**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 2](#_Toc200925297)

[Project Architecture Overview 2](#_Toc200925298)

[Terraform Code Structure 2](#_Toc200925299)

[EC2 Auto Scaling Setup 4](#_Toc200925300)

[RDS Instances 4](#_Toc200925301)

[Application Load Balancer (ALB) 5](#_Toc200925302)

[Dockerized Application Deployment 5](#_Toc200925303)

[Domain and SSL Setup 6](#_Toc200925304)

[Database Access and Dummy Data 6](#_Toc200925305)

[BI Tool Deployment (Metabase) 7](#_Toc200925306)

[Loom Demonstration Video 7](#_Toc200925307)

[GitHub Repository Link 8](#_Toc200925308)

[Conclusion 8](#_Toc200925309)

[Appendix A: Screenshots 9](#_Toc200925310)

# 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

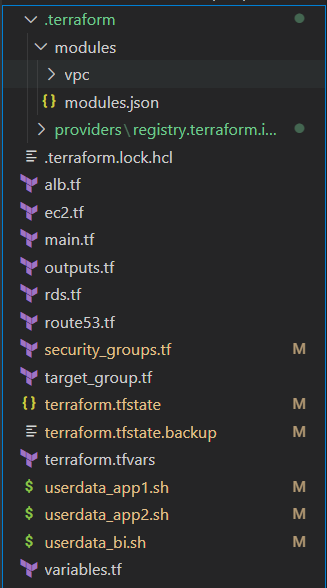
At a high level, the infrastructure consists of:

* 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 tunnelling** 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: t2.micro.
  + 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

For deploying the multi-stage Dockerized application, I used the sample [GitHub repository](https://github.com/Khhafeez47/reactapp.git) provided in the project requirements instead of building a custom frontend and backend from scratch. This approach allowed me to focus on infrastructure automation and deployment processes rather than app development. The core application was deployed as Docker containers on EC2 instances provisioned via Auto Scaling. The setup ensures isolated environments, easier CI/CD integration, and faster deployment times.

* **Dockerfile**: A custom Dockerfile was created for the cloned application code.
  + Backend: Built with **Node.js 20**, exposed on port 3000.
  + Frontend: Built with **React**, served via **Nginx**, exposed on port 80.
* **User Data Scripts** (userdata\_app1.sh, userdata\_app2.sh):
  + Install Docker and Docker Compose.
  + Clone the application from GitHub.
  + Build and run containers automatically on instance launch.
* **GitHub Integration**:
  + Codebase and Dockerfile were hosted on a private GitHub repo.
  + Scripts pull the latest commit on boot, supporting automatic updates.
* **Container Logs & Health**:
  + Application logs were verified via docker logs.
  + ALB health checks validated container health on ports.

# Domain and SSL Setup

To ensure secure access, I used **Route 53** for DNS and **AWS ACM (Certificate Manager)** for SSL certification.

* **Domain Mapping**:
  + A custom domain was purchased and mapped via Route 53 hosted zone.
  + An A-record (alias) points to the ALB DNS.
* **SSL Certificate**:
  + Provisioned via ACM in the same AWS region.
  + Bound to the ALB listener on port 443.
* **HTTP → HTTPS Redirection**:
  + Configured at the ALB level to enforce secure access.
* **Result**: Users accessing the domain are automatically redirected to HTTPS and served securely from the EC2-backed app behind ALB.

# Database Access and Dummy Data

For development and testing, secure access to the RDS databases (MySQL and PostgreSQL) was necessary — without exposing them publicly.

* **SSH Tunnelling**:
  + A bastion EC2 instance (with public IP) was used to create an SSH tunnel.
  + Local port forwarding enabled secure database connections from tools like **DBeaver**.
* **Clients & Tools**:
  + DBeaver was configured with tunnel settings to connect securely.
  + Inserted dummy records into both RDS instances (products, users, metrics).
* **Why It Matters**:
  + Demonstrated secure admin access without exposing database endpoints.
  + Ensured that BI tool and applications could read data from a realistic dataset.

# BI Tool Deployment (Metabase)

A business intelligence tool was deployed via Docker to visualize live data from the RDS instance.

* **Tool Used**: **Metabase**
* **Deployment**:
  + Dockerized version pulled in userdata\_bi.sh.
  + Exposed on port 3000, reverse proxied via Nginx or ALB path-based routing.
* **Database Connection**:
  + Metabase connected securely to the PostgreSQL RDS instance using internal VPC endpoint.
  + Verified by running SQL queries within the Metabase UI.
* **Dashboards**:
  + Built a live dashboard showing:
    - Number of records,
    - Aggregated values (sales/users),
    - Timestamped inserts for live updates.
* **Live Refresh**:
  + Enabled auto-refresh to show real-time data updates from the database.

# Loom Demonstration Video

To provide a complete walkthrough of the infrastructure and deployment, I recorded a demonstration using Loom. This video showcases each component in action and reflects the operational flow of the entire system:

🔗<https://www.loom.com/share/63b17082838e4f0baee3e0ac6fc1fb6a?sid=b424a260-7340-4711-9850-d4106ffafeb3>

**The demo includes:**

* Terraform provisioning of the VPC, subnets, EC2 Auto Scaling Group, RDS instances, and ALB.
* Dockerized application deployment via EC2 user data scripts.
* Route 53 domain mapping and ACM-based SSL setup.
* SSH tunneling to access MySQL/PostgreSQL RDS securely using DBeaver.
* Real-time Metabase dashboard connected to the RDS instance, reflecting dummy data and live updates.

# 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.

🔗 <https://github.com/MaazBatla/DevOps-Project.git>

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.

# Appendix A: Screenshots

The following screenshots along with their description represent the flow of events and the actions I took in sequential order for this project:

VPC:

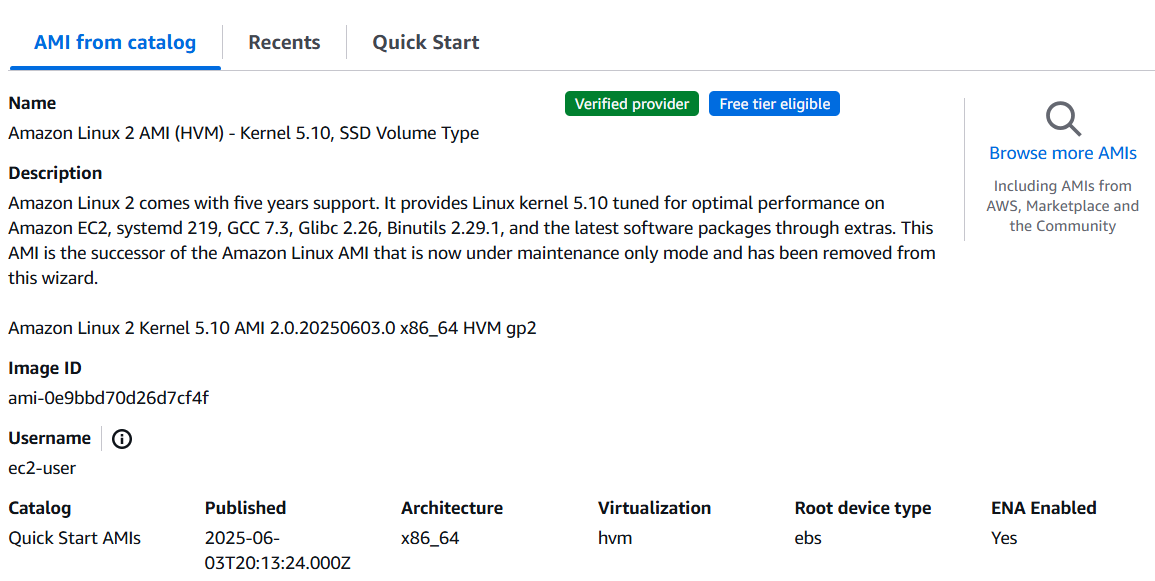
A screenshot of a computer

AI-generated content may be incorrect.

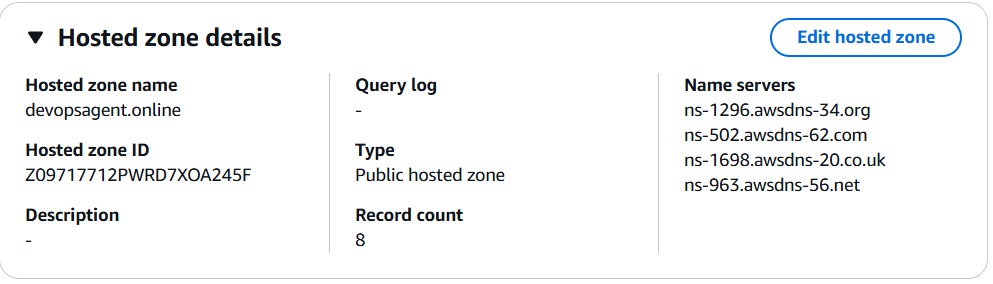
Available Subnets:

A screenshot of a computer

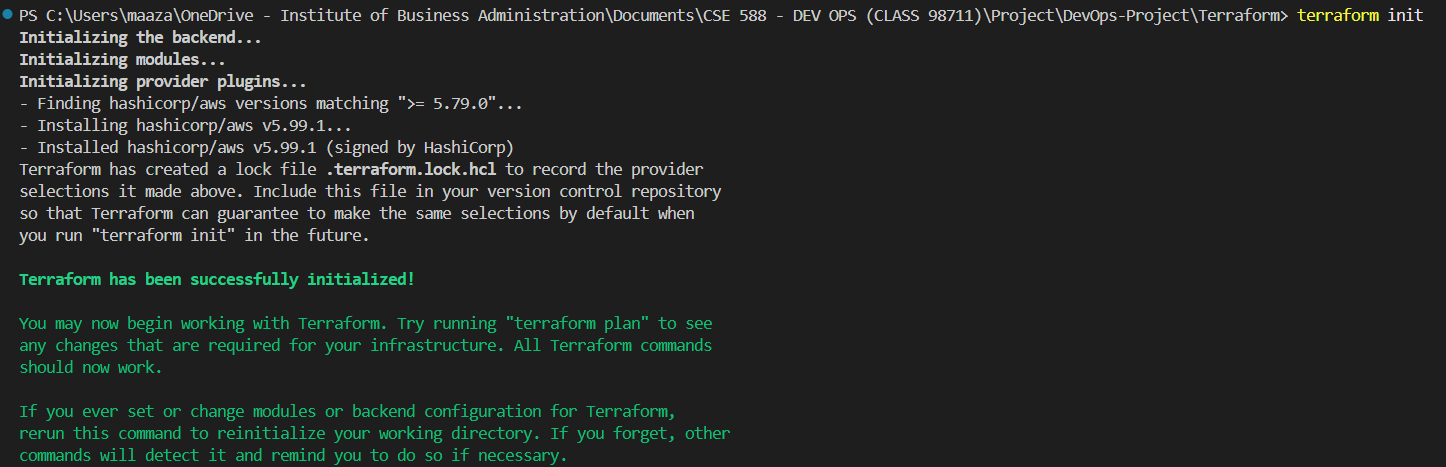
AI-generated content may be incorrect.

Extracting the AMI for Amazon Linux 2 Machines to configure the EC 2 instances:

Route53:



Running the command “terraform init”:

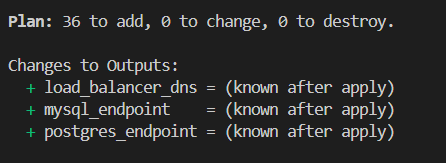


Running the command “terraform validate”:

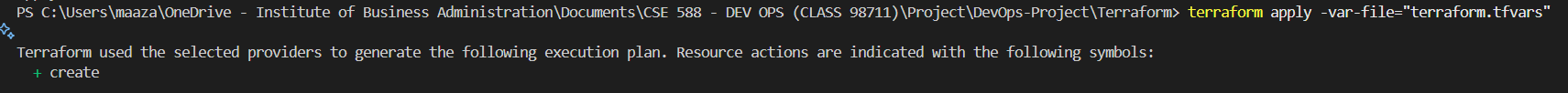


Running the command "terraform plan -var-file=”terraform.tfvars”:





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

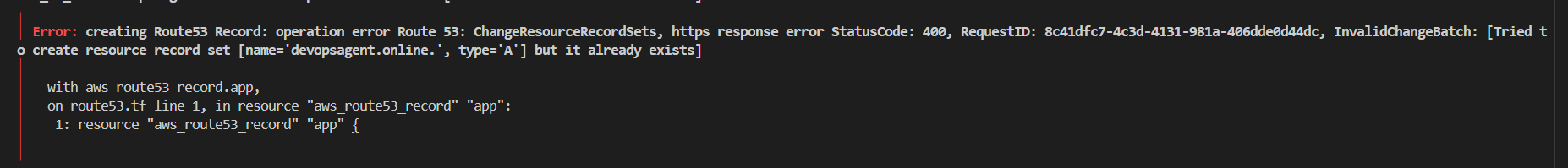


Entering “yes” to confirm execution:

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:

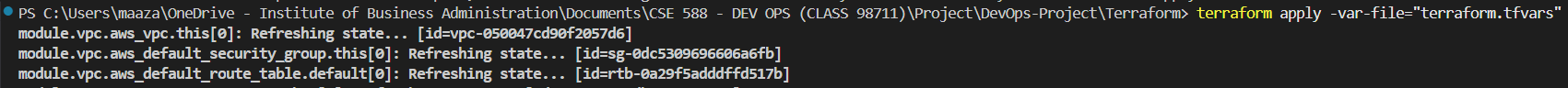


The record was successfully imported:

A computer screen with green text

AI-generated content may be incorrect.

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

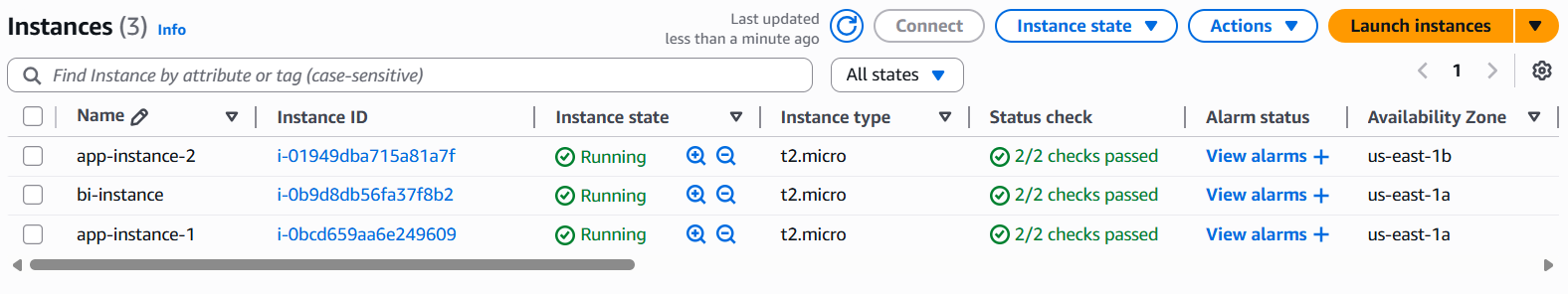


Resources created successfully:

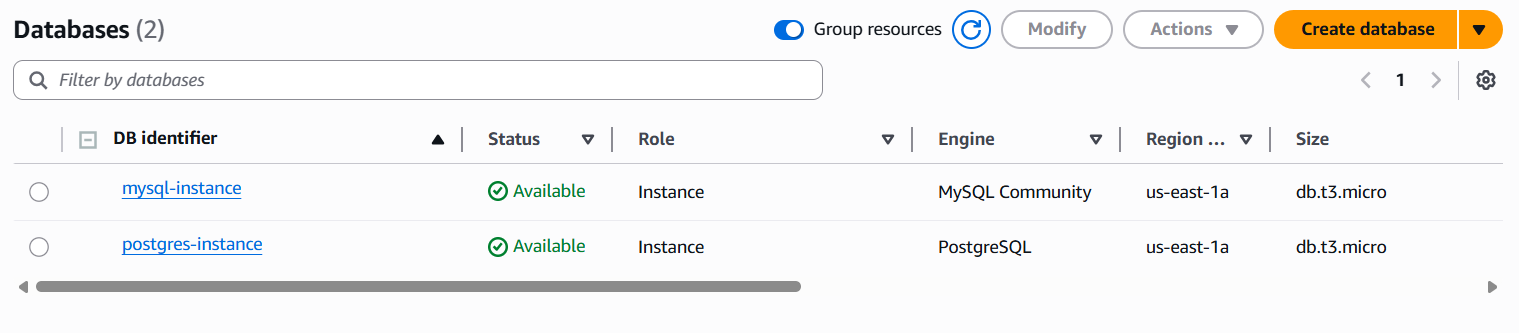
A screen shot of a computer

AI-generated content may be incorrect.

EC2 Instances Running Successfully:



Databases Setup Successfully:



ACM Certificate Issued Successfully:

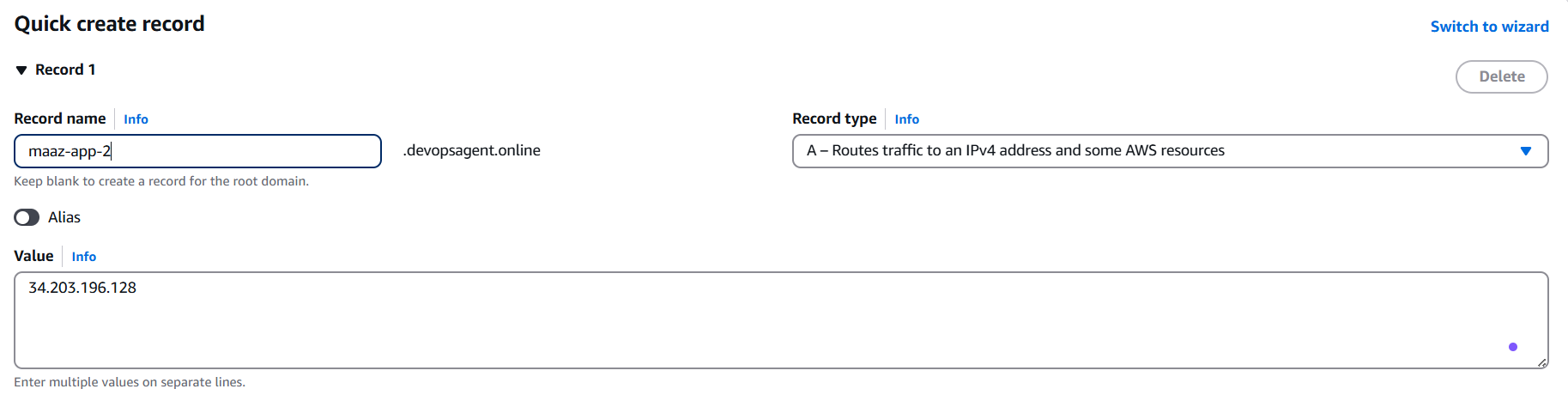
A screenshot of a computer

AI-generated content may be incorrect.

Creating Route53 Records:

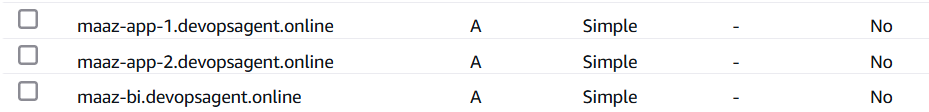
A close-up of a computer screen

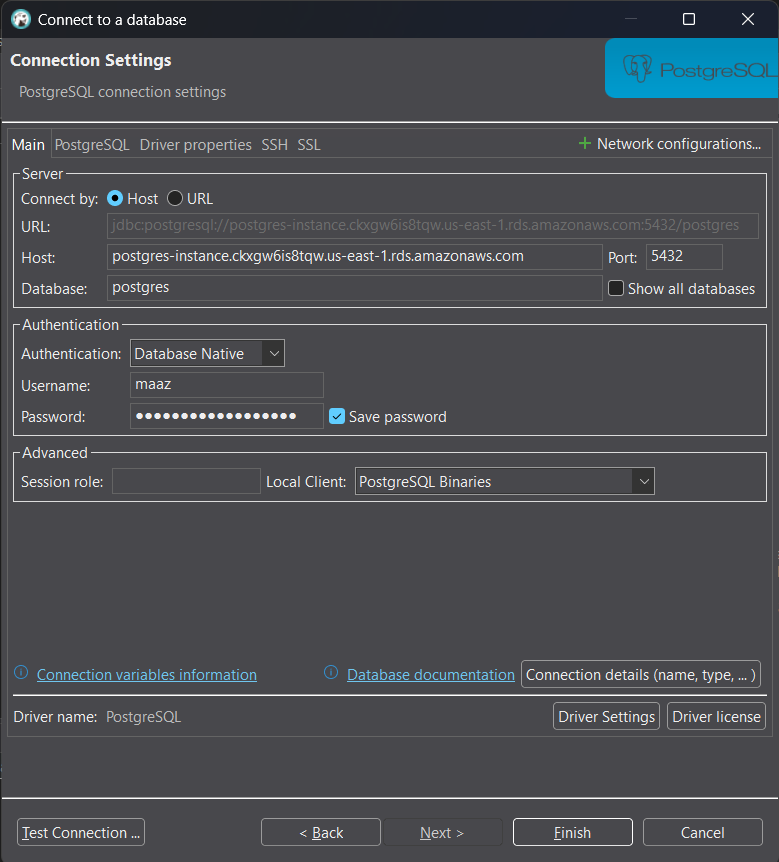
AI-generated content may be incorrect.



A computer screen shot of a computer

AI-generated content may be incorrect.Route53 Records Added Successfully:

Connecting to the created PostgreSQL instance via DBeaver:



Setting up SSH Tunnelling:

A screenshot of a computer

AI-generated content may be incorrect.

SSH Tunnel Connection Successful:

A screenshot of a computer

AI-generated content may be incorrect.

Connected to PostgreSQL instance Successfully:

A screenshot of a computer error

AI-generated content may be incorrect.

Creating a table in the database:

A screenshot of a computer

AI-generated content may be incorrect.

Initial Data Injection:

A screen shot of a computer

AI-generated content may be incorrect.

Setting Up Metabase:

A screenshot of a computer

AI-generated content may be incorrect.

Creating a Dashboard:

A screenshot of a computer

AI-generated content may be incorrect.

Dashboard created Successfully with Initial data:

A screenshot of a computer

AI-generated content may be incorrect.

Using the following script for bulk data injection:

A screenshot of a computer program

AI-generated content may be incorrect.

After bulk data injection:

A screenshot of a computer

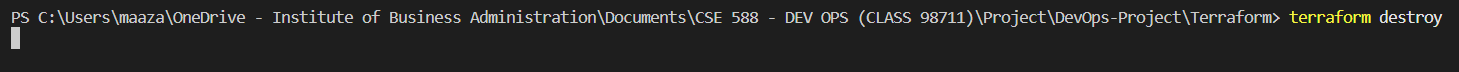
AI-generated content may be incorrect.

Deleting the PostgreSQL connection:

A screen shot of a computer

AI-generated content may be incorrect.

Running “terraform destroy” command:



A screenshot of a computer program

AI-generated content may be incorrect.

All Resources Destroyed Successfully:

