**Course:** Dev Ops

**Instructor:** Khwaja Abdul Hafeez

**Class:** 98711

**Name:** Muhammad Maaz Arsalan Batla

**ERP:** 22794

**Project: AWS Infrastructure Deployment with Terraform, Docker, and BI Integration**

Table of Contents

[Introduction 1](#_Toc200153140)

[Project Architecture Overview 2](#_Toc200153141)

[Terraform Code Structure 2](#_Toc200153142)

[EC2 Auto Scaling Setup 3](#_Toc200153143)

[RDS Instances 3](#_Toc200153144)

[Application Load Balancer (ALB) 3](#_Toc200153145)

[Dockerized Application Deployment 3](#_Toc200153146)

[Domain and SSL Setup 3](#_Toc200153147)

[Database Access and Dummy Data 3](#_Toc200153148)

[BI Tool Deployment (Redash/Metabase) 3](#_Toc200153149)

[Loom Demonstration Video 3](#_Toc200153150)

[GitHub Repository Link 3](#_Toc200153151)

[Conclusion 3](#_Toc200153152)

# Introduction

The goal of this project was to design and deploy a **scalable**, **secure**, and **containerized** infrastructure on **AWS** using **Terraform** as Infrastructure as Code (IaC). The architecture consists of Auto Scaling EC2 instances running Dockerized Node.js applications behind a secure Load Balancer, private RDS databases, and a containerized Business Intelligence (BI) tool (Metabase) for live database visualization. This project simulates a production-grade deployment and demonstrates key DevOps principles including automation, modularity, high availability, and monitoring.

# Project Architecture Overview

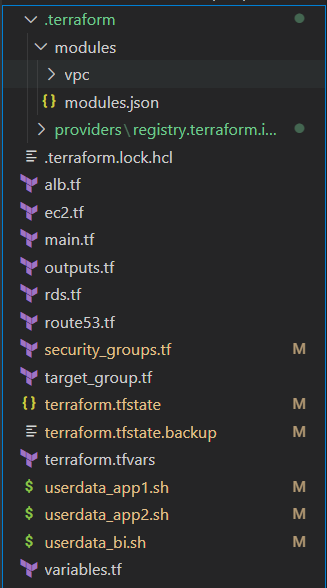
At a high level, the infrastructure consists of:

* A **VPC** with both public and private subnets across multiple Availability Zones.
* An **Auto Scaling Group** of EC2 instances, each bootstrapped with Docker, Node.js 20, and Nginx.
* An **Application Load Balancer (ALB)** forwarding HTTP and HTTPS traffic to the EC2 instances.
* Two **RDS databases** (MySQL and PostgreSQL) deployed in private subnets, inaccessible publicly.
* A **Dockerized BI tool** (Metabase or Redash) deployed on a separate EC2 instance to visualize database updates.
* **Route 53 and ACM** used to associate a custom domain and SSL certificates.
* **SSH tunneling** used to securely access RDS instances for DB client interactions.

# Terraform Code Structure

The infrastructure was provisioned using [Terraform](https://www.terraform.io/) following best practices of modular and reusable code. The directory is organized with individual .tf files for each major component of the infrastructure, making it easy to manage and scale. Here's the breakdown:

* main.tf: Initializes the providers and general configurations.
* ec2.tf: Defines the EC2 launch template and Auto Scaling Group for web and application instances.
* rds.tf: Provisions two RDS instances — one for MySQL and one for PostgreSQL — in private subnets.
* alb.tf: Creates the Application Load Balancer and listener rules.
* target\_group.tf: Sets up target groups for the ALB to forward traffic to EC2 instances.
* security\_groups.tf: Defines security groups for EC2, RDS, and Load Balancer.
* route53.tf: Handles domain name mapping using AWS Route53.
* outputs.tf: Outputs key resources like ALB DNS and RDS endpoints.
* variables.tf: Declares all the variables used across modules.
* terraform.tfvars: Provides values for declared variables.
* .terraform/: Contains the lock file and downloaded provider modules.
* modules/vpc/: Custom VPC module (if used).
* userdata\_app1.sh, userdata\_app2.sh, userdata\_bi.sh: Scripts used as EC2 user data to install necessary software (Nginx, Docker, Node.js, and BI Tool).



# EC2 Auto Scaling Setup

The EC2 Auto Scaling setup ensures high availability and scalability of the application. Here's how it was implemented:

* **Launch Template**:
  + AMI: Amazon Linux 2.
  + Instance Type: e.g., t3.medium.
  + User Data scripts:
    - userdata\_app1.sh and userdata\_app2.sh install:
      * **Nginx** (as a reverse proxy),
      * **Docker** (for containerization),
      * **Node.js 20** (for backend services).
* **Auto Scaling Group**:
  + Attached to multiple subnets across different availability zones.
  + Minimum instances: 2, Desired: 3.
  + Associated with target group to register instances automatically.

# RDS Instances

Two RDS databases were provisioned:

* **MySQL RDS**
* **PostgreSQL RDS**

Key configurations:

* **Subnet Groups**: RDS instances are deployed in private subnets for enhanced security.
* **No Public IPs**: Ensures that databases are not exposed publicly.
* **Security Groups**:
  + Inbound access restricted only to EC2 instances via their security group.
  + Outbound access is open for updates and dependencies.
* **Initialization**: After provisioning, dummy data was inserted via a secure SSH tunnel from EC2 using a client like DBeaver.

# Application Load Balancer (ALB)

An **Application Load Balancer (ALB)** was set up to distribute traffic across the EC2 instances running the containerized application.

* **Listener Configuration**:
  + HTTP (Port 80) and HTTPS (Port 443) enabled.
  + Redirects HTTP to HTTPS to enforce encryption.
* **Target Groups**:
  + Registered with EC2 Auto Scaling Group.
  + Health checks configured on container app ports (i.e., 3000).
* **Security Group**:
  + Only allows inbound traffic on ports 80 and 443.
  + Outbound traffic is unrestricted for ALB to communicate with targets.

# Dockerized Application Deployment

# Domain and SSL Setup

# Database Access and Dummy Data

# BI Tool Deployment (Metabase)

# Loom Demonstration Video

# GitHub Repository Link

All code, including Terraform configurations and Docker deployment scripts, has been organized in a public GitHub repository. This includes:

* Modular Terraform files: main.tf, ec2.tf, rds.tf, alb.tf, route53.tf, etc.
* Supporting shell scripts for Dockerized deployments.
* .tfvars file for variable abstraction and reuse.
* README for setup instructions and architectural overview.

🔗

This repository demonstrates infrastructure as code (IaC), clean modularization, and production-readiness for scalable DevOps pipelines.

# Conclusion

This project was a comprehensive and hands-on implementation of a modern, scalable, and secure cloud architecture using DevOps best practices. Through this exercise, I accomplished the following:

* **Successfully provisioned AWS infrastructure using Terraform**, enabling repeatable and version-controlled deployments.
* **Automated EC2 instance configuration** with user data scripts, ensuring containers are deployed immediately upon instance launch.
* **Ensured security and scalability** with a load-balanced architecture, secure RDS access, and encrypted web traffic via SSL.
* **Implemented observability** using Metabase, showcasing how BI tools integrate with real-time cloud databases for actionable insights.

VPC:

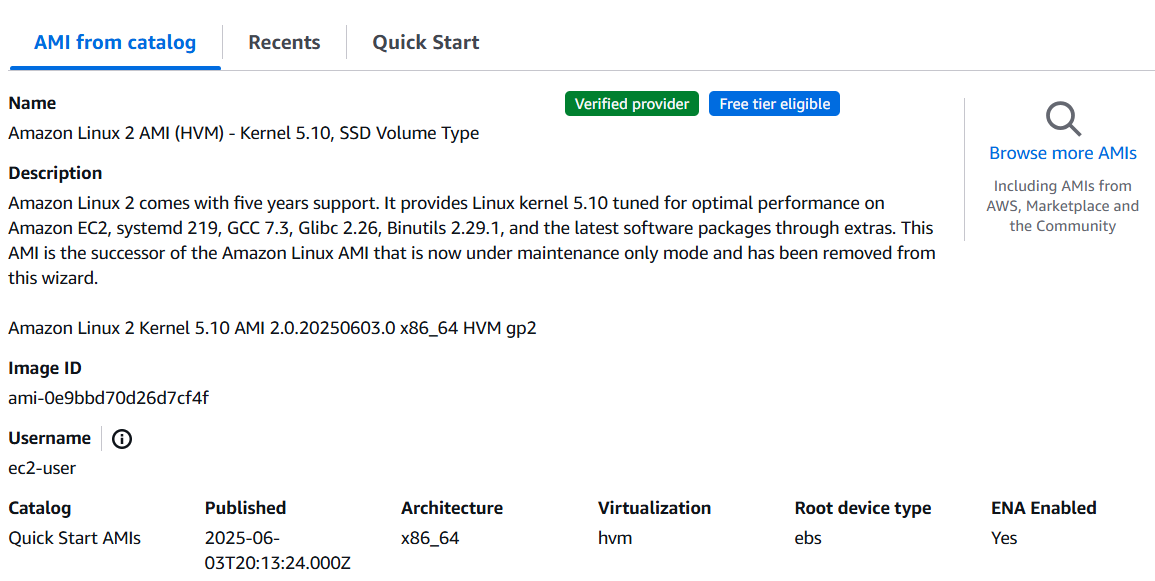
A screenshot of a computer

AI-generated content may be incorrect.

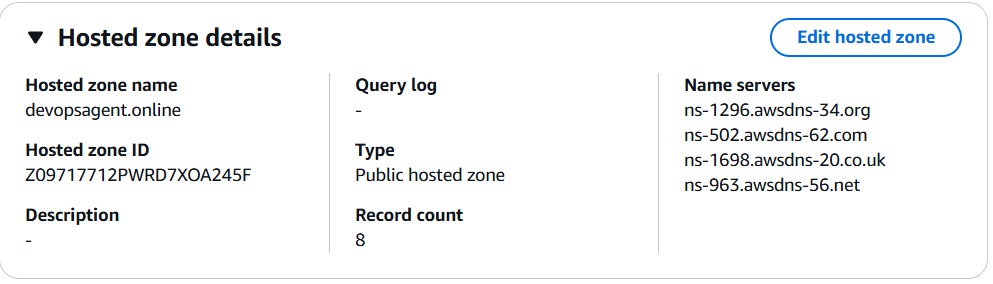
Available Subnets:

A screenshot of a computer

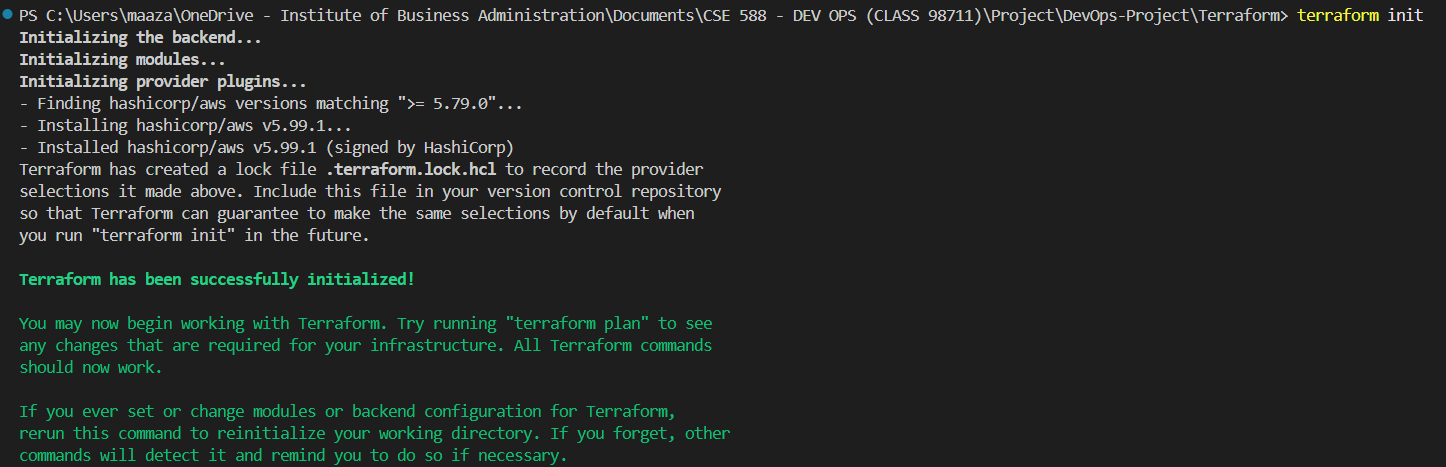
AI-generated content may be incorrect.

EC 2:

Route53:



Running the command “terraform init”:

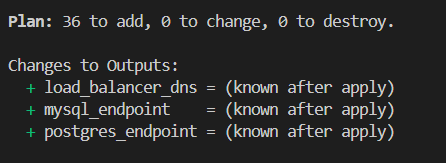


Running the command “terraform validate”:

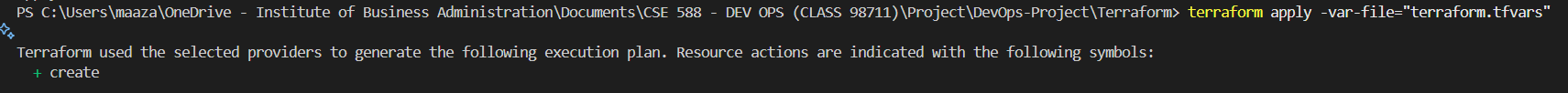


Running the command "terraform plan -var-file=”terraform.tfvars”:





Running the command "terraform apply -var-file= “terraform.tfvars”:

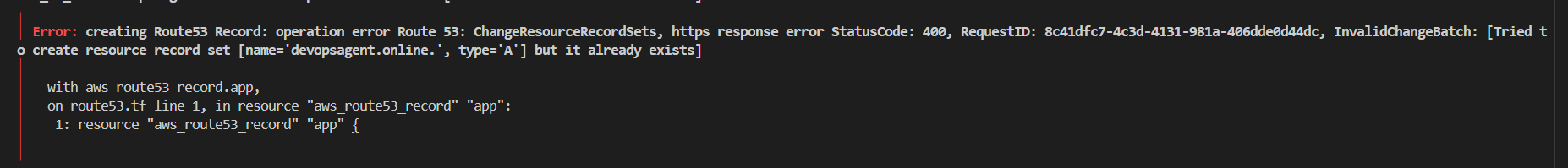


Entering “yes” to confirm execution:

A computer screen shot of a program

AI-generated content may be incorrect.

However, after answering “yes”, I got the following error:



Upon inspection, I realized that this error was caused by the Route53 record already existing and being in possession of my group mate, therefore, I used the below “terraform import” command to import the record into my infrastructure:

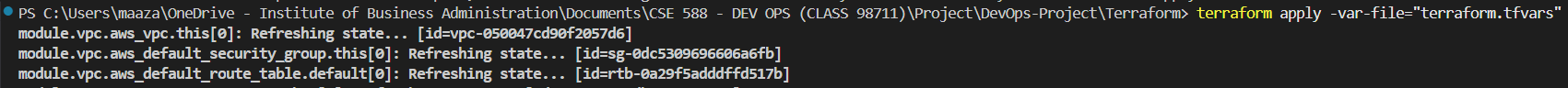


The record was successfully imported:

A computer screen with green text

AI-generated content may be incorrect.

Running “terraform apply -var-file= “terraform.tfvars” command again:

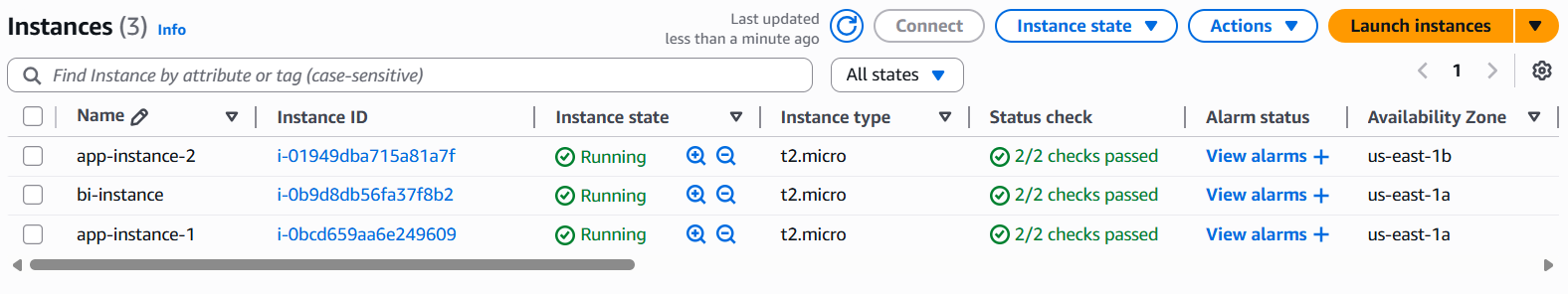


Resources created successfully:

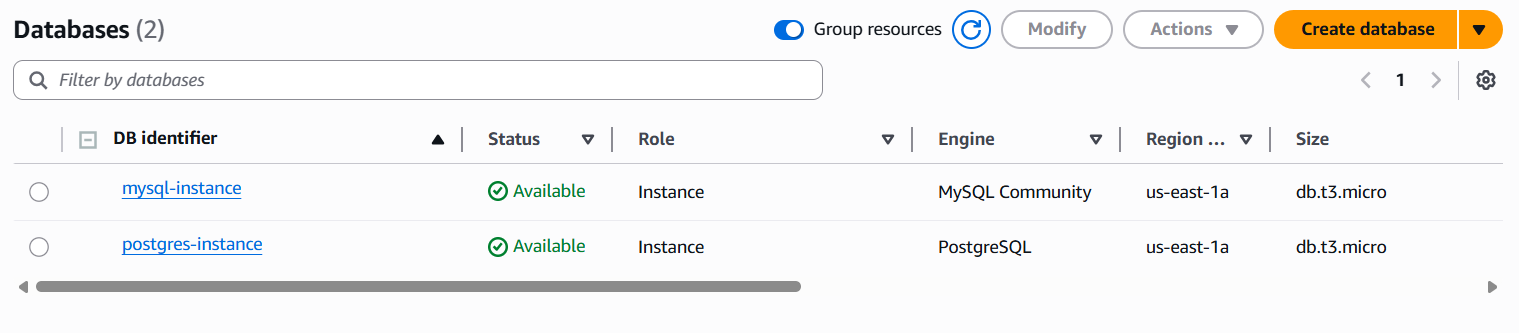
A screen shot of a computer

AI-generated content may be incorrect.

Instances Running Successfully:



Databases Setup Successfully:



ACM Certificate Issued Successfully:

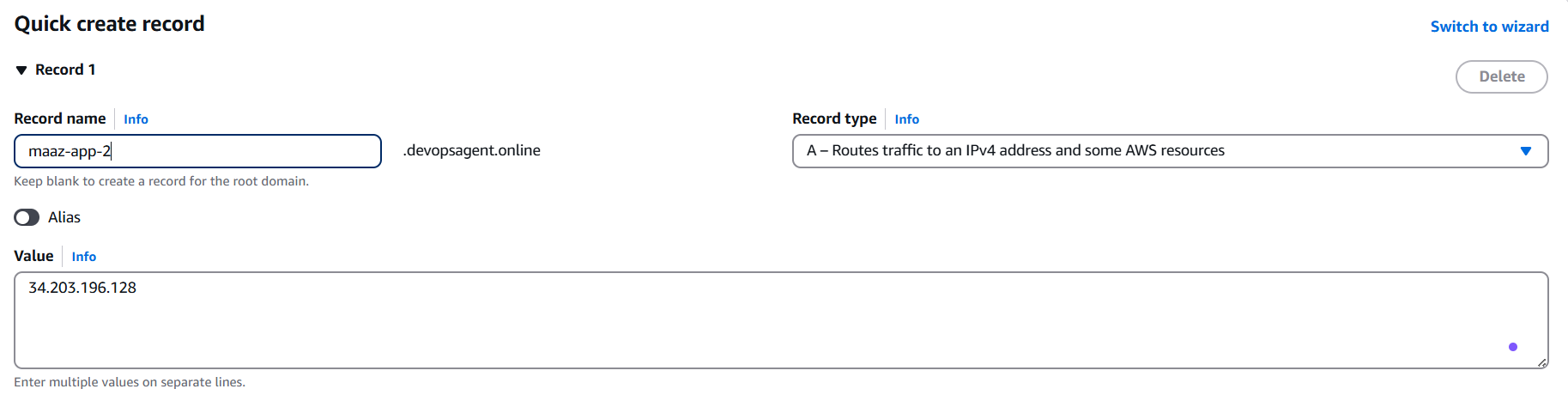
A screenshot of a computer

AI-generated content may be incorrect.

Creating Route53 Records:

A close-up of a computer screen

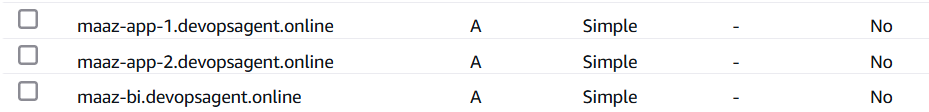
AI-generated content may be incorrect.

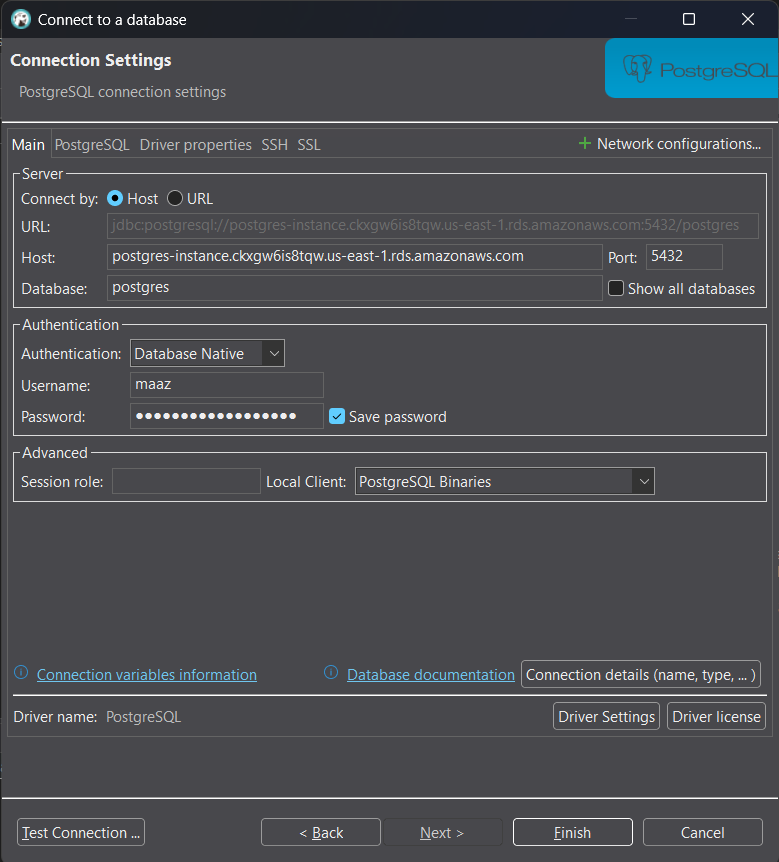


A computer screen shot of a computer

AI-generated content may be incorrect.

Route53 Records Added Successfully:





A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer error

AI-generated content may be incorrect.