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

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:

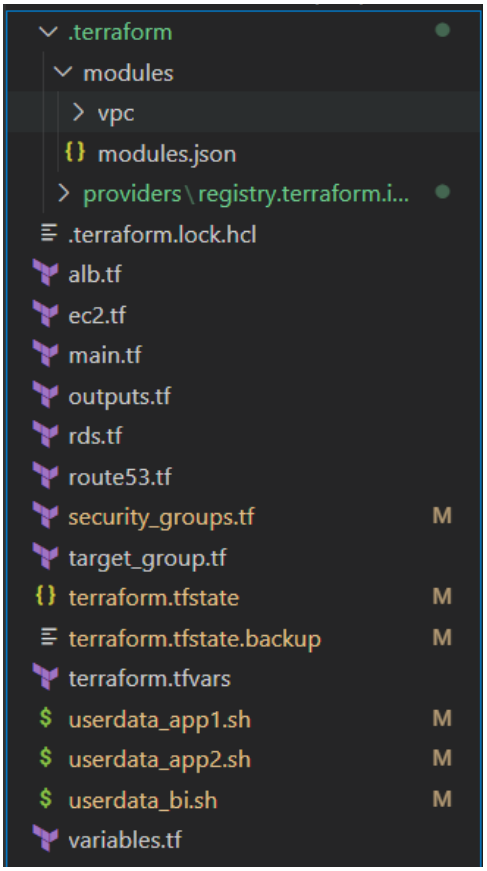
- A **VPC** with both public and private subnets across multiple Availability Zones.
- An **Auto Scaling Group** of EC2 instances, each bootstrapped with Docker, Node.js 20, and Nginx.
- An **Application Load Balancer (ALB)** forwarding HTTP and HTTPS traffic to the EC2 instances.
- Two **RDS databases** (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 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 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

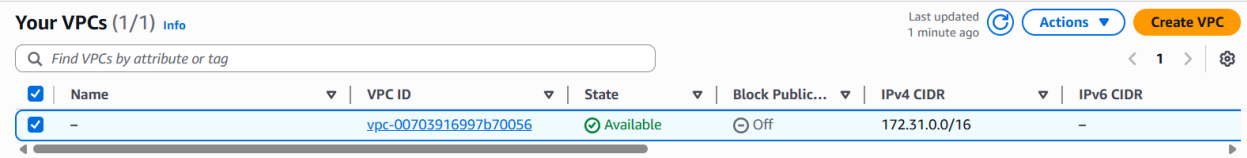
This project was a comprehensive and hands-on implementation of a modern, scalable, and secure cloud architecture using DevOps best practices. Through this exercise, I accomplished the following:

- **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.

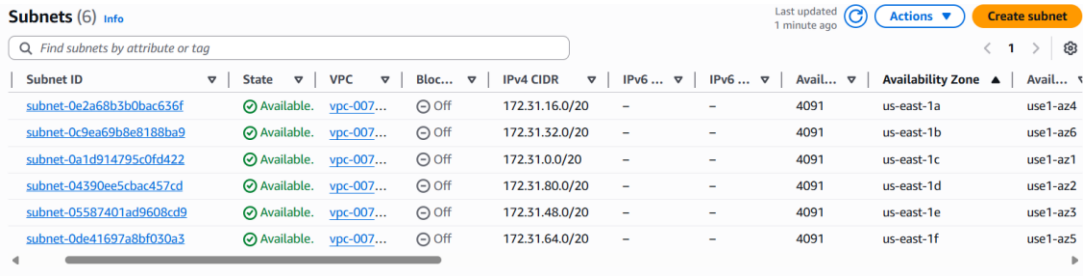
Appendix A: Screenshots

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

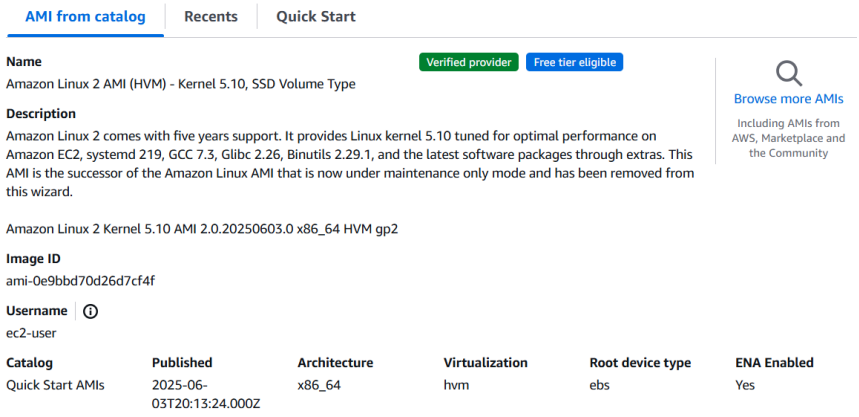
VPC:



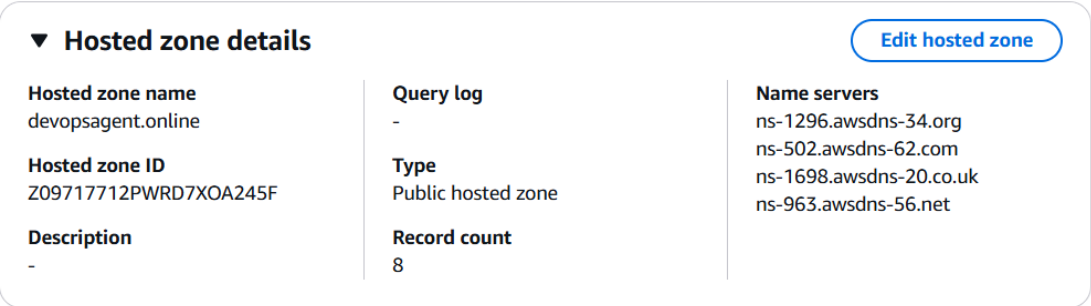
Available Subnets:



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



Route53:



Running the command “terraform init”:

```
PS C:\Users\maaza\OneDrive - Institute of Business Administration\Documents\CSE 588 - DEV OPS (CLASS 98711)\Project\DevOps-Project\Terraform> terraform init
Initializing the backend...
Initializing modules...
Initializing provider plugins...
- Finding hashicorp/aws versions matching ">= 5.79.0"...
- Installing hashicorp/aws v5.99.1...
- Installed hashicorp/aws v5.99.1 (signed by HashiCorp)
Terraform has created a lock file .terraform.lock.hcl to record the provider
selections it made above. Include this file in your version control repository
so that Terraform can guarantee to make the same selections by default when
you run "terraform init" in the future.

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
```

Running the command “terraform validate”:

```
PS C:\Users\maaza\OneDrive - Institute of Business Administration\Documents\CSE 588 - DEV OPS (CLASS 98711)\Project\DevOps-Project\Terraform> terraform validate
Success! The configuration is valid.
```

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

```
PS C:\Users\maaza\OneDrive - Institute of Business Administration\Documents\CSE 588 - DEV OPS (CLASS 98711)\Project\DevOps-Project\Terraform> terraform plan -var-file="terraform.tfvars"
```

```
Plan: 36 to add, 0 to change, 0 to destroy.

Changes to Outputs:
  + load_balancer_dns = (known after apply)
  + mysql_endpoint    = (known after apply)
  + postgres_endpoint = (known after apply)
```

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

```
PS C:\Users\maaza\OneDrive - Institute of Business Administration\Documents\CSE 588 - DEV OPS (CLASS 98711)\Project\DevOps-Project\Terraform> terraform apply -var-file="terraform.tfvars"

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following symbols:
  + create
```

Entering “yes” to confirm execution:

```
Plan: 35 to add, 0 to change, 0 to destroy.

Changes to Outputs:
  + load_balancer_dns = (known after apply)
  + mysql_endpoint    = (known after apply)
  + postgres_endpoint = (known after apply)

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value: yes
```

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

```
Error: creating Route53 Record: operation error Route 53: ChangeResourceRecordSets, https response error StatusCode: 400, RequestID: 8c41dfc7-4c3d-4131-981a-406dde0d44dc, InvalidChangeBatch: [Tried t
o create resource record set [name='devopsagent.online.', type='A'] but it already exists]

  with aws_route53_record.app,
  on route53.tf line 1, in resource "aws_route53_record" "app":
   1: resource "aws_route53_record" "app" {
```

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:

```
PS C:\Users\maaza\OneDrive - Institute of Business Administration\Documents\CSE 588 - DEV OPS (CLASS 98711)\Project\DevOps-Project\Terraform> terraform import aws_route53_record.app Z09717712PWRD7XOA24
5F_devopsagent.online_A
```

The record was successfully imported:

```
aws_route53_record.app: Importing from ID "Z09717712PWRD7XOA245F_devopsagent.online_A"...
aws_route53_record.app: Import prepared!
  Prepared aws_route53_record for import
aws_route53_record.app: Refreshing state... [id=Z09717712PWRD7XOA245F_devopsagent.online_A]

Import successful!

The resources that were imported are shown above. These resources are now in
your Terraform state and will henceforth be managed by Terraform.
```

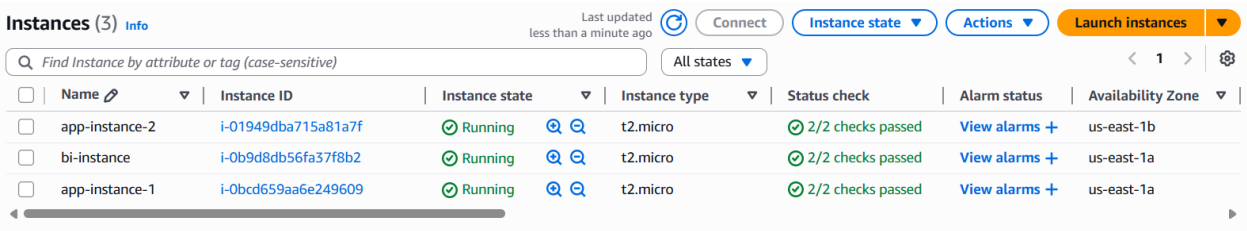
Running “terraform apply -var-file= “terraform.tfvars” command again:

```
PS C:\Users\maaza\OneDrive - Institute of Business Administration\Documents\CSE 588 - DEV OPS (CLASS 98711)\Project\DevOps-Project\Terraform> terraform apply -var-file="terraform.tfvars"
module.vpc.aws_vpc.this[0]: Refreshing state... [id=vpc-050047cd90f2057d6]
module.vpc.aws_default_security_group.this[0]: Refreshing state... [id=sg-0dc5309696606a6fb]
module.vpc.aws_default_route_table.default[0]: Refreshing state... [id=rtb-0a29f5adddffd517b]
```

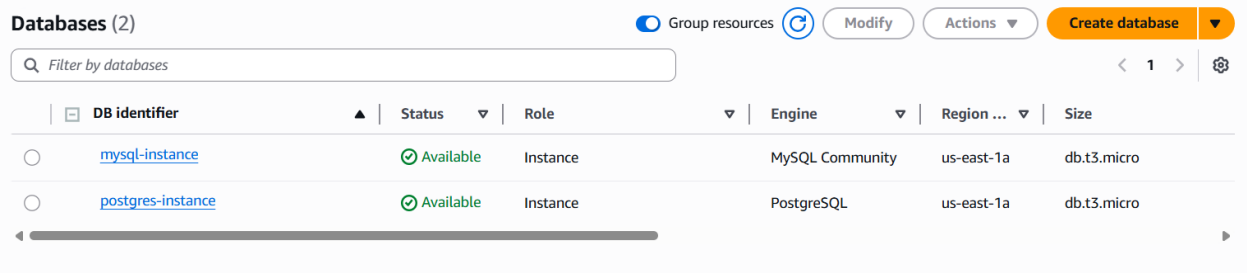
Resources created successfully:

```
Apply complete! Resources: 39 added, 0 changed, 0 destroyed.

Outputs:

load_balancer_dns = "app-lb-1275294627.us-east-1.elb.amazonaws.com"
mysql_endpoint = "mysql-instance.ckxgw6is8tqw.us-east-1.rds.amazonaws.com:3306"
postgres_endpoint = "postgres-instance.ckxgw6is8tqw.us-east-1.rds.amazonaws.com:5432"
```

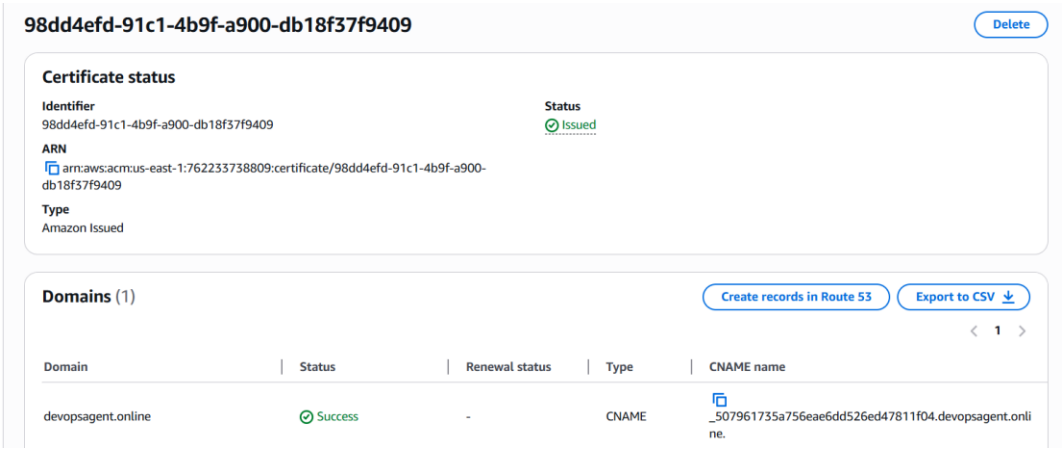
EC2 Instances Running Successfully:



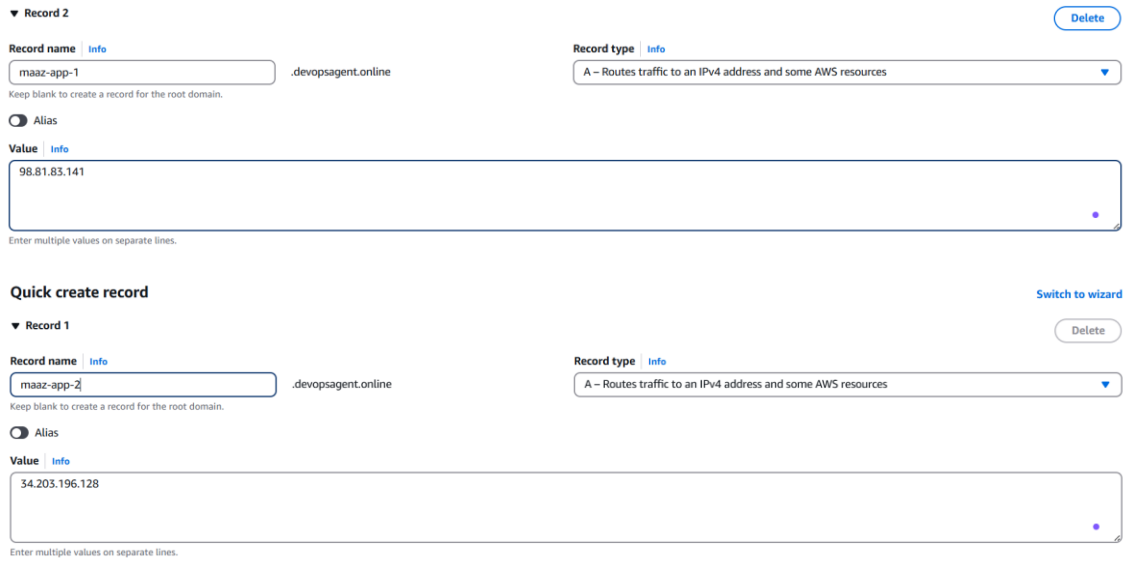
Databases Setup Successfully:

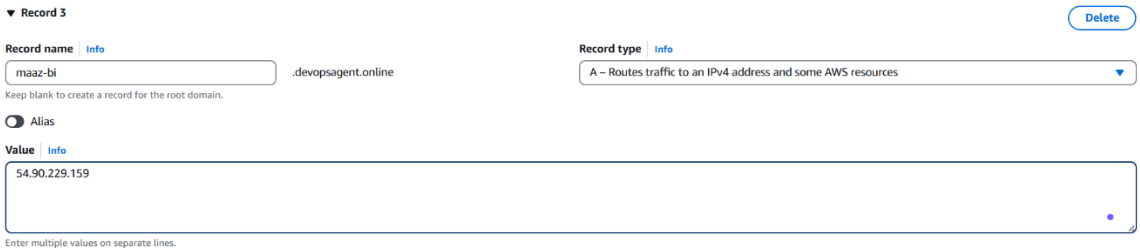


ACM Certificate Issued Successfully:



Creating Route53 Records:

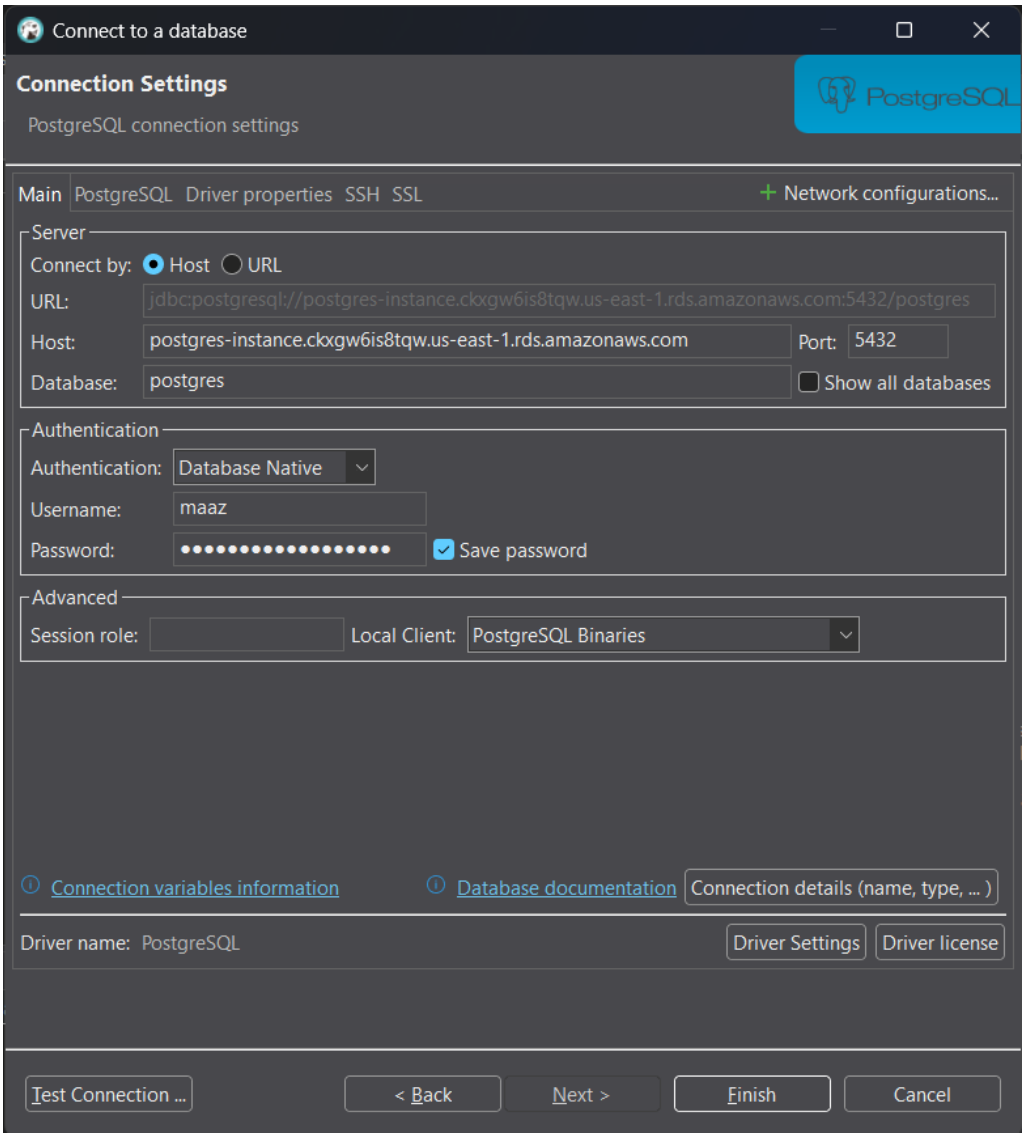




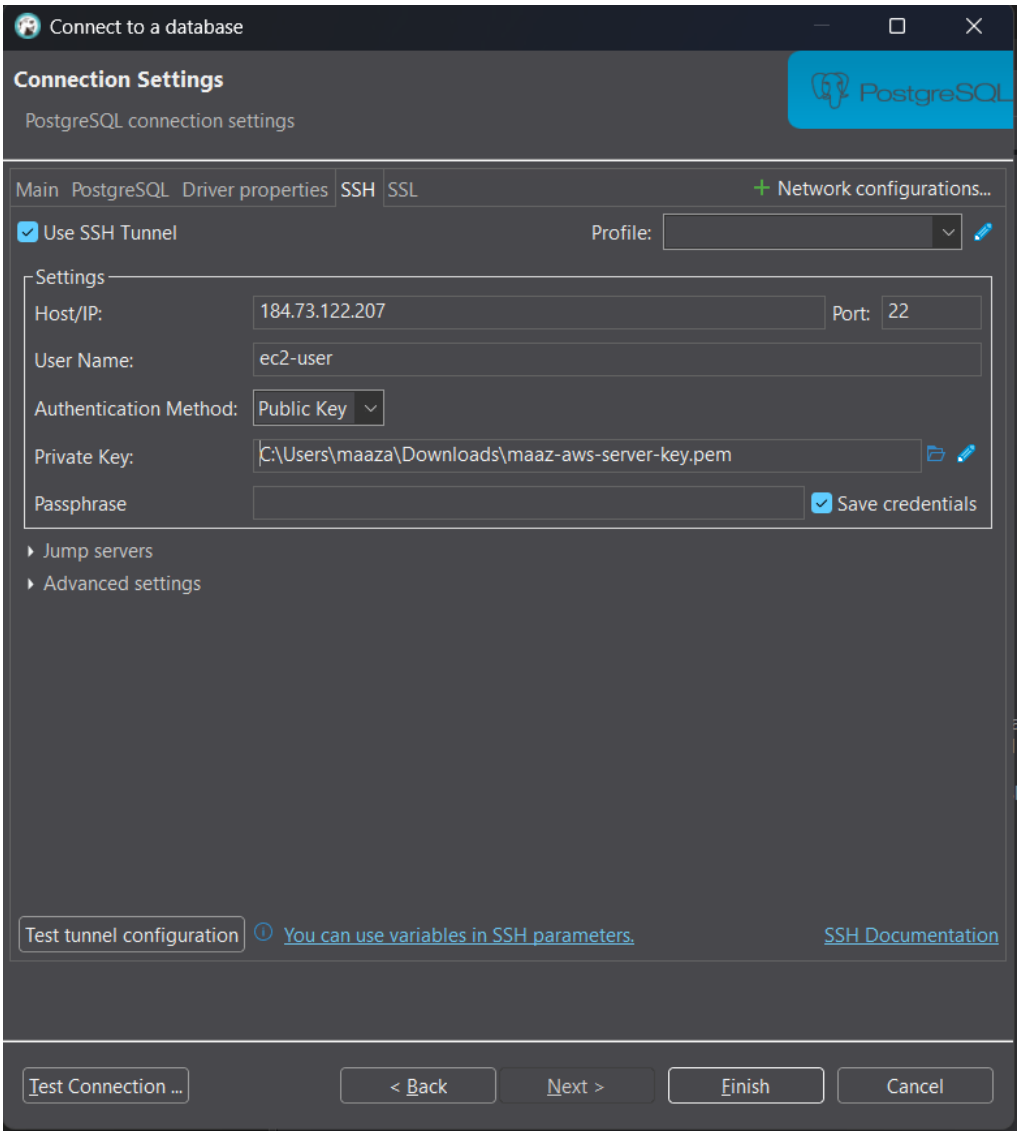
Route53 Records Added Successfully:

☐	maaz-app-1.devopsagent.online	A	Simple	-	No
☐	maaz-app-2.devopsagent.online	A	Simple	-	No
☐	maaz-bi.devopsagent.online	A	Simple	-	No

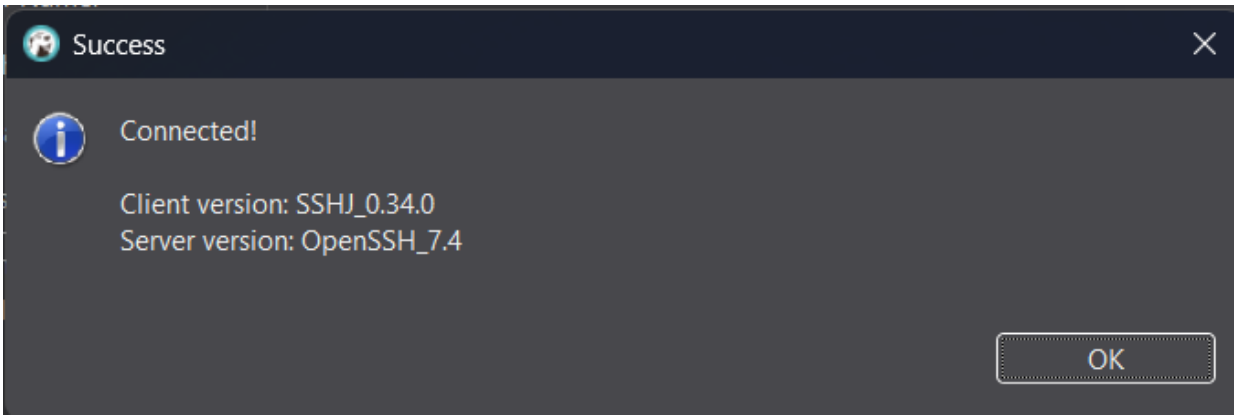
Connecting to the created PostgreSQL instance via DBeaver:



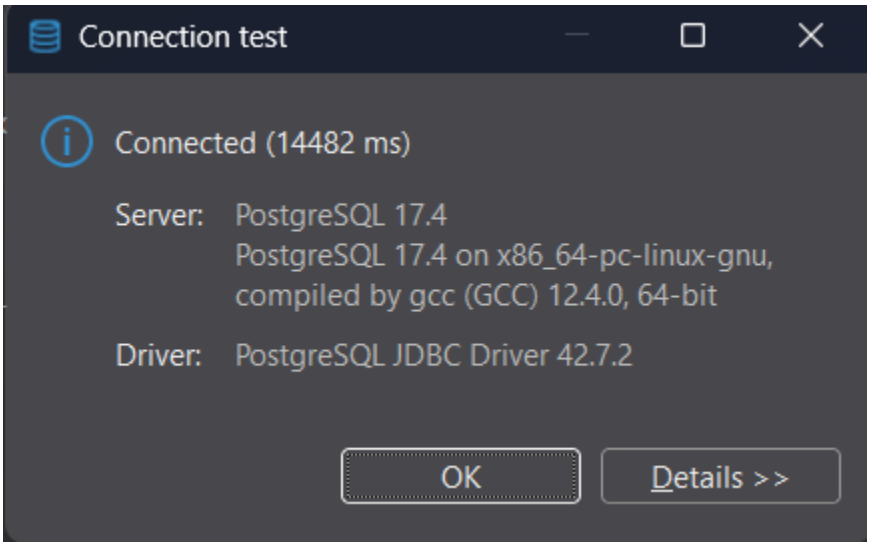
Setting up SSH Tunnelling:



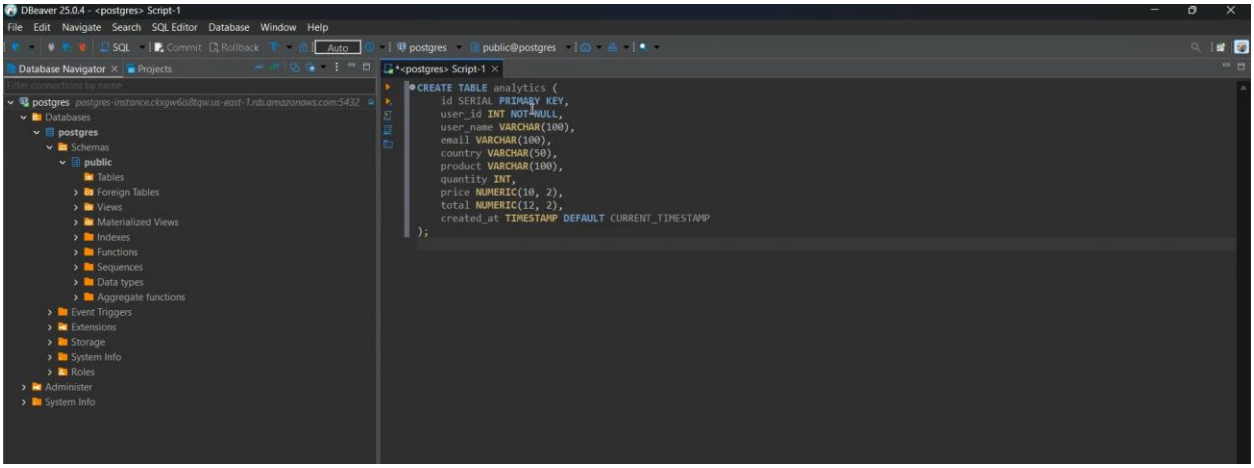
SSH Tunnel Connection Successful:



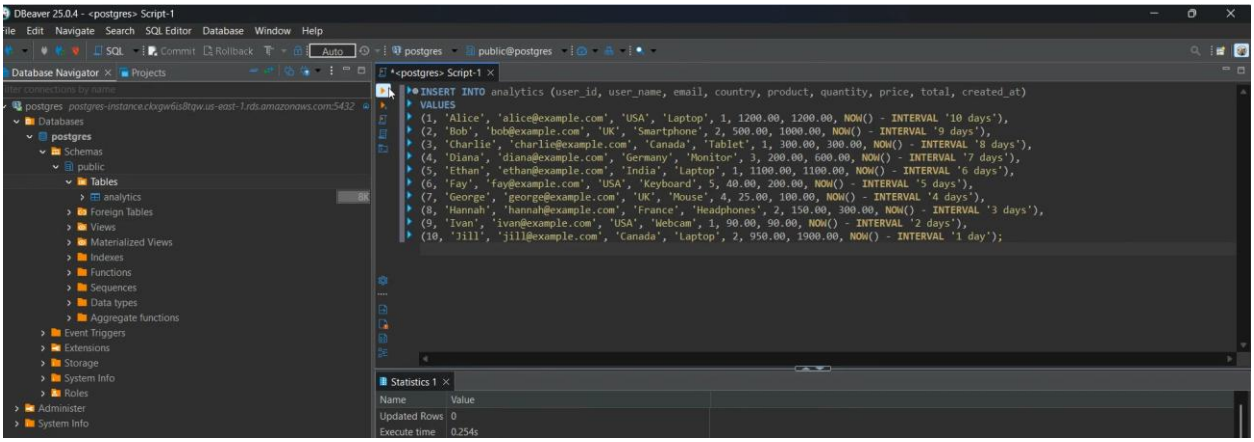
Connected to PostgreSQL instance Successfully:



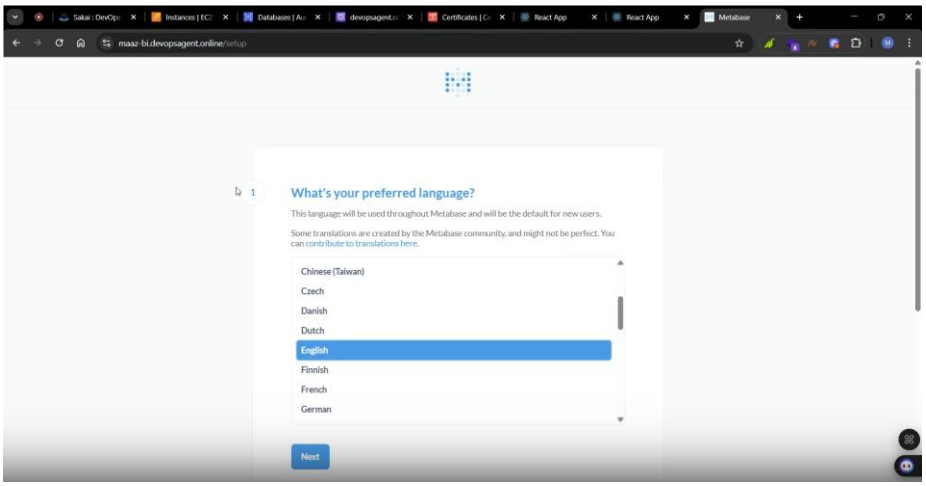
Creating a table in the database:



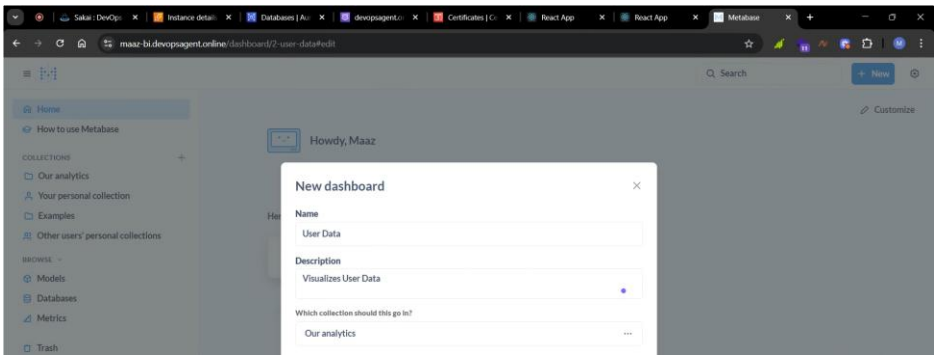
Initial Data Injection:



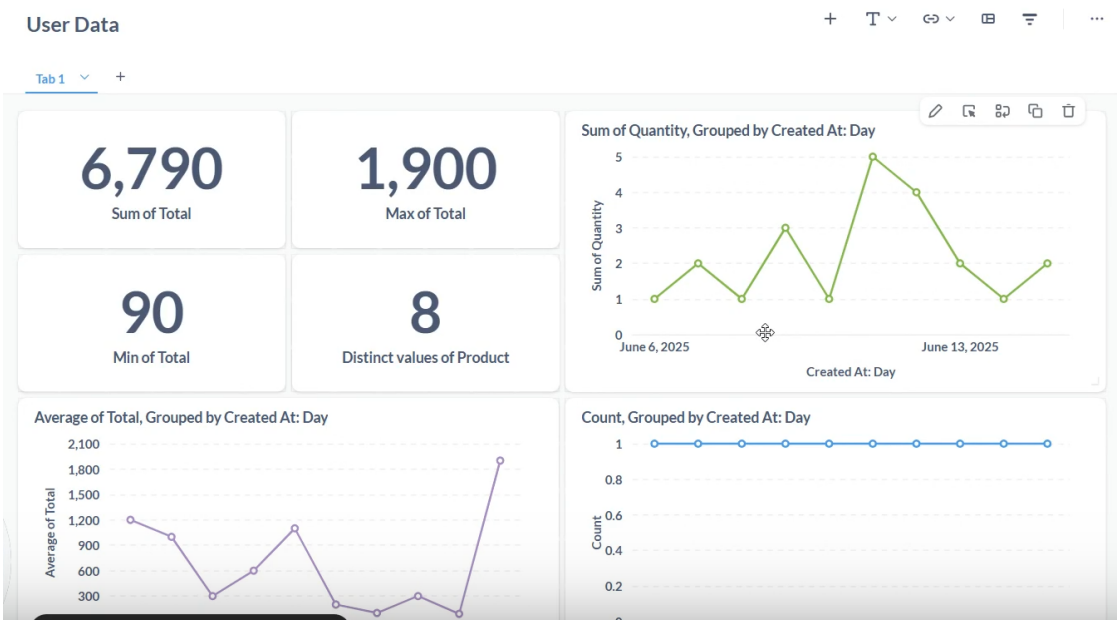
Setting Up Metabase:



Creating a Dashboard:



Dashboard created Successfully with Initial data:

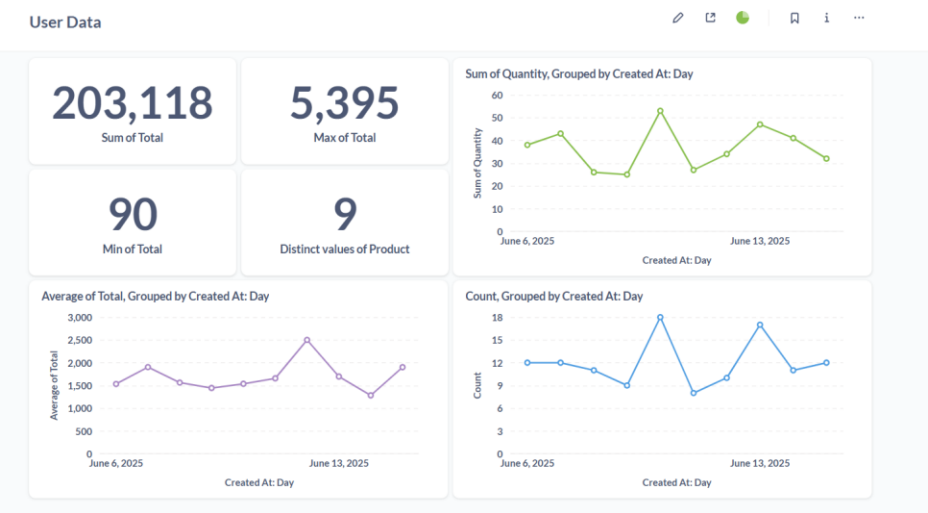


Using the following script for bulk data injection:

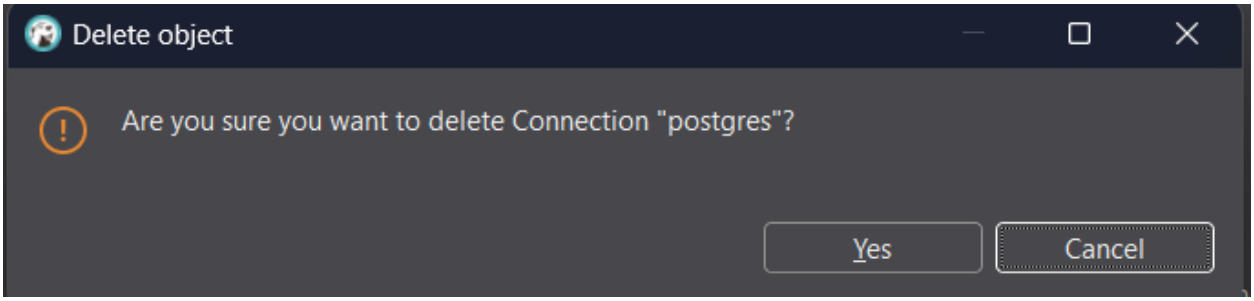
*<postgres> Script-1 | analytics

```
DO $$
DECLARE
  i INT;
  quantity INT;
  price NUMERIC;
BEGIN
  FOR i IN 1..60 LOOP
    quantity := 1 + floor(random() * 5)::INT;
    price := 100 + floor(random() * 1000)::NUMERIC;

    INSERT INTO analytics (
      user_id, user_name, email, country, product, quantity, price, total, created_at
    )
    VALUES (
      i,
      CONCAT('User', i),
      CONCAT('user', i, '@example.com'),
      (ARRAY['USA', 'UK', 'Canada', 'India', 'Germany'])[1 + floor(random() * 5)::INT],
      (ARRAY['Laptop', 'Tablet', 'Phone', 'Monitor', 'Keyboard'])[1 + floor(random() * 5)::INT],
      quantity,
      price,
      quantity * price,
      NOW() - (random() * interval '10 days')
    );
  END LOOP;
END;
$$;
```

After bulk data injection:



Deleting the PostgreSQL connection:



Running “terraform destroy” command:

```
PS C:\Users\maaza\OneDrive - Institute of Business Administration\Documents\CSE 588 - DEV OPS (CLASS 98711)\Project\DevOps-Project\Terraform> terraform destroy
```

```
Plan: 0 to add, 0 to change, 39 to destroy.

Changes to Outputs:
  - load_balancer_dns = "app-alb-1456405498.us-east-1.elb.amazonaws.com" -> null
  - mysql_endpoint    = "mysql-instance.ckxgw6is8tqw.us-east-1.rds.amazonaws.com:3306" -> null
  - postgres_endpoint = "postgres-instance.ckxgw6is8tqw.us-east-1.rds.amazonaws.com:5432" -> null

Do you really want to destroy all resources?
  Terraform will destroy all your managed infrastructure, as shown above.
  There is no undo. Only 'yes' will be accepted to confirm.

  Enter a value: yes
```

All Resources Destroyed Successfully:

```
Destroy complete! Resources: 39 destroyed.
```