End-to-End DevOps, Cloud Infrastructure, Monitoring, and AIOps Project

Comprehensive Documentation

Mahmoud Mohamed (Dockerfile, CI/CD, Monitoring)
Eslam Fouad (CI/CD, Monitoring)
Ezzat Tarek (EKS)
Omar Tamer (Infrastructure)
Rawan Mohamed (AIOps)

November 2025

Contents

1 Full-Stack Application (Node.js + Frontend)

1.1 Purpose

A unified production-ready full-stack application where the backend (Node.js API) and frontend (web UI) are packaged together using a multi-stage Docker build. This ensures minimal image size, improved security, and optimal performance on Kubernetes or ECS.

1.2 Key Features

- Unified backend + frontend production image.
- Final image based on lightweight `node:20-slim`.
- Serves static frontend assets and backend API together.
- Ideal for horizontal scaling on EKS / ECS.
- Clean separation of build and runtime stages.

1.3 Multi-Stage Dockerfile Overview

Stage 1 — Frontend Build

- Base: `node:20`
- Installs frontend dependencies.
- Builds optimized static assets (`dist/`).
- Prepares production-ready frontend files for the final image.

Stage 2 — Backend Build

- Base: `node:20`
- Installs backend dependencies.
- Copies backend application source code.

Stage 3 — Final Runtime Image

- Base: `node:20-slim`
- Copies backend + built frontend from previous stages.
- Installs only production dependencies.

- Exposes port 5000 for API and frontend.
- Starts application using `node server.js`.

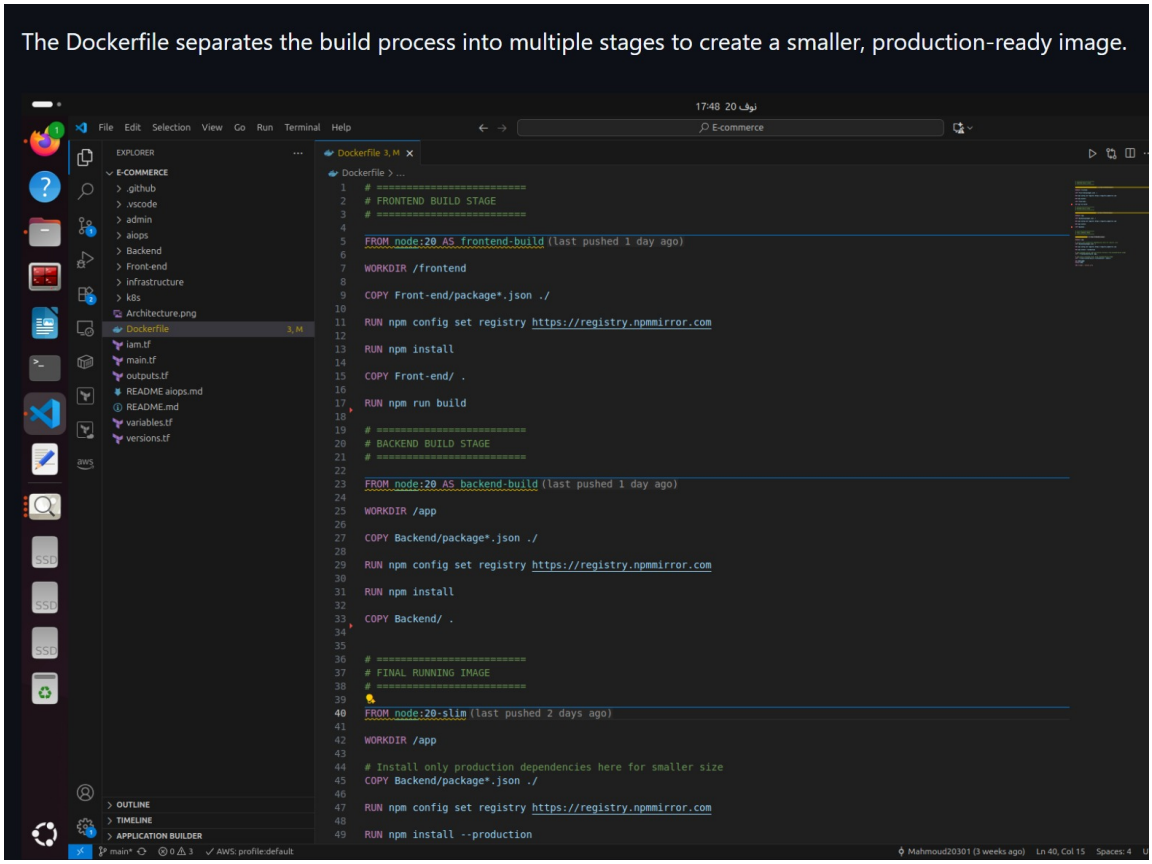

Figure 1: Multi-Stage Docker Build

1.4 Benefit

A minimal, secure, cloud-optimized Docker image suitable for production workloads, combining both frontend and backend efficiently.

2 CI/CD Pipeline (GitHub Actions)

2.1 Goals

- Automate build, scan, test, and deployment workflows.
- Enforce code quality using SonarCloud.
- Secure Docker images using Trivy vulnerability scans.
- Push verified images to Docker Hub and AWS ECR.
- Provide real-time team feedback via Slack.

2.2 Pipeline Stages

2.2.1 SonarCloud Code Analysis

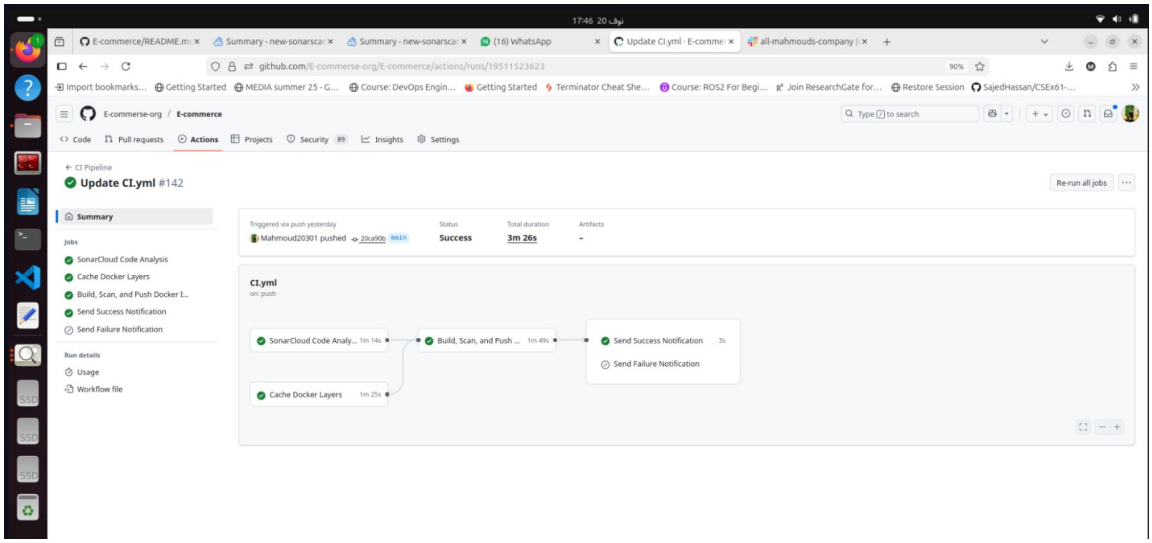

Figure 2: pipeline overview

Performs static code analysis to check code quality and test coverage. Enforces quality gates; deployment blocked if issues are detected.

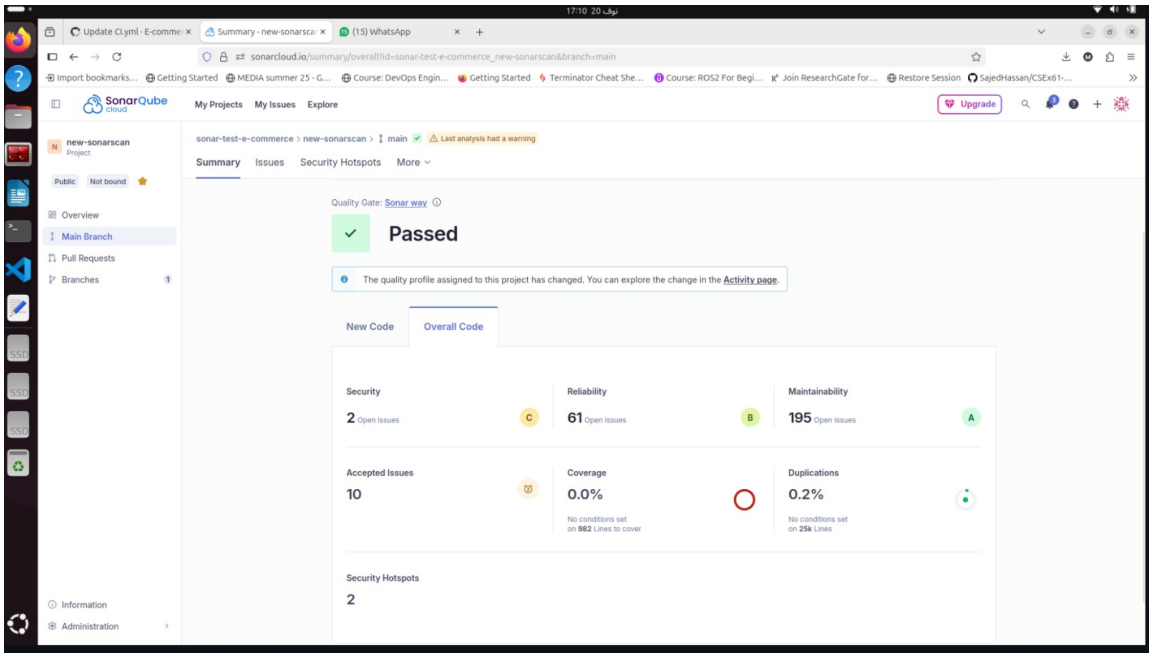

Figure 3: SonarCloud Analysis Results

2.2.2 Docker Layer Caching

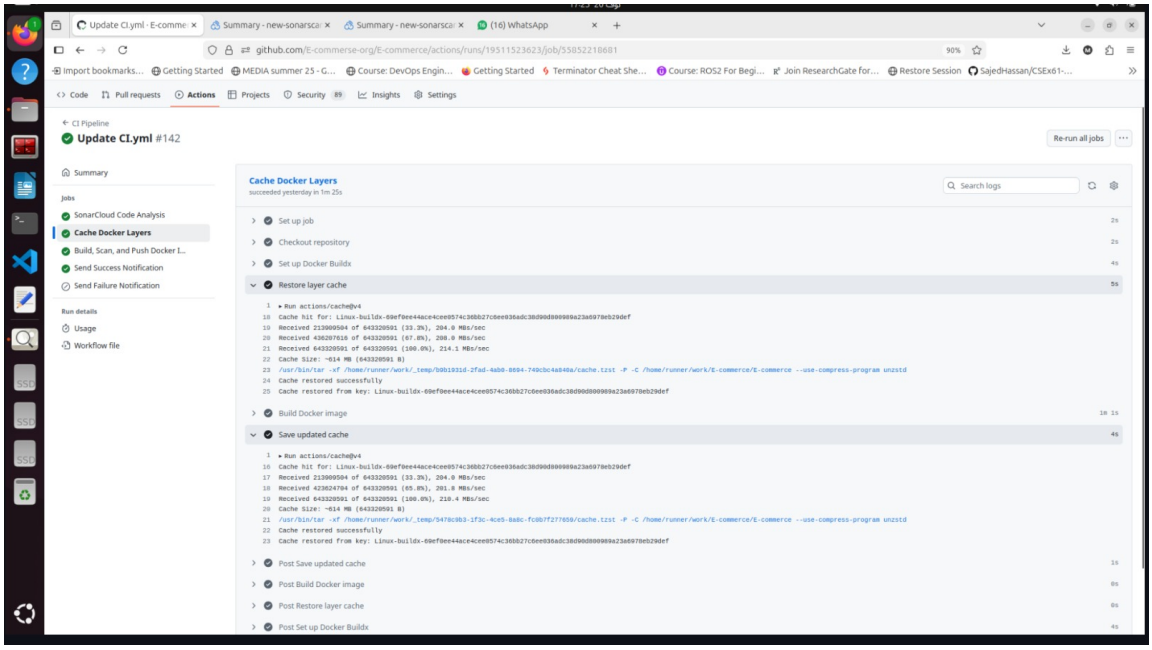

Figure 4: Docker Layer Caching

Restores and caches Docker layers from previous builds to speed up repeated deployments.

2.2.3 Build, Scan, and Push

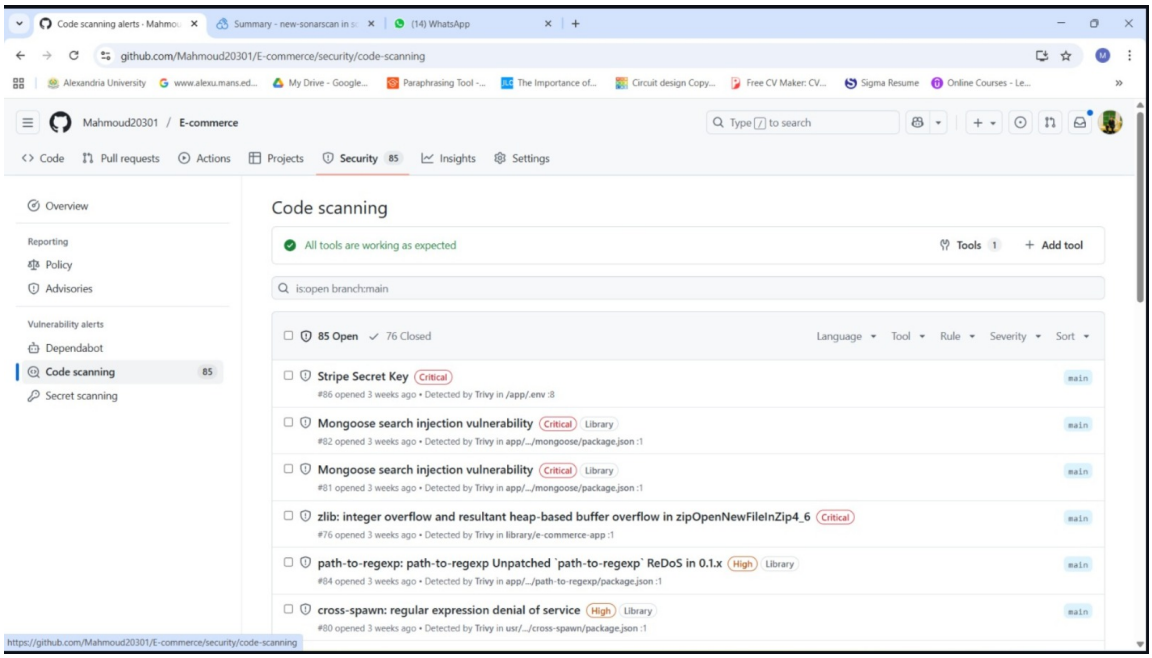

Figure 5: Docker Image Scan

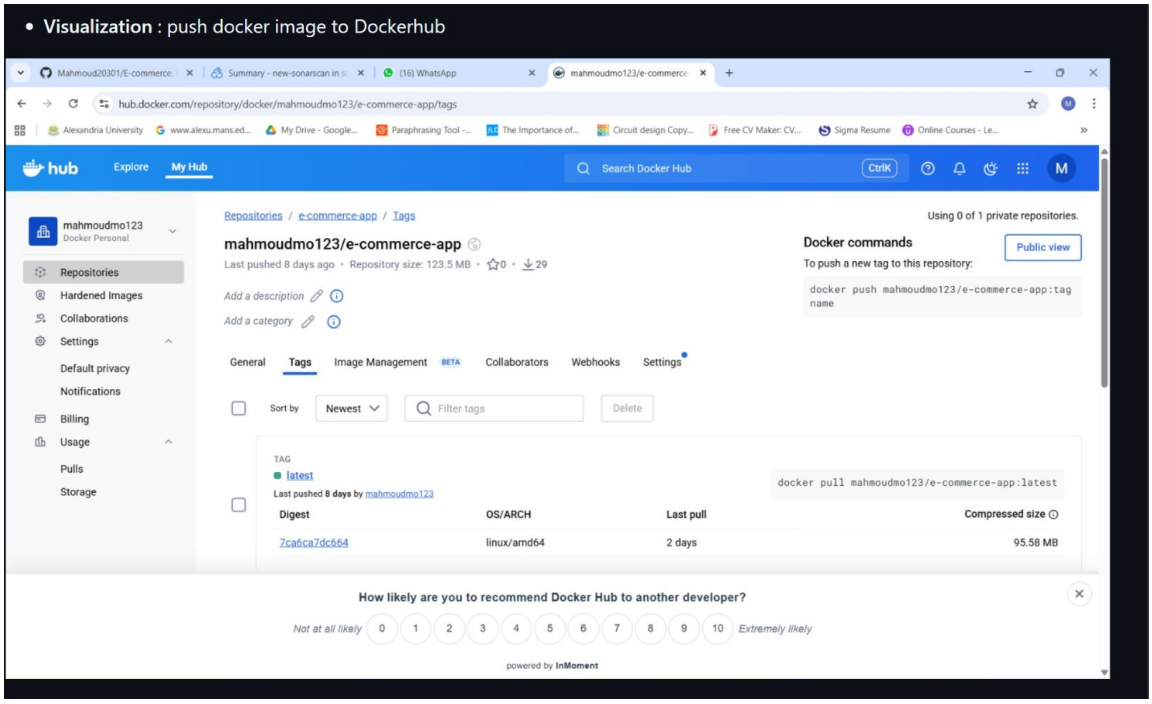

Figure 6: Push to dockerhub

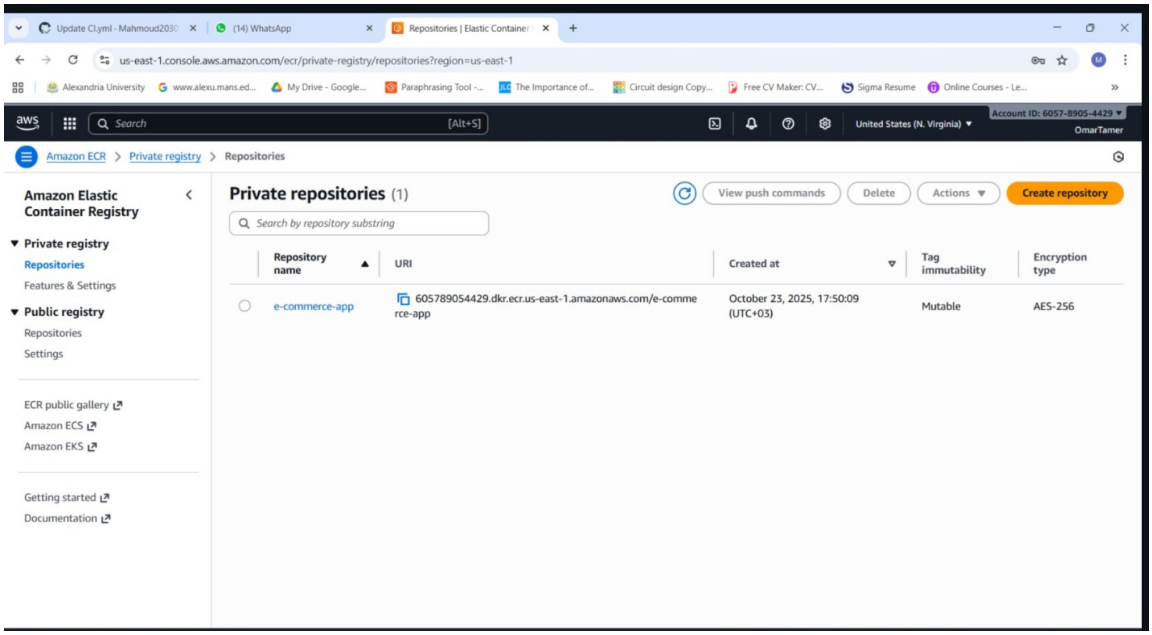

Figure 7: Push to Amazon ecr

Builds production-ready Docker images from the multi-stage Dockerfile, scans for vulnerabilities using Trivy, and pushes verified images to Docker Hub and AWS ECR.

2.2.4 Notifications

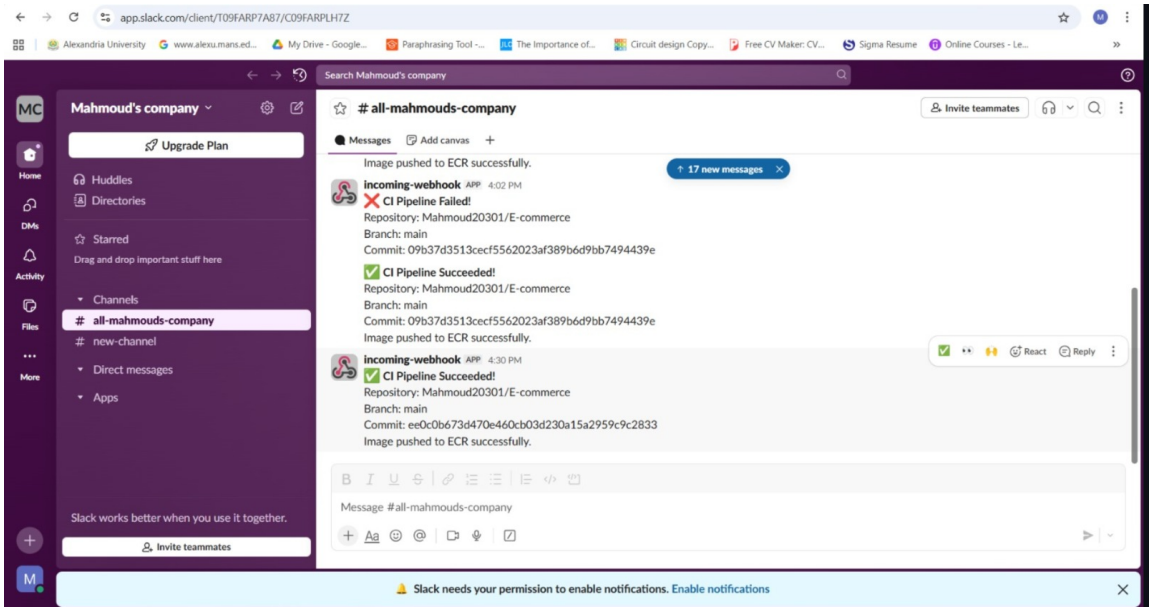

Figure 8: Slack Notification on Success

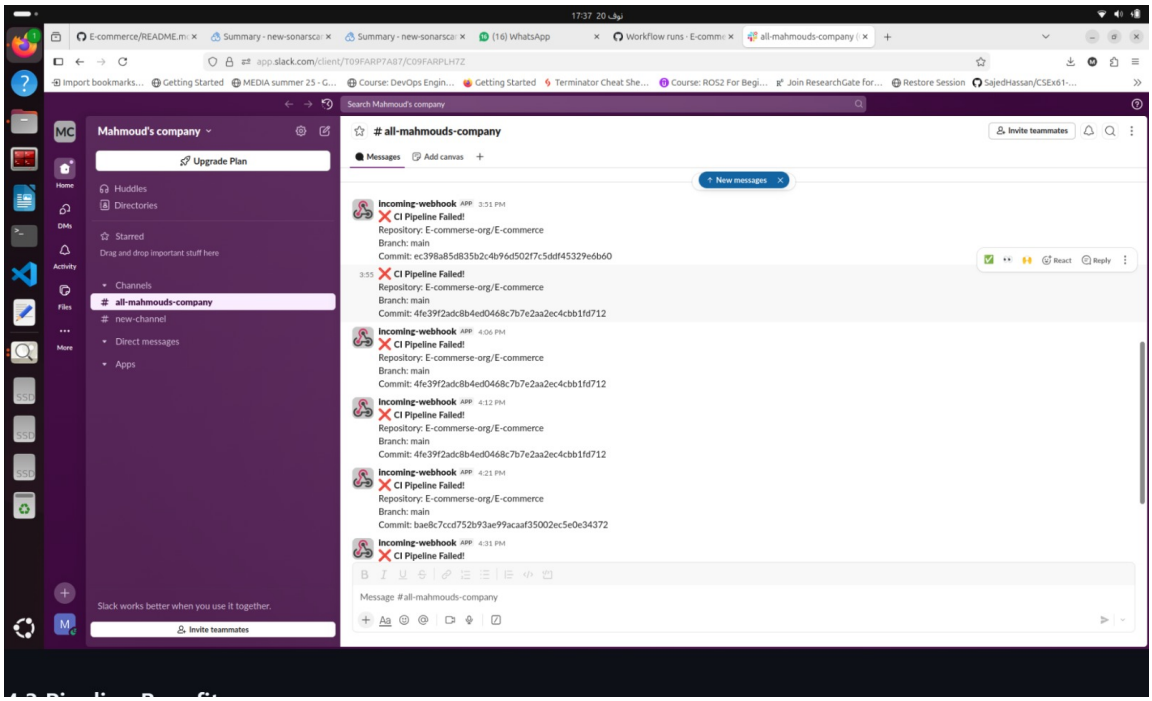

Figure 9: Slack Notification on Failure

Success: Slack message with repository, branch, commit SHA, and image version. Failure: Immediate Slack alert including failed stage details.

3 Kubernetes Monitoring Layer

3.1 Overview

The monitoring stack is deployed on Amazon EKS using Terraform and Helm, providing full observability for the cluster and workloads.

3.2 Components

- Prometheus for metrics collection.
- Grafana for dashboards and visualization.
- Alertmanager for real-time notifications.
- kube-state-metrics for Kubernetes object metrics.
- Node exporters for host-level metrics.
- Custom dashboards and Prometheus recording rules.

3.3 Provisioning Method

- Terraform Helm provider to manage deployment.
- kube-prometheus-stack Helm chart for all monitoring components.
- YAML configuration files for Prometheus and Grafana.

3.4 Outputs

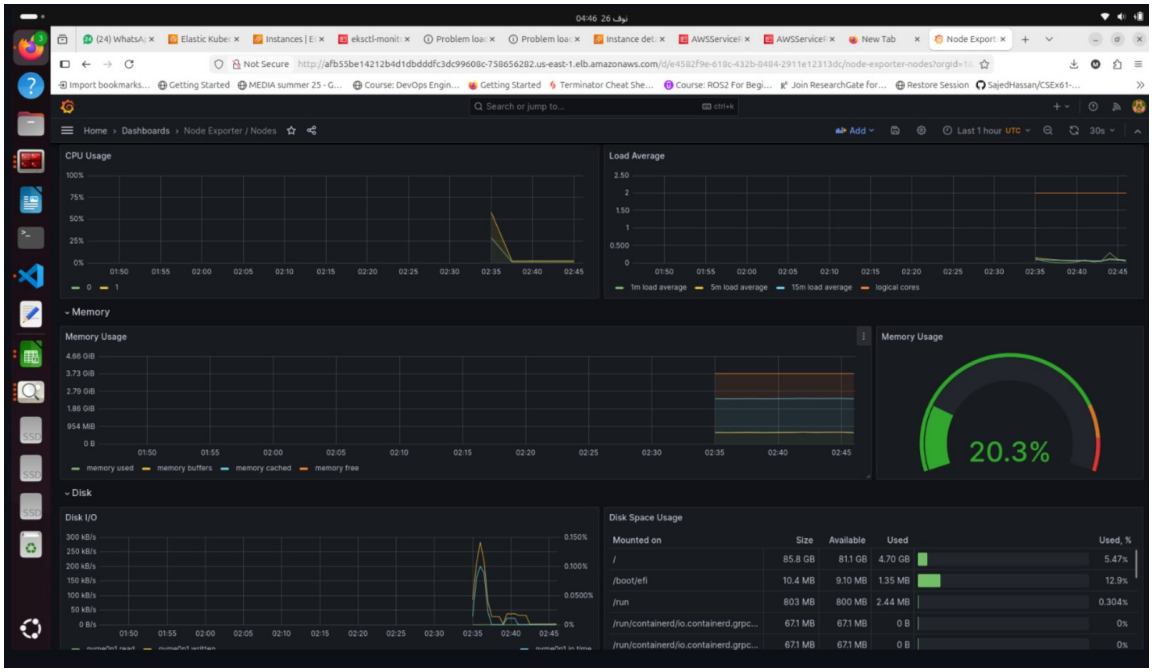

Figure 10: Grafana Dashboard

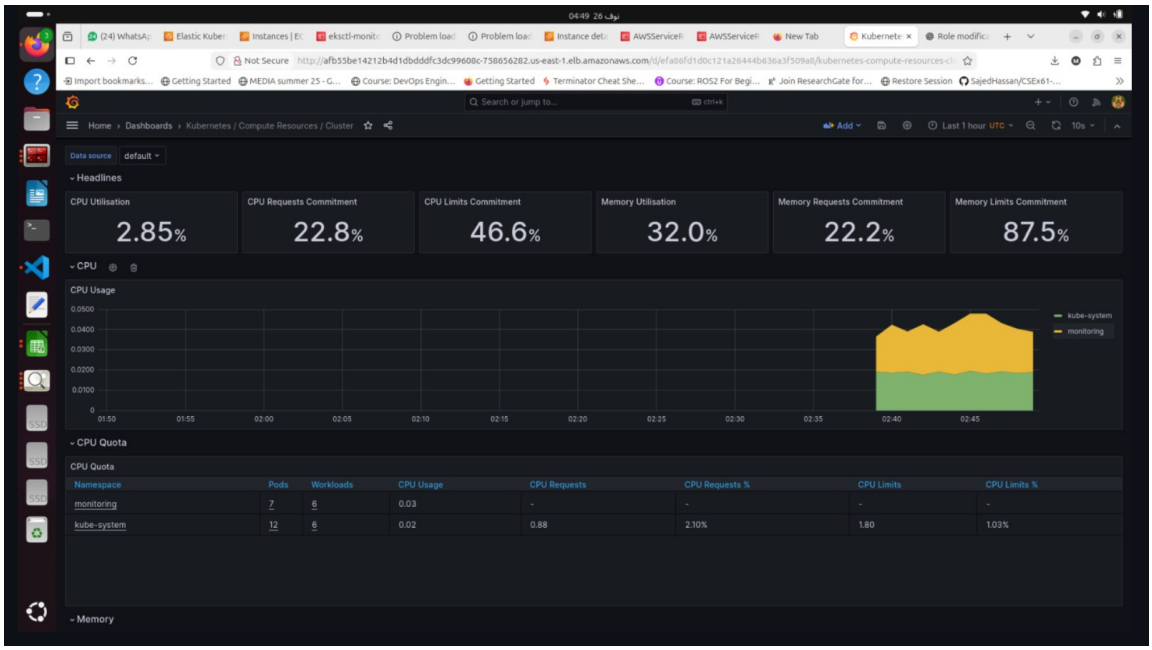

Figure 11: Grafana Dashboard

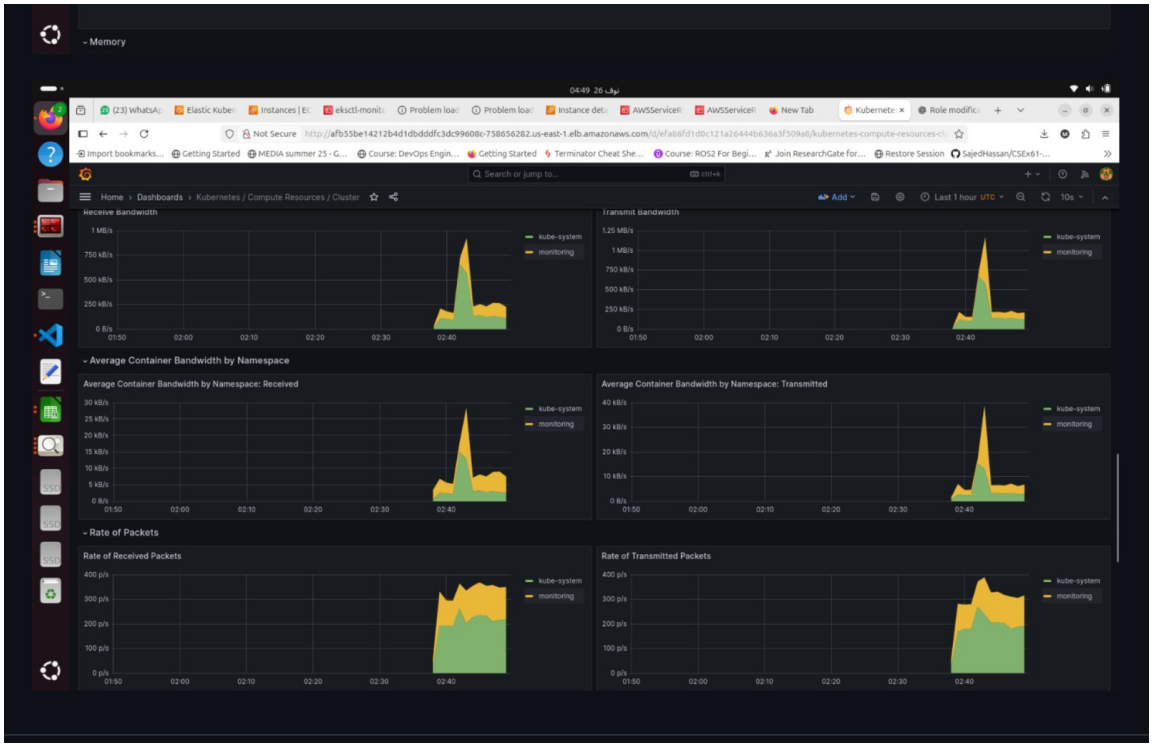

Figure 12: Grafana Dashboard

- Grafana URL.
- Command to retrieve Grafana admin password from Kubernetes Secret.

4 AIOps – Metrics-Based Anomaly Detection

4.1 Purpose

A Python-based anomaly detection system using IsolationForest to monitor Prometheus metrics and identify unusual patterns in real-time.

4.2 Features

- Fetches metrics directly from Prometheus API.
- Trains IsolationForest model on collected data.
- Detects spikes, drops, or other anomalies.
- Sends alerts to Slack, Teams, or custom webhooks.
- Deployable as a Kubernetes Deployment.

4.3 Included Components

- Python detector script.
- Dockerfile for the detector image.
- Kubernetes manifests (Namespace, ServiceAccount, Deployment, Secret).
- Prometheus recording rules (latency, CPU, custom metrics).

4.4 Workflow (Local Minikube Example)

1. Start Minikube with Docker driver.
2. Deploy kube-prometheus-stack via Helm.
3. Build and load the detector image into Minikube.
4. Deploy the detector.
5. Configure Prometheus in-cluster URL as environment variable.
6. Run smoke tests with sample PromQL queries.

5 Terraform Infrastructure (Multi-Cell EKS Architecture)

5.1 Purpose

A fault-isolated Kubernetes architecture with three separate EKS clusters (cell1, cell2, cell3), designed for high availability, scalability, and multi-tenant workloads.

5.2 Architecture Components

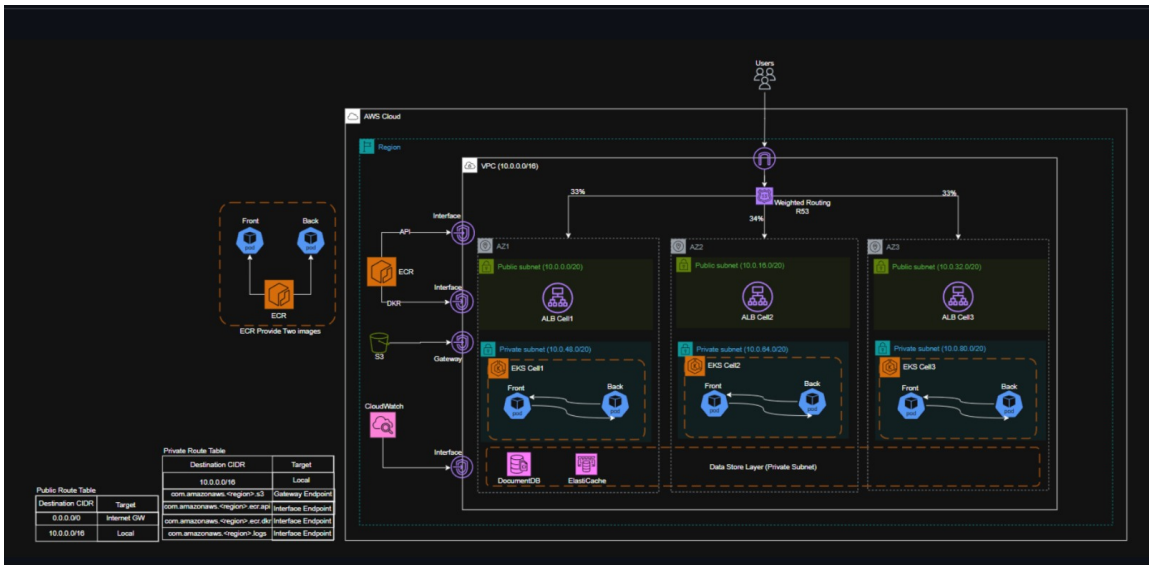

Figure 13: Multi-Cell EKS Architecture

- Custom VPC with 3 public and 6 private subnets.
- Three independent EKS clusters.
- AWS Load Balancer Controller for ALB/NLB ingress.
- Cluster Autoscaler for dynamic node scaling.
- IAM roles and policies per cluster.
- Private VPC endpoints for ECR, S3, CloudWatch.

5.3 Module Structure

```
infrastructure/
 main.tf
 provider.tf
 variables.tf
 outputs.tf
 terraform-dev.tfvars
 terraform-prod.tfvars

 modules/
     network/
     computes/
```

`eks-addons/`

5.4 EKS Configuration

Key Configuration Parameters

Variable	Description	Default
`region`	AWS region for deployment	`us-east-1`
`environment`	Environment name (dev/prod)	`dev`
`cidr_block`	VPC CIDR block	Required
`az`	List of availability zones	Required
`min_node_count`	Minimum nodes per cluster	`1`
`max_node_count`	Maximum nodes per cluster	`3`
`desired_node_count`	Desired nodes per cluster	`1`

Figure 14: EKS Cluster Configuration

- Kubernetes version: 1.29 (configurable).
- Managed node groups with scaling.
- Full control-plane logging enabled (API, audit, scheduler, controller-manager).
- Load Balancer Controller (ALB/NLB ingress) deployed via Helm.

6 Deployment Process

1. Initialize Terraform: `terraform init`
2. Review plan: `terraform plan -var-file=terraform-dev.tfvars`
3. Apply configuration: `terraform apply -var-file=terraform-dev.tfvars`
4. Configure kubectl for each cell:
    - `aws eks update-kubeconfig --name dev-cluster-dev-cell1-cell1`

- `aws eks update-kubeconfig --name dev-cluster-dev-cell2-cell2`
- `aws eks update-kubeconfig --name dev-cluster-dev-cell3-cell3`

5. Deploy NLB service using `nlb-service.yaml`.

7 Project Summary

This project demonstrates expertise in:

- Multi-stage Docker builds for full-stack apps.
- Secure and automated CI/CD pipelines.
- Cloud-native Kubernetes deployment and EKS management.
- Infrastructure as Code (Terraform) for network and compute.
- Observability via Prometheus, Grafana, and Alertmanager.
- Metrics-based AIOps anomaly detection.
- Scalable, fault-isolated multi-cell architecture.
- Best practices for container security, monitoring, and alerting.