Secure and Scalable solution to Deploy a Web Application on AWS Elastic Container Service

A MINI PROJECT REPORT

Submitted by

AFRAZ TANVIR [RA2011003010499]
SUYASH JOSHI [RA2011003010508]
BASIT HASAN [RA2011003010532]

Under the guidance of

Dr. V. Deepan Chakravarthy
(Associate Professor, Department of Computing Technologies)

In partial satisfaction of the requirements for the degree of

BACHELOR OF TECHNOLOGY

in

COMPUTER SCIENCE & ENGINEERING

SCHOOL OF COMPUTING

COLLEGE OF ENGINEERING AND TECHNOLOGY

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

KATTANKULATHUR - 603203

MAY 2023

COLLEGE OF ENGINEERING & TECHNOLOGY
SRM INSTITUTE OF SCIENCE & TECHNOLOGY
S.R.M. NAGAR, KATTANKULATHUR – 603 203

BONAFIDE CERTIFICATE

Certified that this project report **"Secure and Scalable solution to Deploy a Web Application on AWS Elastic Container Service"** is the bonafide work of **" Afraz Tanvir (RA2011003010499), Suyash Joshi (RA2011003010508), Basit Hasan (RA2011003010532)"** of III Year/VI Sem B.tech(CSE) who carried out the mini project work under my supervision for the course 18CSE316J - Essentials in Cloud and Devops in SRM Institute of Science and Technology during the academic year 2022-2023(Even sem).

SIGNATURE

Dr.V.Deepan Chakravarthy
Associate Professor, Department of Computing Technologies
School of computing

ABSTRACT

The deployment of web applications has become increasingly complex in recent years due to the rise of microservices architecture, containerization, and cloud computing. AWS Elastic Container Service (ECS) is a fully managed container orchestration service that simplifies the deployment and management of containerized applications on AWS. This project aims to demonstrate the deployment of a secure and scalable web application on AWS ECS using Docker containerization and infrastructure as code (IaC) tools like Terraform.

The project begins by creating a simple Node.js web application that can be run locally. The application is then containerized using Docker, which allows the application to be packaged and deployed consistently across different environments. The Docker image is then pushed to an AWS Elastic Container Registry (ECR) repository, which provides a secure and scalable way to store and manage Docker images.

Next, an ECS cluster is created with the necessary networking and security configurations. The cluster allows for the deployment and management of multiple Docker containers across different instances, providing fault tolerance and high availability. An ECS task and service are then created to run the Docker image on the ECS cluster, which ensures that the application is always available and responsive.

To handle the incoming traffic, a load balancer is added to the ECS service. The load balancer distributes the incoming traffic across the instances in the ECS cluster, ensuring that the workload is evenly distributed and that no single instance is overloaded.

Finally, Terraform is used to define and provision the infrastructure as code, making it easy to manage and maintain the environment. Terraform allows for the creation of repeatable and consistent infrastructure, which can be versioned and easily rolled back in case of errors or changes.

Overall, this project demonstrates the benefits of using containerization and cloud-based services for deploying and managing web applications in a secure and scalable manner. It shows how AWS ECS, Docker, and Terraform can be used together to create a modern and efficient deployment pipeline, which can adapt to changing requirements and workload.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABBREVIATIONS

AWS	Amazon Web Services
ECS	Elastic Container Service
ECR	Elastic Container Registry
IaC	Infrastructure as Code
Node.js	Node, a JavaScript runtime
npm	Node Package Manager, a package manager for Node.js
Docker	a containerization platform
CLI	Command Line Interface
VPC	Virtual Private Cloud
ALB	Application Load Balancer
EC2	Elastic Compute Cloud
IAM	Identity and Access Management
SG	Security Group

CHAPTER 1

INTRODCUTION

1.1 Aim

The aim of this project is to demonstrate a secure and scalable deployment of the GIPHY website on AWS Elastic Container Service (ECS) using Docker containerization and Terraform infrastructure as code tool. By leveraging the capabilities of AWS ECS, Docker, and Terraform, we aim to showcase the benefits of containerization and infrastructure as code in achieving a secure and efficient deployment of web applications. The project will focus on ensuring high availability, scalability, and security of the GIPHY website, while also providing insights into the best practices for containerized deployment on AWS.

1.2 Background

The deployment of web applications on the cloud has become increasingly popular in recent years due to its scalability, flexibility, and cost-effectiveness. Amazon Web Services (AWS) offers a wide range of cloud computing services, including the Elastic Container Service (ECS), which provides a scalable and efficient way to deploy and manage containerized applications. In this project, we aim to demonstrate a secure and scalable deployment of a web application on AWS ECS using Docker containerization and Terraform infrastructure as code tool.

1.3 Context of the Project

The traditional approach to deploying web applications on servers involves setting up a physical or virtual machine, installing the necessary software, and configuring the system to run the application. This approach has several limitations, such as scalability challenges, maintenance overhead, and the risk of vendor lock-in. Containerization using Docker provides a lightweight and portable alternative that enables developers to package and deploy applications in a consistent and reliable manner.

AWS ECS is a fully managed container orchestration service that simplifies the deployment and management of Docker containers at scale. With ECS, developers can define the infrastructure

resources required for their applications, and ECS handles the scaling, monitoring, and recovery of containers automatically.

1.4 Objectives and Goals

The primary objective of this project is to demonstrate a secure and scalable deployment of a web application on AWS ECS using Docker containerization and Terraform infrastructure as code tool. The project aims to achieve the following goals:

- Develop a simple Node application and Dockerize it
- Create an image repository on AWS ECR and push the Docker image
- Create an AWS ECS cluster and task to run the Docker container
- Implement load balancing using an AWS Application Load Balancer
- Secure the deployment using IAM roles and security groups
- Use Terraform to automate the deployment and management of the infrastructure

1.5 Project scope and limitations

The scope of this project is limited to the deployment of a single containerized web application on AWS ECS. We will focus on using best practices for security, scalability, and maintainability, but the project does not aim to cover all possible scenarios and use cases. The limitations of this project include the following:

- We will not cover the development of the web application or the business logic of the application
- The project does not aim to provide an exhaustive comparison of ECS with other container orchestration services such as Kubernetes
- We will not cover advanced topics such as service discovery, auto-scaling, and monitoring in detail

CHAPTER 2

LITERATURE SURVEY

2.1 Overview of AWS Elastic Container Service and its benefits:

AWS Elastic Container Service (ECS) is a fully managed container orchestration service that enables developers to deploy and manage Docker containers at scale. ECS provides several benefits, such as:

- Scalability: ECS makes it easy to scale the infrastructure resources required for running containerized applications, ensuring that the application can handle high traffic loads.
- High availability: ECS automatically handles the scaling, monitoring, and recovery of containers, ensuring that the application is always available to users.
- Security: ECS provides several security features, such as IAM roles and security groups, to protect the infrastructure and data.
- Cost-effectiveness: ECS offers a pay-as-you-go pricing model, allowing developers to optimize their infrastructure costs based on their usage.

2.2 Overview of Docker containerization and its advantages

Docker is an open-source platform that provides a standardized way to package and deploy applications in containers. Docker containerization offers several advantages, such as:

- Portability: Docker containers can be run on any platform that supports Docker, making it easy to move applications between different environments.
- Consistency: Docker containers provide a consistent runtime environment for applications, ensuring that they behave predictably regardless of the underlying infrastructure.
- Resource efficiency: Docker containers are lightweight and consume fewer resources than traditional virtual machines, enabling developers to run more containers on the same hardware.
- Speed: Docker containers can be started and stopped quickly, enabling rapid deployment and iteration of applications.

2.3 Introduction to Amazon ECR and its features

Amazon Elastic Container Registry (ECR) is a fully managed Docker container registry that makes it easy to store, manage, and deploy Docker images. ECR provides several features, such as:

- Scalability: ECR can handle millions of Docker images and provides high availability and durability for image storage.
- Security: ECR integrates with AWS Identity and Access Management (IAM) to provide granular access control for images and repositories.
- Compatibility: ECR is fully compatible with the Docker command-line interface and other Docker tools, making it easy to integrate with existing Docker workflows.
- Cost-effectiveness: ECR offers a pay-as-you-go pricing model based on the amount of data stored and transferred.

2.4 Importance of Terraform in infrastructure as code

Terraform is an open-source infrastructure as code tool that enables developers to define and manage infrastructure resources in a declarative and version-controlled manner. Terraform provides several benefits, such as:

- Automation: Terraform enables the automation of infrastructure deployment and management, reducing manual overhead and minimizing errors.
- Scalability: Terraform can manage infrastructure resources at any scale, from a single server to a complex multi-cloud environment.
- Consistency: Terraform provides a consistent and reproducible way to deploy and manage infrastructure resources, ensuring that they behave predictably across different environments.
- Collaboration: Terraform supports team collaboration by providing version control and a clear audit trail of infrastructure changes.

CHAPTER 3

SYSTEM DESIGN AND METHODOLOGY

3.1 System Requirements and Specifications

To deploy the GIPHY website on AWS ECS, the following system requirements and specifications are necessary:

Hardware Requirements

- 1 t2.micro EC2 instance for hosting the Docker container.
- 1 Application Load Balancer for distributing traffic to the EC2 instances.

Software Requirements

- Node.js (v12 or higher)
- Docker (v20 or higher)
- Terraform (v1.0 or higher)
- Git

Network Requirements

- The GIPHY website must be accessible via HTTP/HTTPS traffic over port 80/443.
- The EC2 instance must be configured with a security group that allows incoming traffic on port 80/443.

3.2 Tools and Technologies used

Amazon ECS

Amazon Elastic Container Service (ECS) is a fully managed container orchestration service that enables easy deployment, scaling, and management of containerized applications.

Docker

Docker will be used to containerize the GIPHY website, which will enable easy deployment and management of the application.

Amazon ECR

Amazon Elastic Container Registry (ECR) is a fully-managed Docker container registry that enables easy storage, management, and deployment of Docker images.

Terraform

Terraform is an open-source infrastructure as code tool that enables easy management of infrastructure resources on AWS.

Node.js

Node.js is a JavaScript runtime that enables the development of server-side applications in JavaScript.

GitHub

GitHub is a web-based hosting service for version control and collaboration.

AWS CLI

The AWS Command Line Interface (CLI) is a tool that enables management of AWS resources from the command line.

AWS CloudWatch

AWS CloudWatch is a monitoring and logging service that provides real-time metrics and logs for AWS resources.

3.3 System Architecture and Components

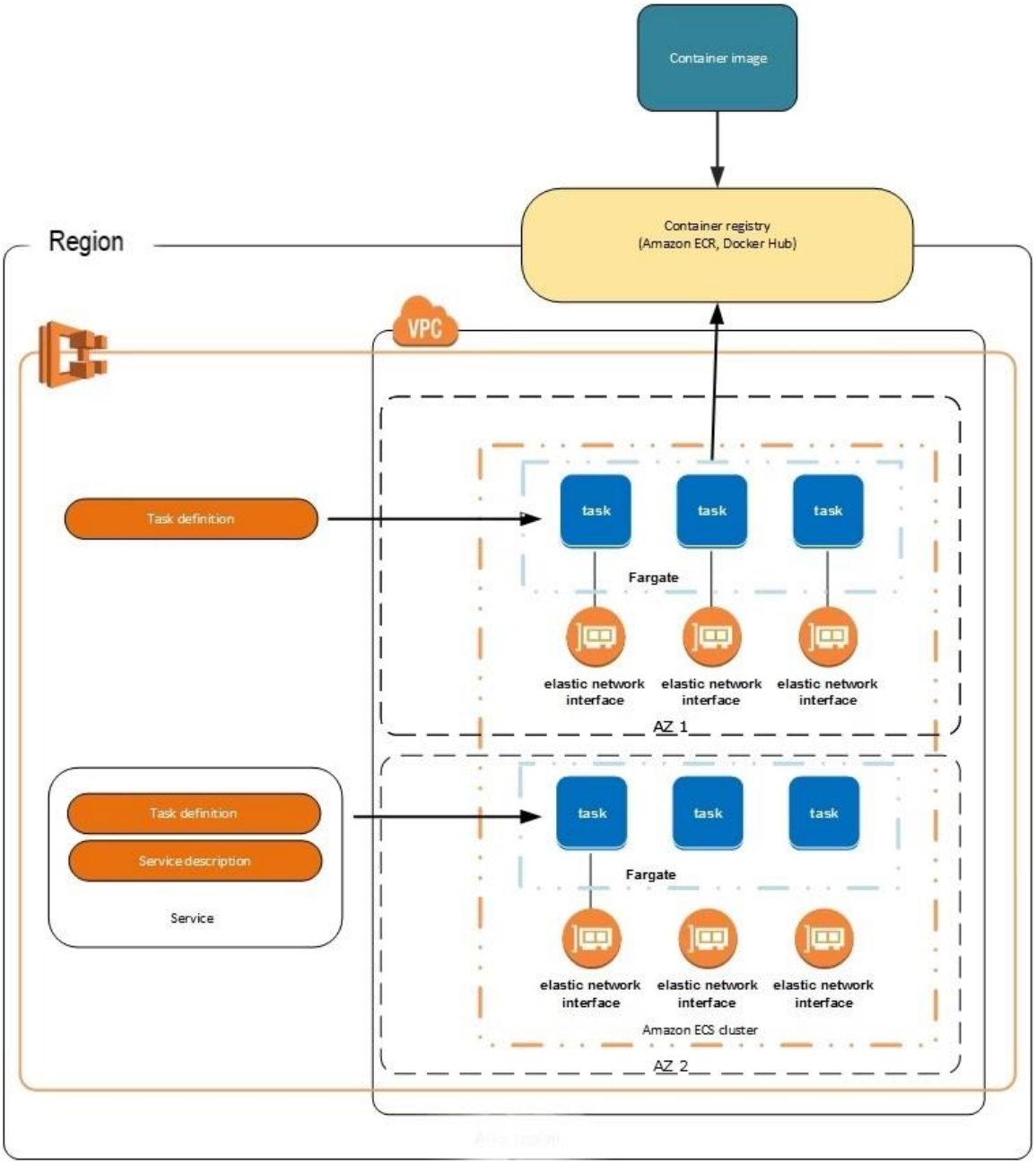

Fig 3.1 Architecture Diagram

The GIPHY website will be deployed on AWS ECS using Docker containers. The system will consist of the following components:

Amazon ECS Cluster

An ECS cluster is a logical grouping of container instances that run together and share the same resources. The GIPHY website will be deployed to an ECS cluster to enable easy scaling and management of the application.

Docker Containers

The GIPHY website will be containerized using Docker, which enables easy deployment and management of the application. The website will be run as a Docker container on an ECS cluster, allowing for easy scaling and management of the application.

Amazon ECR

ECR (Elastic Container Registry) is a fully-managed Docker container registry that makes it easy to store, manage, and deploy Docker container images. The GIPHY website Docker image will be stored in an ECR repository, which will be created and managed by Terraform.

Amazon RDS

Amazon RDS (Relational Database Service) is a fully-managed database service that makes it easy to set up, operate, and scale a relational database. The GIPHY website will use Amazon RDS to store user data.

Application Load Balancer

An application load balancer will be used to distribute traffic to the Docker containers running the GIPHY website. The load balancer will be created and managed by Terraform, and will ensure that traffic is distributed evenly across multiple instances of the application.

Amazon VPC

The GIPHY website will be deployed in an Amazon VPC (Virtual Private Cloud), which provides a logically isolated section of the AWS cloud to launch resources. The VPC will be created and managed by Terraform, and will ensure that the GIPHY website is securely isolated from other AWS resources.

3.4 Methodology Used in System Development

To deploy and manage the GIPHY website on AWS ECS, the following methodology will be used:

Containerization

The GIPHY website will be containerized using Docker, which will enable easy deployment and management of the application. By containerizing the website, we can ensure that the application runs the same way on any machine, regardless of the underlying environment.

Infrastructure as Code

Terraform will be used to manage the infrastructure required to run the GIPHY website on AWS ECS. By using infrastructure as code, we can ensure that our infrastructure is repeatable, versioned, and easily configurable. Terraform will enable us to create and manage resources such as ECS clusters, ECR repositories, load balancers, and VPCs.

Continuous Integration/Continuous Deployment (CI/CD)

We will use a CI/CD pipeline to automate the deployment process of the GIPHY website. Whenever changes are pushed to the source code repository, a build will be triggered which will create a new Docker image, push it to ECR, and update the ECS service to use the latest image. This will ensure that the website is always up-to-date with the latest changes.

Scaling

To ensure that the GIPHY website can handle increased traffic, we will use ECS auto scaling. When CPU utilization or memory usage of the container instances in the ECS cluster exceeds a certain threshold, new container instances will be added to the cluster automatically. This will enable us to handle sudden spikes in traffic without having to manually scale up the application.

Monitoring and Logging

To monitor the performance and health of the GIPHY website, we will use AWS CloudWatch. CloudWatch will provide us with real-time metrics and logs, which will enable us to troubleshoot issues and optimize performance.

Security

To ensure that the GIPHY website is secure, we will use AWS security best practices. This will include using AWS Identity and Access Management (IAM) to manage access to AWS resources, encrypting data at rest and in transit, and using Amazon VPC to isolate the GIPHY website from other AWS resources.

CHAPTER 4

Configuration and Deployment Plan

We have a working website made using node.js and express which uses API to fetch and show GIFs. We will securely and scalaby deploy this web application on AWS Elastic Container Service

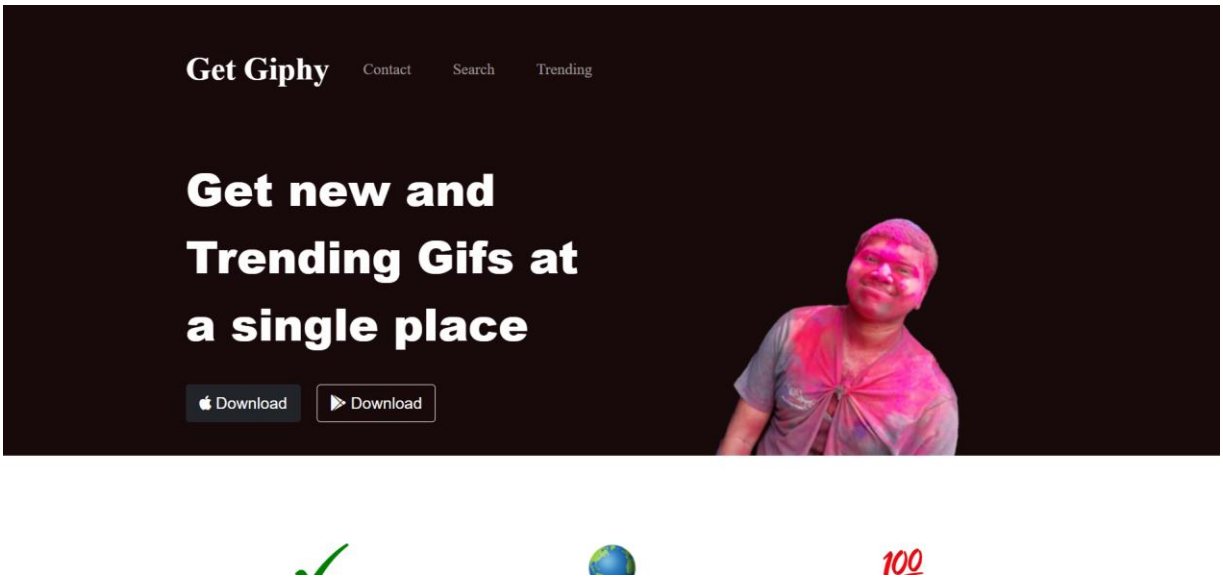

Fig 4.1 Get Giphy Website

4.1 Dockerize the Application

To containerize the GIPHY website, we will be using Docker, a popular open-source containerization platform. Containerizing the application involves creating a Docker image that contains all the necessary dependencies and configuration to run the Node.js application.

To create the Docker image, we will first need to create a Dockerfile that specifies the base image, dependencies, and application code. Once the Dockerfile is created, we will use the Docker command line interface to build the Docker image and tag it with a unique name.

Dockerfile Used

```
# Use an official Node.js image as the base image
FROM node:14
# Set the working directory in the container
WORKDIR /app
# Copy the required files to the container
COPY package*.json ./
COPY /public /app/public
COPY /views /app/views
COPY index.html .
COPY app.js .
# Install the application dependencies
RUN npm install --no-cache
# Specify the command to run when the container starts
CMD [ "node", "app.js" ]
```

4.2 Create an image repository on AWS ECR and push the image

Pushing our container to a container registry service in this case, we will use AWS ECR:

We will use terraform to create our repository. In working directory create a file called main.tf.

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 2.0"
    }
  }
}

provider "aws" {
  region = "us-east-1"
  # any other provider configurations
}
resource "aws_ecr_repository" "get-giphy" {
  name = "get-giphy"
}
```

Now we can push our Node application image up to this repository. Click on the repository and click View push commands. A modal will appear with four commands you need to run locally in order to have your image pushed up to your repository:

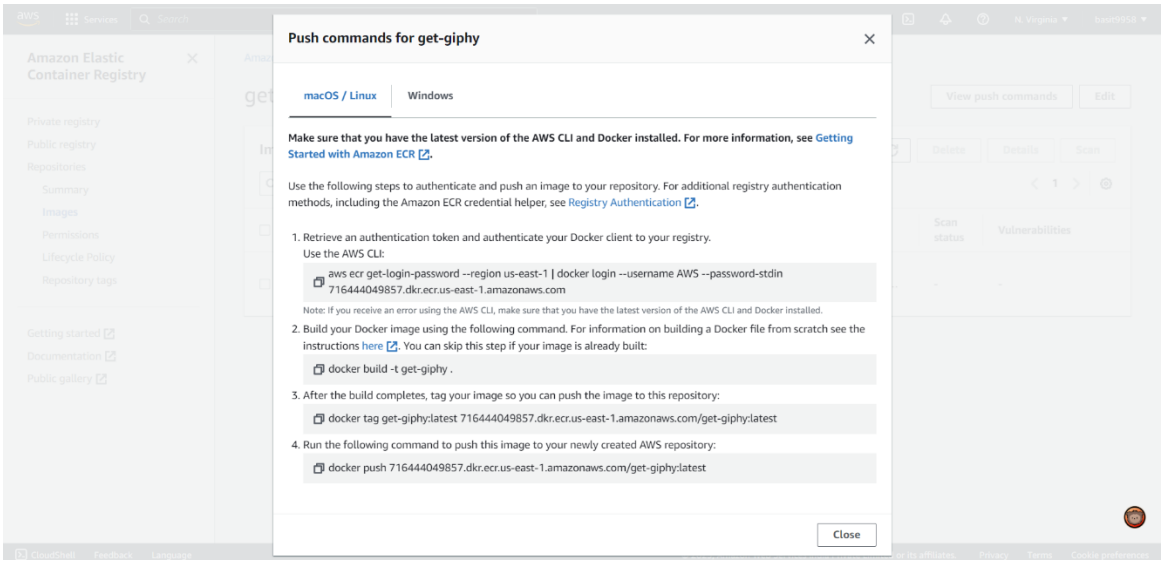

Fig 4.2 Push commands for get giphy

Once you have run these commands, you should see your pushed image in your repository:

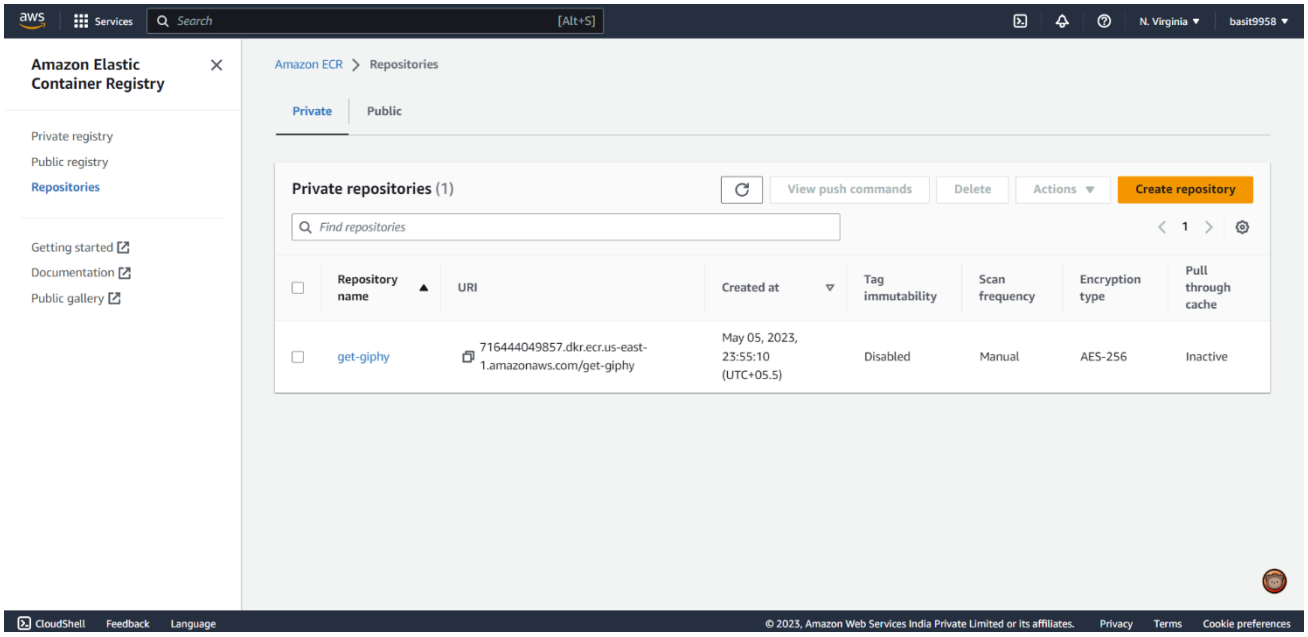

Fig 4.3 Repository

AWS ECS

Amazon Elastic Container Service (Amazon ECS) is a fully managed container orchestration service. AWS ECS is a fantastic service for running your containers. In this guide we will be using ECS Fargate, as this is a serverless compute service that allows you to run containers without provisioning servers.

ECS has three parts: clusters, services, and tasks.

Tasks are JSON files that describe how a container should be run. For example, you need to specify the ports and image location for your application. A service simply runs a specified number of tasks and restarts/kills them as needed. This has similarities to an auto-scaling group for EC2. A cluster is a logical grouping of services and tasks. This will become more clear as we build.

4.3 Create an AWS ECS Cluster

Navigate to the AWS ECS service and click Clusters. You should see a empty page:

Next, add this code to your terraform file and redeploy your infrastructure with terraform apply:

```
resource "aws_ecs_cluster" "my_cluster" {
  name = "my-cluster" # Naming the cluster
}
```

You should then see your new cluster:

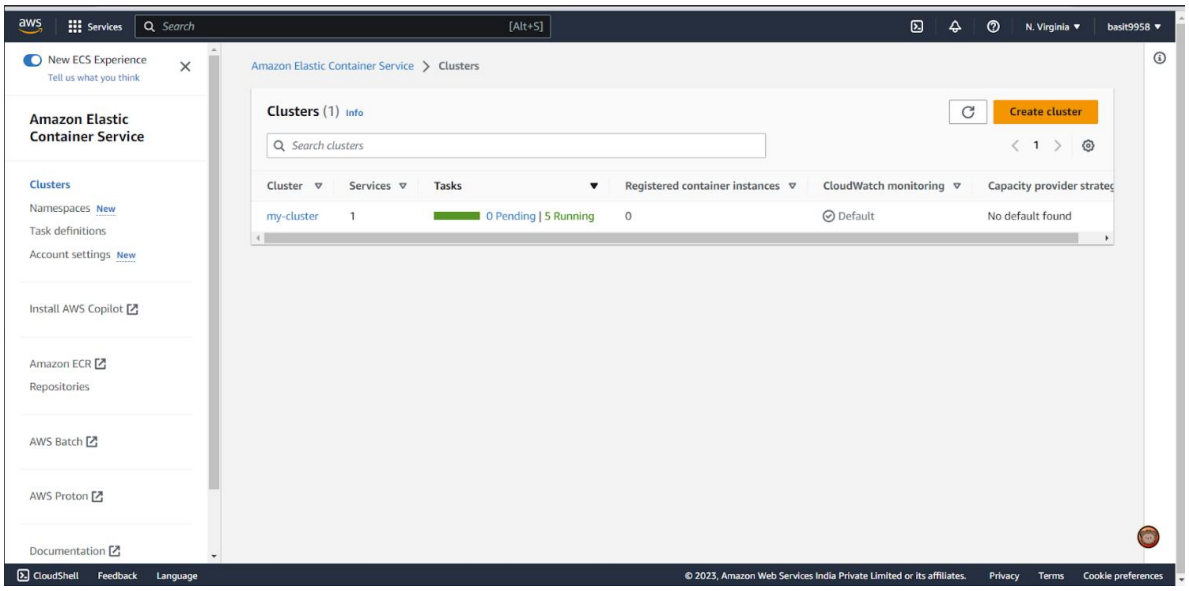

Fig 4.4 Clusters

4.4 Create an AWS ECS Task

Create a task to schedule a deployment inside cluster. Add the following commented code to your terraform script:

```
resource "aws_ecs_task_definition" "my_first_task" {
  family                   = "my-first-task" # Naming our first task
  container_definitions    = <<DEFINITION
  [
    {
      "name": "my-first-task",
      "image": "${aws_ecr_repository.my_first_ecr_repo.repository_url}",
      "essential": true,
      "portMappings": [
        {
          "containerPort": 3000,
          "hostPort": 3000
        }
      ],
      "memory": 512,
      "cpu": 256
    }
  ]
  DEFINITION
  requires_compatibilities = ["FARGATE"] # Stating that we are using ECS Fargate
  network_mode             = "awsvpc"    # Using awsvpc as our network mode as this is required f
  memory                   = 512         # Specifying the memory our container requires
  cpu                      = 256         # Specifying the CPU our container requires
  execution_role_arn       = "${aws_iam_role.ecsTaskExecutionRole.arn}"
}

resource "aws_iam_role" "ecsTaskExecutionRole" {
  name               = "ecsTaskExecutionRole"
  assume_role_policy = "${data.aws_iam_policy_document.assume_role_policy.json}"
}

data "aws_iam_policy_document" "assume_role_policy" {
  statement {
    actions = ["sts:AssumeRole"]

    principals {
      type        = "Service"
      identifiers = ["ecs-tasks.amazonaws.com"]
    }
  }
}

resource "aws_iam_role_policy_attachment" "ecsTaskExecutionRole_policy" {
  role       = "${aws_iam_role.ecsTaskExecutionRole.name}"
  policy_arn = "arn:aws:iam::aws:policy/service-role/AmazonECSTaskExecutionRolePolicy"
}
```

we specify the image by referencing the repository URL of our other terraform resource. Also notice how we provide the port mapping of 3000. We also create an IAM role so that tasks have the correct permissions to execute. If you click Task Definitions in AWS ECS, you should see your new task:

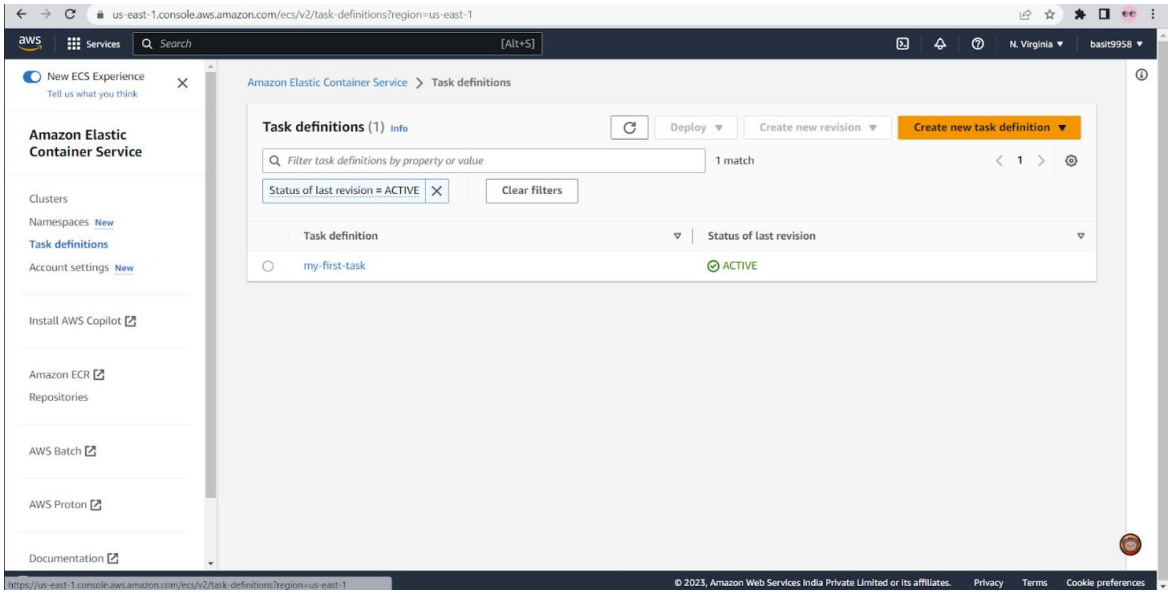

Fig 4.5 Task Definitions

4.5 Create an AWS ECS Service

Create the First Service, Now we have a cluster and a task definition. It is time that we spun up a few containers in our cluster through the creation of a service that will use our newly created task definition as a blueprint. If we examine the documentation in Terraform for an ECS service, we find that we need the following terraform code as a minimum:

```
resource "aws_ecs_service" "my_first_service" {
  name            = "my-first-service"         # Naming our first service
  cluster         = "${aws_ecs_cluster.my_cluster.id}"
     # Referencing our created Cluster
  task_definition = "${aws_ecs_task_definition.my_first_task.arn}"
     # Referencing the task our service will spin up
  launch_type     = "FARGATE"
  desired_count   = 3
   # Setting the number of containers we want deployed to 3
}
```

As we are using Fargate, our tasks need to specify that the network mode is awsvpc. As a result, we need to extend our service to include a network configuration. You may have not known it yet, but our cluster was automatically deployed into your account's default VPC. However, for a service, this needs to be explicitly stated, even if we wish to continue using the default VPC

and subnets. First, we need to create reference resources to the default VPC and subnets so that they can be referenced by our other resources:

```
# Providing a reference to our default VPC
resource "aws_default_vpc" "default_vpc" {
}

# Providing a reference to our default subnets
resource "aws_default_subnet" "default_subnet_a" {
  availability_zone = "us-east-1c"
}

resource "aws_default_subnet" "default_subnet_b" {
  availability_zone = "us-east-1e"
}

resource "aws_default_subnet" "default_subnet_c" {
  availability_zone = "us-east-1f"
}
```

Next, adjust your service to reference the default subnets:

```
resource "aws_ecs_service" "my_first_service" {
  name            = "my-first-service"
# Naming our first service
  cluster         = "${aws_ecs_cluster.my_cluster.id}"
# Referencing our created Cluster
  task_definition = "${aws_ecs_task_definition.my_first_task.arn}"
# Referencing the task our service will spin up
  launch_type     = "FARGATE"
  desired_count   = 3
# Setting the number of containers we want deployed to 3

  network_configuration {
    subnets          = ["${aws_default_subnet.default_subnet_a.id}",
"${aws_default_subnet.default_subnet_b.id}",
"${aws_default_subnet.default_subnet_c.id}"]
    assign_public_ip = true # Providing our containers with public IPs
  }
}
```

Once deployed, click on your cluster, we can see the service:

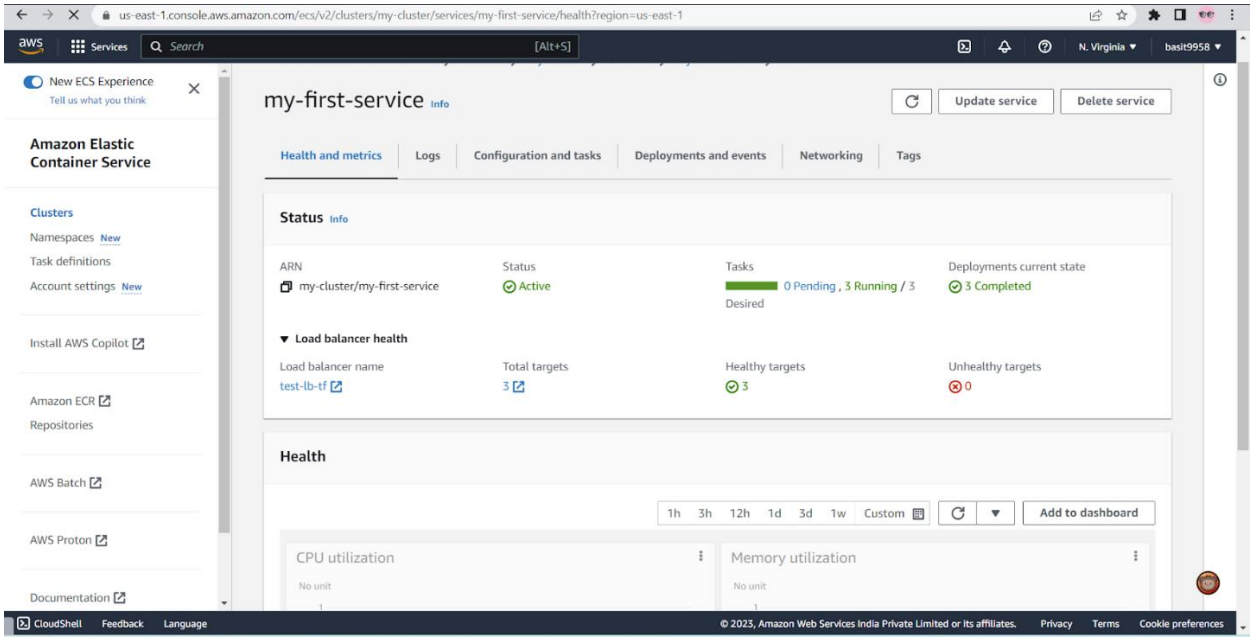

Fig 4.6 First Service

click on services and the Tasks tab, we can also see that three tasks/containers have been spun up:

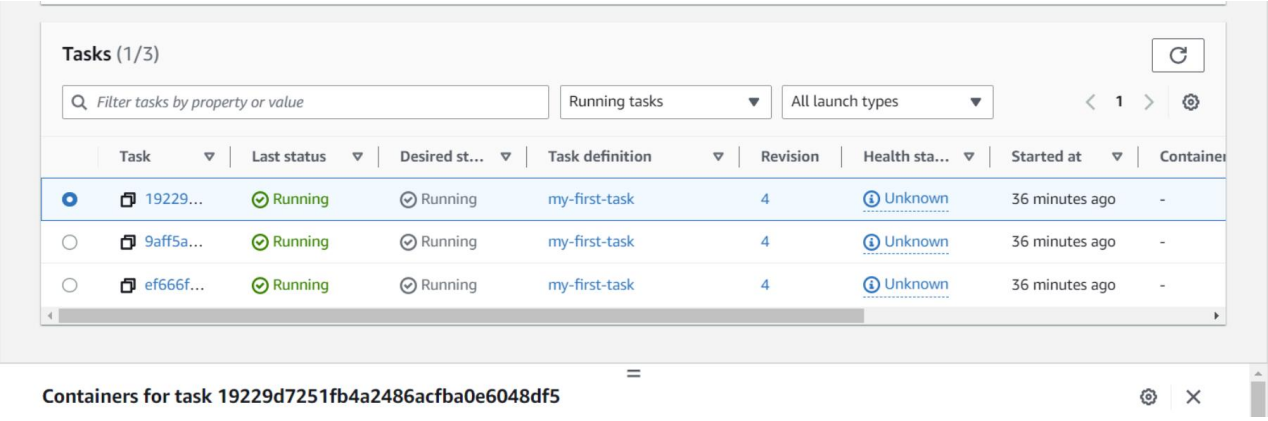

Fig 4.7 Tasks

4.6 Create a Load Balancer

The final step in this process is to create a load balancer through which we can access our containers. The idea is to have a single URL provided by our load balancer that, behind the scenes, will redirect our traffic to our underlying containers. Add the following commented terraform code:

```
resource "aws_alb" "application_load_balancer" {
  name               = "test-lb-tf" # Naming our load balancer
  load_balancer_type = "application"
  subnets = [ # Referencing the default subnets
    "${aws_default_subnet.default_subnet_a.id}",
    "${aws_default_subnet.default_subnet_b.id}",
    "${aws_default_subnet.default_subnet_c.id}"
  ]
  # Referencing the security group
  security_groups = ["${aws_security_group.load_balancer_security_group.id}"]
}

# Creating a security group for the load balancer:
resource "aws_security_group" "load_balancer_security_group" {
  ingress {
    from_port   = 80 # Allowing traffic in from port 80
    to_port     = 80
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"] # Allowing traffic in from all sources
  }

  egress {
    from_port   = 0 # Allowing any incoming port
    to_port     = 0 # Allowing any outgoing port
    protocol    = "-1" # Allowing any outgoing protocol
    cidr_blocks = ["0.0.0.0/0"] # Allowing traffic out to all IP addresses
  }
}
```

We also create a security group for the load balancer. This security group is used to control the traffic allowed to and from the load balancer.To direct traffic we need to create a target group and listener. Each target group is used to route requests to one or more registered targets (in our case, containers). When you create each listener rule, you specify a target group and conditions. Traffic is then forwarded to the corresponding target group. Create these with the following:

```
resource "aws_lb_target_group" "target_group" {
  name        = "target-group"
  port        = 80
  protocol    = "HTTP"
  target_type = "ip"
  vpc_id      = "${aws_default_vpc.default_vpc.id}" # Referencing the default VPC
  health_check {
    matcher = "200,301,302"
    path = "/"
  }
}

resource "aws_lb_listener" "listener" {
  load_balancer_arn = "${aws_alb.application_load_balancer.arn}" # Referencing our load balancer
  port              = "80"
  protocol          = "HTTP"
  default_action {
    type             = "forward"
    target_group_arn = "${aws_lb_target_group.target_group.arn}" # Referencing our tagrte group
  }
}
```

checking the Listeners tab of your load balancer, we see a listener that forwards traffic to your target group:

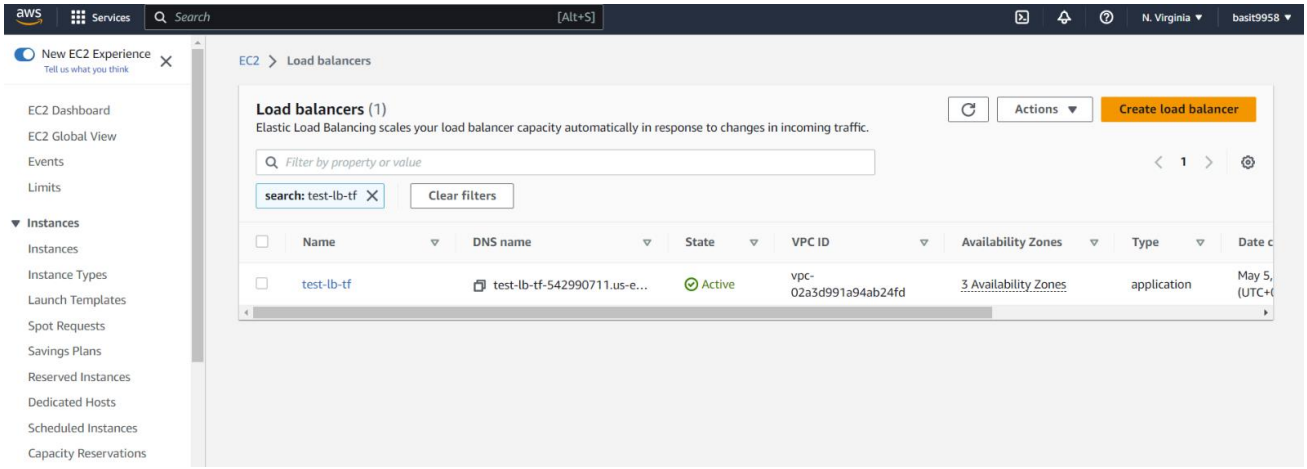

Fig 4.8 Load balancers

we need to create a security group for the ECS service that allows traffic only from the application load balancer security group:

```
resource "aws_ecs_service" "my_first_service" {
  name            = "my-first-service"
# Naming our first service
  cluster         = "${aws_ecs_cluster.my_cluster.id}"
# Referencing our created Cluster
  task_definition = "${aws_ecs_task_definition.my_first_task.arn}"
# Referencing the task our service will spin up
  launch_type     = "FARGATE"
  desired_count   = 3 # Setting the number of containers to 3

  load_balancer {
    target_group_arn = "${aws_lb_target_group.target_group.arn}"
# Referencing our target group
    container_name   = "${aws_ecs_task_definition.my_first_task.family}"
    container_port   = 3000 # Specifying the container port
  }

  network_configuration {
    subnets          = ["${aws_default_subnet.default_subnet_a.id}",
"${aws_default_subnet.default_subnet_b.id}",
"${aws_default_subnet.default_subnet_c.id}"]
    assign_public_ip = true
# Providing our containers with public IPs
    security_groups  =
["${aws_security_group.service_security_group.id}"]
# Setting the security group
  }
```

```
}


resource "aws_security_group" "service_security_group" {
  ingress {
    from_port = 0
    to_port   = 0
    protocol  = "-1"
    # Only allowing traffic in from the load balancer security group
    security_groups =
["${aws_security_group.load_balancer_security_group.id}"]
  }

  egress {
    from_port   = 0 # Allowing any incoming port
    to_port     = 0 # Allowing any outgoing port
    protocol    = "-1" # Allowing any outgoing protocol
    cidr_blocks = ["0.0.0.0/0"] # Allowing traffic out to all IP
addresses
  }
}
```

If we check target containers, they should now be healthy and we should also be able to access your containers through your load balancer DNS: test-lb-tf-542990711.us-east-1.elb.amazonaws.com

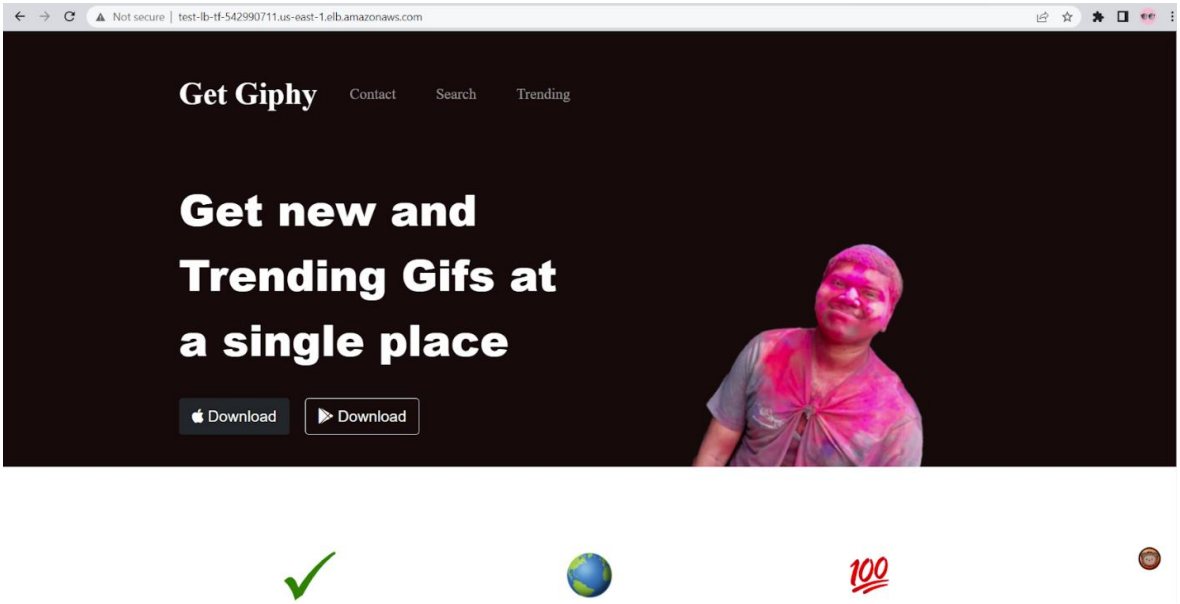

Fig 4.9 Deployed Webpage

CHAPTER 5

RESULTS AND DISCUSSIONS

5.1 Project Outcomes

The "Get-Giphy" web application has been deployed as a Docker image on AWS Elastic Container Registry (ECR) for efficient and reliable container management. The task executions, container services, and health monitoring of the application are being managed through Amazon Elastic Container Service (ECS). By utilizing these tools, the deployment process becomes faster, scalable, and more efficient. Additionally, ECS provides easy management of tasks, load balancing, and auto-scaling to ensure the availability of the application at all times. With ECS, we can also monitor the health of the containers, identify any issues or failures, and quickly respond to them. This architecture provides a stable and cost-effective solution for managing the deployment of the "Get-Giphy" application.

In addition to the previously mentioned tools, a load balancer has been set up to distribute incoming traffic to multiple instances of the "Get-Giphy" application, ensuring that the application is available and responsive to users at all times. The load balancer is configured to automatically scale up or down based on the traffic load, optimizing resource utilization and minimizing costs.

To manually scale up the "Get-Giphy" application, we can use ECS to increase the desired number of tasks. ECS will then automatically create new container instances to run the tasks, which will be registered with the load balancer and added to the pool of available instances. This will distribute the traffic load across the new instances, increasing the capacity of the application.

To manually scale down the application, we can use ECS to decrease the desired number of tasks. ECS will then automatically stop the specified number of tasks, which will deregister from the load balancer and be terminated. This will reduce the capacity of the application and free up resources.

It is important to note that scaling up or down the application should be done carefully, as it can affect the application's performance and cost. It is recommended to monitor the application's metrics, such as CPU utilization and network traffic, to determine when to scale up or down. Additionally, load testing the application can help identify its capacity limits and optimize its performance.
The security group implemented in the "Get-Giphy" project controls inbound and outbound traffic to and from the container instances running the application. The security group acts as a virtual firewall, allowing only the specified traffic to flow in and out of the instances, and blocking all other traffic.

The inbound rules of the security group are configured to allow traffic from specific IP addresses or CIDR ranges, as well as from specific ports. For example, the security group may allow HTTP traffic from the internet to reach the load balancer, which in turn forwards the traffic to the container instances running the application. Additionally, the security group may allow SSH traffic from a specific IP address or range to connect to the instances for administrative purposes.

5.2 Resultants Benefits

1. **Improved portability:** By containerizing the application with Docker, it becomes easier to deploy and run the same application on different environments, without worrying about dependencies and runtime compatibility issues.

2. **Simplified deployment:** With the Docker image hosted on AWS's ECR repository, deploying the application to various environments becomes a simple matter of pulling the image and running it on any Docker host.

3. **Scalability:** AWS's Elastic Container Service (ECS) can be used to manage the Docker containers running the "get-giphy" application. This allows for easy scaling of the application by increasing the number of containers running the application to meet demand.

4. **Isolation:** Running the "get-giphy" application in a Docker container provides a degree of isolation between the application and the host system. This can improve security and reduce the risk of conflicts with other applications running on the same host.

5. **Improved version control:** Docker images can be tagged and versioned, which allows for easy tracking and rollback of changes to the application.

In conclusion, hosting and running the "get-giphy" Docker image on AWS's ECR repository has successfully provide several benefits, including improved portability, simplified deployment, scalability, isolation, and improved version control.

5.3 Analyzing performance metrics

Responding to Incoming Traffic Using Load Balancers

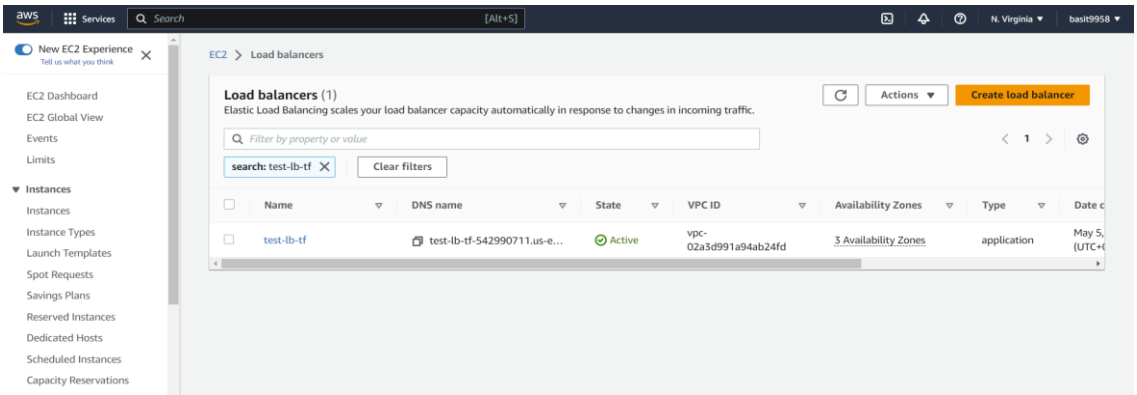

Fig 5.1 Load Balancers

Our project makes the usage of AWS Load Balancers, with which our website can drastically improve and deal with larger amounts of data traffic directed towards our website.AWS Load Balancers are a managed service provided by Amazon Web Services (AWS) that distribute incoming network traffic across multiple targets, such as Amazon EC2 instances, containers, IP addresses, and Lambda functions, to increase the availability and fault tolerance of applications.

The main functions of AWS Load Balancers include:

1. Distributing incoming traffic: Load balancers distribute incoming traffic across multiple targets in a way that ensures that each target receives a fair share of the traffic. This helps to improve the availability and fault tolerance of applications by ensuring that traffic is distributed evenly across all targets, and that no single target becomes overwhelmed with traffic.

2. Scaling applications: Load balancers can automatically scale the number of targets based on the demand for incoming traffic. For example, if the traffic to an application increases, the load balancer can automatically add more targets to handle the increased load. Similarly, if the traffic decreases, the load balancer can remove targets to reduce costs.

3. Health checking: Load balancers regularly check the health of each target to ensure that they are capable of serving traffic. If a target fails a health check, the load balancer can automatically remove it from the pool of available targets and redirect traffic to the healthy targets.

4. SSL termination: Load balancers can also handle SSL termination, which involves decrypting incoming SSL traffic and then forwarding it to the target in unencrypted form. This can help reduce the load on the target instances and improve the performance of the application.

5.4 Controlling flow of network traffic using VPCs

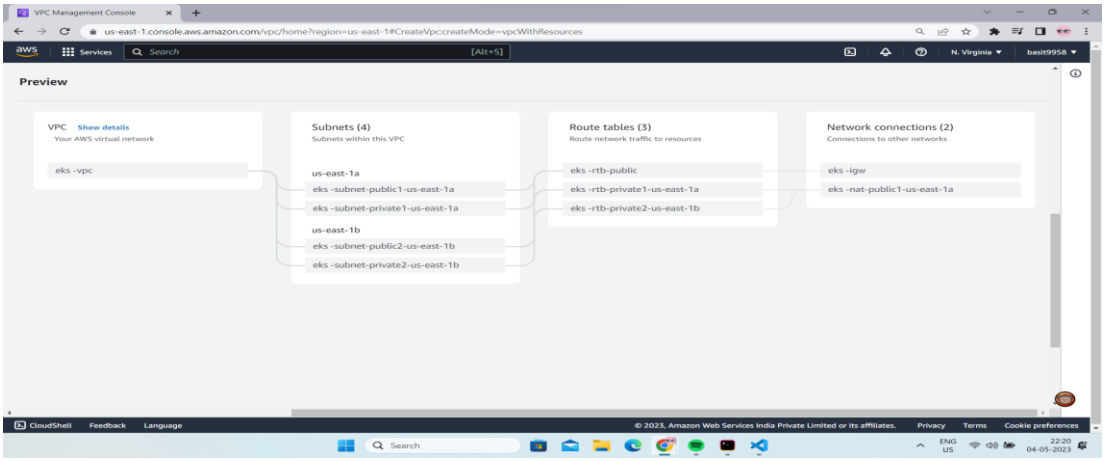

Fig 5.2 VPCs

AWS VPC stands for Amazon Web Services Virtual Private Cloud. It is a networking service that allows you to create a logically isolated section of the AWS Cloud where you can launch AWS resources in a virtual network that you define. With AWS VPC, you can create a private, secure, and scalable environment for your applications and services within the AWS Cloud.VPC allows you to easily scale your website as needed. You can use Auto Scaling groups to automatically adjust capacity based on traffic demand, and you can use Elastic Load Balancing to distribute traffic across multiple instances.

Our project uses AWS VPC, which in turn implements subnets and routing tables to control network traffic between resources in your VPC and the internet or other VPCs. Here's a brief overview of what subnets and routing tables represent:

1. **Subnets:** A subnet is a range of IP addresses within your VPC that can be used to launch resources like EC2 instances, RDS instances, and other services. Each subnet is associated with a specific Availability Zone (AZ), and resources launched in a subnet are assigned IP addresses from the subnet's IP address range. Subnets are used to partition your VPC's IP address range and provide isolation between resources in different subnets.

2. **Routing Tables:** A routing table is a set of rules that determines how network traffic is routed within your VPC and to destinations outside your VPC. Each subnet in your VPC is associated

with a routing table, which is used to determine the next hop for outbound traffic from resources in the subnet.

5.5 Manual scaling of Application

```
resource "aws_ecs_task_definition" "my_first_task" {
  family                   = "my-first-task" # Naming our first task
  container_definitions    = <<DEFINITION
  [
    {
      "name": "my-first-task",
      "image": "${aws_ecr_repository.my_first_ecr_repo.repository_url}",
      "essential": true,
      "portMappings": [
        {
          "containerPort": 3000,
          "hostPort": 3000
        }
      ],
      "memory": 512,
      "cpu": 256
    }
  ]
  DEFINITION
  requires_compatibilities = ["FARGATE"] # Stating that we are using ECS Fargate
  network_mode             = "awsvpc"    # Using awsvpc as our network mode as this is required f
  memory                   = 512         # Specifying the memory our container requires
  cpu                      = 256         # Specifying the CPU our container requires
  execution_role_arn       = "${aws_iam_role.ecsTaskExecutionRole.arn}"
}
```

In the given code snippet, memory and CPU are two parameters used to define the resource requirements of the task that will be run inside a container using AWS Elastic Container Service (ECS) with Fargate launch type.

Memory is a parameter that specifies the amount of memory that is required by the container to run smoothly without any out of memory errors. The value is specified in MiB (megabytes) and the minimum value for Fargate tasks is 256 MiB. In the code snippet, the memory parameter is set to 512 MiB.

CPU is another parameter that specifies the amount of CPU resources that the container requires to run. This value is specified in units and 1 vCPU is equivalent to 1024 units. The minimum value for Fargate tasks is 256 CPU units. In the code snippet, the CPU parameter is set to 256 units.

The memory and CPU parameters are used to determine the amount of resources that are required by a task. These parameters help to ensure that the task has the necessary resources to run without any issues. They also help to optimize resource usage and prevent overprovisioning, which can lead to unnecessary costs.

In order to scale up or down the application using a declarative approach, the memory and CPU parameters can be adjusted accordingly. For example, if the application is experiencing higher traffic and requires more resources, the memory and CPU parameters can be increased to ensure that the application has the necessary resources to run smoothly. Similarly, if the application is experiencing lower traffic, the memory and CPU parameters can be decreased to optimize resource usage and reduce costs.

When deploying the task, the ECS service scheduler ensures that the desired number of tasks are running at all times. This enables us to achieve high availability for our application and also provides a way to scale up or down the application based on the traffic or demand.

By setting the desired_container_count parameter to 3, we are indicating that we want to have three instances of the task running simultaneously. This means that the ECS service scheduler will launch three tasks of the specified task definition and will try to keep them running even if one or more instances fail or become unavailable.

Overall, memory and CPU parameters are important considerations when running containers in ECS Fargate as they help to ensure that the containers have the necessary resources to run smoothly and efficiently. By adjusting these parameters based on the application's requirements, it is possible to scale up or down the application using a declarative approach.

5.6 Metric Analysis

Hosting Website on AWS ERS	Hosting Website Normally
Increased Control Over Scalability through manual required declarations.	Limited to No Control over Scalability
Enhanced Performance	Decreased Performance
Secure and Isolated Environment where access to resources is greatly monitored and controlled.	Challenging to manage and monitor the website's traffic
Website Health monitored and maintained frequently,	Difficult to ensure that the website is always available and responsive
Load balancers can also provide security by filtering traffic and identifying potential threats	Inability to restrict traffic access to webpage
Harder barrier to host and deploy websites. More suited for hosting professional websites for organizations.	Easier to develop and deploy websites through third party hosting sites.

Table 5.1 Metric Analysis

In summary, the choice of hosting service should be based on the specific requirements and goals of the website, and AWS ECS can be a good option for high-traffic and scalable websites.

If the website requires high scalability, availability, and reliability, then hosting it on AWS ECS with Load Balancers and Virtual Private Cloud (VPC) can provide several benefits such as automatic scaling, traffic routing, and increased security. Additionally, AWS ECS provides flexibility in managing Docker containerized applications, making it easier to update, deploy, and roll back the website as needed.However, if the website has lower traffic and does not require high scalability or special requirements, then a traditional web hosting service may be a more cost-effective solution.

CHAPTER 6

CONCLUSIONS AND RECOMMENDATIONS

6.1 Security Measures Implemented

Adding security Group to load balancers

```
# Creating a security group for the load balancer:
resource "aws_security_group" "load_balancer_security_group" {
  ingress {
    from_port   = 80 # Allowing traffic in from port 80
    to_port     = 80
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"] # Allowing traffic in from all sources
  }

  egress {
    from_port   = 0 # Allowing any incoming port
    to_port     = 0 # Allowing any outgoing port
    protocol    = "-1" # Allowing any outgoing protocol
    cidr_blocks = ["0.0.0.0/0"] # Allowing traffic out to all IP addresses
  }
}
```

In the above Terraform code, a security group is created for the load balancer. The security group defines the network traffic rules that control inbound and outbound traffic for the load balancer.

The `ingress` block in the security group allows traffic to come in from port 80 (HTTP), and allows traffic from any source IP address (`0.0.0.0/0`).

The `egress` block in the security group allows any outgoing traffic to any destination IP address (`0.0.0.0/0`). This means that the load balancer is allowed to send traffic to any destination.

By defining these security group rules, we can ensure that the load balancer only allows traffic that is necessary for our application and blocks all other traffic. This adds an extra layer of security to our application, as we can control which traffic is allowed to access our application.

Adding Targets groups and Listeners to Load Balancers

```
resource "aws_lb_target_group" "target_group" {
  name        = "target-group"
  port        = 80
  protocol    = "HTTP"
  target_type = "ip"
  vpc_id      = "${aws_default_vpc.default_vpc.id}" # Referencing the default VPC
  health_check {
    matcher = "200,301,302"
    path = "/"
  }
}

resource "aws_lb_listener" "listener" {
  load_balancer_arn = "${aws_alb.application_load_balancer.arn}" # Referencing our load balancer
  port              = "80"
  protocol          = "HTTP"
  default_action {
    type             = "forward"
    target_group_arn = "${aws_lb_target_group.target_group.arn}" # Referencing our tagrte group
  }
}
```

The above Terraform code creates an AWS target group and an AWS listener for an application load balancer. The target group specifies the port, protocol, target type, and health check settings for the group. It also references the default VPC in which it will operate.

The listener specifies the load balancer ARN, port, and protocol, as well as the default action, which is to forward traffic to the target group. The target group ARN is referenced in the default action.

Overall, these resources define the settings for load balancing traffic and forwarding it to the specified target group.
In the final implementation, we are creating an ECS service that references the ECS cluster, task definition, target group, and security group.

We specify the desired count of containers as 3 and use Fargate as our launch type. We also provide our containers with public IPs, and set the security group to the one we have created earlier.

Inside the service, we specify the load balancer and target group, and also mention the container name and port number. Additionally, we have created a network configuration where we set the subnets and assign public IPs to our containers.

Finally, we create a security group for our service, which only allows traffic in from the load balancer security group. The egress rules allow any outgoing traffic to all IP addresses.

In summary, the above code implements a load balancer with security groups, target groups, and listeners in the ECS cluster. This configuration provides an efficient and secure way to manage and scale containerized applications in AWS.

Added Security Measures

1. Virtual Private Cloud (VPC): AWS VPC allows users to create a private network in the AWS Cloud, which is isolated from other networks. This means that the Docker containers running on the ERS are isolated from other networks and are protected against external threats.

2. Load Balancers: Load balancers are used to distribute traffic across multiple instances of the Docker containers. This provides high availability and scalability, ensuring that the application can handle large volumes of traffic without compromising performance or security.

3. Access Control: Access control is implemented using IAM (Identity and Access Management) to restrict access to the Docker containers running on ERS. Users can create IAM roles and policies to control access to the resources, allowing only authorized users to interact with the containers.

4. Encryption: AWS ERS supports the use of SSL/TLS encryption to protect data in transit between clients and the Docker containers running on ERS.

5. Network Security: Security groups and network ACLs are used to restrict inbound and outbound traffic to and from the Docker containers. This helps to ensure that only authorized traffic is allowed in and out of the containers.

6. Monitoring: AWS provides several monitoring tools, such as Amazon CloudWatch, which can be used to monitor the Docker containers running on ERS for security events, performance issues, and other metrics. This helps to detect and respond to potential security threats and maintain the security posture of the environment.

6.2 Project Findings

Hosting a website on AWS ERS (Elastic Container Service) using VPC (Virtual Private Cloud) and Load Balancers provides several benefits, including:

1. **High availability:** Load balancers distribute traffic across multiple instances of the Docker containers, providing high availability and scalability, ensuring that the application can handle large volumes of traffic without compromising performance or security.

2. **Security:** AWS ERS provides several security measures, including VPC, access control, encryption, network security, and monitoring to protect containerized applications running on the platform from external threats and ensure data privacy and confidentiality.

3. **Cost-effective:** AWS ERS is a cost-effective solution for hosting containerized applications, as users only pay for the resources they use, and can easily scale up or down based on the demand.

4. **Easy management:** AWS ERS allows users to easily manage and deploy containerized applications, using the Amazon ECS console, AWS CLI (Command Line Interface), or SDKs (Software Development Kits).

5. **Integration with other AWS services:** AWS ERS integrates seamlessly with other AWS services, such as Amazon S3, Amazon RDS, and Amazon CloudFront, allowing users to easily incorporate these services into their containerized applications.

In summary, hosting a website on AWS ERS using VPC and Load Balancers provides several benefits, including high availability, security, cost-effectiveness, easy management, and integration with other AWS services.

6.3 Future Addons/Improvements

There are several future addons and improvements that can be made to this project, including:

1. **Autoscaling:** Autoscaling can be added to the Load Balancers to automatically scale the number of instances of the Docker containers based on the demand. This can help to optimize resource usage and ensure that the application can handle sudden spikes in traffic.

2. **Service discovery:** Service discovery can be added to the Load Balancers to automatically discover and route traffic to the appropriate instances of the Docker containers based on the domain name. This can help to simplify the deployment and management of containerized applications.

3. **Container orchestration:** Container orchestration tools, such as Kubernetes or Amazon ECS, can be used to automate the deployment, scaling, and management of containerized applications. This can help to improve the efficiency and reliability of the application deployment process.

4. **Database optimization:** Database optimization techniques, such as caching and indexing, can be used to improve the performance and scalability of the database used by the application.

5. **Content delivery network (CDN):** A CDN can be used to cache and serve static content, such as images and videos, from edge locations around the world, improving the performance and user experience of the website.

6. **Continuous integration and deployment (CI/CD):** A CI/CD pipeline can be set up to automate the build, testing, and deployment of the containerized application, ensuring that new changes are deployed quickly and reliably.

In summary, there are several future addons and improvements that can be made to this project, including autoscaling, service discovery, container orchestration, database optimization, CDN, and CI/CD. These addons and improvements can help to improve the performance, scalability, and reliability of the application and provide a better user experience for the end-users.

REFERENCES

1. Amazon Elastic Container Registry (ECR): https://aws.amazon.com/ecr/

2. Elastic Load Balancing: https://aws.amazon.com/elasticloadbalancing/

3. Amazon Virtual Private Cloud (VPC): https://aws.amazon.com/vpc/

4. AWS ECS Developer Guide: https://docs.aws.amazon.com/AmazonECS/latest/developerguide/Welcome.html

5. Amazon EKS documentation: https://aws.amazon.com/eks/getting-started/

6. Deploying a Docker container to Amazon ECS: https://aws.amazon.com/getting-started/hands-on/deploy-docker-containers/

7. AWS Fargate Developer Guide: https://docs.aws.amazon.com/AmazonECS/latest/developerguide/AWS_Fargate.html