DEMO DOCUMENTATION: DEPLOYING APPLICATIONS ON AWS ECS (FARGATE VS EC2)

Index

This documentation aims to empower DevOps and Cloud Engineers with practical knowledge to choose and deploy ECS-based architectures effectively.

1. INTRODUCTION

This document provides a detailed walkthrough of two approaches to deploying containerized applications using AWS Elastic Container Service (ECS):

- Fargate (serverless compute engine)
- EC2 (provisioned compute instances)

2. OBJECTIVE

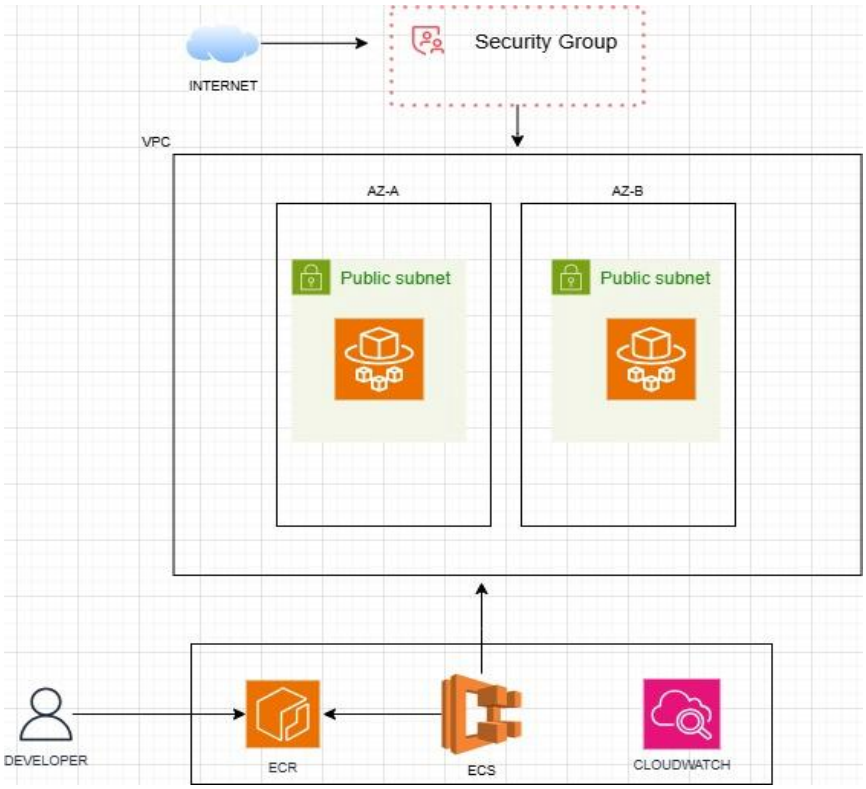
To compare resource management, scalability, and operational overhead between AWS ECS Fargate and ECS EC2 launch types.

3. PREREQUISITES

- AWS Account with appropriate permissions
- Dockerized application (e.g., Node.js or Flask app)
- AWS CLI & Terraform installed with aws credentials configured

4. ARCHITECTURE OVERVIEW

ECS with Fargate:



The workflow operates as follows:

- Developer pushes image to ECR.

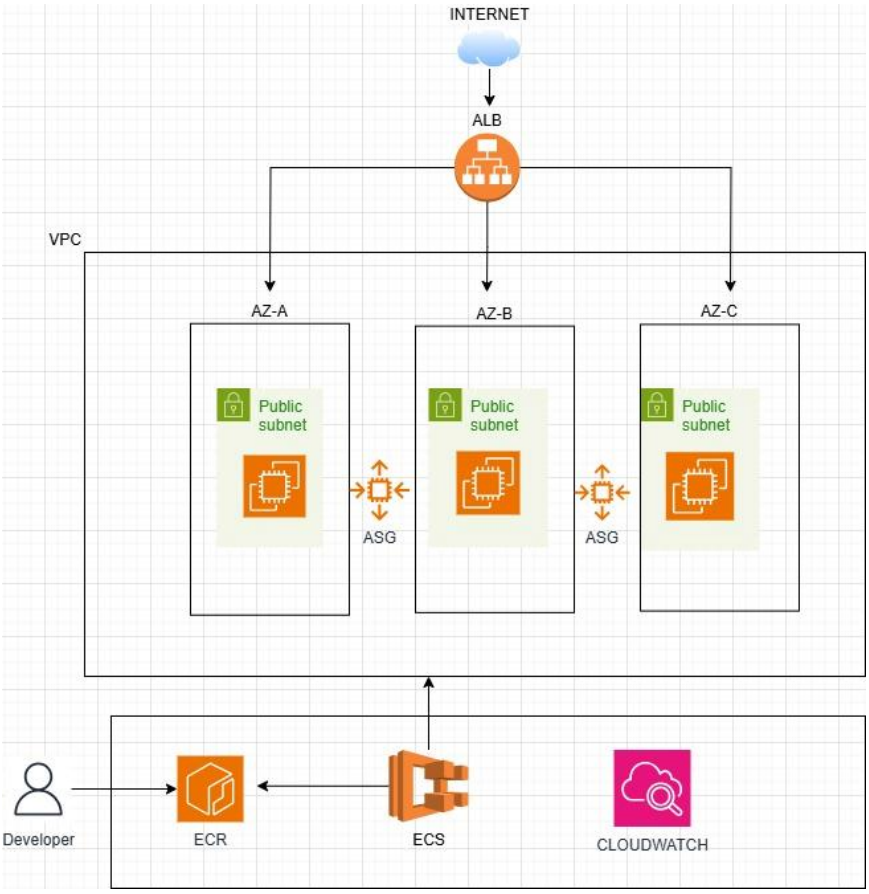
- ECS Service deploys and manages tasks using that image.

- Tasks run in Fargate with networking provided by the default VPC/subnets.

- Security groups control access to the tasks.

- Container logs are sent to CloudWatch.

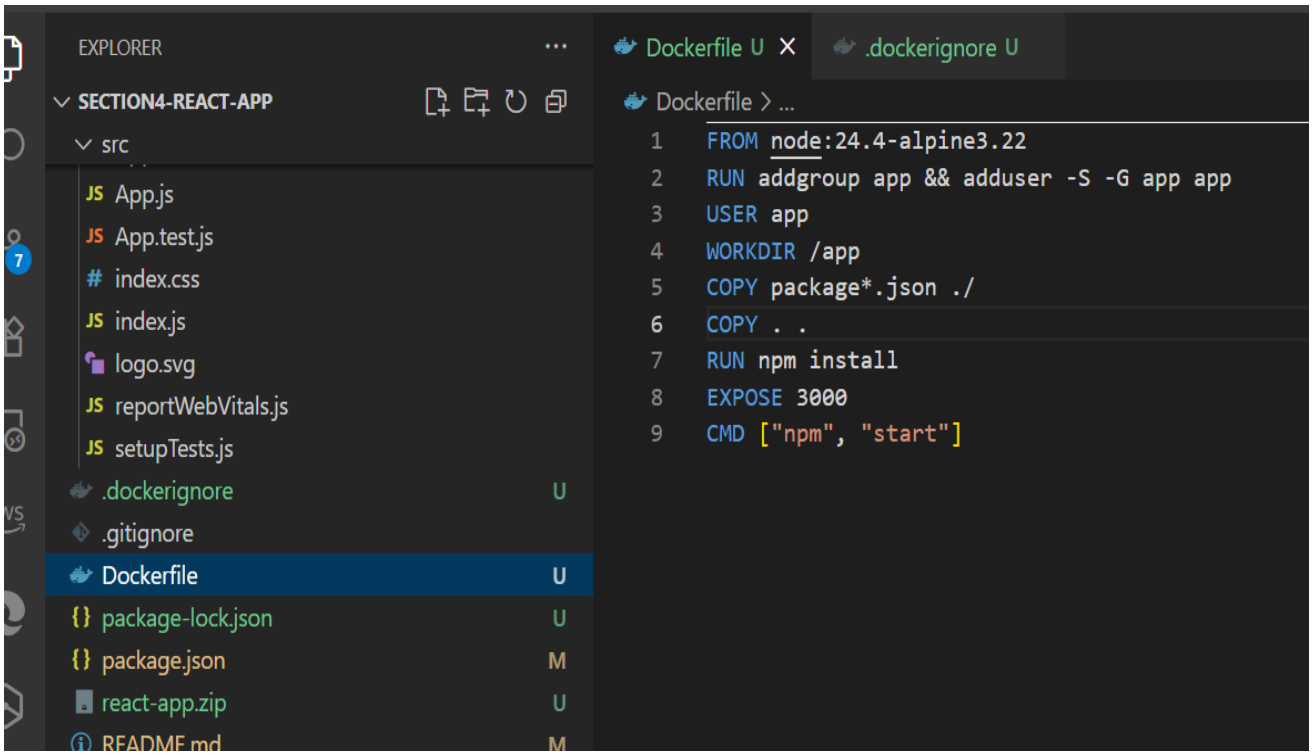
ECS with EC2:



- Developer pushes image to ECR.
- ECS agent on EC2 pulls image from ECR
  Creates containers based on task definition
  Manages container lifecycle (start, stop, restart)
  Registers container IPs with ALB target group

- ALB distributes traffic to various instances in subnets.
- ASG to scale instances when there is no capacity for new containers.
- CloudWatch monitors everything
  Container logs → CloudWatch Logs (/ecs/react-app)
  ECS metrics → Service health, task count
  EC2 metrics → CPU, memory usage
  ALB metrics → Request count, response times

This is the Dockerfile which was used to build the react app on node:alpine and exposed on port 3000 which will be used in the deployment.



5.DEPLOYMENT OF A CONTAINERIZED APPLICATION ON AMAZON ECS (EC2 LAUNCH TYPE) USING TERRAFORM

Overview

The snippets of Terraform configuration below automates the provisioning and deployment of a containerized application to **Amazon ECS using the EC2 launch type**. The architecture utilizes a fully managed ECS Cluster running on EC2 instances, with scalable backend compute, application-level load balancing, secure networking, and logging via AWS CloudWatch.

Infrastructure Components Deployed

VPC & Subnets

```hcl
# Get default VPC
data "aws_vpc" "default" {
  default = true
}

# Get default subnets (exclude us-east-1e)
data "aws_subnets" "default" {
  filter {
    name   = "vpc-id"
    values = [data.aws_vpc.default.id]
  }

  filter {
    name   = "availability-zone"
    values = ["us-east-1a", "us-east-1b", "us-east-1c", "us-east-1d", "us-east-1f"]
  }
}
```

- The configuration dynamically fetches the default VPC and a set of available subnets across selected availability zones in us-east-1.
- Ensures high availability and proper distribution of resources.
- us-east-1e was excluded because the type of instance being provisioned wasn't available in that Availability zone.

ECR Repository

```hcl
# ECR Repository
resource "aws_ecr_repository" "app" {
  name         = var.app_name
  force_delete = true
}
```

- Creates an Amazon Elastic Container Registry (aws_ecr_repository.app) to store the Docker image (:latest tag assumed in this demo).
- Enables force_delete to automatically delete any images in the repository during teardown.

ECS Cluster

```hcl
# ECS Cluster
resource "aws_ecs_cluster" "main" {
  name = "${var.app_name}-cluster"
}
```

- An ECS cluster (aws_ecs_cluster.main) is created to host EC2 container instances.

IAM Roles

```hcl
# IAM Role for ECS Task Execution
resource "aws_iam_role" "ecs_execution_role" {
  name = "${var.app_name}-ecs-execution-role"

  assume_role_policy = jsonencode({
    Version = "2012-10-17"
    Statement = [
      {
        Action = "sts:AssumeRole"
        Effect = "Allow"
        Principal = {
          Service = "ecs-tasks.amazonaws.com"
        }
      }
    ]
  })
}

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

```hcl
# IAM Role for ECS EC2 instances
resource "aws_iam_role" "ecs_instance_role" {
  name = "${var.app_name}-ecs-instance-role"

  assume_role_policy = jsonencode({
    Version = "2012-10-17"
    Statement = [
      {
        Action = "sts:AssumeRole"
        Effect = "Allow"
        Principal = {
          Service = "ec2.amazonaws.com"
        }
      }
    ]
  })
}

resource "aws_iam_role_policy_attachment" "ecs_instance_role_policy" {
  role       = aws_iam_role.ecs_instance_role.name
  policy_arn = "arn:aws:iam::aws:policy/service-role/AmazonEC2ContainerServiceforEC2Role"
}

resource "aws_iam_instance_profile" "ecs_instance_profile" {
  name = "${var.app_name}-ecs-instance-profile"
  role = aws_iam_role.ecs_instance_role.name
}
```

Creates two roles:

- **Execution Role** for ECS tasks to interact with AWS services like ECR and CloudWatch.
- **Instance Role** for ECS EC2 instances to register into the cluster and manage containers.

CloudWatch Logging

```hcl
# CloudWatch Log Group
resource "aws_cloudwatch_log_group" "app" {
  name              = "/ecs/${var.app_name}"
  retention_in_days = 1
}
```

- A log group (aws_cloudwatch_log_group.app) is provisioned for centralized logging from ECS containers.

EC2 Launch Infrastructure

```hcl
# Get latest ECS-optimized AMI
data "aws_ami" "ecs_optimized" {
  most_recent = true
  owners      = ["amazon"]

  filter {
    name   = "name"
    values = ["amzn2-ami-ecs-hvm-*-x86_64-ebs"]
  }
}

# Launch Template
resource "aws_launch_template" "ecs" {
  name_prefix   = "${var.app_name}-ecs"
  image_id      = data.aws_ami.ecs_optimized.id
  instance_type = "t3.micro"

  vpc_security_group_ids = [aws_security_group.ecs_instances.id]

  iam_instance_profile {
    name = aws_iam_instance_profile.ecs_instance_profile.name
  }

  user_data = base64encode(<<-EOF
    #!/bin/bash
    echo ECS_CLUSTER=${aws_ecs_cluster.main.name} >> /etc/ecs/ecs.config
  EOF
  )

  tag_specifications {
    resource_type = "instance"
    tags = {
      Name = "${var.app_name}-ecs-instance"
    }
  }
```

```hcl
# Auto Scaling Group
resource "aws_autoscaling_group" "ecs" {
  name                = "${var.app_name}-ecs-asg"
  vpc_zone_identifier = data.aws_subnets.default.ids
  health_check_type   = "ELB"
  min_size            = 1
  max_size            = 3
  desired_capacity    = var.desired_count

  launch_template {
    id      = aws_launch_template.ecs.id
    version = "$Latest"
  }

  tag {
    key                 = "AmazonECSManaged"
    value               = true
    propagate_at_launch = false
  }
}
```

- A **Launch Template** is configured to spin up EC2 instances using the ECS-optimized Amazon Linux 2 AMI. The user data in the launch template appends the name of the cluster to an ecs.config file upon launching an instance, the ECS agent reads this file and registers with the cluster. Without the user data, EC2 instances wouldn't know which ECS cluster to join.

- An **Auto Scaling Group (ASG)** is used to maintain the desired number of ECS EC2 instances (min 1, max 3), desired capacity was set as a variable which is 1. The ASG is triggered by ECS automatically when it tries to place a container and there is no capacity on the EC2 to accommodate the container.

Security Groups

```
# Security Group for ALB
resource "aws_security_group" "alb" {
  name_prefix = "${var.app_name}-alb"
  vpc_id      = data.aws_vpc.default.id

  ingress {
    from_port   = 80
    to_port     = 80
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  egress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

```
# Security Group for ECS EC2 instances
resource "aws_security_group" "ecs_instances" {
  name_prefix = "${var.app_name}-ecs-instances"
  vpc_id      = data.aws_vpc.default.id

  ingress {
    from_port   = 22
    to_port     = 22
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  egress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

```
# Security Group for ECS Tasks (awsvpc mode)
resource "aws_security_group" "ecs_tasks" {
  name_prefix = "${var.app_name}-ecs-tasks"
  vpc_id      = data.aws_vpc.default.id

  ingress {
    from_port       = var.container_port
    to_port         = var.container_port
    protocol        = "tcp"
    security_groups = [aws_security_group.alb.id]
  }

  egress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

- Created for:
    - The Application Load Balancer (ALB)

      Purpose: Controls traffic to/from the Application Load Balancer.

      Allows: HTTP (port 80) from the internet.

      Outbound: All traffic to anywhere (needed to reach containers).
    - ECS EC2 instances (includes SSH access)

      Purpose: Controls traffic to/from the EC2 host instances.

      Allows: SSH (port 22) for admin access (restrict to specific IP ranges for better Security).

      Outbound: All traffic to anywhere (needed for updates, container pulls).

- ECS tasks (for container communication)

  Purpose: Controls traffic to/from the containers themselves.

  Allows: Traffic on container port (3000) ONLY from the ALB.

  Outbound: All traffic to anywhere (needed for API calls, dependencies).

This setup provides defense-in-depth, containers can only be accessed through the ALB, not directly from the internet, creating a secure architecture.

Application Load Balancer (ALB)

```hcl
# Application Load Balancer
resource "aws_lb" "app" {
  name               = "${var.app_name}-alb"
  internal           = false
  load_balancer_type = "application"
  security_groups    = [aws_security_group.alb.id]
  subnets            = data.aws_subnets.default.ids
}

resource "aws_lb_target_group" "app" {
  name        = "react-app-tg-updated"
  port        = var.container_port
  protocol    = "HTTP"
  vpc_id      = data.aws_vpc.default.id
  target_type = "ip"

  health_check {
    enabled             = true
    healthy_threshold   = 2
    interval            = 30
    matcher             = "200"
    path                = "/"
    port                = "traffic-port"
    protocol            = "HTTP"
    timeout             = 5
    unhealthy_threshold = 2
  }

  lifecycle {
    create_before_destroy = true
  }
}
```

```hcl
resource "aws_lb_listener" "app" {
  load_balancer_arn = aws_lb.app.arn
  port              = "80"
  protocol          = "HTTP"

  default_action {
    type             = "forward"
    target_group_arn = aws_lb_target_group.app.arn
  }
}
```

- An **ALB** is created to expose the application on port 80.
- Health checks are defined to route traffic only to healthy ECS containers.
- A listener is added to forward HTTP traffic to the appropriate target group.

ECS Task Definition

```
# ECS Task Definition
resource "aws_ecs_task_definition" "app" {
  family                   = var.app_name
  network_mode             = "awsvpc"
  requires_compatibilities = ["EC2"]
  execution_role_arn       = aws_iam_role.ecs_execution_role.arn

  container_definitions = jsonencode([
    {
      name   = var.app_name
      image  = "${aws_ecr_repository.app.repository_url}:latest"
      memory = 512

      portMappings = [
        {
          containerPort = var.container_port
          protocol      = "tcp"
        }
      ]

      logConfiguration = {
        logDriver = "awslogs"
        options = {
          "awslogs-group"         = aws_cloudwatch_log_group.app.name
          "awslogs-region"        = var.aws_region
          "awslogs-stream-prefix" = "ecs"
        }
      }

      essential = true
    }
  ])
}
```

- Defines the Docker container configuration using:
    - The ECR image
    - Container port mapping
    - Log configuration
- Network mode is set to awsvpc, allowing the task to get its own ENI(Elastic Network Interface) and IP address. This helps with direct vpc connectivity, integrates with ALB by setting target type= “ip” on the ALB (ALB connects directly to container IPs) and better security isolation (task-level security groups)

ECS Service

```
# ECS Service
resource "aws_ecs_service" "app" {
  name            = "${var.app_name}-service"
  cluster         = aws_ecs_cluster.main.id
  task_definition = aws_ecs_task_definition.app.arn
  desired_count   = var.desired_count
  launch_type     = "EC2"

  network_configuration {
    subnets         = data.aws_subnets.default.ids
    security_groups = [aws_security_group.ecs_tasks.id]
  }

  load_balancer {
    target_group_arn = aws_lb_target_group.app.arn
    container_name   = var.app_name
    container_port   = var.container_port
  }

  depends_on = [aws_lb_listener.app]
}

```

- Launches the defined number of ECS tasks on EC2 instances.
- Registers the containers with the target group attached to the ALB.
- Manages scaling and deployment of the application in the ECS cluster.

Service Launch:

  - The ECS service pulls the image from ECR and launches it in containers across the EC2 instances.
  - The ALB routes external HTTP traffic to healthy containers based on the defined listener and target group.

Deployment Flow Summary

Infrastructure Provisioning:

- Run terraform apply to provision VPC-related data sources, ECS Cluster, IAM roles, security groups, load balancer, and EC2 instances.

Docker Image Handling:

After terraform apply, run the following commands to build and push your image to ECR. These commands can be added to the outputs file in the terraform configuration so you get the details of the ECR url and respective variables. The build command can be ignored if the image is already built. You can build and start the container locally before pushing it to the container registry.

```
output "docker_commands" {
  description = "Commands to build and push Docker image"
  value = <<-EOT
    # 1. Get ECR login token
    aws ecr get-login-password --region ${var.aws_region} | docker login --username AWS --password-stdin ${aws_ecr_repository.app.repository_url}

    # 2. Build Docker image
    docker build -t ${var.app_name} .

    # 3. Tag image for ECR
    docker tag ${var.app_name}:latest ${aws_ecr_repository.app.repository_url}:latest

    # 4. Push image to ECR
    docker push ${aws_ecr_repository.app.repository_url}:latest


  EOT
```

Benefits of This Architecture

- **Elastic Scaling**: Easily adjust the desired count of ECS tasks or EC2 instances.
- **Security**: Isolated container networking, IAM-based access controls, and granular security groups.
- **Observability**: Real-time log streaming to AWS CloudWatch for diagnostics.

Deployment of a Containerized Application on Amazon ECS Fargate Using Terraform

Overview

The snippets of Terraform configuration below provisions a **serverless containerized application** using **Amazon ECS Fargate**, eliminating the need to manage EC2 infrastructure. It sets up the networking, security, IAM roles, ECS Cluster, Task Definition, and ECS Service to run your container in a secure, scalable, and cost-efficient manner.

Key Components and Workflow

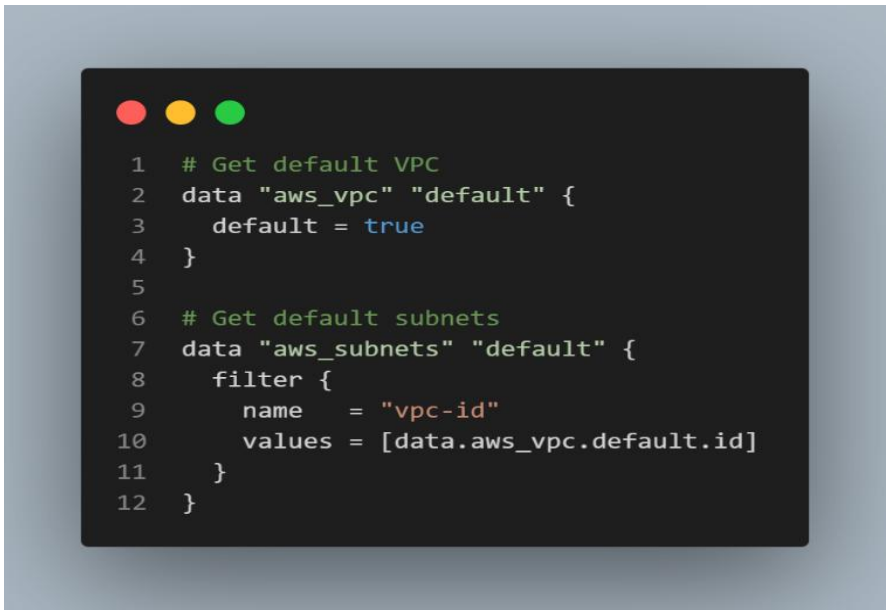
Provider & Region Configuration

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 5.0"
    }
  }
}

provider "aws" {
  region = var.aws_region
}
```

- Declares the **AWS provider** and specifies the target AWS region from a variable (var.aws_region).
- Ensures compatibility with the required AWS provider version ~> 5.0.

Networking Setup

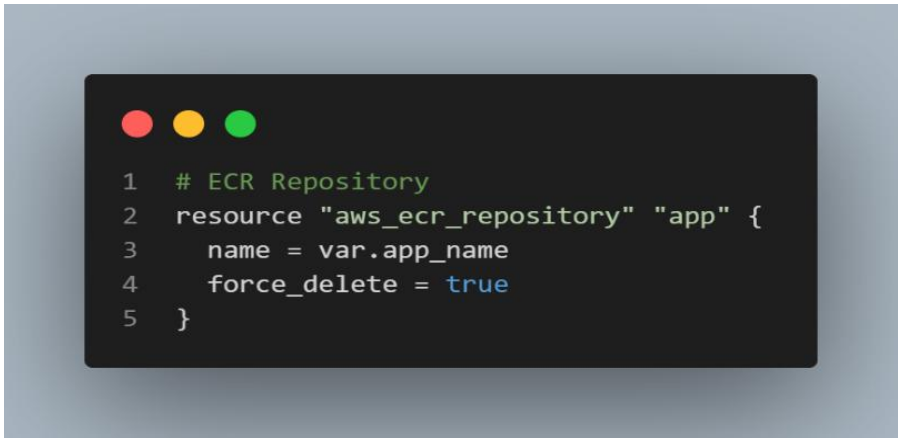


```hcl
# Get default VPC
data "aws_vpc" "default" {
  default = true
}

# Get default subnets
data "aws_subnets" "default" {
  filter {
    name   = "vpc-id"
    values = [data.aws_vpc.default.id]
  }
}
```

- Uses **data sources** to retrieve the **default VPC** and its associated **subnets** for deploying the ECS service.
- These subnets are used to place the Fargate tasks.

ECR Repository



```hcl
# ECR Repository
resource "aws_ecr_repository" "app" {
  name = var.app_name
  force_delete = true
}
```

- Creates an **Elastic Container Registry (ECR)** repository to store your container image (latest tag assumed).
- Enables force_delete to automatically delete any images in the repository during teardown.

ECS Cluster

```
# ECS Cluster
resource "aws_ecs_cluster" "main" {
  name = "${var.app_name}-cluster"
}
```

- Provisions an **ECS Cluster** (aws_ecs_cluster.main) to host your Fargate services.
- No EC2 provisioning is necessary—Fargate abstracts the compute layer.

IAM Role for Task Execution

```
# IAM Role for ECS Task Execution
resource "aws_iam_role" "ecs_execution_role" {
  name = "${var.app_name}-ecs-execution-role"

  assume_role_policy = jsonencode({
    Version = "2012-10-17"
    Statement = [
      {
        Action = "sts:AssumeRole"
        Effect = "Allow"
        Principal = {
          Service = "ecs-tasks.amazonaws.com"
        }
      }
    ]
  })
}

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

- Creates an **IAM role** with a trust policy allowing ECS tasks to assume it.
- Attaches the **AmazonECSTaskExecutionRolePolicy** to allow:
    - Pulling images from ECR
    - Pushing logs to CloudWatch

Security Group

```hcl
# Security Group
resource "aws_security_group" "ecs_tasks" {
  name_prefix = "${var.app_name}-ecs-tasks"
  vpc_id      = data.aws_vpc.default.id

  ingress {
    from_port   = var.container_port
    to_port     = var.container_port
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  egress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

- Defines a **security group** to:
    - Allow inbound traffic to your container on var.container_port(3000)
    - Allow all outbound traffic to the internet
- The security group is associated with the ECS service for Fargate networking.

CloudWatch Logs

```hcl
# CloudWatch Log Group
resource "aws_cloudwatch_log_group" "app" {
  name              = "/ecs/${var.app_name}"
  retention_in_days = 1
}
```

- Creates a **CloudWatch Log Group** to store logs generated by the ECS task containers.
- Logs are streamed using the awslogs driver.

ECS Task Definition

```
# ECS Task Definition
resource "aws_ecs_task_definition" "app" {
  family                   = var.app_name
  network_mode             = "awsvpc"
  requires_compatibilities = ["FARGATE"]
  cpu                      = "256"
  memory                   = "512"
  execution_role_arn       = aws_iam_role.ecs_execution_role.arn

  container_definitions = jsonencode([
    {
      name  = var.app_name
      image = "${aws_ecr_repository.app.repository_url}:latest"

      portMappings = [
        {
          containerPort = var.container_port
          protocol      = "tcp"
        }
      ]

      logConfiguration = {
        logDriver = "awslogs"
        options = {
```

- Specifies:
    - awsvpc network mode (required by Fargate)
    - Fargate-specific resources: 256 CPU units and 512 MiB memory
    - The container image from ECR
    - Port mapping (e.g., 80 or 3000 depending on your app)
    - Log configuration pointing to CloudWatch

ECS Fargate Service

```
# ECS Service
resource "aws_ecs_service" "app" {
  name            = "${var.app_name}-service"
  cluster         = aws_ecs_cluster.main.id
  task_definition = aws_ecs_task_definition.app.arn
  desired_count   = var.desired_count
  launch_type     = "FARGATE"

  network_configuration {
    subnets          = data.aws_subnets.default.ids
    security_groups  = [aws_security_group.ecs_tasks.id]
    assign_public_ip = true
  }
}
```

- Launches the ECS service with:
    - **Fargate** launch type (no EC2 instances required)
    - Reference to the task definition created above
    - Subnets for network placement
    - Public IP assignment for external accessibility
    - Attached security group
    - Desired count of containers specified by var.desired_count (1)

ECS Service Starts:

- ECS Fargate launches the container from ECR.
- Logs are streamed to CloudWatch.
- If configured with public IP, the container is accessible externally.

Deployment Workflow

Infrastructure Provisioning:

- Run terraform apply to provision:
  - ECS Cluster
  - IAM roles
  - ECR repository
  - Security groups
  - Task definition
  - ECS Service

Build & Push Docker Image:

After terraform apply, run the following commands to build and push your image to ECR. These commands can be added to the outputs file in the terraform configuration so you get the details of the ECR url and respective variables. The build command can be ignored if the image is already built. You can build and start the container locally before pushing it to the container registry.

```
output "docker_commands" {
  description = "Commands to build and push Docker image"
  value = <<-EOT
    # 1. Get ECR login token
    aws ecr get-login-password --region ${var.aws_region} | docker login --username AWS --password-stdin ${aws_ecr_repository.app.repository_url}

    # 2. Build Docker image
    docker build -t ${var.app_name} .

    # 3. Tag image for ECR
    docker tag ${var.app_name}:latest ${aws_ecr_repository.app.repository_url}:latest

    # 4. Push image to ECR
    docker push ${aws_ecr_repository.app.repository_url}:latest


  EOT
```

Why Use Fargate?

- **Serverless Containers**: No EC2 provisioning or management required.
- **Granular Costing**: Pay per task-level resource usage.
- **Scalability**: Easily adjust desired_count for auto-scaling.

6. OBSERVATIONS & KEY DIFFERENCES

Feature	ECS Fargate	ECS EC2
Infrastructure	Managed	Self-managed
Scaling	Automatic	Manual or ASG
Cost	Pay-per-task	Pay for instances
Customization	Limited	High

7. SECURITY CONSIDERATIONS

- Ensure IAM roles follow least privilege
- EC2 instances or Fargate tasks can be put in private subnets, EC2 instances and Fargate in private subnets can be accessed by ALB and will need a NAT Gateway for pulling images and making API calls.
- Enable encryption (EFS or EBS, secrets management)

8. COST IMPLICATIONS

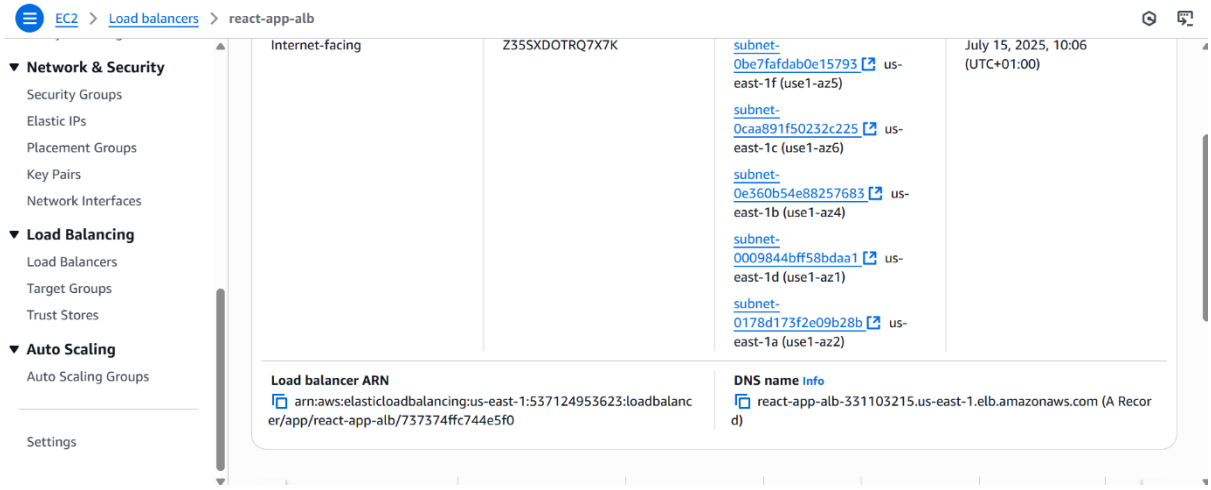
- Fargate: Ideal for bursty workloads, reduced admin
- EC2: Cost-efficient for consistent workloads

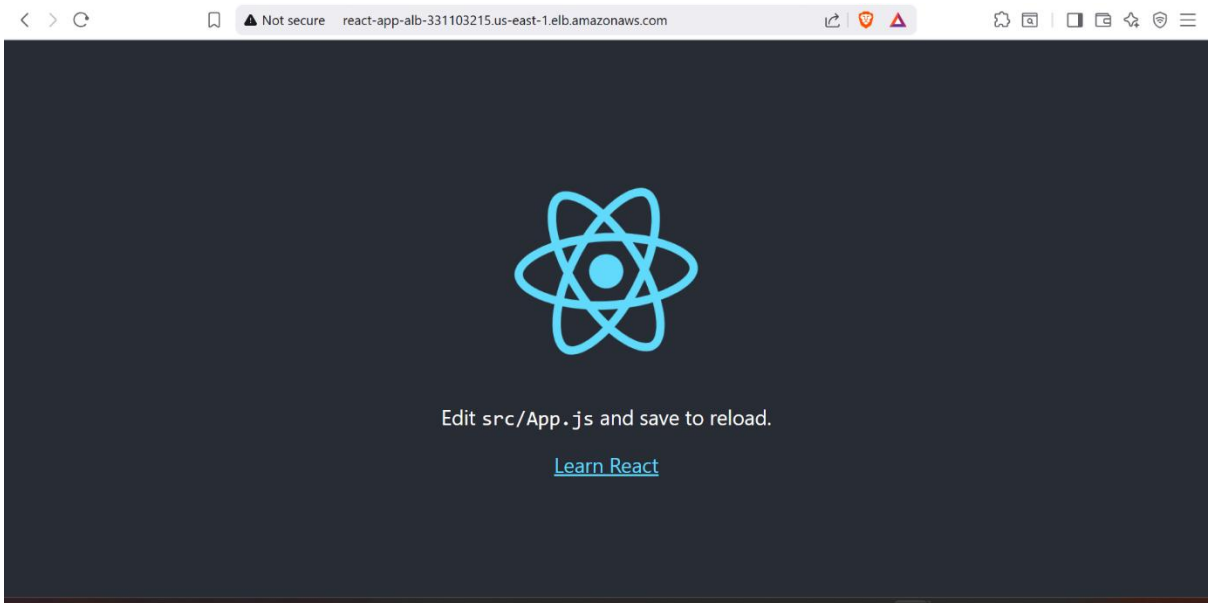
9. FINAL THOUGHTS & RECOMMENDATIONS

- Use **Fargate** for simpler, serverless deployments and rapid scaling
- Use **EC2** for customized environments, consistent usage patterns, or legacy compatibility

10. **PROOF OF CONCEPT**

EC2 Launch type: The pictures below shows the DNS of the ALB in the AWS console and how the application is being accessed. For EC2 Launch type, the DNS of the ALB will be copied into a browser to access the application.





Fargate Launch type: The pictures below shows a fargate task running in the AWS console. For Fargate Launch type the public IP address of the task is appended with the port the container is listening on to access the application.