

TASK 5

**ECS Fargate Nginx Server Deployment
with ALB and EFS using Terraform**

Umar Satti

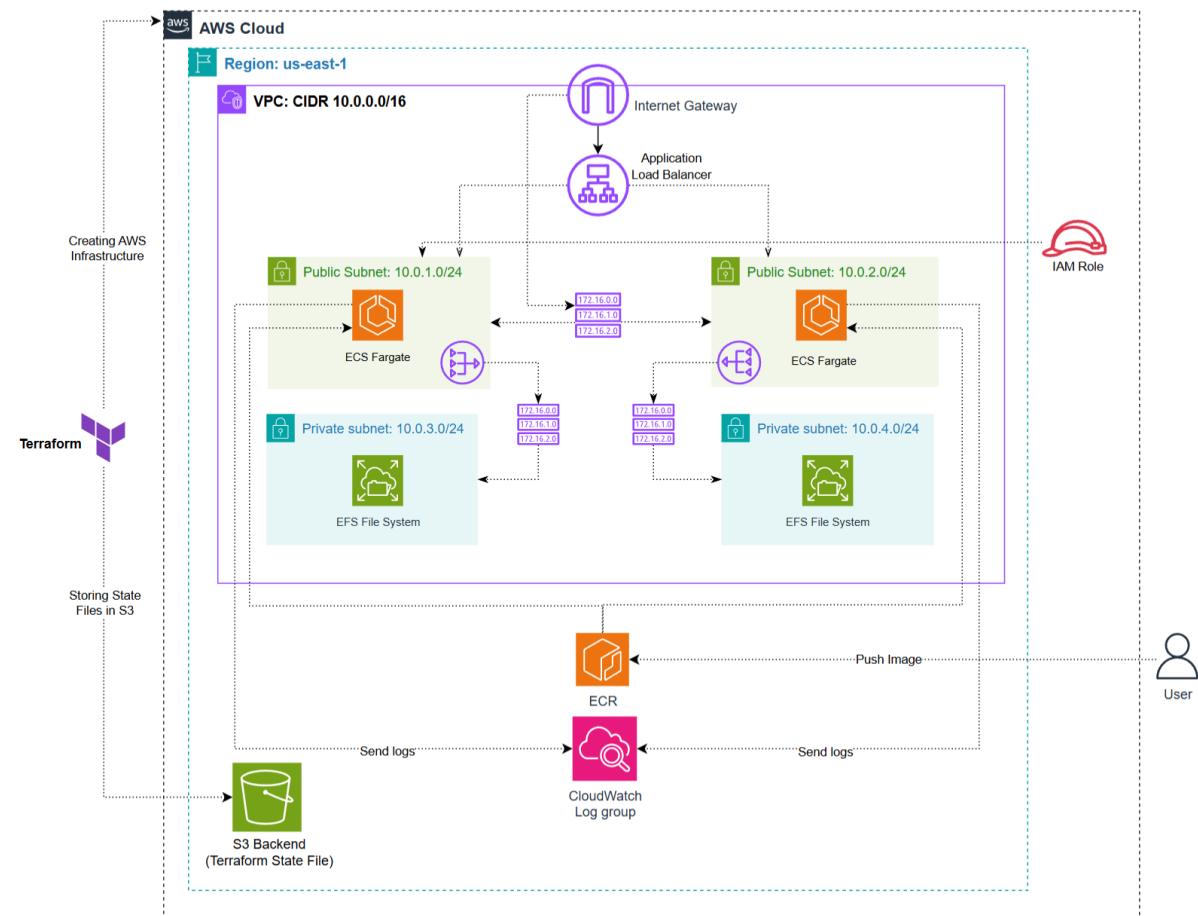
Table of Contents

Task Description	3
Architecture Diagram.....	3
Task 1.1: NGINX and Dockerfile Configuration.....	4
Task 1.2: Test in Local Environment.....	5
Pre-requisites.....	5
Task 1.3: Create Terraform Project Structure	7
Task 1.4: Create S3 Bucket for Terraform Remote Backend	8
Task 1.5: Root Directory Files.....	10
Task 1.6: Configure VPC Module	17
Task 1.7: Configure Elastic Container Registry (ECR) Module.....	26
Task 1.8: Elastic File System (EFS) Module.....	28
Task 1.9: Configure Task Definition Module.....	30
Task 1.10: Elastic Container Service (ECS) Module.....	36
Task 1.11: Application Load Balancer Module	39
Task 1.12: Execute Terraform Commands	42
Task 1.13: Validate Infrastructure Deployment in AWS Console.....	43
Task 1.11.2: Push Docker Image to ECR	56
Task 1.11.3: Confirm Application is Accessible	58
Task 1.12: Clean Up	59
Task 1.13: Troubleshooting	59

Task Description

Set up a highly available and scalable infrastructure on Amazon ECS Fargate using Terraform. Includes a VPC, security groups, ECS cluster, ECS service with a custom Docker image, ALB, and EFS for persistent storage.

Architecture Diagram



Task 1.1: NGINX and Dockerfile Configuration

The first step of the project is building a custom Docker image that runs an NGINX server hosting a static webpage. This image is then uploaded to ECR and used by ECS Fargate.

Step 1: Create Static Website (index.html)

A simple HTML page is placed in the /static/index.html directory. It serves as the application content rendered by the NGINX container.



The screenshot shows a code editor window with the file 'index.html' open. The code is a standard HTML document with meta tags for charset and viewport, a title, and two headings ('Hello from Nginx on Fargate!' and 'This is a test page for the ECS Fargate deployment with ALB and EFS.'):

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>ECS Fargate Nginx Test</title>
</head>
<body>
    <h1>Hello from Nginx on Fargate!</h1>
    <h2>This is a test page for the ECS Fargate deployment with ALB and EFS.</h2>
</body>
</html>
```

Step 2: Create Dockerfile

- **Custom configuration:** Removing the default config avoids the standard welcome page and allows full control over routing and behavior.
- **Static file deployment:** The static site is included directly in the container, so ECS tasks launch instantly without fetching external data.
- **Foreground execution:** daemon off ensures the container stays alive and does not exit immediately.
- **Port exposure:** Required for the ALB and ECS service to route incoming web traffic.

Step 3: Create NGINX configuration file

- **Root directory mapping:** Points NGINX to the correct path where the container stores static files.
- **server_name _:** This ensures NGINX accepts traffic regardless of domain.
- **try_files directive:** Provides proper 404 handling if files are missing.

With the Dockerfile, NGINX configuration, and static website in place, the image can now be tested locally before uploading to Amazon ECR. This ensures that the application runs successfully in a controlled environment and behaves as expected.

Task 1.2: Test in Local Environment

Before deploying the containerized NGINX application to AWS ECS Fargate, the image must first be tested locally. This ensures that:

- The Dockerfile is valid.
- NGINX configuration works correctly.
- Static content is being served properly.
- No runtime errors occur before pushing the image to ECR.

Testing locally removes guesswork and validates that the container will behave correctly once deployed in AWS.

Pre-requisites

- **Docker Desktop installed** and running
- Project directory containing:
 - /static/index.html
 - /Dockerfile
 - /nginx.conf

Step 1: Build the Docker Image

Navigate to the root of the project folder (where the Dockerfile exists), and run:

- `docker build -t nginx-app:latest .`

Explanation

- **docker build** compiles the Docker image using the Dockerfile in the current directory.
- **-t nginx-app:latest** assigns a name (nginx-app) and tag (latest).
- The resulting image becomes the local container that mirrors what will later be deployed on ECS.

Step 2: Run the Container Locally

Start the container using the following command:

- `docker run -d -p 8080:80 --name nginx-app nginx-app:latest`

Explanation

- **-d** runs the container in detached mode
- **-p 8080:80** maps local port **8080** to container port **80** (where NGINX listens)
- **--name nginx-app** assigns a readable container name
- **nginx-app:latest** is the image created earlier

Step 3: Verify Container Status

- Once running, inspect the container:
 - `docker ps -a`
- To confirm the image exists:
 - `docker images`
- What to verify:
 - The container should show **STATUS: Up**
 - The image **nginx-app** should be listed locally

```
PS D:\Cloudelligent\Task-5-ECS-Fargate-Nginx-Server-Deployment-with-ALB-and-EFS-using-Terraform> docker build -t nginx-app:latest .
[*] Building 2.3s (10/10) FINISHED
=> [internal] load build definition from Dockerfile
=> [internal] load build context
=> transferring dockerfile: 454B
=> [internal] load metadata for docker.io/library/nginx:stable-alpine
=> [auth] library/nginx:pull token for registry-1.docker.io
=> [internal] load dockerignore
=> transferring ignore file: 2B
=> [internal] load build context
=> transferring context: 958
=> [1/4] FROM docker.io/library/nginx:stable-alpine@sha256:30f1c0d78e0ad60901648be663a710bdadf19e4c10ac6782c235200619158284
=> CACHED [2/4] RUN rm /etc/nginx/conf.d/default.conf
=> CACHED [3/4] COPY nginx.conf /etc/nginx/conf.d/nginx.conf
=> CACHED [4/4] COPY /static/index.html /usr/share/nginx/html/index.html
=> exporting to image
=> exporting layers
=> writing image sha256:d81a3e9c91c173ed62ae6b30669b8a04599099c5fe7cc388b30fe2150ee9242
=> naming to docker.io/library/nginx-app.latest
View build details: docker-desktop://dashboard/build/desktop-linux/desktop-linux/h0f3wcery4vvhjzlvmmchqlq

What's next:
View a summary of image vulnerabilities and recommendations + docker scan quickview
PS D:\Cloudelligent\Task-5-ECS-Fargate-Nginx-Server-Deployment-with-ALB-and-EFS-using-Terraform> docker run -d -p 8080:80 --name nginx-app nginx-app:latest
baa7d50973d69678c1cx7f969278568b916b3db694cb56327f315c6120992b7e
PS D:\Cloudelligent\Task-5-ECS-Fargate-Nginx-Server-Deployment-with-ALB-and-EFS-using-Terraform> docker ps -a
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
baa7d50973d6 nginx-app:latest "/docker-entrypoint..." 5 seconds ago Up 4 seconds 0.0.0.0:8080->80/tcp, [::]:8080->80/tcp nginx-app
PS D:\Cloudelligent\Task-5-ECS-Fargate-Nginx-Server-Deployment-with-ALB-and-EFS-using-Terraform>
```

Step 4: Test in Browser

Open any browser and go to <http://localhost:8080>. The page should display the custom webpage written in index.html



Hello from Nginx on Fargate!

This is a test page for the ECS Fargate deployment with ALB and EFS.

This confirms:

- Dockerfile works
- NGINX config is valid

- Static file was copied correctly
- Container exposes port 80 properly
- Image is ready to be pushed to Amazon ECR

```
PS D:\Cloudelligent\Task-5-ECS-Fargate-Nginx-Server-Deployment-with-ALB-and-EFS-using-Terraform> docker ps -a
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
baa7d50973d6 nginx-app:latest "/docker-entrypoint..." 2 minutes ago Up About a minute 0.0.0.0:8080->80/tcp, [::]:8080->80/tcp nginx-app
PS D:\Cloudelligent\Task-5-ECS-Fargate-Nginx-Server-Deployment-with-ALB-and-EFS-using-Terraform> docker stop baa7d50973d6
baa7d50973d6
PS D:\Cloudelligent\Task-5-ECS-Fargate-Nginx-Server-Deployment-with-ALB-and-EFS-using-Terraform> docker rm baa7d50973d6
PS D:\Cloudelligent\Task-5-ECS-Fargate-Nginx-Server-Deployment-with-ALB-and-EFS-using-Terraform> |
```

Task 1.3: Create Terraform Project Structure

Steps:

1. Create a `terraform.tf` file in the root directory. This file defines the AWS provider, provider version, AWS region, and the S3 backend for storing Terraform state files.
2. Create an S3 bucket in the same AWS region to store Terraform state files (explained in Task 1.4 below).
3. Create a `main.tf` file in the root directory. This connects modules together, passes outputs from one module to another, and passes variables between them.
4. Create a `variables.tf` file in the root directory. This file defines variables by description and type as key-value pairs. These contain arguments for parameters such as VPC name, CIDR block, ECR URIs, and more.
5. Create an `outputs.tf` file in the root directory. This file exposes important module outputs such as ALB DNS name after deployment.
6. Create a `modules` directory that contains 6 total modules including VPC, Task definition, ECR, ECS, EFS, and ALB. Each module contains its own individual `main.tf`, `variables.tf`, and `outputs.tf` files.

The terraform project directory structure looks like this:

```
D:.
  .terraform.lock.hcl
  main.tf
  outputs.tf
  terraform.tf
  terraform.tfvars
  variables.tf

  .terraform
    terraform.tfstate
    modules
      modules.json
    providers
      registry.terraform.io
        hashicorp
          aws
            6.21.0
              windows_amd64
                LICENSE.txt
                terraform-provider-aws_v6.21.0_x5.exe

  modules
    alb
      main.tf
      outputs.tf
      variables.tf

    ecr
      main.tf
      outputs.tf
      variables.tf

    ecs
      main.tf
      outputs.tf
      variables.tf

    efs
      main.tf
      outputs.tf
      variables.tf

    task_definition
      main.tf
      outputs.tf
      variables.tf

    vpc
      main.tf
      outputs.tf
      variables.tf
```

Task 1.4: Create S3 Bucket for Terraform Remote Backend

Steps:

1. Log in to the AWS Management Console. Navigate to S3 using the search bar at the top of the console page.
2. Click on **Create Bucket** button.
3. Choose **General Purpose**, add a globally unique bucket name, and make sure the AWS Region is the same as Terraform.
4. Click **Create Bucket**.
5. Update **terraform.tf** file in root directory to reference this S3 bucket in the backend block.

```

backend "s3" {
  bucket      = "umarsatti-terraform-s3-bucket-state-file"
  key         = "Task-5/terraform.tfstate"
  region      = "us-west-2"
  encrypt     = true
  use_lockfile = true
}

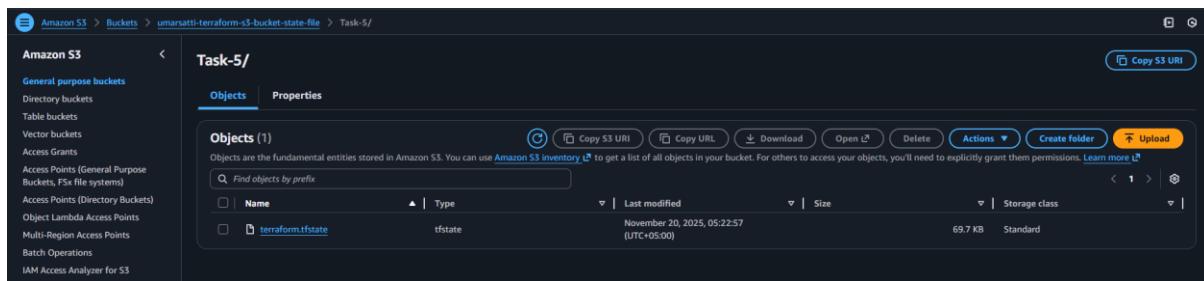
```

Once the S3 bucket is created in the AWS Management Console and referenced in the Terraform backend configuration, Terraform automatically begins storing and versioning state files in this bucket.

In this case, the S3 bucket named **umarsatti-terraform-s3-bucket-state-file** is used as the remote backend, as defined in the **provider.tf** file. The backend block ensures all state information is centralized, secure, and persistent across multiple users or workstations.

The screenshot shown below shows the exact file path inside the S3 bucket:

S3 > Buckets > umarsatti-terraform-state-file-s3-bucket > Task-5 > terraform.tfstate



This confirms that:

- Terraform successfully initialized the backend and wrote the state file to the S3 bucket.
- The **terraform.tfstate** file contains metadata about all deployed AWS resources (VPC, Subnets, Security groups, etc.).
- Every terraform plan, apply, or destroy operation reads and updates this file automatically.
- The locking mechanism (enabled by **use_lockfile = true**) ensures that no two processes modify the state simultaneously, preventing state corruption.

Task 1.5: Root Directory Files

The root directory of the Terraform project acts as the **control layer** for the entire infrastructure deployment.

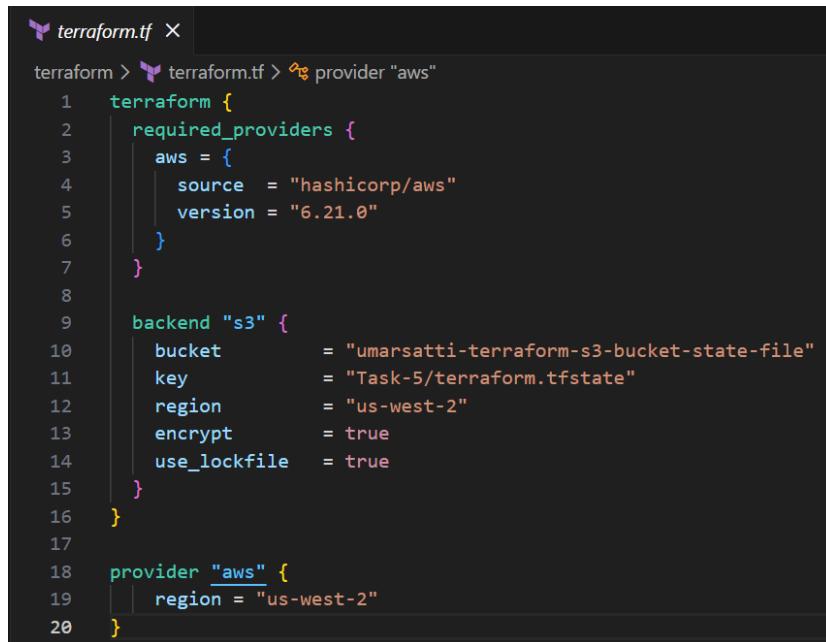
While each AWS service (VPC, ECS, ALB, EFS, ECR, Task Definition) is isolated into its own module, the root directory ties everything together by:

- Defining the provider and backend.
- Passing variables into modules.
- Connecting module outputs to other modules.
- Exposing global outputs after deployment.
- Organizing environment-specific configuration (tfvars).

This ensures that the project is modular, scalable, reusable, and easy to maintain.

terraform.tf:

This file configures the AWS provider, specifies its version, and defines the remote backend used for storing Terraform state files. It also enables state locking using S3 and controls the region of deployment.



```
terraform > terraform.tf > provider "aws"
1  terraform {
2    required_providers {
3      aws = {
4        source  = "hashicorp/aws"
5        version = "6.21.0"
6      }
7    }
8
9    backend "s3" {
10      bucket      = "umarsatti-terraform-s3-bucket-state-file"
11      key         = "Task-5/terraform.tfstate"
12      region      = "us-west-2"
13      encrypt     = true
14      use_lockfile = true
15    }
16  }
17
18  provider "aws" {
19    region = "us-west-2"
20  }
```

Explanation:

- **backend "s3"** block defines where the Terraform state file is stored.
- **use_lockfile = true** ensures no two Terraform operations run at the same time.
- The remote backend provides resilience and collaboration.

main.tf:

This is the central orchestrator of the infrastructure. It declares and connects all Terraform modules.

Purpose

- Loads modules for VPC, ECR, Task Definition, EFS, ECS, and ALB.
- Passes required inputs into each module.
- Wires module outputs into dependent modules.
- Creates dependency order between infrastructure components.

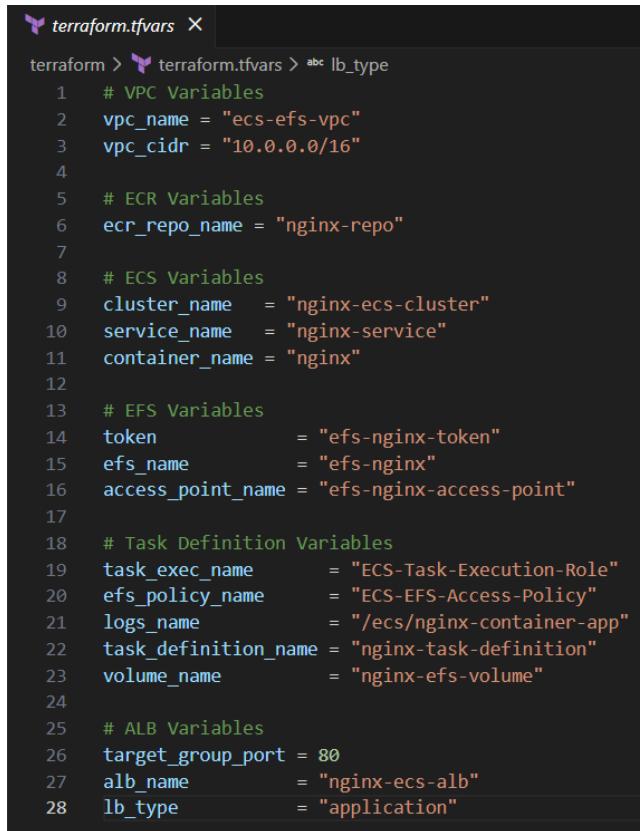
```
main.tf
 terraform > main.tf > module "task_definition"
1   module "vpc" {
2     source  = "./modules/vpc"
3     vpc_name = var.vpc_name
4     vpc_cidr = var.vpc_cidr
5   }
6
7   module "ecr" {
8     source      = "./modules/ecr"
9     ecr_repo_name = var.ecr_repo_name
10  }
11
12  module "task_definition" {
13    source        = "./modules/task_definition"
14    ecr_repo_url = module.ecr.ecr_repo_url
15    efs_fs_id    = module.efs.efs_file_system_id
16    efs_ap_id    = module.efs.efs_access_point_id
17    task_exec_name = var.task_exec_name
18    efs_policy_name = var.efs_policy_name
19    logs_name    = var.logs_name
20    task_definition_name = var.task_definition_name
21    volume_name   = var.volume_name
22  }
23
24  module "efs" {
25    source      = "./modules/efs"
26    private_subnet_ids = module.vpc.private_subnets
27    security_group_id = module.vpc.efs_sg_id
28    token       = var.token
29    efs_name    = var.efs_name
30    access_point_name = var.access_point_name
31  }
32
33  module "ecs" {
34    source      = "./modules/ecs"
35    task_definition_arn = module.task_definition.task_definition_arn
36    public_subnets = module.vpc.public_subnets
37    security_group_id = module.vpc.ecs_sg_id
38    target_group_arn = module.alb.target_group_arn
39    service_name   = var.service_name
40    container_name = var.container_name
41    cluster_name   = var.cluster_name
42  }
43
44  module "alb" {
45    source      = "./modules/alb"
46    vpc_id      = module.vpc.vpc_id
47    target_group_port = var.target_group_port
48    alb_security_group_id = module.vpc.alb_sg_id
49    public_subnets = module.vpc.public_subnets
50    alb_name     = var.alb_name
51    lb_type      = var.lb_type
52  }
53
```

Explanation:

- The VPC module is loaded first because most other modules depend on it.
- ECR is independent and provides an ECR repo URI for the Task Definition.
- The Task Definition depends on EFS and ECR.
- ECS requires Task Definition, ALB target group, VPC subnets, and ECS SG.
- ALB needs VPC ID, subnets, and the ALB SG.

This modular design separates networking, database, security, compute, and container image handling into cleanly isolated units.

terraform.tfvars



```

terraform > terraform.tfvars > abc_lb_type
  1 # VPC Variables
  2 vpc_name = "ecs-efs-vpc"
  3 vpc_cidr = "10.0.0.0/16"
  4
  5 # ECR Variables
  6 ecr_repo_name = "nginx-repo"
  7
  8 # ECS Variables
  9 cluster_name = "nginx-ecs-cluster"
10 service_name = "nginx-service"
11 container_name = "nginx"
12
13 # EFS Variables
14 token = "efs-nginx-token"
15 efs_name = "efs-nginx"
16 access_point_name = "efs-nginx-access-point"
17
18 # Task Definition Variables
19 task_exec_name = "ECS-Task-Execution-Role"
20 efs_policy_name = "ECS-EFS-Access-Policy"
21 logs_name = "/ecs/nginx-container-app"
22 task_definition_name = "nginx-task-definition"
23 volume_name = "nginx-efs-volume"
24
25 # ALB Variables
26 target_group_port = 80
27 alb_name = "nginx-ecs-alb"
28 lb_type = "application"

```

The `terraform.tfvars` file provides the actual input values Terraform uses during deployment. It separates configuration data from the module code, ensuring cleaner, reusable, and environment-specific setups. Each variable defined here directly corresponds to the infrastructure components deployed in AWS.

VPC Variables

- `vpc_name` and `vpc_cidr` specify the name and IP range of the VPC that forms the foundational network for the entire environment.
- These values ensure your VPC is uniquely identifiable and properly segmented.

ECR Variables

- **ecr_repo_name** defines the name of the Amazon ECR repository where the Nginx Docker image will be stored.
- Terraform uses this to create a private container registry for ECS deployments.

ECS Variables

- **cluster_name**, **service_name**, and **container_name** provide the names for the ECS Cluster, ECS Service, and container definition.
- These values ensure ECS resources are created with consistent and meaningful names.

EFS Variables

- **token**, **efs_name**, and **access_point_name** define the EFS file system and its access point.
- These values are used by ECS tasks to mount persistent shared storage using the EFS Access Point.

Task Definition Variables

- **task_exec_name** and **efs_policy_name** define IAM resources required by ECS Tasks.
 - Execution role: allows ECS to pull images, publish logs, etc.
 - EFS policy: grants tasks permission to mount EFS.
- **logs_name** specifies the CloudWatch Logs group for container logs.
- **task_definition_name** sets the name of the ECS Task Definition.
- **volume_name** defines the name of the EFS volume attached to the task.

ALB Variables

- **target_group_port** specifies which port the ALB Target Group listens on (HTTP 80 in this case).
- **alb_name** defines the Application Load Balancer's name.
- **lb_type** specifies the load balancer type (here it is set to "application" for ALB).

variables.tf:

The **variables.tf** file defines all input variables required by the Terraform root module. These variables parameterize the infrastructure, making the configuration modular, reusable, and environment-agnostic. Each variable includes a type and description to improve clarity and ensure Terraform validates values before applying changes.

VPC Variables

```
variables.tf ×
terraform > variables.tf > ...
1  # VPC Variables
2  variable "vpc_name" {
3    type      = string
4    description = "VPC Name"
5  }
6
7  variable "vpc_cidr" {
8    type      = string
9    description = "VPC IPv4 CIDR Block"
10 }
```

- **vpc_name** and **vpc_cidr** specify the foundational networking configuration.
- These values determine the VPC's name and address range, which all other resources depend on.

ECR Variables

```
# ECR Variables
variable "ecr_repo_name" {
  type      = string
  description = "NGINX ECR repository name"
}
```

- **ecr_repo_name** provides the name of the Amazon ECR repository where the Nginx container image will be stored.
- This allows the ECR module to dynamically create and manage the repository.

ECS Variables

```
# ECS Variables
variable "cluster_name" {
  type      = string
  description = "ECS Cluster name"
}

variable "service_name" {
  type      = string
  description = "ECS Service name"
}

variable "container_name" {
  type      = string
  description = "NGINX Container name"
}
```

- **cluster_name**, **service_name**, and **container_name** define the ECS Cluster, ECS Service, and the container name used inside the Task Definition.
- These variables ensure consistent naming across compute resources.

EFS Variables

```

# EFS Variables
variable "token" {
  type      = string
  description = "EFS file system creation token name"
}

variable "efs_name" {
  type      = string
  description = "EFS file system name tag"
}

variable "access_point_name" {
  type      = string
  description = "EFS access point name"
}

```

- **token, efs_name, and access_point_name** provide values required to create the EFS file system and Access Point.
- These variables allow ECS tasks to attach persistent storage.

Task Definition & IAM Variables

```

# Task Definition Variables
variable "task_exec_name" {
  type      = string
  description = "Task Execution IAM Role name"
}

variable "efs_policy_name" {
  type      = string
  description = "EFS IAM policy name"
}

variable "logs_name" {
  type      = string
  description = "ECS CloudWatch Logs group"
}

variable "task_definition_name" {
  type      = string
  description = "Task definition name"
}

variable "volume_name" {
  type      = string
  description = "EFS mount volume name"
}

```

- **task_exec_name** defines the name of the ECS Task Execution Role.
- **efs_policy_name** specifies the IAM policy that grants EFS access permissions.
- **logs_name** defines the CloudWatch Log Group for storing container logs.
- **task_definition_name** provides the Task Definition identifier.
- **volume_name** specifies the EFS volume name used within the ECS task.

ALB & Target Group Variables

```

variable "volume_name" {
  type     = string
  description = "EFS mount volume name"
}

# ALB and Target Group Variables
variable "target_group_port" {
  type     = number
  description = "Target Group port number"
}

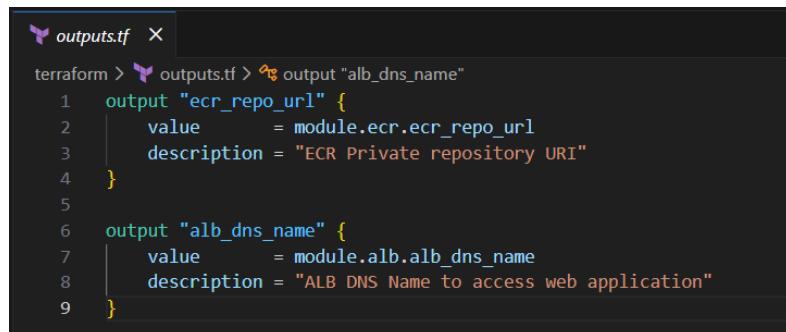
variable "alb_name" {
  type     = string
  description = "Application Load Balancer name"
}

variable "lb_type" {
  type     = string
  description = "Load Balancer type"
}

```

- **target_group_port** determines the port the ALB Target Group listens on (e.g., HTTP 80).
- **alb_name** sets the name of the Application Load Balancer.
- **lb_type** defines the type of load balancer (e.g., "application" for ALB).

outputs.tf:



```

outputs.tf ✘
terraform > outputs.tf > ↗ output "alb_dns_name"
1   output "ecr_repo_url" {
2     value      = module.ecr.ecr_repo_url
3     description = "ECR Private repository URI"
4   }
5
6   output "alb_dns_name" {
7     value      = module.alb.alb_dns_name
8     description = "ALB DNS Name to access web application"
9   }

```

This file exposes important outputs after Terraform completes its deployment.

- **ecr_repo_url** returns the full private ECR repository URI where container images are pushed.
- **alb_dns_name** outputs the Application Load Balancer's DNS name, which is used to access the deployed web application.

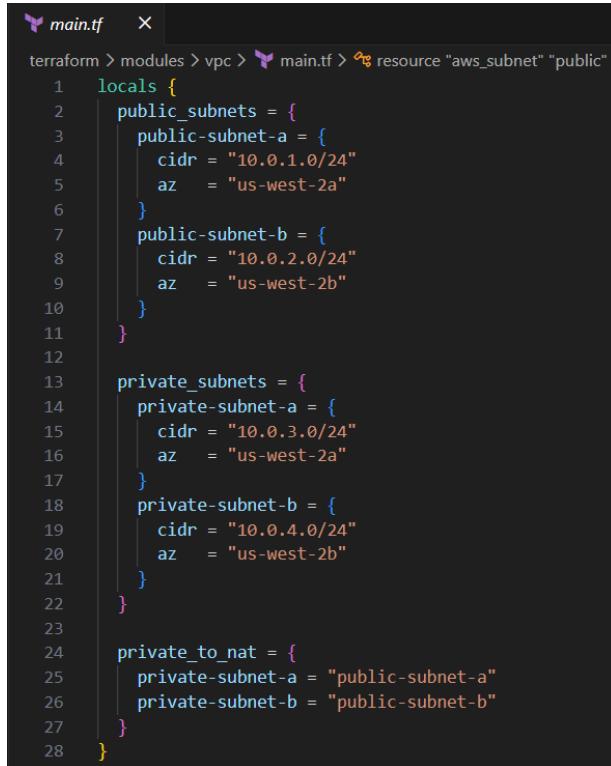
These outputs are useful for verification and for quickly locating key resource details without navigating the AWS console. They also make it easy to reuse infrastructure components in future stages or other dependent systems.

Task 1.6: Configure VPC Module

This section explains each Terraform configuration file located inside the **VPC module** (modules/vpc) and how they work together to create the complete networking layer required for the ECS Fargate, ALB, and EFS infrastructure.

main.tf

Local Variables



```
main.tf
  ...
  terraform > modules > vpc > main.tf > resource "aws_subnet" "public"
  1   locals {
  2     public_subnets = {
  3       public-subnet-a = {
  4         cidr = "10.0.1.0/24"
  5         az   = "us-west-2a"
  6       }
  7       public-subnet-b = {
  8         cidr = "10.0.2.0/24"
  9         az   = "us-west-2b"
 10      }
 11    }
 12
 13    private_subnets = {
 14      private-subnet-a = {
 15        cidr = "10.0.3.0/24"
 16        az   = "us-west-2a"
 17      }
 18      private-subnet-b = {
 19        cidr = "10.0.4.0/24"
 20        az   = "us-west-2b"
 21      }
 22    }
 23
 24    private_to_nat = {
 25      private-subnet-a = "public-subnet-a"
 26      private-subnet-b = "public-subnet-b"
 27    }
 28  }
```

The module begins with a **locals** block defining three important data structures:

1. public_subnets

A map describing all public subnets, including:

- Subnet name (key)
- CIDR block along with its actual value (value)
- Availability Zone along with its actual value (value)

2. private_subnets

A map defining all private subnets with similar structure. These subnets are used for ECS tasks and EFS mount targets, keeping them isolated from the public internet.

3. private_to_nat

A mapping that assigns each private subnet to the correct NAT Gateway residing in the matching public subnet (such as private subnet in Availability Zone B to public subnet in Availability Zone B). This ensures proper outbound routing for private resources.

Purpose of locals

- Allows clean, dynamic subnet creation using **for_each** argument
- Simplifies scaling to more subnets/AZs
- Eliminates repetitive code

Virtual Private Cloud (VPC)

```
# VPC
resource "aws_vpc" "main" {
  cidr_block      = var.vpc_cidr
  enable_dns_hostnames = true
  tags = {
    Name = var.vpc_name
  }
}
```

This resource creates the main VPC that will hold all networking components.

- **cidr_block** is passed from the root module, making the VPC customizable.
- **enable_dns_hostnames = true** allows resources like ECS tasks and EFS endpoints to use internal DNS.
- Tagged using the **vpc_name** variable for easy identification.

Public Subnets

```
# Subnets

# Public Subnet
resource "aws_subnet" "public" {
  for_each = local.public_subnets

  vpc_id      = aws_vpc.main.id
  cidr_block   = each.value.cidr
  availability_zone = each.value.az
  map_public_ip_on_launch = true

  tags = {
    Name = each.key
  }
}
```

Public subnets are created dynamically using **for_each = local.public_subnets**. Each subnet receives:

- The CIDR block defined in locals
- An availability zone
- Automatic public IP assignment using **map_public_ip_on_launch = true**
- A friendly tag equal to the map key (e.g., "public-subnet-a")

These subnets are used for:

- ALB (Application Load Balancer)
- NAT Gateways
- ECS

Private Subnets

```
# Private Subnet
resource "aws_subnet" "private" {
  for_each = local.private_subnets

  vpc_id          = aws_vpc.main.id
  cidr_block      = each.value.cidr
  availability_zone = each.value.az

  tags = {
    Name = each.key
  }
}
```

Created using **for_each = local.private_subnets**. Each private subnet receives:

- CIDR and AZ values from locals (local variables)
- No public IP assignment
- Tags matching their map key

Private subnets are used for EFS mount targets and do not have direct Internet access.

Internet Gateway (IGW)

```

# Internet Gateway
resource "aws_internet_gateway" "igw" {
  vpc_id = aws_vpc.main.id

  tags = {
    Name = "${var.vpc_name}-igw"
  }
}

```

Creates and attaches an Internet Gateway to the VPC.

- Allows public subnets to communicate with the Internet
- Required before NAT Gateways can function

Tagged for easy visibility.

Elastic IPs (EIP) for NAT Gateways

```

# Multi-AZ NAT Gateway
resource "aws_eip" "nat_eip" {
  for_each = aws_subnet.public
  domain   = "vpc"

  tags = { Name = "nat-eip-${each.key}" }
}

```

A separate EIP is created for each public subnet using **for_each = aws_subnet.public**. This ensures one NAT Gateway per Availability Zone, supporting high availability.

NAT Gateways

```

resource "aws_nat_gateway" "nat" {
  for_each = aws_subnet.public

  allocation_id = aws_eip.nat_eip[each.key].id
  subnet_id     = each.value.id

  tags = {
    Name = "nat-gw-${each.key}"
  }
}

```

Each NAT Gateway is deployed inside a public subnet:

- Uses the corresponding Elastic IP
- Allows outbound Internet access for private subnets

- Ensures private ECS tasks can pull images from ECR

The module deploys **one NAT per public subnet**, aligning with AWS best practices.

Route Tables

Public Route Table

```
# Route Tables

# Public Route Table
resource "aws_route_table" "public" {
  vpc_id = aws_vpc.main.id

  route {
    cidr_block = "0.0.0.0/0"
    gateway_id = aws_internet_gateway.igw.id
  }

  tags = {
    Name = "public-rt"
  }
}
```

Used by all public subnets.

- Routes all outbound traffic (0.0.0.0/0) to the Internet Gateway
- Ensures ALB and NAT Gateways have Internet access

Private Route Tables

```
# Private Route Tables
resource "aws_route_table" "private" {
  for_each = aws_subnet.private
  vpc_id   = aws_vpc.main.id

  tags = {
    Name = "${each.key}-rt"
  }
}
```

A separate private route table is created for each private subnet using **for_each = aws_subnet.private**.

Each private route table:

- Routes Internet traffic through the **correct NAT Gateway** due to local.private_to_nat variable
- Keeps private resources hidden from the public Internet

```
resource "aws_route" "private_nat_route" {
  for_each = aws_subnet.private

  route_table_id = aws_route_table.private[each.key].id
  destination_cidr_block = "0.0.0.0/0"

  nat_gateway_id = aws_nat_gateway.nat[
    local.private_to_nat[each.key]
  ].id
}
```

Route Table Associations

Public Associations

```
# Route Table Associations

# Public Route Table Association
resource "aws_route_table_association" "public" {
  for_each = aws_subnet.public

  subnet_id      = each.value.id
  route_table_id = aws_route_table.public.id
}
```

Links each public subnet to the public route table.

Private Associations

```
# Private Route Table Association
resource "aws_route_table_association" "private" {
  for_each = aws_subnet.private

  subnet_id      = each.value.id
  route_table_id = aws_route_table.private[each.key].id
}
```

Links each private subnet to its corresponding private route table.

These associations finalize the network routing behavior.

Security Groups

Three security groups are created inside the VPC:

1. ALB Security Group

```

# ALB Security Group
resource "aws_security_group" "alb_sg" {
  name      = "ALB-SG"
  description = "Allows HTTP traffic from the internet"
  vpc_id    = aws_vpc.main.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"]
  }

  tags = {
    Name = "ALB-SG"
  }
}

```

Allows:

- Inbound HTTP traffic from the Internet on port 80
- All outbound traffic allowed

Attached to the Application Load Balancer.

2. ECS Security Group

```

# ECS Security Group
resource "aws_security_group" "ecs_sg" {
  name      = "ECS-SG"
  description = "Allows HTTP traffic from the ALB-SG"
  vpc_id    = aws_vpc.main.id

  ingress {
    from_port  = 80
    to_port    = 80
    protocol   = "tcp"
    security_groups = [aws_security_group.alb_sg.id]
  }

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

  tags = {
    Name = "ECS-SG"
  }
}

```

Allows:

- Inbound traffic **only from the ALB** on port 80

- Outbound traffic allowed

This ensures:

- Only the ALB can communicate with ECS tasks
- ECS tasks cannot be accessed directly from the public Internet

3. EFS Security Group

```
# EFS Security Group
resource "aws_security_group" "efs_sg" {
  name          = "EFS-SG"
  description   = "Allows NFS traffic from ECS-SG"
  vpc_id        = aws_vpc.main.id

  ingress {
    from_port    = 2049
    to_port     = 2049
    protocol    = "tcp"
    security_groups = [aws_security_group.ecs_sg.id]
  }

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

  tags = {
    Name = "EFS-SG"
  }
}
```

Allows:

- Inbound traffic to port **2049** (NFS)
- Only from the ECS security group

Purpose:

- ECS tasks can mount EFS
- EFS is protected from all external traffic

This creates a secure flow in which traffic routes from **Internet to ALB to ECS Tasks to EFS**

variables.tf

```
variables.tf
variable "vpc_name" {
  type = string
}
variable "vpc_cidr" {
  type = string
}
```

This module accepts two variables:

1. vpc_name

Used for tagging the VPC for identification.

2. vpc_cidr

Defines the IPv4 range used by the VPC.

These variables allow the VPC module to be reused across different environments (dev, staging, prod).

outputs.tf

```
outputs.tf
output "vpc_id" {
  description = "VPC ID"
  value       = aws_vpc.main.id
}

output "public_subnets" {
  description = "VPC Public Subnet IDs"
  value       = [for s in aws_subnet.public : s.id]
}

output "private_subnets" {
  description = "VPC Private Subnet IDs"
  value       = [for s in aws_subnet.private : s.id]
}

output "efs_sg_id" {
  description = "EFS Private Security Group ID"
  value       = aws_security_group.efs_sg.id
}

output "ecs_sg_id" {
  description = "ECS Service Security Group ID"
  value       = aws_security_group.ecs_sg.id
}

output "alb_sg_id" {
  description = "ALB Security Group ID"
  value       = aws_security_group.alb_sg.id
}
```

The following outputs provide critical information for other modules.

1. vpc_id

Needed by:

- ALB module
- ECS module
- EFS module

2. public_subnets

A list of public subnet IDs, used by:

- ALB for load balancer placement
- NAT Gateway deployment
- ECS services if needed

3. private_subnets

A list of private subnet IDs, used by:

- ECS tasks
- EFS mount targets

4. efs_sg_id

Security group ID used by EFS module to allow ECS tasks to access the file system.

5. ecs_sg_id

Security group for ECS tasks, passed to the ECS module.

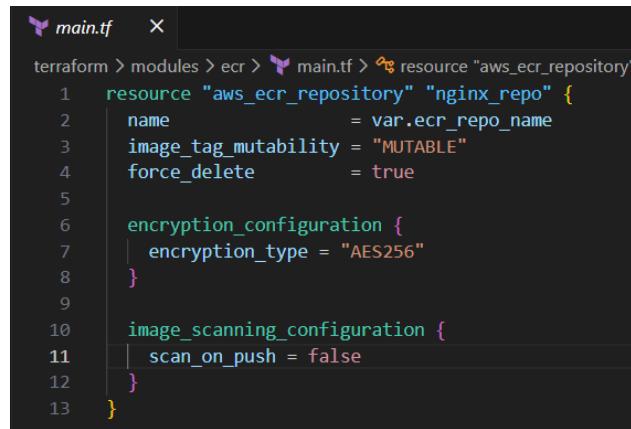
6. alb_sg_id

Security group for ALB, passed to the ALB module.

Task 1.7: Configure Elastic Container Registry (ECR) Module

This section explains all Terraform configuration files inside the **ECR module** (`modules/ecr`). This module provisions a private Elastic Container Registry that stores the Docker image used by the Nginx application running on ECS Fargate.

main.tf



```
main.tf
terraform > modules > ecr > main.tf > resource "aws_ecr_repository" "nginx_repo" {
  1   name          = var.ecr_repo_name
  2   image_tag_mutability = "MUTABLE"
  3   force_delete    = true
  4
  5   encryption_configuration {
  6     encryption_type = "AES256"
  7   }
  8
  9   image_scanning_configuration {
 10     scan_on_push = false
 11   }
 12 }
 13 }
```

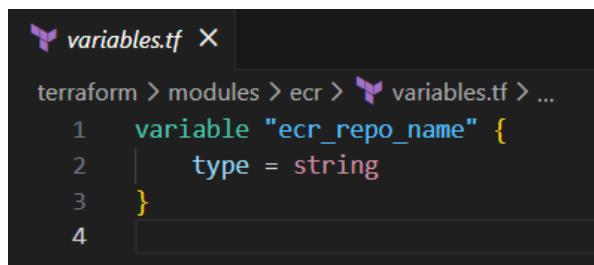
This file creates a single Amazon ECR repository:

Explanation

- **name** pulled from the input variable **ecr_repo_name**, allowing flexible naming for different environments.
- **image_tag_mutability = "MUTABLE"** allows tags (such as latest) to be overwritten. This is helpful during rapid development and testing, where images are frequently updated.
- **force_delete = true** ensures that Terraform can destroy the ECR repository even if images still exist, preventing cleanup issues during repeated deployments.
- **AES-256 encryption** allows all container images stored in the registry to be encrypted by default for security.
- **scan_on_push = false** disables image vulnerability scanning on upload. This simplifies CI/CD pipelines but can be enabled later for production security requirements.

Note: This registry provides a secure and private location to store the Nginx Docker image. Later in the deployment process, the ECS Task Definition will reference this repository URL when pulling the application image.

variables.tf

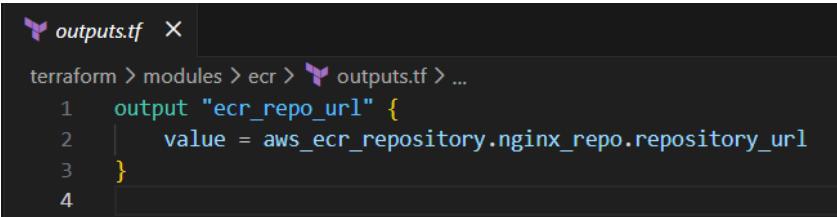


```
variables.tf
terraform > modules > ecr > variables.tf > ...
variable "ecr_repo_name" {
  1   type = string
  2 }
  3
  4 
```

- Defines a single variable **ecr_repo_name**, representing the repository name used when creating the ECR registry.

- Keeping this as an external input (in `terraform.tfvars`) ensures:
 - The module is reusable for any application.
 - Repository names can differ across environments without modifying module code.

outputs.tf



```

outputs.tf  ×

terraform > modules > ecr > outputs.tf > ...
1   output "ecr_repo_url" {
2     value = aws_ecr_repository.nginx_repo.repository_url
3   }
4

```

- Exposes `ecr_repo_url`, the full ECR URI such as:
123456789012.dkr.ecr.us-west-2.amazonaws.com/nginx-repo
- This output is essential because **ECS Task Definition** needs this URL to pull the Docker image.

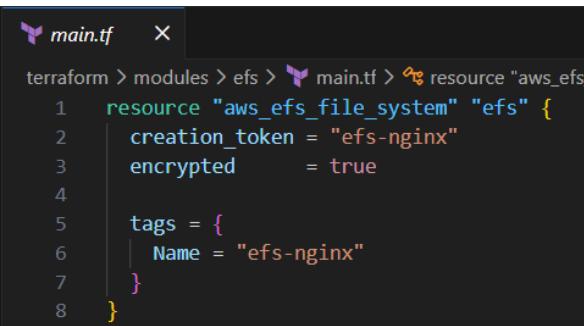
Task 1.8: Elastic File System (EFS) Module

This section explains all Terraform configuration files inside the **EFS module** (`modules/efs`). The purpose of this module is to create a shared Elastic File System that the ECS tasks will use to store or persist application data.

main.tf

This file provisions the EFS File System, its Mount Targets, and an Access Point.

EFS File System



```

main.tf  ×

resource "aws_efs_file_system" "efs" {
  creation_token = "efs-nginx"
  encrypted      = true
  tags = {
    Name = "efs-nginx"
  }
}

```

Explanation:

- **creation_token**
Ensures the EFS filesystem is created idempotently. If Terraform runs multiple times, AWS will not create duplicate EFS volumes.
- **encrypted = true**
Enables server-side encryption using AWS-managed keys.
This secures all data stored in the filesystem.
- **tags**
Helps identify this EFS filesystem inside AWS.

This creates the core network file system that ECS tasks will mount.

EFS Mount Targets

```
resource "aws_efs_mount_target" "efs_mount" {
  for_each = {
    subnet_1 = var.private_subnet_ids[0]
    subnet_2 = var.private_subnet_ids[1]
  }

  file_system_id  = aws_efs_file_system.efs.id
  subnet_id       = each.value
  security_groups = [var.security_group_id]
}
```

Explanation:

- EFS requires **one mount target per Availability Zone (AZ)** for ECS tasks to connect to it.
- The **for_each** block creates two mount targets, one for each private subnet.
- This ensures:
 - High availability
 - ECS tasks in any AZ can connect to EFS with low latency

Key inputs:

- **var.private_subnet_ids**
Passed from the VPC module, ensuring mount targets are created in private subnets only.
- **security_groups**
EFS mount targets use the provided security group, allowing only ECS tasks to connect using NFS port 2049.

EFS Access Point

```
resource "aws_efs_access_point" "efs_access" {
    file_system_id = aws_efs_file_system.efs.id

    posix_user {
        gid = "1000"
        uid = "1000"
    }

    root_directory {
        creation_info {
            owner_uid = "1000"
            owner_gid = "1000"
            permissions = "777"
        }

        path = "/"
    }

    tags = {
        Name = "efs-nginx-access-point"
    }
}
```

Explanation:

An Access Point provides a standardized entry path for ECS containers.

- **posix_user block** sets a default Linux user (UID/GID 1000) for file access.
- **root_directory block** ensures:
 - A root directory exists for the application
 - It is owned by UID/GID 1000
 - It has open permissions (777) for simplicity and compatibility with containers
- **Access Point benefits:**
 - Consistent permissions
 - Isolated directory
 - Cleaner ECS task volumes configuration

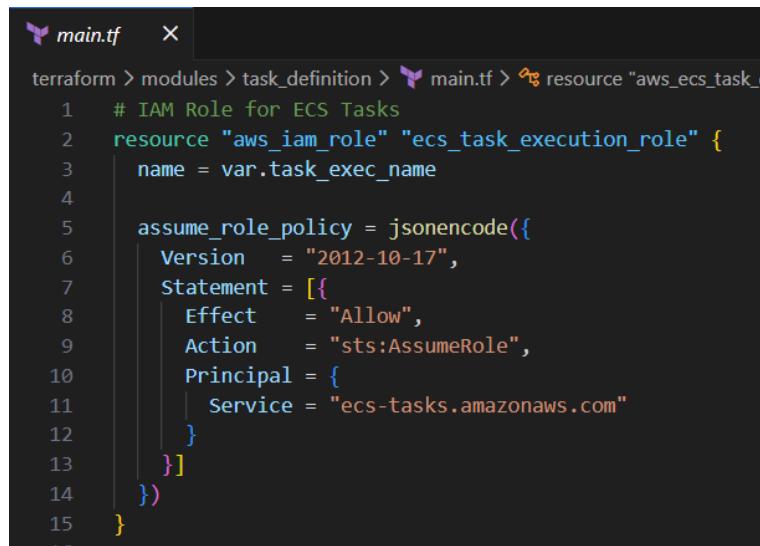
This Access Point will later be referenced directly in the ECS task definition.

Task 1.9: Configure Task Definition Module

This section explains the Terraform files inside the Task Definition module (modules/task_definition), which defines how an **Nginx container** should run inside ECS Fargate with EFS storage.

main.tf

ECS Task Execution Role



```
main.tf
 terraform > modules > task_definition > main.tf > resource "aws_ecs_task_definition" "task_definition" {
 1   name = var.task_exec_name
 2
 3   # IAM Role for ECS Tasks
 4   resource "aws_iam_role" "ecs_task_execution_role" {
 5     name = var.task_exec_name
 6
 7     assume_role_policy = jsonencode({
 8       Version  = "2012-10-17",
 9       Statement = [
10         {
11           Effect  = "Allow",
12           Action   = "sts:AssumeRole",
13           Principal = {
14             Service = "ecs-tasks.amazonaws.com"
15           }
16         }
17     })
18   }
19 }
```

- Creates an **IAM Role** that ECS tasks assume during execution.
- `assume_role_policy` allows ECS to use this role.
- This role enables ECS tasks to:
 - Pull container images from ECR
 - Send logs to CloudWatch
 - Access AWS APIs during runtime, including mounting EFS volumes

IAM Policy Attachment (AWS Managed)

```
# ECS Task Execution IAM Policy Attachment
resource "aws_iam_role_policy_attachment" "ecs_exec_policy" {
  role      = aws_iam_role.ecs_task_execution_role.name
  policy_arn = "arn:aws:iam::aws:policy/service-role/AmazonECSTaskExecutionRolePolicy"
}
```

- Attaches the **AmazonECSTaskExecutionRolePolicy**, which provides:
 - ECR read access
 - CloudWatch Logs permissions
 - Basic ECS runtime permissions

Custom EFS Access Policy

```

# Custom ECS and EFS Access IAM Policy
resource "aws_iam_policy" "efs_policy" {
  name      = var.efs_policy_name
  description = "Custom policy which allows ECS tasks to mount and access EFS file systems"

  policy = jsonencode({
    Version = "2012-10-17"
    Statement = [
      {
        Effect = "Allow"
        Action = [
          "elasticfilesystem:ClientMount",
          "elasticfilesystem:ClientWrite",
          "elasticfilesystem:ClientRootAccess",
          "elasticfilesystem:DescribeMountTargets",
          "elasticfilesystem:DescribeFileSystems"
        ],
        Resource = "*"
      }
    ]
  })
}

```

- Creates a custom IAM policy granting ECS tasks permission to:
 - Mount EFS volumes (ClientMount)
 - Write to EFS (ClientWrite)
 - Access the root directory of EFS (ClientRootAccess)
 - Describe EFS file systems and mount targets
- Attaches this policy to the ECS Task Execution Role.
- This allows ECS tasks to securely access the specified EFS file system and access point.

```

# Custom ECS and EFS Access IAM Policy Attachment
resource "aws_iam_role_policy_attachment" "ecs_efs_policy" {
  role      = aws_iam_role.ecs_task_execution_role.name
  policy_arn = aws_iam_policy.efs_policy.arn
}

```

CloudWatch Log Group

```

# CloudWatch Log Group
resource "aws_cloudwatch_log_group" "ecs_logs" {
  name           = var.logs_name
  retention_in_days = 7
}

```

- Creates a CloudWatch Logs group for container logs.
- retention_in_days = 7 ensures logs are automatically deleted after 7 days, controlling costs.

ECS Task Definition

```

# ECS Task Definition
resource "aws_ecs_task_definition" "task_definition" {
  family           = var.task_definition_name
  network_mode    = "awsvpc"
  requires_compatibilities = ["FARGATE"]
  cpu              = "512"
  memory           = "1024"

  execution_role_arn = aws_iam_role.ecs_task_execution_role.arn
  task_role_arn      = aws_iam_role.ecs_task_execution_role.arn

  volume {
    name = var.volume_name

    efs_volume_configuration {
      file_system_id      = var.efs_fs_id
      transit_encryption = "ENABLED"

      authorization_config {
        access_point_id = var.efs_ap_id
        iam             = "ENABLED"
      }
    }
  }

  container_definitions = jsonencode([
  {
    name      = "nginx"
    image     = "${var.ecr_repo_url}:latest"
    essential = true
    cpu       = 200
    memory   = 512

    portMappings = [
      {
        containerPort = 80
        protocol     = "tcp"
      }
    ]

    mountPoints = [
      {
        sourceVolume  = var.volume_name
        containerPath = "/mnt/data"
        readOnly     = false
      }
    ]

    logConfiguration = {
      logDriver = "awslogs"
      options = {
        awslogs-group      = aws_cloudwatch_log_group.ecs_logs.name
        awslogs-region     = "us-west-2"
        awslogs-stream-prefix = "nginx"
      }
    }
  ]
])
}

```

Defines how the Nginx container runs on ECS Fargate. Key settings:

- **General Settings**
 - family: Groups revisions of the same task definition.
 - network_mode = "awsvpc": Required for Fargate tasks.
 - requires_compatibilitys = ["FARGATE"] specifies the launch type.
 - cpu = 512 and memory = 1024 allocate task-level CPU and memory.
- **Task and Execution Role**
 - Both execution_role_arn and task_role_arn use the ECS Task Execution Role created above.
 - This enables container operations like pulling images, logging, and accessing EFS.
- **Volumes and EFS Configuration**
 - A volume is defined using EFS:
 - file_system_id and access_point_id are provided via variables.

- transit_encryption = "ENABLED" ensures encrypted network traffic between ECS and EFS.
- iam = "ENABLED" in authorization_config ensures IAM-based access control for the volume.

- **Container Definitions**

- Defines a single **Nginx container**:
 - image: Pulled from the ECR repository (`${var.ecr_repo_url}:latest`)
 - cpu and memory define container resource allocation.
 - essential = true ensures ECS stops the task if the container fails.

- **Port Mapping**

- Maps container port 80 to allow traffic routing from ECS.

- **Mount Points**

- Mounts the EFS volume at /mnt/data inside the container.
- readOnly = false allows writing to the volume.

- **Log Configuration**

- Sends container logs to CloudWatch using the log group defined above.
- Uses awslogs driver with region and stream prefix configured.

variables.tf

```
variables.tf ×
terraform > modules > task_definition > variables.tf >
1  variable "ecr_repo_url" {
2    | type = string
3  }
4
5  variable "efs_fs_id" {
6    | type = string
7  }
8
9  variable "efs_ap_id" {
10   | type = string
11 }
12
13 variable "task_exec_name" {
14   | type = string
15 }
16
17 variable "efs_policy_name" {
18   | type = string
19 }
20
21 variable "logs_name" {
22   | type = string
23 }
24
25 variable "task_definition_name" {
26   | type = string
27 }
28
29 variable "volume_name" {
30   | type = string
31 }
```

Defines input variables for the module:

- **ecr_repo_url**: URL of the ECR image repository
- **efs_fs_id**: ID of the EFS filesystem
- **efs_ap_id**: ID of the EFS access point
- **task_exec_name**: Name of the ECS Task Execution Role
- **efs_policy_name**: Name of the custom EFS policy
- **logs_name**: Name of the CloudWatch log group
- **task_definition_name**: ECS Task Definition family name
- **volume_name**: Name of the volume used for mounting EFS

outputs.tf

```
outputs.tf ×
terraform > modules > task_definition > outputs.tf > output "task_definition_arn"
1  output "task_definition_arn" {
2    | value      = aws_ecs_task_definition.task_definition.arn
3    | description = "ARN of Task Definition"
4  }
```

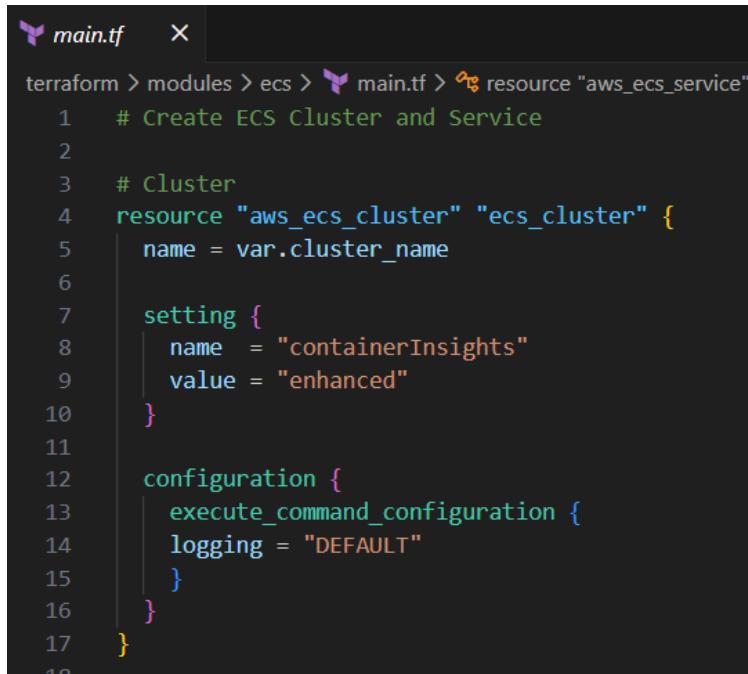
- `task_definition_arn`:
 - Exposes the ARN of the ECS Task Definition.
 - Required by the ECS Service module to deploy tasks using this definition.

Task 1.10: Elastic Container Service (ECS) Module

This module defines the ECS infrastructure needed to run the containerized application defined in the Task Definition module. It creates an **ECS cluster** and an **ECS Fargate service** that runs tasks using that cluster.

main.tf

ECS Cluster



```

main.tf
 terraform > modules > ecs > main.tf > resource "aws_ecs_service"
 1 # Create ECS Cluster and Service
 2
 3 # Cluster
 4 resource "aws_ecs_cluster" "ecs_cluster" {
 5   name = var.cluster_name
 6
 7   setting {
 8     name  = "containerInsights"
 9     value = "enhanced"
10   }
11
12   configuration {
13     execute_command_configuration {
14       logging = "DEFAULT"
15     }
16   }
17 }
18

```

- `aws_ecs_cluster` creates an ECS cluster to run tasks.
- Key settings:
 - `name`: Cluster name, provided via variable `cluster_name`.
 - `setting`: Enables **Container Insights** (enhanced) for monitoring and observability.
 - `configuration.execute_command_configuration.logging`: Enables ECS Execute Command logging to CloudWatch for debugging.

Cluster Capacity Providers

```

resource "aws_ecs_cluster_capacity_providers" "capacity" {
  cluster_name = aws_ecs_cluster.ecs_cluster.name

  capacity_providers = ["FARGATE"]

  default_capacity_provider_strategy {
    base           = 1
    weight         = 100
    capacity_provider = "FARGATE"
  }
}

```

- `aws_ecs_cluster_capacity_providers` sets the **capacity provider strategy** for the cluster.
- Here, it uses **FARGATE** as the capacity provider, with a base of 1 task and weight of 100.
- This ensures all tasks in this cluster run on Fargate (serverless ECS compute).

ECS Service

```

# Service

resource "aws_ecs_service" "service" {
  name          = var.service_name
  cluster        = aws_ecs_cluster.ecs_cluster.id
  task_definition = var.task_definition_arn
  desired_count   = 2
  launch_type     = "FARGATE"
  platform_version = "LATEST"
  scheduling_strategy = "REPLICA"
  enable_execute_command = true

  deployment_configuration {
    strategy = "ROLLING"
  }

  network_configuration {
    assign_public_ip = true
    security_groups  = [var.security_group_id]
    subnets          = var.public_subnets
  }

  load_balancer {
    target_group_arn = var.target_group_arn
    container_name   = var.container_name
    container_port   = 80
  }
}

```

ECS Service deploys and manages tasks using the ECS cluster.

Key features:

- **General Settings**
 - `name`: Name of the ECS service (`service_name`)
 - `cluster`: ECS cluster ID

- task_definition: Uses the ARN of the ECS Task Definition (from Task 1.9)
- desired_count = 2: Runs 2 task replicas
- launch_type = "FARGATE": Serverless compute
- platform_version = "LATEST": Ensures newest Fargate features
- scheduling_strategy = "REPLICA": ECS maintains the desired number of task replicas
- enable_execute_command = true: Allows interactive command execution in containers

- **Deployment Configuration**

- strategy = "ROLLING": Updates tasks gradually during deployment to avoid downtime

- **Network Configuration**

- assign_public_ip = true: Tasks get public IPs
- security_groups: Attaches specified security group
- subnets: Deploys tasks into public subnets

- **Load Balancer Integration**

- Configures ECS tasks to work with an **Application Load Balancer (ALB)**
- target_group_arn: Target group where traffic is sent
- container_name and container_port: Maps traffic to the correct container and port (port 80)

variables.tf

```
variables.tf ×
terraform > modules > ecs > variables.tf > variables.tf
1 variable "task_definition_arn" {
2   type = string
3 }
4
5 variable "public_subnets" {
6   type = any
7 }
8
9 variable "security_group_id" {
10  type = string
11 }
12
13 variable "target_group_arn" {
14  type = string
15 }
16
17 variable "cluster_name" {
18  type = string
19 }
20
21 variable "service_name" {
22  type = string
23 }
24
25 variable "container_name" {
26  type = string
27 }
```

Defines input variables for the module:

- **task_definition_arn**: ARN of the ECS Task Definition to run
- **public_subnets**: List of public subnet IDs for Fargate tasks
- **security_group_id**: Security group ID for task networking
- **target_group_arn**: ARN of the ALB target group
- **cluster_name**: Name of the ECS cluster
- **service_name**: Name of the ECS service
- **container_name**: Name of the container in the task definition (used for ALB routing)

Task 1.11: Application Load Balancer Module

This module provisions an **Application Load Balancer** and its supporting resources, enabling ECS services to receive HTTP traffic from the internet.

main.tf

Application Load Balancer (aws_lb)

```

  main.tf
  terraform > modules > alb > main.tf > ...
1   resource "aws_lb" "alb" {
2     name          = var.alb_name
3     internal      = false
4     load_balancer_type = var.lb_type
5     security_groups = [var.alb_security_group_id]
6     subnets        = var.public_subnets
7
8     enable_deletion_protection = false
9
10    tags = {
11      Name = var.alb_name
12    }
13  }
14
15  resource "aws_lb_target_group" "ip_target_group" {
16    name          = "nginx-target-group"
17    port          = var.target_group_port
18    protocol      = "HTTP"
19    protocol_version = "HTTP1"
20    target_type   = "ip"
21    vpc_id        = var.vpc_id
22  }
23
24  resource "aws_lb_listener" "listener" {
25    load_balancer_arn = aws_lb.alb.arn
26    port              = var.target_group_port
27    protocol          = "HTTP"
28
29    default_action {
30      type      = "forward"
31      target_group_arn = aws_lb_target_group.ip_target_group.arn
32    }
33  }
34

```

- Creates an **internet-facing ALB** that distributes incoming traffic to ECS tasks.
- Key settings:
 - **name:** Name of the ALB (from `alb_name` variable).
 - **internal = false:** Makes the ALB publicly accessible.
 - **load_balancer_type:** Set via variable (ALB or NLB)
 - **security_groups:** Attach a security group controlling access to the ALB.
 - **subnets:** Deploys the ALB in public subnets for internet access.
 - **enable_deletion_protection = false:** Allows ALB deletion if needed.

Target Group (aws_lb_target_group)

- Defines a **target group** for routing traffic from the ALB to ECS tasks.
- Key settings:
 - **name:** Target group name, here "nginx-target-group".
 - **port & protocol:** Traffic is sent over HTTP to the container port.
 - **protocol_version = "HTTP1":** Uses HTTP/1.1.

- **target_type = "ip"**: ECS tasks registered by IP address, required for Fargate.
- **vpc_id**: Associates the target group with the correct VPC.

Listener (aws_lb_listener)

- Configures the **listener** for the ALB that receives incoming requests.
- Key settings:
 - **load_balancer_arn**: Connects the listener to the ALB.
 - **port & protocol**: Listens on port 80 using HTTP.
 - **default_action**: Forwards all requests to the **ip_target_group** target group.

variables.tf

```

variables.tf ×
terraform > modules > alb > variables.tf > variable "lb_type"
1   variable "vpc_id" {
2     type = string
3   }
4
5   variable "public_subnets" {
6     type = any
7   }
8
9   variable "alb_security_group_id" {
10    type = string
11  }
12
13  variable "target_group_port" {
14    type = number
15  }
16
17  variable "alb_name" {
18    type = string
19  }
20
21  variable "lb_type" {
22    type = string
23  }

```

Defines inputs for the module:

- **vpc_id**: VPC where ALB and target group reside.
- **public_subnets**: Public subnets to deploy the ALB.
- **alb_security_group_id**: Security group for ALB access.
- **target_group_port**: Port ECS tasks listen on (container port, e.g., 80).
- **alb_name**: Name of the ALB.
- **lb_type**: Load balancer type (application for ALB, network for NLB).

outputs.tf

```
outputs.tf ×
terraform > modules > alb > outputs.tf > output "alb_dns_name" > value
1   output "alb_arn" {
2     value      = aws_lb.alb.arn
3     description = "ARN for Application Load Balancer"
4   }
5
6   output "target_group_arn" {
7     value      = aws_lb_target_group.ip_target_group.arn
8     description = "ARN for ALB Target Group"
9   }
10
11  output "alb_dns_name" {
12    value      = aws_lb.alb.dns_name
13    description = "ALB DNS Name to access web application"
14 }
```

Exposes key ALB information for other modules (e.g., ECS service):

- **alb_arn:** ARN of the ALB for referencing in ECS service.
- **target_group_arn:** ARN of the target group for ECS service registration.
- **alb_dns_name:** Public DNS name of the ALB to access the web application.

Task 1.12: Execute Terraform Commands

This section documents the series of Terraform CLI commands executed to deploy, validate, and destroy the WordPress environment.

Note: To perform this task, the user must be in the root directory of Terraform project where the provider.tf and the root main.tf files are stored.

Step 1: terraform init

Initializes the working directory by downloading the required provider plugins and connecting them to the configured backend (S3).

Step 2: terraform validate

Performs a syntax and logic check on all configuration files in the directory. Outputs an error if the logic or syntax is incorrect.

Step 3: terraform plan

Generates an execution plan showing all actions Terraform will perform to reach the desired state (resource creation, updates, or deletions).

Step 4: terraform apply

terraform apply --auto-approve command creates 39 resources. After infrastructure creation, the terminal displays the following outputs:

- alb_dns_name = "nginx-ecs-alb-1498337006.us-west-2.elb.amazonaws.com"
- ecr_repo_url = "504649076991.dkr.ecr.us-west-2.amazonaws.com/nginx-repo"

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

Outputs:

alb_dns_name = "nginx-ecs-alb-1498337006.us-west-2.elb.amazonaws.com"
ecr_repo_url = "504649076991.dkr.ecr.us-west-2.amazonaws.com/nginx-repo"
○ PS D:\Cloudelligent\Task-5-ECS-Fargate-Nginx-Server-Deployment-with-ALB-and-EFS-using-Terraform>
```

Task 1.13: Validate Infrastructure Deployment in AWS Console

After running **terraform apply**, Terraform provisions the entire containerized application environment, including networking, compute, storage, IAM, and orchestration resources. In this step, verify that all deployed components exist and are configured correctly in the AWS Console.

Verify VPC Creation

1. Sign in to the AWS Management Console.
2. Navigate to **VPC** using the search bar and click **Your VPCs**.
3. Verify that a VPC named **ecs-vpc** (or your custom name) has been created.

Expected Configuration:

- **CIDR Block:** 10.0.0.0/16
- **DNS Hostnames:** Enabled
- **Purpose:** This VPC acts as the isolated network environment for the ECS Fargate and EFS application

The screenshot shows the AWS VPC console. On the left, there's a navigation sidebar with options like 'VPC dashboard', 'AWS Global View', 'Virtual private cloud', 'Security', and 'PrivateLink and Lattice'. The main area displays a table titled 'Your VPCs (1/5) Info' with columns for Name, VPC ID, State, Block Public..., IPv4 CIDR, IPv6 CIDR, DHCP option set, and Main route table. One row is selected: 'ecs-efs-vpc' (vpc-0fbe6c9daab0312e4). A detailed view panel on the right shows the 'Details' tab for this VPC, including its VPC ID, DNS resolution status, security group, and network ACL. It also lists subnets, route tables, and network connections.

Verify Internet Gateway (IGW) Creation

1. In the VPC console, choose **Internet Gateways**.
2. Confirm that an Internet Gateway named **ecs-efs-vpc-igw** exists and is attached to the VPC created earlier. This gateway allows the ECS service to communicate with the internet.

The screenshot shows the AWS VPC console under the 'Internet gateways' section. The left sidebar includes 'VPC dashboard', 'AWS Global View', 'Virtual private cloud', 'Route tables', and 'Internet gateways'. The main area shows a table with one entry: 'ecs-efs-vpc-igw' (igw-02e49cd83d813ace), which is attached to the 'ecs-efs-vpc' VPC. A detailed view panel on the right shows the 'Details' tab for this gateway, confirming it is attached to the correct VPC.

Verify Subnets Creation

Name	Subnet ID	VPC	Block Public...	IPv4 CIDR	IPv6 CIDR	IPv6 CI
public-subnet-a	subnet-057229f2adc6206b1	vpc-0fbe6c9daab0312e4 ecs...	Off	10.0.1.0/24	-	-
private-subnet-b	subnet-0f1e8b1d1ec9a430d	vpc-0fbe6c9daab0312e4 ecs...	Off	10.0.4.0/24	-	-
public-subnet-b	subnet-071584407c6fa0dd3	vpc-0fbe6c9daab0312e4 ecs...	Off	10.0.2.0/24	-	-
private-subnet-a	subnet-0500682eb56912ea7	vpc-0fbe6c9daab0312e4 ecs...	Off	10.0.3.0/24	-	-

Public Subnets

1. In the VPC console, select **Subnets** in the left navigation bar.
2. Confirm the existence of:
 - public-subnet-a with IPv4 CIDR **10.0.1.0/24**
 - public-subnet-b with IPv4 CIDR **10.0.2.0/24**

These subnets host the Application Load Balancer (ALB) and ECS Fargate ENIs (public IP assigned).

Details	Flow logs	Route table	Network ACL	CIDR reservations	Sharing	Tags
Details						
Subnet ID subnet-057229f2adc6206b1	Subnet ARN arn:aws:ec2:us-west-2:504649076991:subnet/subnet-057229f2adc6206b1	State Available	Block Public Access Off			
IPv4 CIDR 10.0.1.0/24	Available IPv4 addresses 249	IPv6 CIDR -	IPv6 CIDR association ID -			
Availability Zone usw2-az1 (us-west-2a)	Network border group us-west-2	VPC vpc-0fbe6c9daab0312e4 ecs-efs-vpc	Route table rtb-00970a8f31d86d2dd public-rt			
Network ACL acl-07d096003fb489e4	Default subnet No	Auto-assign public IPv4 address Yes	Auto-assign IPv6 address No			
Auto-assign customer-owned IPv4 address No	Customer-owned IPv4 pool -	Outpost ID -	IPv4 CIDR reservations -			
IPv6 CIDR reservations -	IPv6-only No	Hostname type IP name	Resource name DNS A record Disabled			
Resource name DNS AAAA record Disabled	DNS64 Disabled	Owner 504649076991				

Details	Flow logs	Route table	Network ACL	CIDR reservations	Sharing	Tags
Details						
Subnet ID subnet-071584407c6fa0dd3	Subnet ARN arn:aws:ec2:us-west-2:504649076991:subnet/subnet-071584407c6fa0dd3	State Available	Block Public Access Off			
IPv4 CIDR 10.0.2.0/24	Available IPv4 addresses 249	IPv6 CIDR -	IPv6 CIDR association ID -			
Availability Zone usw2-az2 (us-west-2b)	Network border group us-west-2	VPC vpc-0fbe6c9daab0312e4 ecs-efs-vpc	Route table rtb-00970a8f31d86d2dd public-rt			
Network ACL acl-07d096003fb489e4	Default subnet No	Auto-assign public IPv4 address Yes	Auto-assign IPv6 address No			
Auto-assign customer-owned IPv4 address No	Customer-owned IPv4 pool -	Outpost ID -	IPv4 CIDR reservations -			
IPv6 CIDR reservations -	IPv6-only No	Hostname type IP name	Resource name DNS A record Disabled			
Resource name DNS AAAA record Disabled	DNS64 Disabled	Owner 504649076991				

Private Subnets

1. In the VPC console, select **Subnets**.
2. Confirm the existence of:

- private-subnet-a with IPv4 CIDR **10.0.3.0/24**
- private-subnet-b with IPv4 CIDR **10.0.4.0/24**

These subnets host:

- Private ECS networking (for internal workloads)
- EFS mount access
- Any private resources (if applicable)

The image contains two screenshots of the AWS VPC Subnet configuration page. Both screenshots show the 'Details' tab selected for two different subnets.

subnet-0500682eb569125a7 / private-subnet-a

Setting	Value
Subnet ID	subnet-0500682eb569125a7
IPv4 CIDR	10.0.3.0/24
Availability Zone	usw2-az2 (us-west-2a)
Network ACL	acl-07d096003efb489e4
Auto-assign customer-owned IPv4 address	No
IPv6 CIDR reservations	-
Resource name DNS AAAA record	Disabled
Subnet ARN	arn:aws:ec2:us-west-2:504649076991:subnet/subnet-0500682eb569125a7
Available IPv4 addresses	250
Network border group	us-west-2
Default subnet	No
Customer-owned IPv4 pool	-
IPv6-only	No
DNS64	Disabled
State	Available
IPv6 CIDR	-
VPC	vpc-0fbe6c9daab0312e4 ecs-efs-vpc
Auto-assign public IPv4 address	No
Outpost ID	-
Hostname type	IP name
Owner	504649076991
Block Public Access	Off
IPv6 CIDR association ID	-
Route table	rtb-03afa18bd52ac58e0 private-subnet-a-rt
Auto-assign IPv6 address	No
IPv4 CIDR reservations	-
Resource name DNS A record	Disabled

subnet-0f1e8b1d1ec9ac30d / private-subnet-b

Setting	Value
Subnet ID	subnet-0f1e8b1d1ec9ac30d
IPv4 CIDR	10.0.4.0/24
Availability Zone	usw2-az2 (us-west-2b)
Network ACL	acl-07d096003efb489e4
Auto-assign customer-owned IPv4 address	No
IPv6 CIDR reservations	-
Resource name DNS AAAA record	Disabled
Subnet ARN	arn:aws:ec2:us-west-2:504649076991:subnet/subnet-0f1e8b1d1ec9ac30d
Available IPv4 addresses	250
Network border group	us-west-2
Default subnet	No
Customer-owned IPv4 pool	-
IPv6-only	No
DNS64	Disabled
State	Available
IPv6 CIDR	-
VPC	vpc-0fbe6c9daab0312e4 ecs-efs-vpc
Auto-assign public IPv4 address	No
Outpost ID	-
Hostname type	IP name
Owner	504649076991
Block Public Access	Off
IPv6 CIDR association ID	-
Route table	rtb-0bb804d71ca1accc private-subnet-b-rt
Auto-assign IPv6 address	No
IPv4 CIDR reservations	-
Resource name DNS A record	Disabled

Verify NAT Gateways and Elastic IPs

1. In the VPC console, select **NAT Gateways** located in the left navigation bar.
2. Verify the existence of two NAT Gateways:
 - One in each public subnet
 - Each associated with its allocated Elastic IP
 - Status should be **Available**
 - Names: nat-gw-public-subnet-a / nat-gw-public-subnet-b

NAT gateways (2) <small>Info</small>								
<input type="text" value="Find NAT gateways by attribute or tag"/> Actions Create NAT gateway								
VPC	Name	NAT gateway ID	Connectivity...	Status	State message	Primary public I...	Primary private I...	Primary network interface II
vpc-0fbe6c9daab0312e4	nat-gw-public-subnet-a	nat-07e7363983d47a48d	Public	Available	-	54.191.105.159	10.0.1.41	eni-011f0fbc183c801e3
vpc-0fbe6c9daab0312e4	nat-gw-public-subnet-b	nat-014164dda0b35bbc2	Public	Available	-	52.42.172.5	10.0.2.17	eni-011f0fbc183c801e3

nat-07e7363983d47a48d / nat-gw-public-subnet-a								
Details	Secondary IPv4 addresses	Monitoring	Tags					
Details								
NAT gateway ID nat-07e7363983d47a48d	Connectivity type Public	State Available	State message -					
NAT gateway ARN arn:aws:ec2:us-west-2:504649076991:natgateway/nat-07e7363983d47a48d	Primary public IPv4 address 54.191.105.159	Primary private IPv4 address 10.0.1.41	Created Thursday, November 20, 2025 at 05:19:45 GMT+5					
VPC vpc-0fbe6c9daab0312e4 / ecs-efs-vpc	Subnet subnet-057229f2adc6206b1 / public-subnet-a	Subnet subnet-0500682eb56912...	Deleted -					

nat-014164dda0b35bbc2 / nat-gw-public-subnet-b								
Details	Secondary IPv4 addresses	Monitoring	Tags					
Details								
NAT gateway ID nat-014164dda0b35bbc2	Connectivity type Public	State Available	State message -					
NAT gateway ARN arn:aws:ec2:us-west-2:504649076991:natgateway/nat-014164dda0b35bbc2	Primary public IPv4 address 52.42.172.5	Primary private IPv4 address 10.0.2.17	Created Thursday, November 20, 2025 at 05:19:45 GMT+5					
VPC vpc-0fbe6c9daab0312e4 / ecs-efs-vpc	Subnet subnet-071584407c6fa0dd3 / public-subnet-b	Subnet subnet-0500682eb56912...	Deleted -					

Verify Route Tables

Route tables (4) <small>Info</small>								
<input type="text" value="Find route tables by attribute or tag"/> Actions Create route table								
VPC	Name	Route table ID	Explicit subnet assoc...	Edge associations	Main	VPC	Owner ID	Last updated
vpc-0fbe6c9daab0312e4	private-subnet-a-rt	rtb-014fa18bd57ac58e0	subnet-0500682eb56912...	-	No	vpc-0fbe6c9daab0312e4 ecs-efs-vpc	504649076991	4 minutes ago
vpc-0fbe6c9daab0312e4	private-subnet-b-rt	rtb-0bb80d71ca1acccc	subnet-0f1e8b1d1e9ac3...	-	No	vpc-0fbe6c9daab0312e4 ecs-efs-vpc	504649076991	
vpc-0fbe6c9daab0312e4	-	rtb-0e7ea0274e4d854b2	-	-	Yes	vpc-0fbe6c9daab0312e4 ecs-efs-vpc	504649076991	
vpc-0fbe6c9daab0312e4	public-rt	rtb-00970a8f31d86d2dd	2 subnets	-	No	vpc-0fbe6c9daab0312e4 ecs-efs-vpc	504649076991	

Verify Public Route Table and Routes Creation

- In the VPC console, choose **Route tables** located on the left navigation panel.
- Select the route table named **Public-Route-Table**.
- Check that it contains a route to the Internet Gateway with the destination **0.0.0.0/0**.
- Under **Subnet Associations**, verify that **public-subnet-a** and **public-subnet-b** are associated with this route table as shown under **Explicit subnet associations (2)**.

rtb-00970a8f31d86d2dd / public-rt								
Details	Routes	Subnet associations	Edge associations	Route propagation	Tags			
Routes (2)								
Destination	Target	Status	Propagated	Route Origin		Both	Edit routes	
0.0.0.0/0	igw-02e49cdd83d813ace	Active	No	Create Route				
10.0.0.0/16	local	Active	No	Create Route Table				

rtb-00970a8f31d86d2dd / public-rt					
Details	Routes	Subnet associations	Edge associations	Route propagation	Tags
Explicit subnet associations (2)					
Edit subnet associations					
Name	Subnet ID	IPv4 CIDR	IPv6 CIDR		
public-subnet-a	subnet-057229f2adc6206b1	10.0.1.0/24	-		
public-subnet-b	subnet-071584407c6fa0dd3	10.0.2.0/24	-		

Verify Private Route Tables and Routes Creation

1. In the VPC console, choose **Route tables** located on the left navigation panel.
2. Select both the route tables named **private-subnet-a-rt** and **private-subnet-b-rt**.
3. Check that both contain a route to the NAT gateway with the destination **0.0.0.0/0**.
4. Under **Subnet Associations**, verify that **private-subnet-a** and **private-subnet-b** are associated with their own route table shown under **Explicit subnet associations (1)**.

rtb-034fa18bd52ac58e0 / private-subnet-a-rt					
Details	Routes	Subnet associations	Edge associations	Route propagation	Tags
Routes (2)					
Edit routes					
Destination	Target	Status	Propagated	Route Origin	
0.0.0.0/0	nat-07e7363983d47a48d	Active	No	Create Route	
10.0.0.0/16	local	Active	No	Create Route Table	

rtb-034fa18bd52ac58e0 / private-subnet-a-rt					
Details	Routes	Subnet associations	Edge associations	Route propagation	Tags
Explicit subnet associations (1)					
Edit subnet associations					
Name	Subnet ID	IPv4 CIDR	IPv6 CIDR		
private-subnet-a	subnet-0500682eb569125a7	10.0.3.0/24	-		

rtb-0bb804d71ca1acccc / private-subnet-b-rt					
Details	Routes	Subnet associations	Edge associations	Route propagation	Tags
Routes (2)					
Edit routes					
Destination	Target	Status	Propagated	Route Origin	
0.0.0.0/0	nat-014164dda0b35bbc2	Active	No	Create Route	
10.0.0.0/16	local	Active	No	Create Route Table	

rtb-0bb804d71ca1acccc / private-subnet-b-rt					
Details	Routes	Subnet associations	Edge associations	Route propagation	Tags
Explicit subnet associations (1)					
Edit subnet associations					
Name	Subnet ID	IPv4 CIDR	IPv6 CIDR		
private-subnet-b	subnet-0f1e8b1d1ec9ac30d	10.0.4.0/24	-		

Verify Security Group Creation

Security Groups (4) <small>Info</small>					
<input type="text" value="Find security groups by attribute or tag"/> Actions <small>Export security groups to CSV</small> Create security group					
VPC ID = vpc-0fbe6cdaab0312e4	<input type="button" value="Clear filters"/>				
Name	Security group ID	Security group name	VPC ID	Description	Owner
ECS-SG	sg-0517afa73c643061e	ECS-SG	vpc-0fbe6cdaab0312e4	Allows HTTP traffic from the ALB-SG	504649076991
ALB-SG	sg-0fe2af8e6cafcd2d0	ALB-SG	vpc-0fbe6cdaab0312e4	Allows HTTP traffic from the internet	504649076991
-	sg-005f4fc2cb3de0a7	default	vpc-0fbe6cdaab0312e4	default VPC security group	504649076991
EFS-SG	sg-068c1ad62ec4d8f39	EFS-SG	vpc-0fbe6cdaab0312e4	Allows NFS traffic from ECS-SG	504649076991

Select a security group

Load Balancer Security Group

- In the VPC console, choose **Security groups** located on the left navigation panel.
- Locate the **ALB-SG** security group
- It should have the following inbound rules:

Type	Protocol	Port range	Source
HTTP	TCP	80	0.0.0.0/0

- Outbound rules should allow all traffic.

Inbound rules (1)						
<input type="text" value="Search"/> Manage tags <input type="button" value="Edit inbound rules"/>						
Name	Security group rule ID	IP version	Type	Protocol	Port range	Source
-	sgr-052f59108cd0ffa89	IPv4	HTTP	TCP	80	0.0.0.0/0

ECS Security Group

- In the VPC console, choose **Security groups** located on the left navigation panel.
- Locate the **ECS-SG** security group
- It should have the following inbound rules:

Type	Protocol	Port range	Source
HTTP	TCP	80	ALB-SG

- Outbound rules should allow all traffic.

Inbound rules (1)						
<input type="text" value="Search"/> Manage tags <input type="button" value="Edit inbound rules"/>						
Name	Security group rule ID	IP version	Type	Protocol	Port range	Source
-	sgr-09e172bb72ffa8e0	-	HTTP	TCP	80	sg-0fe2af8e6cafcd2d0 / ALB-SG

EFS Security Group

1. In the VPC console, choose **Security groups** located on the left navigation panel.
2. Locate the **EFS-SG** security group
3. It should have the following inbound rules:

Type	Protocol	Port range	Source
HTTP	TCP	2049	ECS-SG

4. Outbound rules should allow all traffic.

The screenshot shows the AWS VPC Security Groups Inbound Rules page for a security group named 'EFS-SG'. The 'Inbound rules (1)' section displays a single rule: 'sgr-0878549c0ee7def9' (Security group rule ID) for NFS (IP version) over TCP (Protocol) port 2049 (Port range) from source 'sg-0517afa73c643061e / ECS-SG' (Source). There are buttons for 'Manage tags' and 'Edit inbound rules' at the top right.

Verify ECR Creation

1. Sign in to the AWS Management Console.
2. Navigate to **ECR** service using the search bar at the top and then click **Repositories** under **Private registry** located in the left navigation bar.
3. Verify that the **nginx-repo** repository was created with a URI.

The screenshot shows the AWS ECR Private repositories page. It lists two repositories: 'nginx-repo' (URI: 504649076991.dkr.ecr.us-west-2.amazonaws.com/nginx-repo). The repository is created at November 20, 2025, 05:19:15 (UTC+05), is mutable, and uses AES-256 encryption. There are buttons for 'View push commands', 'Delete', 'Actions', and 'Create repository' at the top right.

Verify EFS File System and Access Point

The screenshot shows the AWS EFS File systems page. It lists one file system named 'efs-nginx' (File system ID: fs-04dd76219c9e574e8). The file system is encrypted, has a total size of 6.00 KiB, and is available. It was created on Nov 20, 2025, 00:19:15 GMT, and is located in the Regional availability zone. There are buttons for 'View details', 'Delete', and 'Create file system' at the top right.

1. Navigate to **EFS** console using the search bar.
2. Select **File systems** in the left navigation bar and verify that the file system named **efs-nginx** is deployed and assigned to the VPC created earlier
3. Under Networks tab, confirm that EFS file system is deployed in two Availability Zones (us-west-2a and us-west-2b)

Amazon resource name (ARN): arn:aws:elasticfilesystem:us-west-2:504640870991:file-system/fs-04dd76219c9e374e8

Performance mode: General Purpose

Throughput mode: Bursting

Lifecycle management: Transition into Inrequent Access (IA): None
Transition into Archive None
Transition into Standard: None

Availability zone: Regional

Automatic backups: Disabled

Encrypted: 745bed3-4d17-4e30-84fc-db434ebff4dd (aws/elasticfilesystem)

File system state: Available

DNS name: fs-04dd76219c9e374e8.efs.us-west-2.amazonaws.com

Replication overwrite protection: Enabled

Metered size | **Monitoring** | **Tags** | **File system policy** | **Access points** | **Network** | **Replication**

Network

Availability zone (AZ-ID)	Mount target ID	Subnet ID	VPC ID	Mount target state	IPv4 address	IPv6 address	Network interface ID	Security groups
us-west-2a (usw2-az1)	fsmt-081f07a794d4f96f2	subnet-0500682eb569125a	vpc-0fbe6c9daab0512e	Available	10.0.3.176	N/A	eni-0e5c288d072613aa	sg-06bc1ad62e4d8f39 (EFS-SG)
us-west-2b (usw2-az2)	fsmt-0294bfb50696967d4	subnet-0f1e8bd1d1ec9ac30	vpc-0fbe6c9daab0512e	Available	10.0.4.70	N/A	eni-0fd8872f71b2ca8a0	sg-06bc1ad62e4d8f39 (EFS-SG)

4. Now click **Access Points** in the left navigation bar.
5. Confirm that access point named **efs-nginx-access-point** exists.

Access points (1)

Name	Access point ID	File system ID	Path	POSIX user	Creation info	State
efs-nginx-access-point	fsap-088700a719dd0dd70	fs-04dd76219c9e374e8	/	1000:1000	1000:1000 (777)	Available

efs-nginx-access-point (fsap-088700a719dd0dd70)

Details

File system ID fs-04dd76219c9e374e8	POSIX user	Root directory creation permissions
Root directory path /	User ID 1000	Owner user ID 1000
State Available	Group ID 1000	Owner group ID 1000
Secondary group IDs	Permissions	Permissions 777

Access point ARN: arn:aws:elasticfilesystem:us-west-2:504640870991:access-point/fsap-088700a719dd0dd70

Tags

Tag key	Tag value
Name	efs-nginx-access-point

Verify IAM Roles and Policies

The screenshot shows the AWS IAM service interface. In the left navigation bar, 'Access management' is expanded, showing 'Roles' selected. The main area displays a list of roles with a search bar at the top. One role, 'ECS-Task-Execution-Role', is highlighted. Below the list, there's a section titled 'Roles Anywhere' with options for 'Access AWS from your non AWS workloads' and 'Temporary credentials'. A 'Manage' button is located in the top right corner of this section.

1. Navigate to **IAM** service and then select **Roles** in the left navigation bar.
2. Use search to locate the **ECS-Task-Execution-Role**.
3. Click on this IAM role and confirm attached policies:
 - o **AmazonECSTaskExecutionRolePolicy**
 - o Custom EFS access policy named **ECS-EFS-Access-Policy**

The screenshot shows the detailed view of the 'ECS-Task-Execution-Role'. The left sidebar shows 'Access management' expanded with 'Roles' selected. The main panel shows the role's summary, including its ARN (arn:aws:iam::504649076991:role/ECS-Task-Execution-Role) and maximum session duration (1 hour). The 'Permissions' tab is active, showing two attached policies: 'AmazonECSTaskExecutionRolePolicy' (AWS managed) and 'ECS-EFS-Access-Policy' (Customer managed). There are buttons for 'Simulate', 'Remove', and 'Add permissions'.

Verify CloudWatch Log Group

1. Navigate to **CloudWatch** service and select **Log groups** in the left navigation bar.
2. Verify the log group exists:
 - o Name: /ecs/nginx-container-app

The screenshot shows the 'Log groups' page in the CloudWatch service. The left sidebar shows 'Logs' expanded with 'Log groups' selected. The main area lists 21 log groups, with one named '/ecs/nginx-container-app' highlighted. The search bar at the top contains the name of the log group. There are filters and actions buttons at the top right.

Verify Task Definition Creation

The screenshot shows the 'Task definitions' page in the AWS Management Console. The left sidebar has 'Task definitions' selected under 'Amazon Elastic Container Service'. The main area lists six task definitions: 'nginx-task-definition' (Active), 'ecs-project1' (Active), 'ecs-project1-taskdef' (Active), 'nginx-task-definition' (Active), 'nginx-task-definition-with-efs' (Active), and 'test-def' (Active). A search bar at the top allows filtering by name.

1. Sign in to the AWS Management Console.
2. Navigate to **ECS** service using the search bar at the top and then click **Task definitions** located in the left navigation bar.
3. Verify that a task definition named **nginx-task-definition** is created.

The screenshot shows the details of the 'nginx-task-definition' task definition. It indicates there are 2 revisions. Revision 'nginx-task-definition:7' is selected and marked as 'Active'. Revision 'nginx-task-definition:1' is also listed as 'Active'.

Verify ECS Cluster, Service, and Tasks

1. Sign in to the AWS Management Console.
2. Navigate to **ECS** service using the search bar at the top.
3. Click **Clusters** located in the left navigation bar.
4. Verify that a cluster named **nginx-ecs-cluster** is already created.
5. Click on this **nginx-ecs-cluster**.

The screenshot shows the 'Services' page for the 'nginx-ecs-cluster'. The cluster overview shows ARN: 'arn:aws:ecs:us-west-2:504649076991:cluster/nginx-ecs-cluster' and Status: 'Active'. The 'Services' section shows one service, 'nginx-service', with ARN: 'arn:aws:ecs:us-west-2:504649076991:service/nginx-ecs-cluster/nginx-service' and Status: 'Active'. The 'Tasks' section shows 2 pending tasks. The bottom table lists the service with its ARN, status, scheduling strategy (FARGATE), and deployment count (0/2 Tasks running).

6. There should be a **Service** running with an **Active** status and **Deployments and tasks** showing **Tasks** running.
7. Click on **Tasks** tab. It should show Tasks in a **Stopped** state. This is because the **Tasks** are failing as there are **NO** images that have been **Pushed** to the ECR repositories (e.g. wordpress image).

Verify ALB and Target Group

1. Navigate to EC2 service and then click **Load Balancers** located in the left navigation bar.
2. Verify:
 - o ALB named **ecs-alb** exists
 - o Assigned to public subnets
 - o Using ALB-SG

3. Click **Listeners**
- o **HTTP:80** Forward to Target Group

Load balancer: nginx-ecs-alb

Listeners and rules (1) Info

A listener checks for connection requests on its configured protocol and port. Traffic received by the listener is routed according to the default action and any additional rules.

Protocol:Port	Default action	Rules	ARN	Security policy	Default SSL/TLS certificate	mTLS	Trust store
HTTP:80	Forward to target group nginx-target-group 1 (100%)	1 rule	arn:aws:elasticloadbalancing:us-west-2:2504649076991:targetgroup/nginx-target-group/208e6049a046f5df	Not applicable	Not applicable	Not applicable	Not applicable

4. Navigate to Target Groups

- Confirm the target group exists
- Targets should show **healthy** after tasks are running and the image has been pushed to ECR.

Note: It should display 0 Healthy targets before Docker image has been pushed to ECR repository. This screenshot is taken after the image was pushed to ECR repository.

EC2 > Target groups

Target groups (1 / 1) Info | What's new?

Name	ARN	Port	Protocol	Target type	Load balancer	VPC ID
nginx-target-group	arn:aws:elasticloadbalancing:us-west-2:2504649076991:targetgroup/nginx-target-group/208e6049a046f5df	80	HTTP	IP	nginx-ecs-alb	vpc-0fbe6c9daab0312e4

Target group: nginx-target-group

Details | Targets | Monitoring | Health checks | Attributes | Tags

Details

Target type: IP | Protocol: Port | Protocol version: HTTP | VPC: vpc-0fbe6c9daab0312e4

IP address type: IPv4 | Load balancer: nginx-ecs-alb

Total targets: 2 | Healthy: 2 | Unhealthy: 0 | Unused: 0 | Initial: 0 | Draining: 0

Load balancer: nginx-ecs-alb

Resource map Info

View, explore, and troubleshoot your load balancer's architecture.

Overview | Unhealthy target map | Show resource details

nginx-ecs-alb

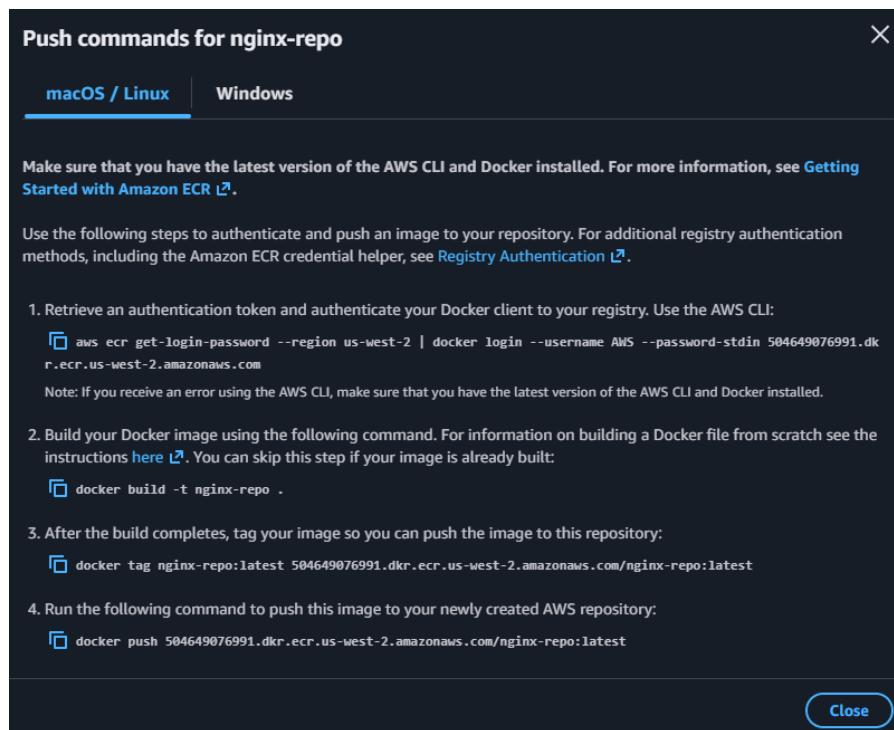
Last fetched seconds ago | Export

```

graph LR
    Listener[Listeners (1)] --- Rule[Rules (1)]
    Rule --- TargetGroup[Target groups (1) Info]
    TargetGroup --- Target[Targets (2)]
    Listener -- "HTTP:80, 1 rule" --> Rule
    Rule -- "Priority default, Forward to target group, Conditions (if), If no other rule applies" --> TargetGroup
    TargetGroup -- "IP: HTTP, nginx-target-group, 2 targets, 2 healthy" --> Target
    Target -- "10.0.1.175, Port 80, Healthy" --> Target
    Target -- "10.0.2.10, Port 80, Healthy" --> Target
  
```

Task 1.11.2: Push Docker Image to ECR

1. Navigate back to **ECR** service using the search bar.
2. Select .
3. Click the repository named **nginx-repo**.
4. Click **View push commands**.
5. Run the displayed commands locally on a terminal (Docker Desktop must be running).



- `aws ecr get-login-password --region us-west-2 | docker login --username AWS --password-stdin <Account_ID>.dkr.ecr.us-west-2.amazonaws.com`
- `docker build -t nginx-repo .`
- `docker tag nginx-repo:latest <Account_ID>.dkr.ecr.us-west-2.amazonaws.com/nginx-repo:latest`
- `docker push <Account_ID>.dkr.ecr.us-west-2.amazonaws.com/nginx-repo:latest`

```

umars@Umar-Satti MINGW64 /d/Cloudelligent/Task-5-ECS-Fargate-Nginx-Server-Deployment-with-ALB-and-EFS-using-Terraform
● $ aws ecr get-login-password --region us-west-2 | docker login --username AWS --password-stdin 504649076991.dkr.ecr.us-west-2.amazonaws.com
Login Succeeded

umars@Umar-Satti MINGW64 /d/Cloudelligent/Task-5-ECS-Fargate-Nginx-Server-Deployment-with-ALB-and-EFS-using-Terraform
● $ docker build -t nginx-repo .
[+] Building 2.1s (10/10) FINISHED
=> [internal] load build definition from Dockerfile
=> transferring dockerfile: 643B
=> [internal] load metadata for docker.io/library/nginx:stable-alpine
=> [auth] library/nginx:pull token for registry-1.docker.io
=> [internal] load .dockerignore
=> => transferring context: 2B
=> [1/4] FROM docker.io/library/nginx:stable-alpine@sha256:30f1c0d78e0ad60901648be663a710bdadf19e4c10ac6782c235200619158284
=> [internal] load build context
=> => transferring context: 958
=> CACHED [2/4] RUN rm /etc/nginx/conf.d/default.conf
=> CACHED [3/4] COPY nginx.conf /etc/nginx/conf.d/nginx.conf
=> CACHED [4/4] COPY /static/index.html /usr/share/nginx/html/index.html
=> exporting to image
=> => exporting layers
=> => writing image sha256:d81a36e9c91c173ed62ae6b30669b8a04599099c5fe7cc388b30fe2150ee9242
=> => naming to docker.io/library/nginx-repo

View build details: docker-desktop://dashboard/build/desktop-linux/desktop-linux/ial16zy9dpde2khsig929jfmk

What's next:
  View a summary of image vulnerabilities and recommendations → docker scout quickview

umars@Umar-Satti MINGW64 /d/Cloudelligent/Task-5-ECS-Fargate-Nginx-Server-Deployment-with-ALB-and-EFS-using-Terraform
○ $ 

```

```

umars@Umar-Satti MINGW64 /d/Cloudelligent/Task-5-ECS-Fargate-Nginx-Server-Deployment-with-ALB-and-EFS-using-Terraform
● $ docker tag nginx-repo:latest 504649076991.dkr.ecr.us-west-2.amazonaws.com/nginx-repo:latest

umars@Umar-Satti MINGW64 /d/Cloudelligent/Task-5-ECS-Fargate-Nginx-Server-Deployment-with-ALB-and-EFS-using-Terraform
● $ docker push 504649076991.dkr.ecr.us-west-2.amazonaws.com/nginx-repo:latest
The push refers to repository [504649076991.dkr.ecr.us-west-2.amazonaws.com/nginx-repo]
c191707ab34d: Pushed
9a2661e813ed: Pushed
a028359e97dd: Pushed
90ec27130398: Pushed
a231a657395e: Pushed
5f23a9cf34f1: Pushed
7d9abb9ab3b7: Pushed
34034c523565: Pushed
570a1c87f279: Pushed
412e147b334c: Pushed
922ec217407c: Pushed
latest: digest: sha256:2346d324794e3fc27690bc493a75892d1fdff243fe7d82c5259e8e0c6b987e55 size: 2610

```

Verify the images have been pushed

1. Click the **nginx-repo** repository.
2. An image should be available with the **latest** tag.

Image tags	Type	Created at	Image size	Image digest	Last pulled at
latest	Image	November 20, 2025, 05:51:42 (UTC+05)	21.00	sha256:2346d324794e3fc2...	November 20, 2025, 05:51:59 (UTC+05)

Verify the Service and Tasks are running

1. Navigate to **ECS** service using the search bar at the top and then click **Clusters** located in the left navigation bar.
2. Under the **Service** tab, it should display two tasks running successfully (2/2).

The screenshot shows the AWS ECS console for the 'nginx-ecs-cluster'. The 'Services' tab is selected. Key details include:

- ARN:** arn:aws:ecs:us-west-2:504649076991:cluster/nginx-ecs-cluster
- Status:** Active
- CloudWatch monitoring:** Enabled (green)
- Registered container instances:** 2
- Tasks:** Pending 0, Running 2

The 'Services' table lists one service entry:

Service name	ARN	Status	Launch type	Scheduling strategy	Task definition	Deployments and tasks
nginx-service	arn:aws:ecs:us-west-2:504649076991:task-definition/nginx-task-definition	Active	REPLICAS	FARGATE	nginx-task-defi...	2/2 Tasks running

Task 1.11.3: Confirm Application is Accessible

1. Navigation to EC2 service using the search bar.
2. Scroll down and click **Load Balancers** located in the left navigation bar.
3. Locate the **ALB DNS Name** assigned and copy it.
4. Open a web browser and paste the DNS.
5. It should display a static webpage.



Hello from Nginx on Fargate!

This is a test page for the ECS Fargate deployment with ALB and EFS.

Task 1.12: Clean Up

To delete the resources, run `terraform destroy` or `terraform destroy --auto-approve`

Task 1.13: Troubleshooting

Issue 1: EFS Not Mounting to ECS Tasks

Problem Description:

ECS Tasks consistently failed to start, and the task logs displayed EFS mount errors such as “failed to resolve EFS mount target.”

Root Cause:

The Terraform VPC module did not have `enable_dns_hostnames = true` enabled, which is required for EFS.

EFS mount targets depend on DNS names (e.g., `fs-xxxx.efs.<region>.amazonaws.com`). If DNS hostnames are disabled, ECS cannot resolve the EFS DNS endpoint, causing the mount operation to fail.

Solution:

The VPC configuration in Terraform was updated to include:

- `enable_dns_hostnames = true`

Once applied, EFS DNS names resolved correctly and ECS tasks were able to mount the file system successfully.

Issue 2: EFS Mount Failure Due to Invalid Container Path

Problem Description:

Even after fixing DNS hostname support, ECS tasks still failed with mount-related errors. The container mount path was shown as invalid in task logs.

Root Cause:

The **container path** for the EFS volume was incorrectly set to ‘/’, which is the root directory of the container filesystem. ECS does not allow mounting over the container root path.

Solution:

The EFS volume mount path inside the container was changed to a valid directory such as:

- `/mnt/data`
- `/mnt/efs`

After updating the ECS task definition, the tasks started successfully and the EFS volume mounted as expected.

Issue 3: Terraform State Lock Preventing Destroy Operation

Problem Description:

Attempting to execute `terraform destroy --auto-approve` resulted in the following message:

- Error acquiring the state lock

Root Cause:

Changes were made to Terraform configuration files after the resources were deployed, causing Terraform to detect a state mismatch and lock the state to prevent accidental corruption.

Solution:

The state lock was manually released using:

- `terraform force-unlock <LOCK_ID>`

After unlocking, `terraform destroy` was completed successfully.

```
PS D:\Cloudelligent\Task-5-ECS-Fargate-Nginx-Server-Deployment-with-ALB-and-EFS-using-Terraform\terraform> terraform destroy --auto-approve
Acquiring state lock. This may take a few moments...

Error: Error acquiring the state lock

Error message: operation error S3: PutObject, https response error StatusCode: 412, RequestID: 63F36R4K0A9S3Y6X, HostID: 3oIaqRuedob8ikSVGQyTfg1NCwRnm/M/1DWg5bNczcGE02cpyCgbW9LuVQgNPtbVzWbZZhueeY=, api error PreconditionFailed: At least one of the pre-conditions you specified did not hold

Lock Info:
ID: db22a328-9875-3f6a-eac0-72baa9ad0f60
Path: umarsatti-terraform-s3-bucket-state-file/Task-5/terraform.tfstate
Operation: OperationTypeApply
Who: UMAR-SATTI\umars@Umar-Satti
Version: 1.12.2
Created: 2025-11-20 01:02:07.924961 +0000 UTC
Info:

Terraform acquires a state lock to protect the state from being written by multiple users at the same time. Please resolve the issue above and try again. For most commands, you can disable locking with the "-lock=false" flag, but this is not recommended.

● PS D:\Cloudelligent\Task-5-ECS-Fargate-Nginx-Server-Deployment-with-ALB-and-EFS-using-Terraform\terraform> terraform force-unlock db22a328-9875-3f6a-eac0-72baa9ad0f60
Do you really want to force-unlock?
Terraform will remove the lock on the remote state.
This will allow local Terraform commands to modify this state, even though it may still be in use. Only 'yes' will be accepted to confirm.

Enter a value: yes

Terraform state has been successfully unlocked!

The state has been unlocked, and Terraform commands should now be able to obtain a new lock on the remote state.
○ PS D:\Cloudelligent\Task-5-ECS-Fargate-Nginx-Server-Deployment-with-ALB-and-EFS-using-Terraform\terraform> terraform destroy --auto-approve
```