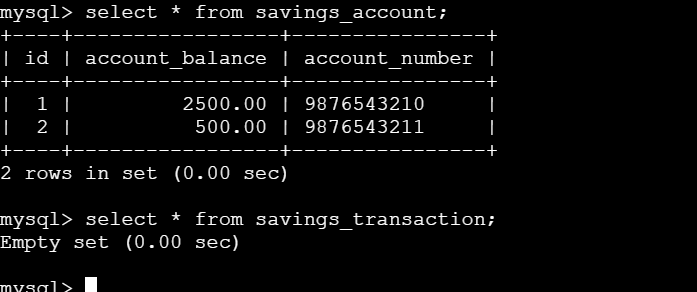






A screen shot of a computer

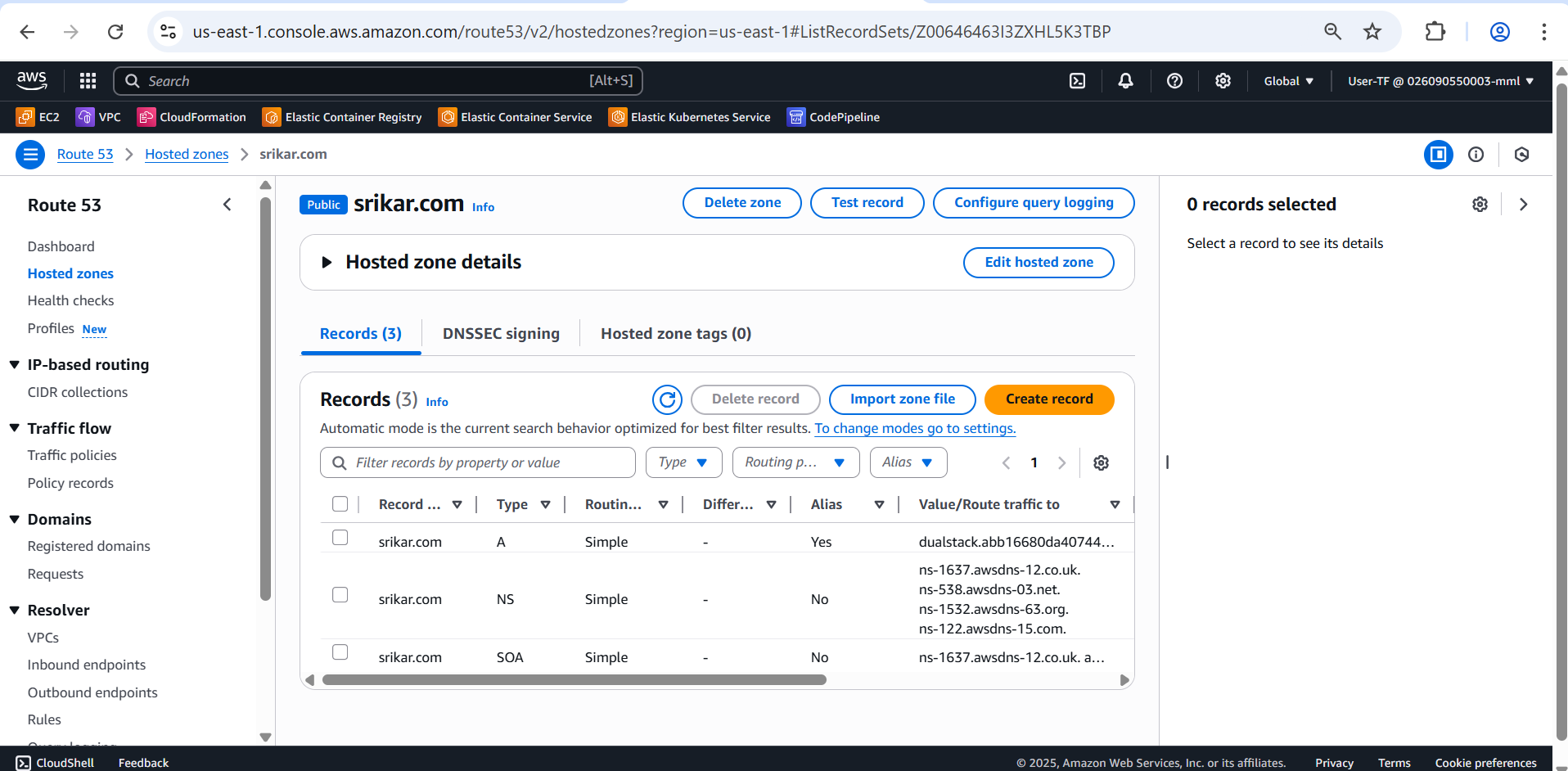
AI-generated content may be incorrect.



**10. Multi-Region Disaster Recovery and High Availability with Route 53**

A critical aspect of this project is its multi-region deployment strategy, leveraging AWS Route 53 for global traffic management and disaster recovery.

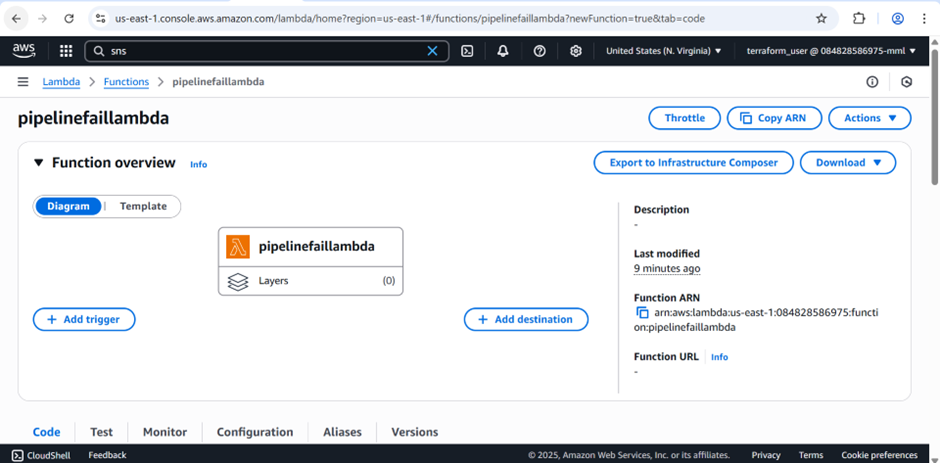
* **Multi-Region ALB Targets**: Route 53 can route traffic to Application Load Balancers (ALBs) deployed in different AWS regions, serving as entry points for the application.
* **Route 53 Failover Policy**: Configured to automatically redirect traffic to a healthy region in case of an outage in the primary region. This is crucial for maintaining high availability.
* **Automated Health Checks**: Route 53 performs automated health checks on the ALBs or other endpoints in each region to determine their health status and facilitate failover.
* **Custom Domain DNS**: The application is accessible via a custom domain, http://srikar.com (as seen in Image 24, with srikan.com/em for the frontend), managed by Route 53.
* **Seamless Cross-Region Failover**: In the event of a regional disaster, traffic is seamlessly routed to the operational region without manual intervention, ensuring business continuity.
* **High Availability / Disaster Recovery**: This multi-region setup provides robust high availability and disaster recovery capabilities, making the online banking application resilient to regional failures.



**11. Event-Driven Automation with Amazon EventBridge**

Amazon EventBridge plays a key role in enabling real-time, event-driven automation within the architecture.

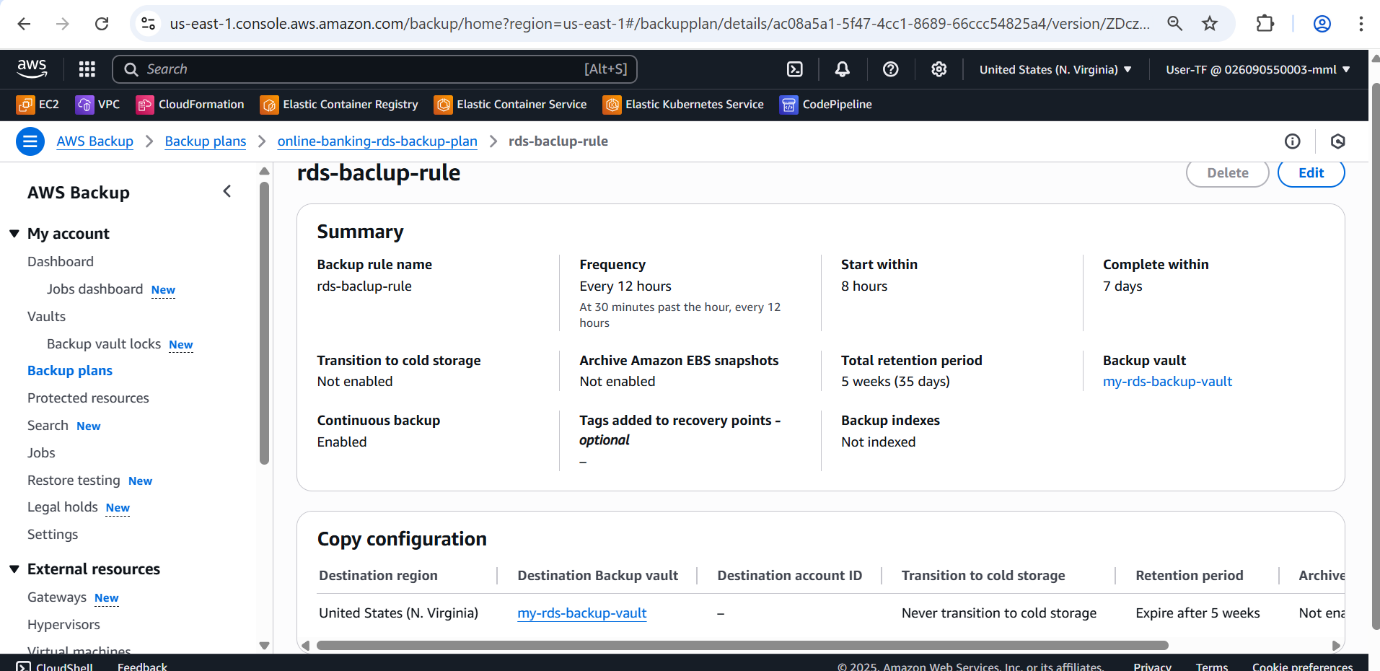
* **Event-Driven Automation**: EventBridge allows you to build scalable, event-driven applications by routing events from various sources to targets.
* **EventBridge Rule Trigger**: Specific rules are configured to detect defined events (e.g., changes in AWS services, custom application events).
* **Lambda Function Invocation**: These rules can trigger AWS Lambda functions, enabling serverless compute for automated responses or data processing.
* **SNS Topic Notification**: Events can also send notifications to Amazon SNS (Simple Notification Service) topics, which can then deliver alerts to subscribers (e.g., email, SMS). The provided screenshot (Image 12) shows an SNS subscription confirmation for arn:aws:sns:us-east-1:026090550003:Capstone\_topic, indicating automated alerting is in place.
* **Real-time Alerts & Automated Responses**: This setup facilitates real-time alerts on critical events and enables automated corrective actions or workflows.



**12. Data Protection with AWS Backup**

AWS Backup provides centralized, automated, and policy-based backup management across AWS services.

* **Centralized Backup Management**: Offers a single dashboard to manage backups for various AWS resources.
* **Automated Backup Plans**: Configured to create backup plans with defined schedules (e.g., Every 12 hours as seen in Image 25 for rds-backup-rule).
* **Custom Retention Policies**: Allows setting custom retention periods for backups (e.g., 5 weeks (35 days) total retention).
* **Cross-Region Backup Copies**: Supports copying backups to other AWS regions for enhanced disaster recovery.
* **Backup Vaults (Secure Storage)**: Backups are stored securely in designated backup vaults (e.g., my-rds-backup-vault).
* **Database Disaster Recovery**: Essential for the MySQL database, ensuring that point-in-time recovery is possible and that data can be restored in case of data loss or corruption. The online-banking-rds-backup-plan is a specific example of this.



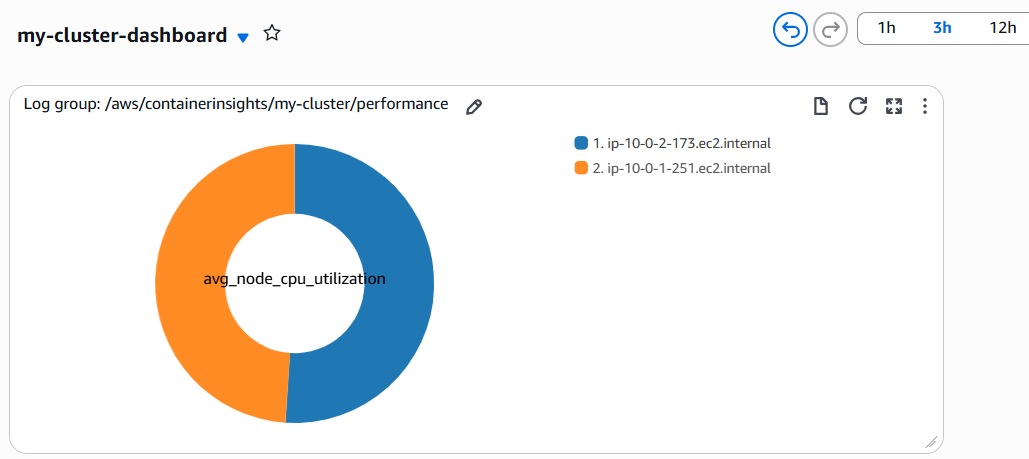
**13. Monitoring and Observability with CloudWatch and Grafana**

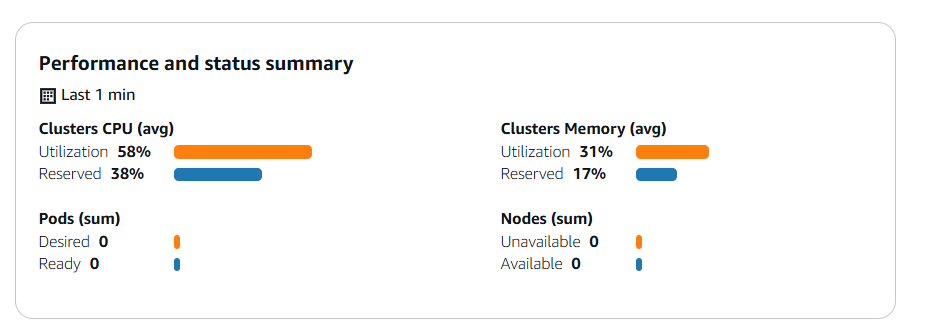
Effective monitoring and logging are crucial for observing application health, diagnosing issues, and ensuring performance in a cloud-native environment.

**13.1. Amazon CloudWatch**

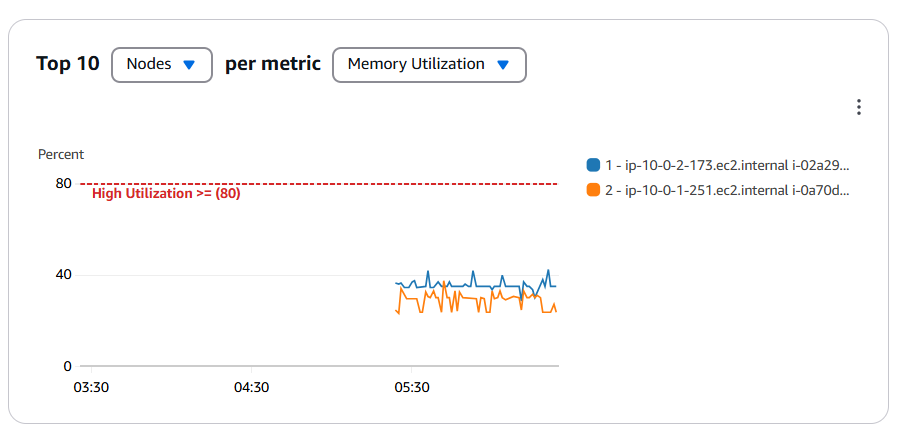
CloudWatch is the native AWS monitoring and observability service.

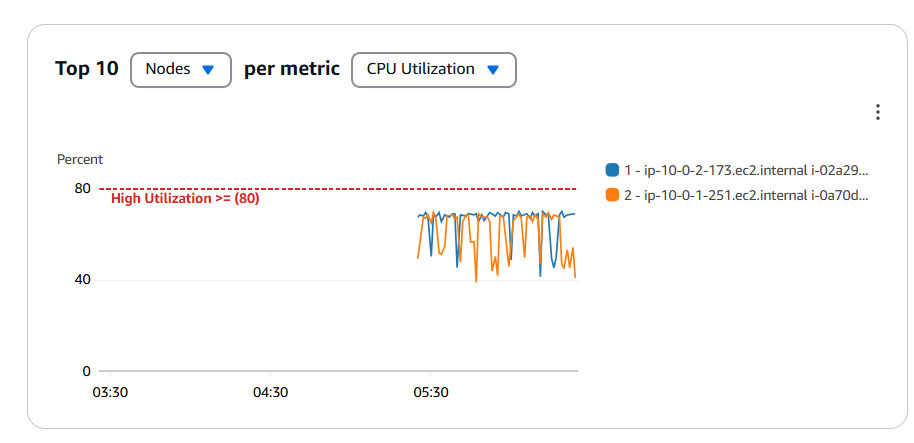
* **Key Capabilities**:
  + **Log Collection**: Aggregates application and infrastructure logs from EKS pods, EC2 instances, and other AWS services.
  + **Metrics**: Collects detailed performance data (CPU usage, memory, network I/O, disk I/O, etc.) for EKS clusters (nodes, pods, containers) and RDS instances. Screenshots (Images 13, 14, 15, 16) show CPU utilization metrics, cluster performance summaries (CPU, Memory utilization, pods, nodes), and cluster state summaries.
  + **Alarms**: Configures alerts for predefined thresholds on metrics. An example screenshot (Image 13) shows a CPU\_utilization alarm set at 75%. These alarms can trigger SNS notifications.
  + **Dashboards**: Visualizes key metrics and logs through customizable dashboards for operational insights, as seen in Images 14, 15, 16 showing performance and status summaries and cluster state.
* **Benefits**: Proactive issue detection, real-time performance monitoring, faster troubleshooting, and ensuring application health and availability.



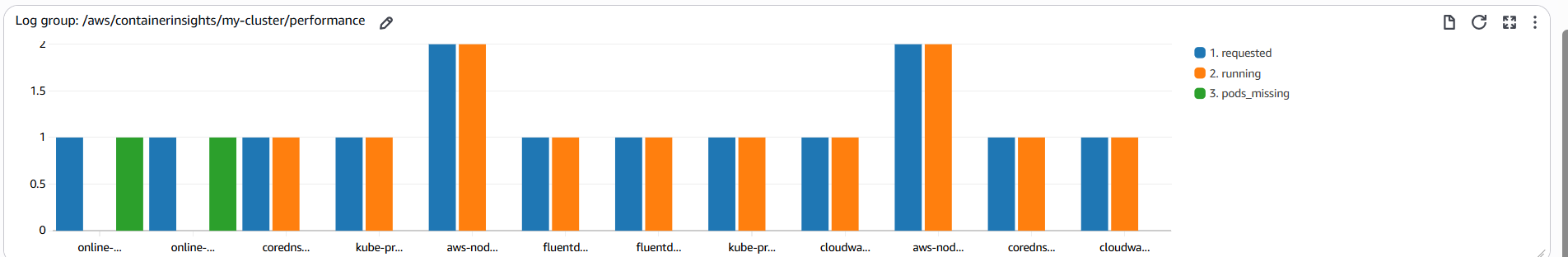


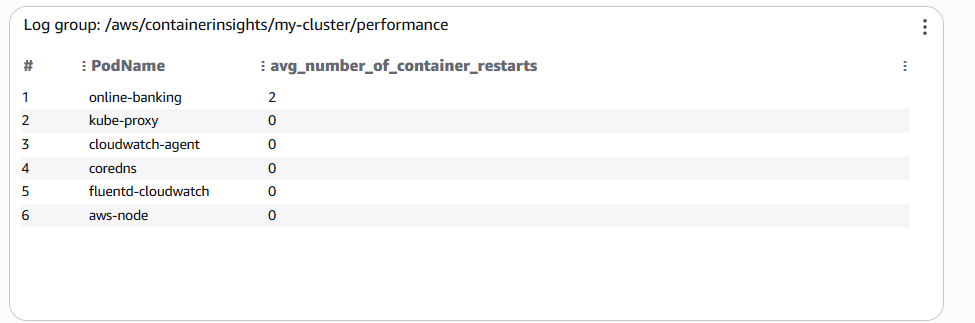




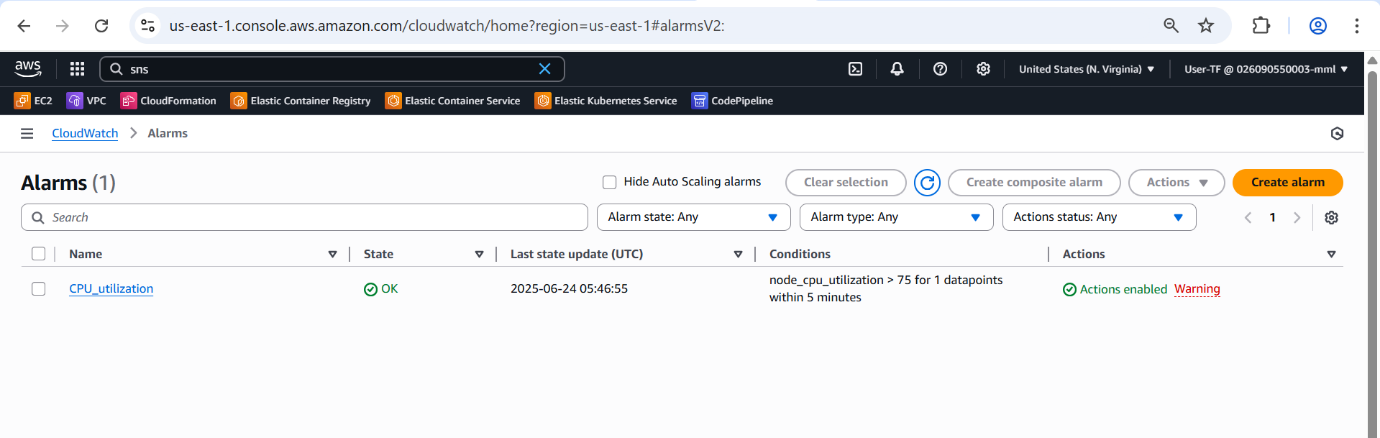


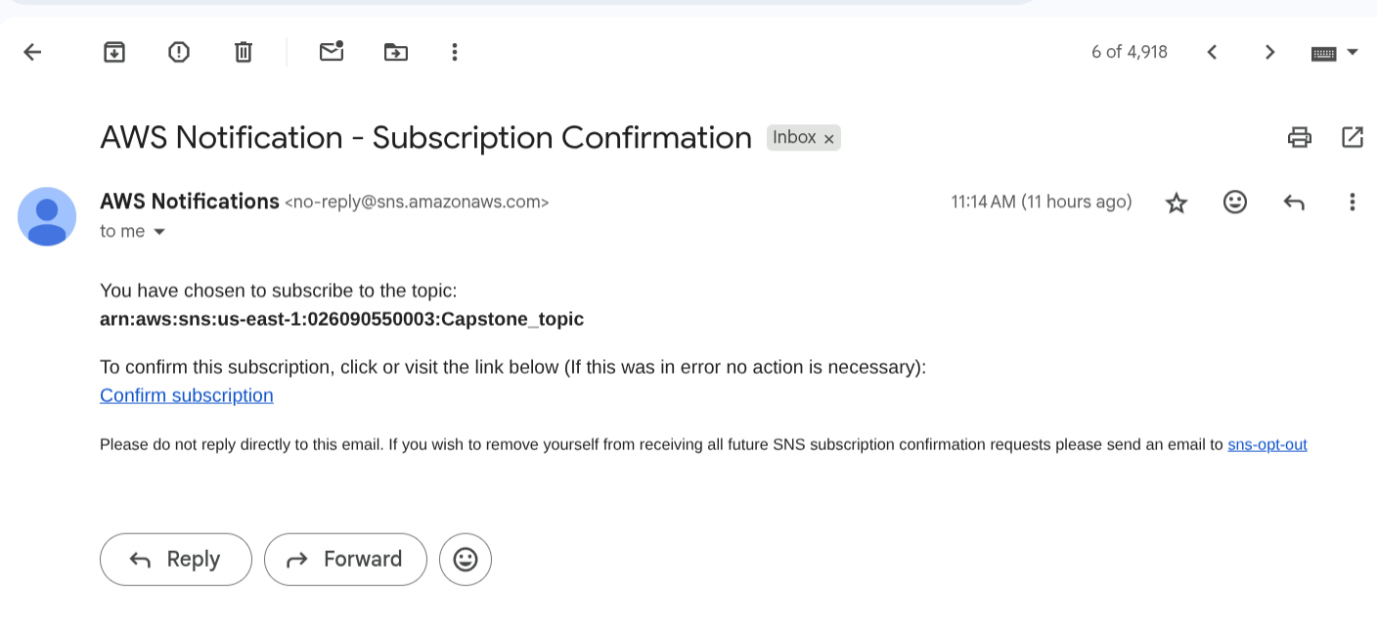
A white background with black text

AI-generated content may be incorrect.



**13.2 Alarm Notification**

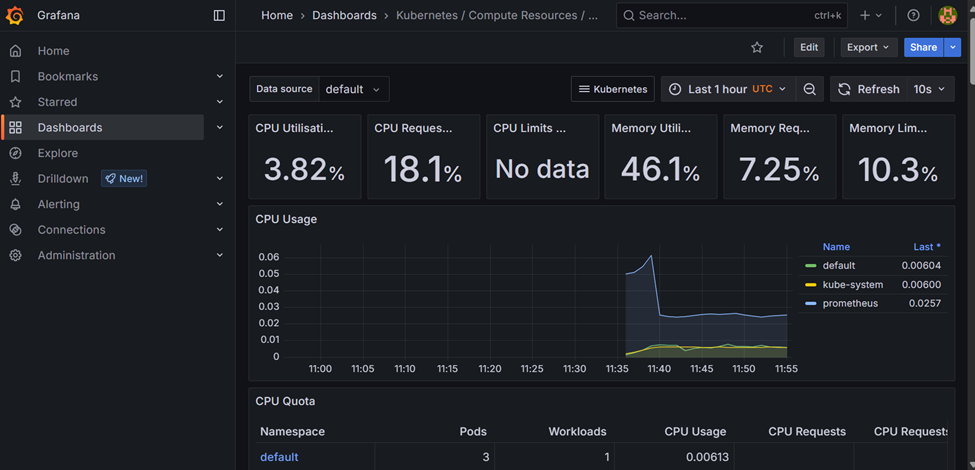


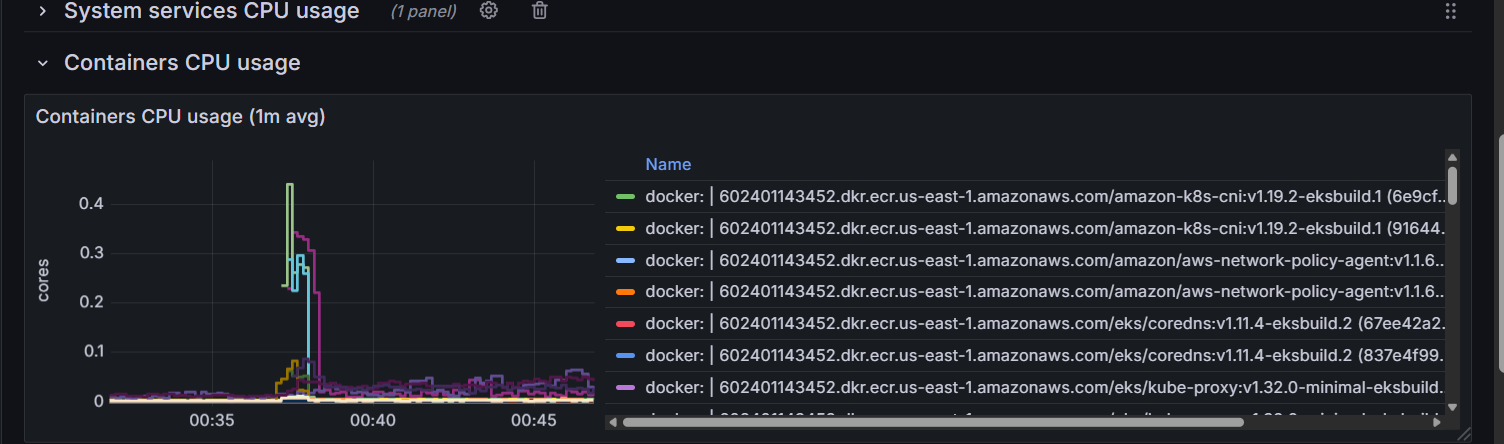


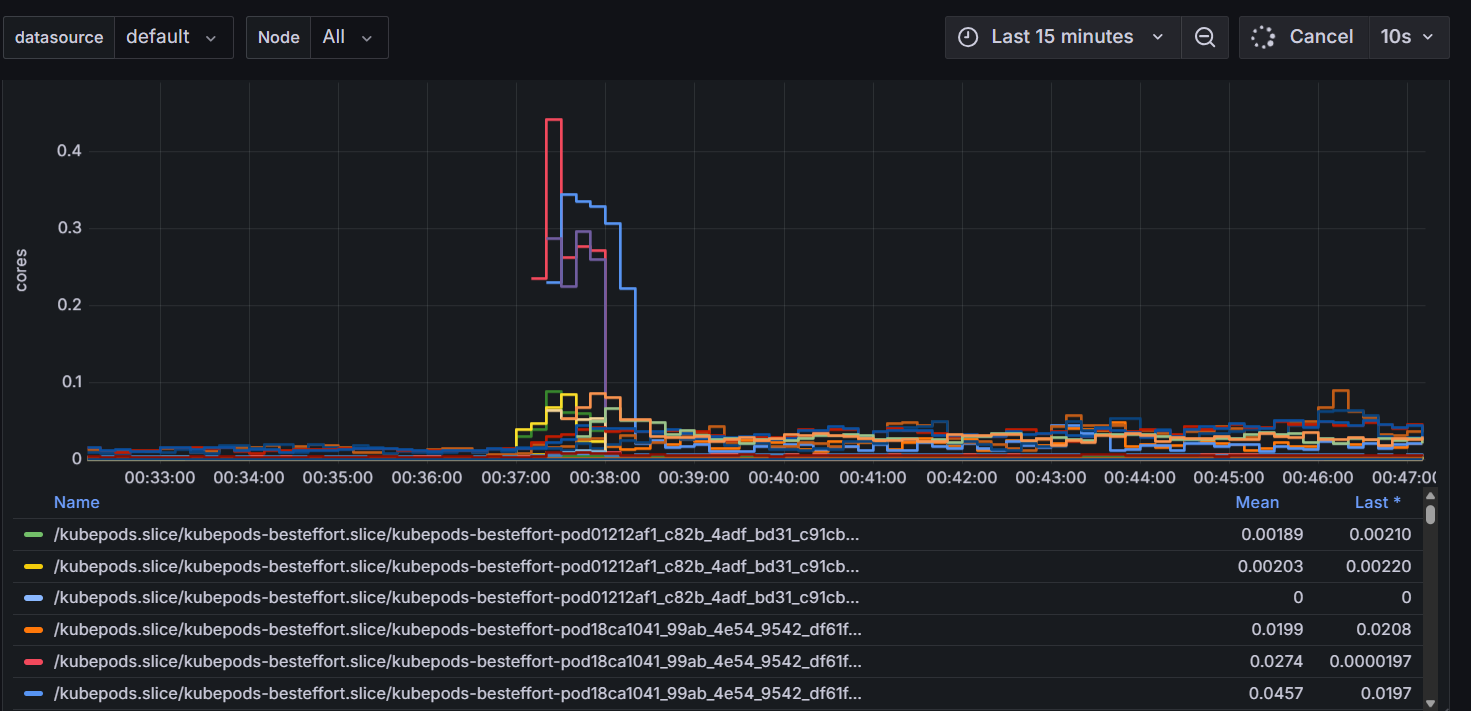
**13.3. GRAFANA**

Grafana is an open-source platform for analytics and monitoring.

* **Purpose**: Used to visualize time-series data from various sources, providing rich and customizable dashboards.
* **Integration**: Can integrate with Amazon CloudWatch, Prometheus (if deployed in EKS), or other data sources to pull metrics for visualization. The screenshots (Images 17, 18, 19) clearly show Grafana dashboards monitoring Kubernetes compute resources, including CPU utilization, CPU requests, memory utilization, memory requests, and individual container CPU usage.
* **Benefits**: Offers advanced visualization capabilities, custom dashboards for specific operational needs, and the ability to correlate different metrics across the infrastructure.







**14. Scaling and High Availability**

The architecture is designed for high scalability and availability, leveraging Kubernetes and AWS services.

* **Horizontal Pod Autoscaler (HPA)**: Configured in Kubernetes to automatically scale the number of application pods up or down based on observed CPU utilization or other custom metrics, ensuring the application can handle varying loads.
* **Cluster Autoscaler**: Integrates with EKS to automatically scale the underlying EC2 instances (worker nodes) of your Kubernetes cluster up or down based on pending pod resource requests, ensuring enough capacity for HPA.
* **Multi-AZ Deployment for RDS**: Amazon RDS allows you to deploy your MySQL database in a Multi-AZ configuration, which provides automatic failover to a synchronous standby replica in a different Availability Zone in case of primary database instance failure, significantly enhancing database availability.
* **Load Balancing**: The Kubernetes Service of type LoadBalancer (typically backed by an AWS Elastic Load Balancer like ALB or NLB) distributes incoming traffic across your application pods, providing fault tolerance and load distribution. Route 53 then extends this load balancing across regions.

**15. Key Project Outcomes**

The successful deployment of this online banking application highlights several key achievements:

* **Multi-tier Application Deployment**: Successfully deployed a multi-tier application across **two AWS regions (us-east-1 and us-west-1)**, demonstrating robust geographical distribution.
* **Infrastructure as Code (IaC) Adoption**: Utilized **different IaC tools (AWS CloudFormation and Terraform)** for infrastructure provisioning in different regions, showcasing versatility and best practices for repeatable, version-controlled infrastructure.
* **Secure EKS Communication**: Ensured frontend interaction with the backend occurs over secure EKS communication channels.
* **Automated CI/CD Pipelines**: Implemented end-to-end CI/CD pipelines that automate code delivery, infrastructure provisioning, and application deployment, significantly reducing manual effort and potential errors.
* **Scalable, Fault-Tolerant, and Region-Resilient Application**: The architecture is designed to be highly scalable, fault-tolerant, and resilient to regional failures, critical for a banking application.
* **Continuous Monitoring for Quality and Security**: Code quality and security are continuously monitored via SonarQube and Trivy scans.
* **Real-time Monitoring and Alerting**: Achieved through integrated CloudWatch, Grafana, and EventBridge, providing proactive insights and notifications.

**16. GitHub Repository and Links**

The source code and infrastructure templates for this project are available in the following GitHub repositories:

* **Application-Repository**:

<https://github.com/Srikar2610/AWS_Capstone_Project.git>

* **Cloud-Formation Infra template Repository**: <https://github.com/Srikar2610/Capstone_CFT_Repo.git>
* **Terraform Infra template Repository**: <https://github.com/Srikar2610/Capstone_Terraform_repo.git>

The application's custom domain name is configured as [**http://srikar.com.**](http://srikar.com.)

**17. Local Development Setup**

For developers working on the project:

* **Java Development Kit (JDK)**: Ensure JDK 21 is installed, aligning with the project's pom.xml and Dockerfile.
* **Maven**: Install Maven for building and managing dependencies locally.
* **Docker**: Install Docker Desktop for building and running containers locally.
* **IDE**: Use a modern Integrated Development Environment (IDE) like IntelliJ IDEA or VS Code with relevant Spring Boot extensions for enhanced productivity.
* **Local MySQL**: For development and testing purposes, a local MySQL instance (or a containerized one using Docker Compose) can be used. Database connection details should be configured in application.properties or application.yml for local profiles.
* **Kubectl**: Install kubectl to interact with local Kubernetes environments (e.g., Minikube, Kind) if local Kubernetes testing is desired.

**18. Prerequisites for AWS Deployment**

Before running this CI/CD pipeline and deploying the application, ensure the following AWS resources and configurations are in place:

* **AWS Account**: An active AWS account with appropriate billing configured.
* **VPC and Subnets**: A Virtual Private Cloud (VPC) with well-designed public and private subnets spanning multiple Availability Zones to ensure network isolation and high availability.
* **Amazon RDS MySQL Instance**: A MySQL database instance named mydb running in Amazon RDS, configured for high availability (Multi-AZ) and accessible from your EKS cluster's VPC. Ensure necessary security groups are in place.
* **Amazon ECR Repository**: A private ECR repository named online-banking to store your Docker images.
* **Amazon EKS Cluster**: A running Kubernetes cluster (EKS) in each target region (us-east-1, us-west-1) configured with appropriate IAM roles for worker nodes (allowing ECR image pulls) and a functional kubeconfig.
* **IAM Roles and Permissions**:
  + **CodeBuild Service Role**: Must have permissions to:
    - Interact with ECR (ecr:GetAuthorizationToken, ecr:BatchCheckLayerAvailability, ecr:PutImage, etc.).
    - Access S3 (for source code and artifacts).
    - Describe RDS instances (rds:DescribeDBInstances).
    - Log to CloudWatch Logs.
  + **EKS Worker Node Roles**: Must have permissions to pull images from ECR.
  + **CodePipeline Service Role**: Required if using AWS CodePipeline to orchestrate CodeBuild, with permissions to trigger CodeBuild, deploy to EKS (via kubectl or eksctl), and access S3 for artifacts.
* **Kubernetes Configuration**: Ensure k8s/deployment.yaml and k8s/service.yaml exist in your source repository with the IMAGE\_PLACEHOLDER correctly defined in the deployment manifest, ready for sed replacement.
* **Maven Settings File**: The maven-settings.xml file must be present in the root of your source repository alongside the Dockerfile to enable Maven mirror functionality.