Deployed Infrastructure

This document provides the visual evidence to accompany the solution I built for the Simetrik Cloud Engineer technical test. All the infrastructure described in the main README.md file was deployed using the Terraform code in this repository.

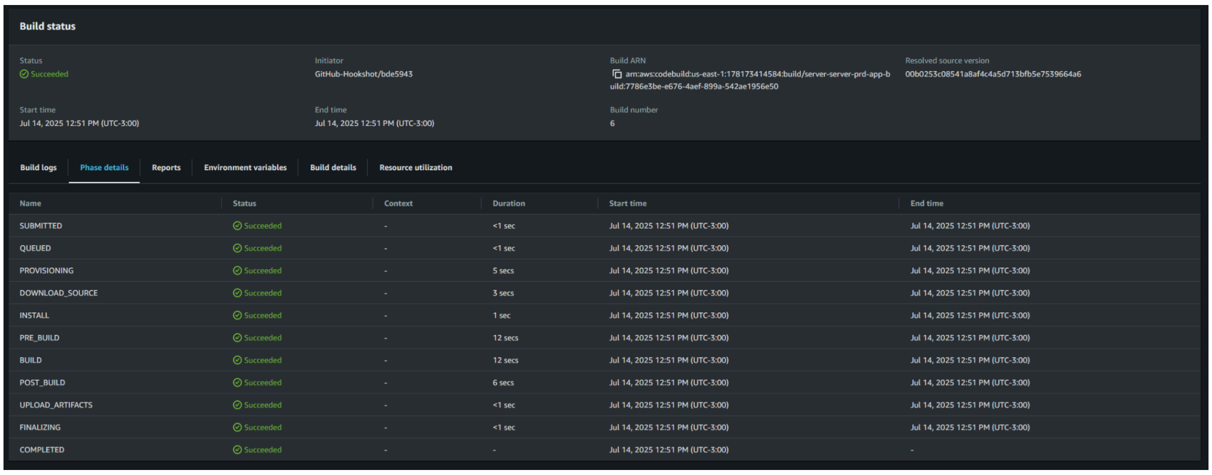
The following screenshots from the AWS Console that the key components of the architecture are successfully provisioned and running. Each image includes a brief explanation of what it demonstrates.

As of submitting this document, the final end-to-end test is pending due to the standard propagation time for DNS. I have fully automated the creation of the Route 53 zone and the ACM certificate request in Terraform. As the evidence shows, the validation CNAME record has been successfully created. The final step is simply waiting for AWS's external validation process to complete, which will allow the ALB Ingress address to resolve. I expect this to be fully functional for our live validation session.

The screenshots will be also available as .png files inside the deliverables folder for better visualization.

1. Automated Deployment via CodeBuild

To meet the CI/CD requirement, I configured an AWS CodeBuild pipeline that triggers automatically on every push to the main branch of the application's GitHub repository. The screenshot below shows the successful execution of this pipeline, which handled everything from building the Docker image to deploying the new version to the EKS cluster.



Build status

Status	Initiator	Build ARN	Resolved source version
✓ Succeeded	GitHub-Hookshot/bde5943	arn:aws:codebuild:us-east-1:178173414584:build/server-server-prd-app-build:7786e3be-e676-4aef-899a-542ae1956e50	00b0253c08541a8af4c4a5d713bfb5e7539664a6

Start time	End time	Build number
Jul 14, 2025 12:51 PM (UTC-3:00)	Jul 14, 2025 12:51 PM (UTC-3:00)	6

Name	Status	Context	Duration	Start time	End time
SUBMITTED	✓ Succeeded	-	<1 sec	Jul 14, 2025 12:51 PM (UTC-3:00)	Jul 14, 2025 12:51 PM (UTC-3:00)
QUEUED	✓ Succeeded	-	<1 sec	Jul 14, 2025 12:51 PM (UTC-3:00)	Jul 14, 2025 12:51 PM (UTC-3:00)
PROVISIONING	✓ Succeeded	-	5 secs	Jul 14, 2025 12:51 PM (UTC-3:00)	Jul 14, 2025 12:51 PM (UTC-3:00)
DOWNLOAD_SOURCE	✓ Succeeded	-	3 secs	Jul 14, 2025 12:51 PM (UTC-3:00)	Jul 14, 2025 12:51 PM (UTC-3:00)
INSTALL	✓ Succeeded	-	1 sec	Jul 14, 2025 12:51 PM (UTC-3:00)	Jul 14, 2025 12:51 PM (UTC-3:00)
PRE_BUILD	✓ Succeeded	-	12 secs	Jul 14, 2025 12:51 PM (UTC-3:00)	Jul 14, 2025 12:51 PM (UTC-3:00)
BUILD	✓ Succeeded	-	12 secs	Jul 14, 2025 12:51 PM (UTC-3:00)	Jul 14, 2025 12:51 PM (UTC-3:00)
POST_BUILD	✓ Succeeded	-	6 secs	Jul 14, 2025 12:51 PM (UTC-3:00)	Jul 14, 2025 12:51 PM (UTC-3:00)
UPLOAD_ARTIFACTS	✓ Succeeded	-	<1 sec	Jul 14, 2025 12:51 PM (UTC-3:00)	Jul 14, 2025 12:51 PM (UTC-3:00)
FINALIZING	✓ Succeeded	-	<1 sec	Jul 14, 2025 12:51 PM (UTC-3:00)	Jul 14, 2025 12:51 PM (UTC-3:00)
COMPLETED	✓ Succeeded	-	-	Jul 14, 2025 12:51 PM (UTC-3:00)	-

2. Container Image in ECR

A key output of the successful CodeBuild pipeline is the containerized application image. As required, this image is pushed to and stored in a private Amazon ECR repository, which was also provisioned by Terraform. The screenshot below shows the image tagged with its unique Git commit hash, ensuring versioning and traceability for our deployments.

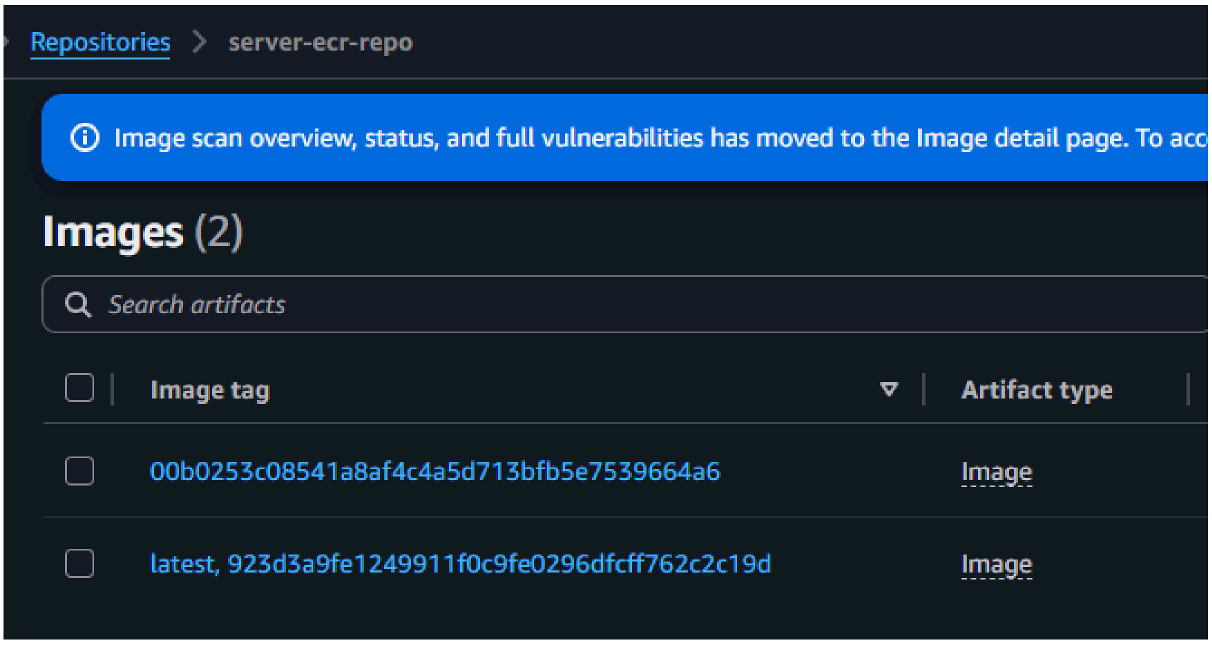
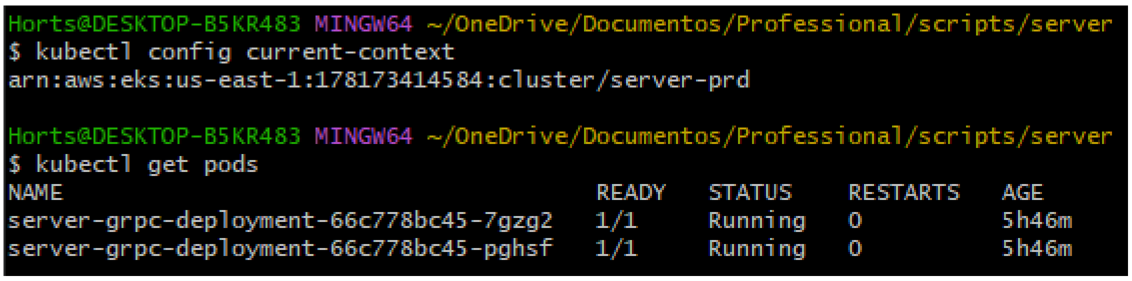

Repositories > server-ecr-repo
Image scan overview, status, and full vulnerabilities has moved to the Image detail page. To acc
Images (2)
Search artifacts

Image tag	Artifact type
00b0253c08541a8af4c4a5d713bfb5e7539664a6	Image
latest, 923d3a9fe1249911f0c9fe0296dfcff762c2c19d	Image

3. Live Application Pods on EKS

Following the successful deployment from the CI/CD pipeline, the containerized application is now running on the EKS cluster. The screenshot below shows the output of `kubectl get pods`, confirming that two replicas of the server application are running and healthy, distributed across the worker nodes for high availability.



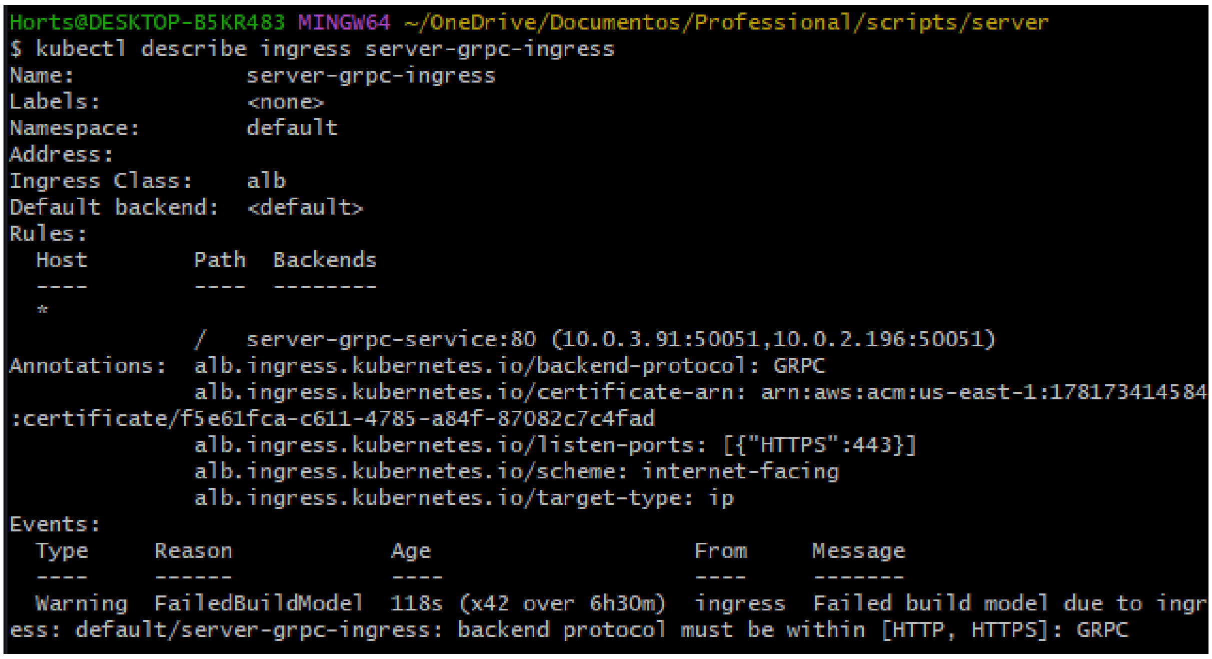
```
Horts@DESKTOP-B5KR483 MINGW64 ~/OneDrive/Documentos/Professional/scripts/server
$ kubectl config current-context
arn:aws:eks:us-east-1:178173414584:cluster/server-prd

Horts@DESKTOP-B5KR483 MINGW64 ~/OneDrive/Documentos/Professional/scripts/server
$ kubectl get pods
NAME                                    READY   STATUS    RESTARTS   AGE
server-grpc-deployment-66c778bc45-7gzg2   1/1     Running   0          5h46m
server-grpc-deployment-66c778bc45-pghsf   1/1     Running   0          5h46m
```

4. Ingress Configuration for the Application Load Balancer

To expose the application securely, I created a Kubernetes `Ingress` resource. The annotations on this resource instruct the AWS Load Balancer Controller to provision an internet-facing Application Load Balancer configured for GRPC over an HTTPS listener.

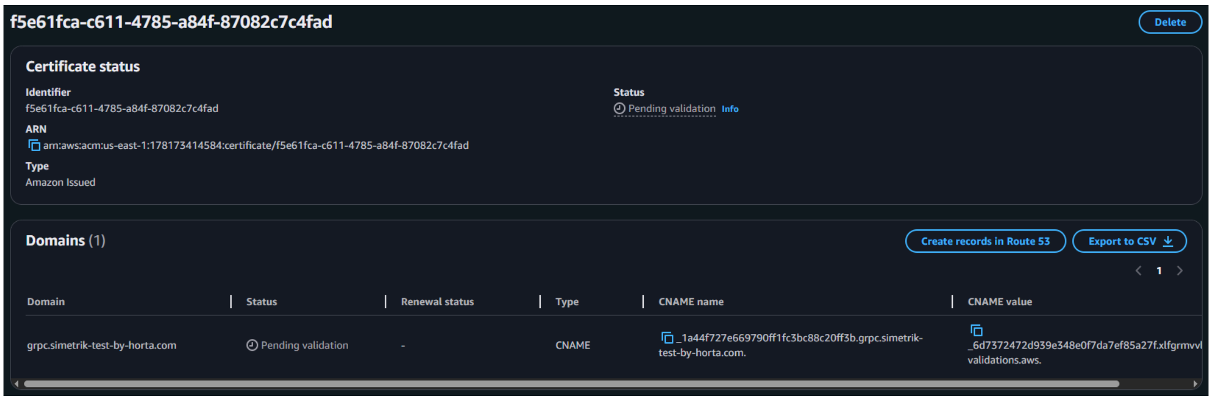
The screenshot below shows the details of this Ingress. As you can see in the Events section, the final step is currently pending the validation of the ACM certificate, which was provisioned automatically by Terraform. This is a standard DNS propagation delay, and I expect the public address to be available for our live review session.



```
Horts@DESKTOP-B5KR483 MINGW64 ~/OneDrive/Documentos/Professional/scripts/server
$ kubectl describe ingress server-grpc-ingress
Name:             server-grpc-ingress
Labels:           <none>
Namespace:        default
Address:
Ingress Class:    alb
Default backend:  <default>
Rules:
  Host        Path  Backends
  ----        ----  --------
  *
              /   server-grpc-service:80 (10.0.3.91:50051,10.0.2.196:50051)
Annotations:  alb.ingress.kubernetes.io/backend-protocol: GRPC
              alb.ingress.kubernetes.io/certificate-arn: arn:aws:acm:us-east-1:178173414584
:certificate/f5e61fca-c611-4785-a84f-87082c7c4fad
              alb.ingress.kubernetes.io/listen-ports: [{"HTTPS":443}]
              alb.ingress.kubernetes.io/scheme: internet-facing
              alb.ingress.kubernetes.io/target-type: ip
Events:
  Type     Reason            Age                   From     Message
  ----     ------            ----                  ----     -------
  Warning  FailedBuildModel  118s (x42 over 6h30m)  ingress  Failed build model due to ingr
ess: default/server-grpc-ingress: backend protocol must be within [HTTP, HTTPS]: GRPC
```

5. Automated Certificate Provisioning

To enable the required HTTPS listener on the Application Load Balancer, I automated the creation and validation of a public TLS certificate using Terraform. The screenshot below from the AWS Certificate Manager console shows the certificate that was successfully requested for our domain. Its validation is currently pending the standard DNS propagation delay.



f5e61fca-c611-4785-a84f-87082c7c4fad

Certificate status

Identifier
f5e61fca-c611-4785-a84f-87082c7c4fad

Status
Pending validation Info

ARN
arn:aws:acm:us-east-1:178173414584:certificate/f5e61fca-c611-4785-a84f-87082c7c4fad

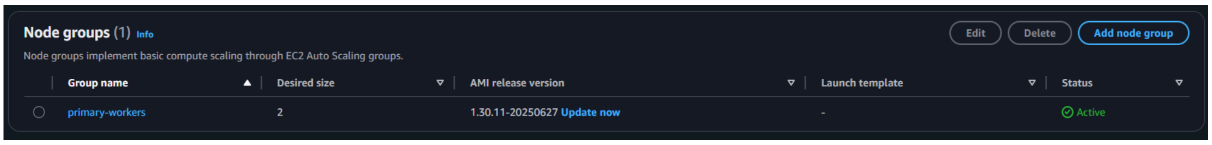
Type
Amazon Issued

Domains (1)

Domain	Status	Renewal status	Type	CNAME name	CNAME value
grpc.simetrik-test-by-horta.com	Pending validation	-	CNAME	_1a44f727e669790ff1fc3bc88c20ff3b.grpc.simetrik-test-by-horta.com.	_6d7372472d939e348e0f7da7ef85a27f.xlfgrmvvl validations.aws.

6. EKS Cluster and Compute Resources

At the heart of the deployment is the Amazon EKS cluster, which was provisioned by the `eks` Terraform module. The screenshot below from the EKS console confirms that the cluster (server-prd) is active and that its associated EC2 Node Group is running with the desired number of instances, ready to host our application pods.



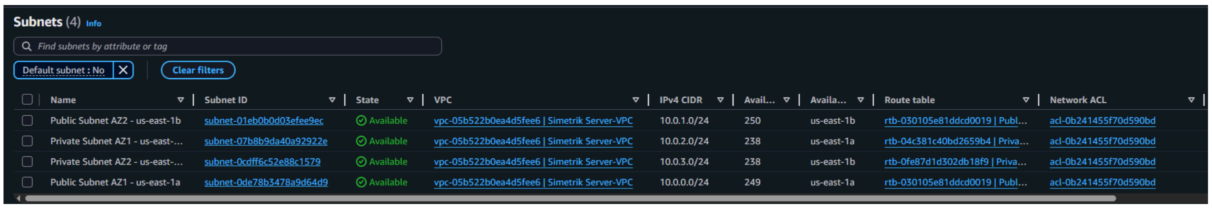
Node groups (1) Info

Node groups implement basic compute scaling through EC2 Auto Scaling groups.

Group name	Desired size	AMI release version	Launch template	Status
primary-workers	2	1.30.11-20250627 Update now	-	Active

7. Custom VPC and Subnet FoundationEKS Cluster and Compute Resources

Using the reusable `networking` Terraform module, I provisioned a custom VPC to create a secure, isolated environment. The screenshot below shows the subnets created by the module, highlighting the public/private strategy across two Availability Zones to ensure both security and high availability for the entire deployment.



Subnets (4) Info

Name	Subnet ID	State	VPC	IPv4 CIDR	Avail...	Availa...	Route table	Network ACL
Public Subnet AZ2 - us-east-1b	subnet-01eb0b0d03efee9ec	Available	vpc-05b522b0ea4d5fee6 | Simetrik Server-VPC	10.0.1.0/24	250	us-east-1b	rtb-030105e81ddcd0019 | Publ...	acl-0b241455f70d590bd
Private Subnet AZ1 - us-east-...	subnet-07b8b9da40a92922e	Available	vpc-05b522b0ea4d5fee6 | Simetrik Server-VPC	10.0.2.0/24	238	us-east-1a	rtb-04c381c40bd2659b4 | Priva...	acl-0b241455f70d590bd
Private Subnet AZ2 - us-east-...	subnet-0cdff6c52e88c1579	Available	vpc-05b522b0ea4d5fee6 | Simetrik Server-VPC	10.0.3.0/24	238	us-east-1b	rtb-0fe87d1d302db18f9 | Priva...	acl-0b241455f70d590bd
Public Subnet AZ1 - us-east-1a	subnet-0de78b3478a9d64d9	Available	vpc-05b522b0ea4d5fee6 | Simetrik Server-VPC	10.0.0.0/24	249	us-east-1a	rtb-030105e81ddcd0019 | Publ...	acl-0b241455f70d590bd

Simetrik Cloud Engineer Technical Test - Hortencia Santos

Welcome to my solution for the Simetrik Cloud Engineer technical test. This repository is structured to provide a clear overview of the applications, the infrastructure as code, and all supporting materials.

Here's a breakdown of what you'll find:

- **Applications**: This directory contains the two Python applications built with the gRPC protocol. You'll find both the `server` and `client` code here. The code is created to work on the AWS infrastructure so it won't run locally (I have a copy that does, in case you're interested in some hard core mega programming)
- **Terraform Modules**: Here you can find the two reusable required Terraform modules I created. The `networking` module sets up the VPC and subnets, while the `eks` module handles the deployment of the EKS cluster, CI/CD pipeline, and all related resources - each infrastructure necessary is in a separate .tf file. There's also a third module, `ec2`, that I used to deploy the client side application to test the server app.
- **Terraform Deployments**: Separated each deployment into its own root directory to manage each environment's remote state independently.
- **Scripts**: To meet the requirement of using an Application Load Balancer, I created a script to install the `aws-load-balancer-controller` inside the EKS cluster. You can find it in this directory.
- **Deliverables**: All the required deliverables, including the architecture diagram and the documentation in PDF format, are located here.

Table of Contents

Project Overview

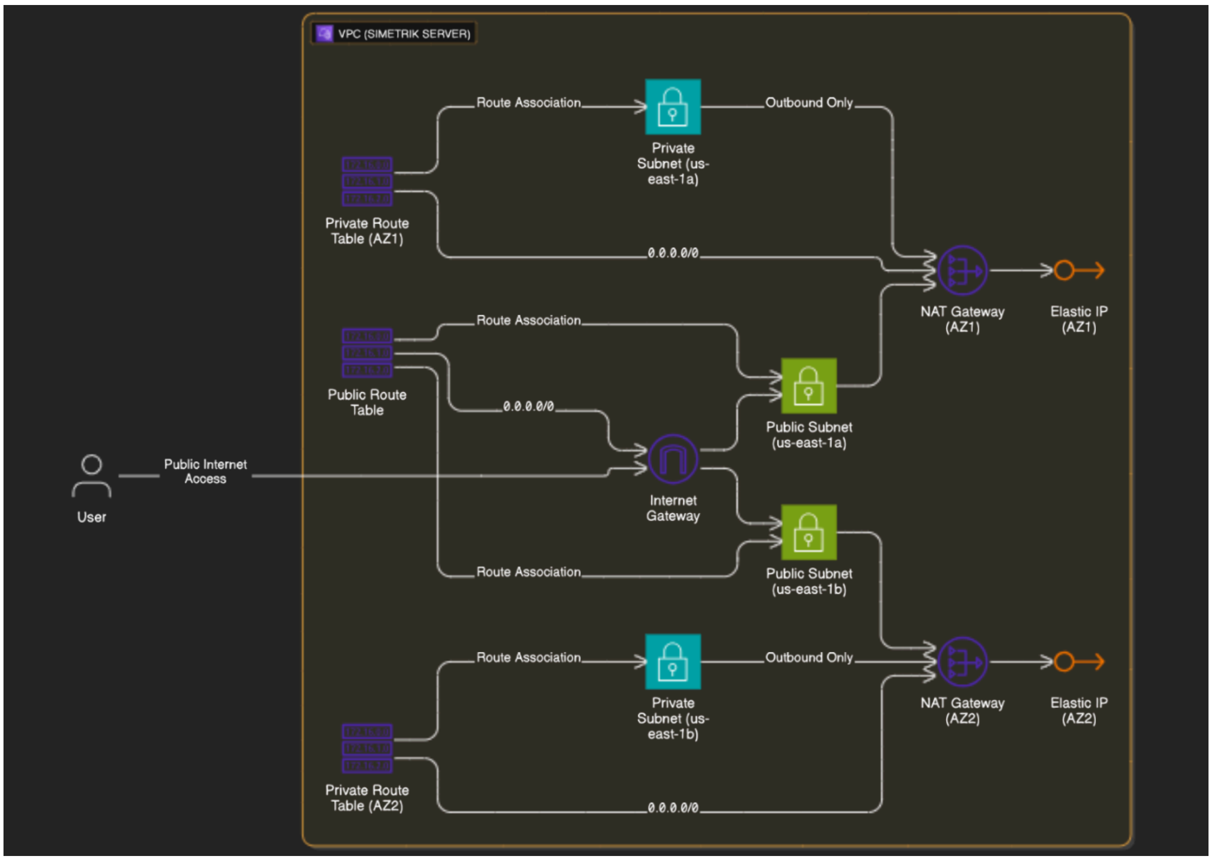
The goal of this project is to deploy a containerized gRPC server application onto a secure and scalable EKS cluster. The entire infrastructure is defined as code using reusable Terraform modules, and the application deployment is fully automated via a CI/CD pipeline using AWS CodeBuild.

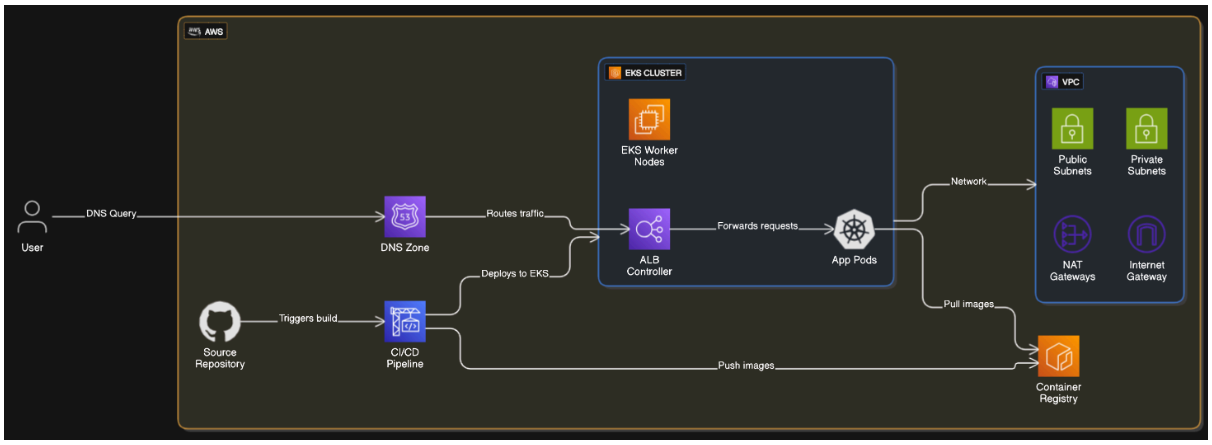
Key Features:

- **Infrastructure as Code (IaC):** All AWS resources are managed by Terraform, ensuring repeatability and version control.
- **Remote State Management:** The state of the infrastructure is preserved remotely in S3 buckets, enabling collaboration and preventing state loss - with state lock stored in dynamodb.
- **Reusable Modules:** The infrastructure is split into `networking`, `eks`, and `ec2` modules for maximum reusability across different environments.
- **Containerization:** The application is containerized with Docker and stored in a secure AWS ECR repository, which is also provisioned by Terraform.
- **Kubernetes Orchestration:** Amazon EKS is used to manage, scale, and ensure the high availability of the application.
- **CI/CD Automation:** A CodeBuild pipeline automatically builds and deploys the application to EKS upon every push to the `main` branch of the configured GitHub repository.
- **Secure Exposure:** The application is securely exposed to the internet via an Application Load Balancer (ALB) using an HTTPS listener and a TLS certificate managed by ACM.

Architecture Diagram

Bellow are the diagrams for the Networking architecture and the Application architecture:





Note: the diagrams were generated based on the state files by app.eraser.io/ - unfortunately I wasn't able to build from scratch due to some time management issues

The core architecture for this project is deployed in `us-east-1`, where the server application runs on an EKS cluster. To simplify testing and keep costs low, I've configured the client application to run from a simple EC2 instance within the same environment.

Core Components:

1. **VPC:** A custom VPC provides network isolation. It contains:

    a. **Public Subnets:** Span two Availability Zones for high availability. The Application Load Balancer and NAT Gateways reside here. An Internet Gateway provides internet access to the resources rquired in these subnets.
    b. **Private Subnets:** Also two Availability Zones. The EKS worker nodes run here, ensuring they are not directly exposed to the internet. They access the internet via the NAT Gateways for tasks like pulling public images.
2. **Amazon EKS Cluster:**
    a. The managed EKS control plane resides in the AWS-managed VPC.
    b. Worker nodes (EC2 instances) are deployed into the private subnets of our custom VPC.
    c. The **AWS Load Balancer Controller** runs within the cluster, watching for `Ingress` resources.
3. **Application Load Balancer (ALB):**
    a. When a Kubernetes `Ingress` object is created, the controller automatically provisions an internet-facing ALB in the public subnets.
    b. It uses an HTTPS listener on port 443 with a TLS certificate from ACM to securely terminate traffic.
    c. It routes GRPC traffic to the application pods running on the worker nodes.
4. **CI/CD Pipeline:**
    a. A **GitHub** repository hosts the application source code.
    b. An **AWS CodeBuild** project is triggered by a webhook on every `git push` to the `main` branch.
    c. The pipeline builds the Docker image, pushes it to a private **Amazon ECR** repository, and runs `kubectl` commands to deploy the new image to the EKS cluster.
5. **Route 53 & ACM:**
    a. A **Route 53 Hosted Zone** is created to manage DNS for a test domain.
    b. **AWS Certificate Manager (ACM)** automatically requests and validates a public TLS certificate using the Route 53 zone, which is then used by the ALB.

Networking Explanation

For this project, I built the network foundation using a dedicated Terraform module to ensure it's secure, highly available, and reusable and also to meet the criterias of the test.

1. **VPC and Subnet Strategy:** I started by creating a custom VPC to provide a logically isolated network for the entire deployment. Within this VPC, I implemented a classic two-tier design with public and private subnets.
    a. **Public Subnets:** These are for resources that need to be directly accessible from the internet, like the Application Load Balancer. There are two public subnets, each in a different Availability Zone (AZ), to ensure high availability. These subnets have a route to an **Internet Gateway**, which allows them to communicate with the web outside the VPC isolation.

   b. **Private Subnets:** This is where the EKS worker nodes (EC2 instances) are deployed preventing them from having a public IP address and from direct internet exposure.
2. **NAT Gateways for Egress:** The NAT Gateways are used for the egress traffic from the worker nodes to execute some activites such as pulling the container images and downloading software packages. The NAT Gateways are deployed in each public subnet. The private subnets have a route that directs all internet-bound traffic to the NAT Gateway, which then forwards the traffic to the Internet Gateway.
3. **Security Layers:**
   a. **Security Groups:** I use security groups as stateful firewalls for the resources. The ALB's security group is configured to allow inbound HTTPS traffic (port 443) from the internet. The EKS worker node security group is configured to only accept traffic from the ALB on the application's port, creating a secure path from the load balancer to the application.
   b. **NACLs (Network ACLs):** As an additional layer to filter traffic entering and leaving the subnets.
4. **Application Traffic Flow (Ingress):** As required, the ALB is deployed on the public subnet and manages the application traffic. The ALB receives the HTTPS application request via the DNS and then the GRPC traffic to the private IP address of an application pod running on a worker node in one of the private subnets.

Deployment Guide

This guide is a step-by-step on how to deploy the entire infrastructure and application from scratch.

Prerequisites

1. An AWS account with the necessary permissions inside your environment.
2. The AWS CLI installed and configured with a profile.
3. Terraform installed.
4. A GitHub personal access token with `repo` and `admin:repo_hook` permissions, stored as a secret in AWS Secrets Manager.
5. An EC2 Key Pair created in the `us-east-1` region.
6. A S3 bucket to store the state of your infrastructure remotely.
7. A dynamodb table to store the state locks - to prevent state loss

Deploying the Infrastructure

1. Clone the infrastructure repository.
2. Create a `terraform.tfvars` file with the required variable values (region, project name, etc.). These variables are defined inside the root folders of each infrastructure.
3. Initialize Terraform for the defined backend environment. The `backend.tf` files on each root module correspond to different keys or buckets, edit accordingly:
   I usually run terraform init with the -reconfigure flag if there's already a deprecated

local state

```
terraform init -reconfigure
```

I usually run terraform init with the -reconfigure flag if there's already a deprecated local state and without the flag if it's the first deploy and there's no remote state

4. Run the plan to verify the changes:

```
terraform plan
```

5. Apply the configuration to create the resources:

```
terraform apply
```

6. *Note: The first apply may fail with an `Unauthorized` error while creating the `aws-auth` ConfigMap. This is expected. The user running `apply` must be added to the `admin_user_arns` variable, and then `apply` can be run again.*
   Note 2: If the `aws-auth` ConfigMap already exists, a one-time `terraform import` command is required.

Deploying the Application

1. Clone the gRPC server application repository.
2. Push the code to the `main` branch of the repo you're defining for the pipeline.
3. This will automatically trigger the CodeBuild pipeline, which builds the Docker image, pushes it to ECR, and deploys it to the EKS cluster.

Testing the Application

Note: make sure your kubernetes credentials are configure on your local machine and you have kubectl installed

1. Get the public address of the Application Load Balancer:

```
kubectl get ingress server-grpc-ingress
```

2. Deploy the client application (e.g., a local machine or a separate EC2 instance - like I did).
3. Set the `GRPC_SERVER_ADDRESS` environment variable to the ALB address from the previous step, including the port `:443`.
4. Run the client application to test the connection.

Destroying the Infrastructure

To avoid ongoing costs, it's crucial to destroy all created resources when you are finished.

1. First, destroy the EKS module resources:

```
# Ensure you are initialized for the correct state file
terraform destroy -target=module.eks
```

2. Then, destroy the networking resources:

```
# Ensure you are initialized for the correct state file
 terraform destroy -target=module.networking
```