Sardar Noor Ul Hassan

Cloud Intern

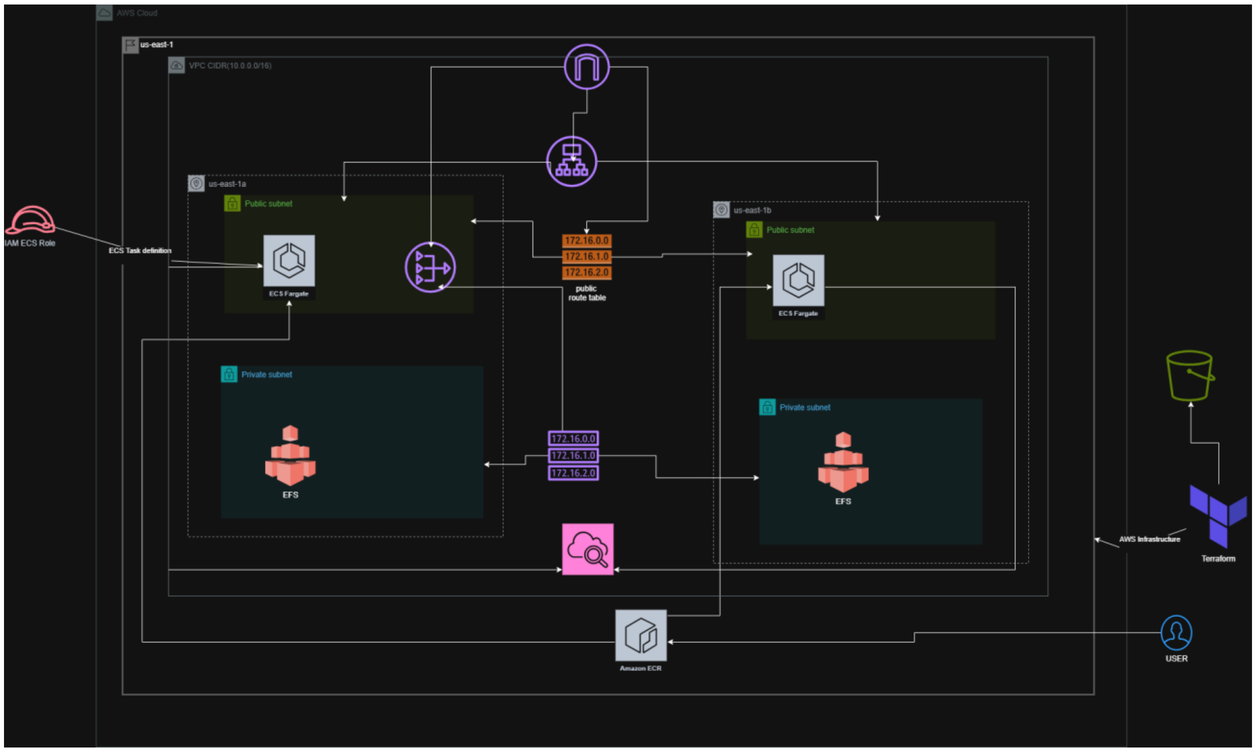
ECS Fargate Nginx Server Deployment with ALB and EFS using Terraform

Contents

1. Overview

In this project I deployed a highly available Nginx web server on **Amazon ECS Fargate**, fronted by an **Application Load Balancer (ALB)** and using **Amazon EFS** for persistent storage.
Everything is provisioned using **Terraform** (with a remote S3 backend), and the application itself runs from a **custom Docker image** stored in **ECR**.

Architecture Diagram



2. Design and create a highly available VPC with multiple public and private subnets

1.1 What I wanted to achieve

For this project, I needed a properly structured, isolated VPC that could support a production-grade ECS Fargate environment. My main goals were:

To use non-overlapping CIDR blocks so the VPC can be extended or peered later without conflicts.

To spread the network across multiple Availability Zones for fault tolerance and high availability.

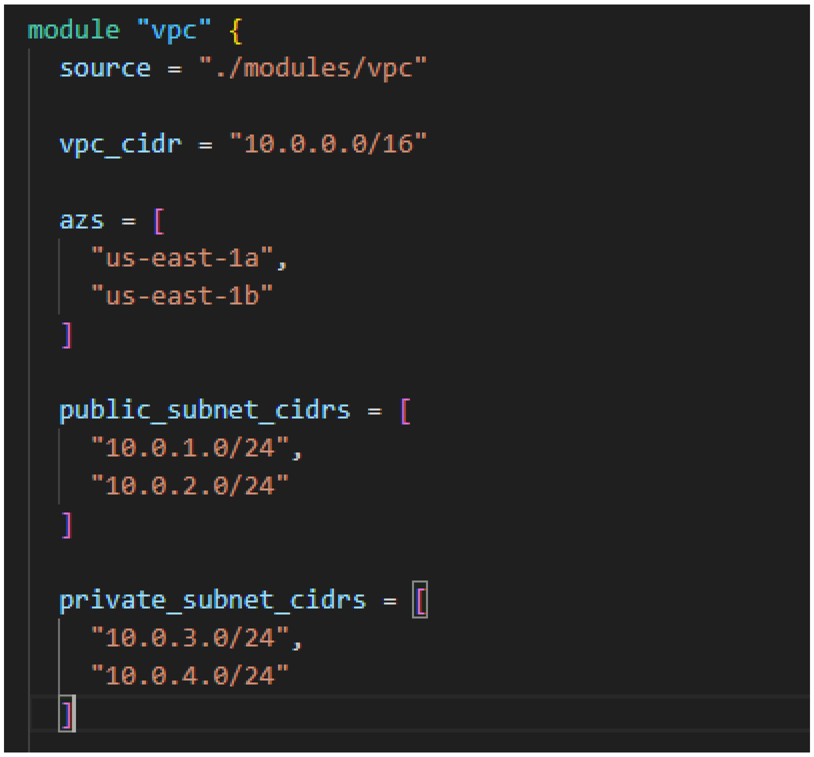
To create public subnets for internet-facing components such as the ALB and Fargate tasks (in this project).

To also create private subnets, which form the application/backend layer of the VPC and support services like EFS or any future internal microservices.

1.2 Design decisions

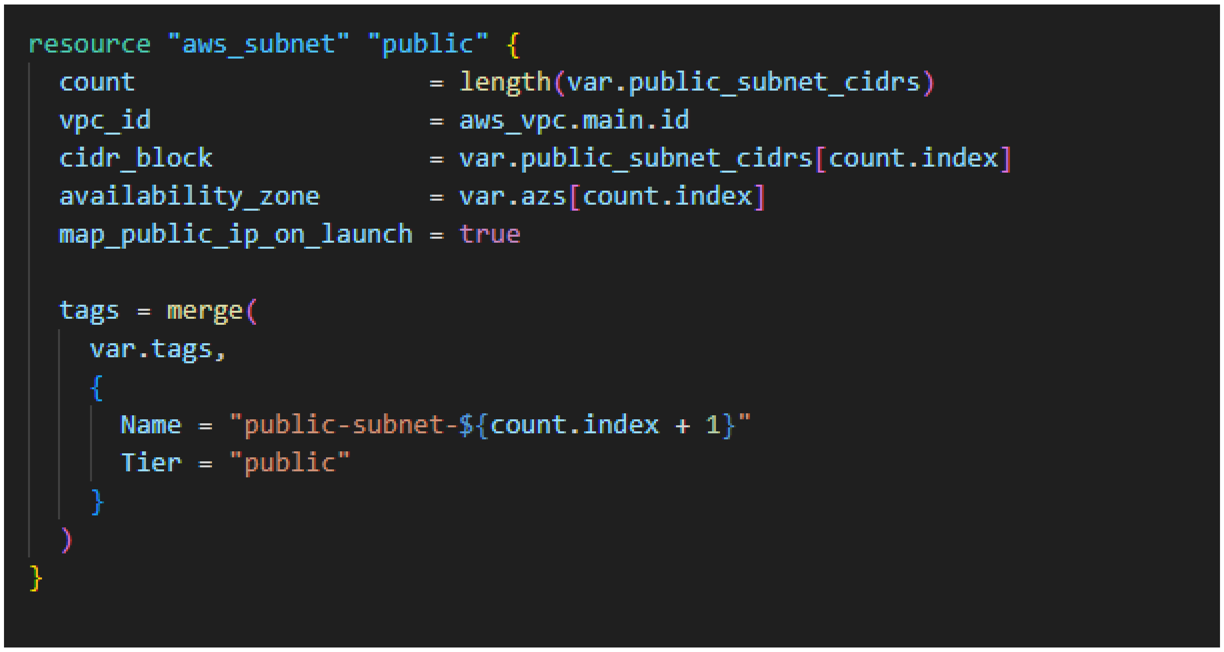
• VPC CIDR

I selected a CIDR range such as 10.0.0.0/16.
This gives a large IP space (up to 65k IPs), which is more than enough to divide into multiple public and private subnets, and it leaves room for future services, NAT Gateways, EFS mount targets, and potential VPC peering.



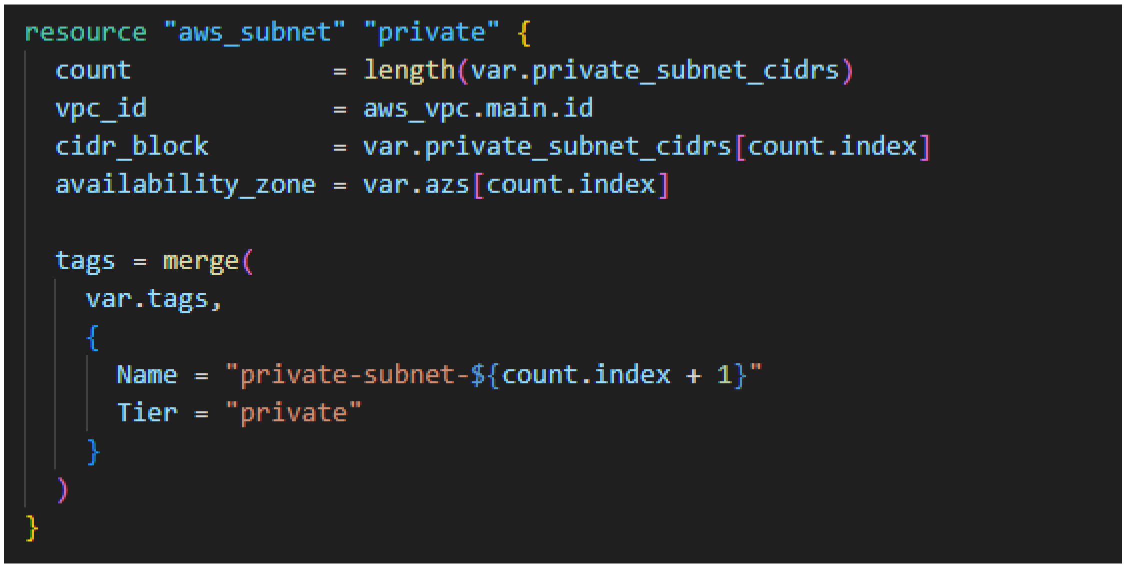
```
module "vpc" {
  source = "./modules/vpc"

  vpc_cidr = "10.0.0.0/16"

  azs = [
    "us-east-1a",
    "us-east-1b"
  ]

  public_subnet_cidrs = [
    "10.0.1.0/24",
    "10.0.2.0/24"
  ]

  private_subnet_cidrs = [
    "10.0.3.0/24",
    "10.0.4.0/24"
  ]
```

- Public Subnets (2 total)



```
resource "aws_subnet" "public" {
  count                   = length(var.public_subnet_cidrs)
  vpc_id                  = aws_vpc.main.id
  cidr_block              = var.public_subnet_cidrs[count.index]
  availability_zone       = var.azs[count.index]
  map_public_ip_on_launch = true

  tags = merge(
    var.tags,
    {
      Name = "public-subnet-${count.index + 1}"
      Tier = "public"
    }
  )
}
```

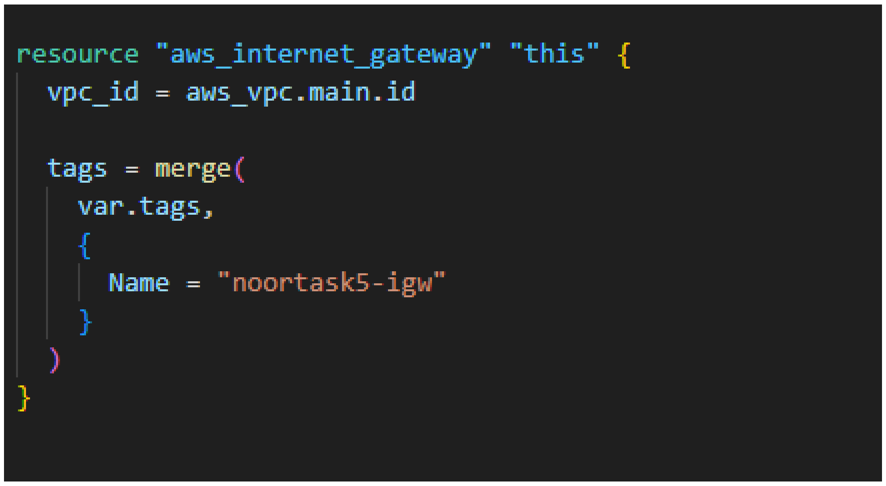
• Private Subnets (2 total)



```
resource "aws_subnet" "private" {
  count             = length(var.private_subnet_cidrs)
  vpc_id            = aws_vpc.main.id
  cidr_block        = var.private_subnet_cidrs[count.index]
  availability_zone = var.azs[count.index]

  tags = merge(
    var.tags,
    {
      Name = "private-subnet-${count.index + 1}"
      Tier = "private"
    }
  )
}
```

• Internet Gateway (IGW)

I attached an Internet Gateway to the VPC so public subnets can provide outbound internet access and the ALB can receive incoming traffic from the internet.



```
resource "aws_internet_gateway" "this" {
  vpc_id = aws_vpc.main.id

  tags = merge(
    var.tags,
    {
      Name = "noortask5-igw"
    }
  )
}
```

• NAT Gateway (for private subnets)

Because private subnets cannot directly access the internet, I deployed a NAT Gateway in one of the public subnets, along with an Elastic IP address.

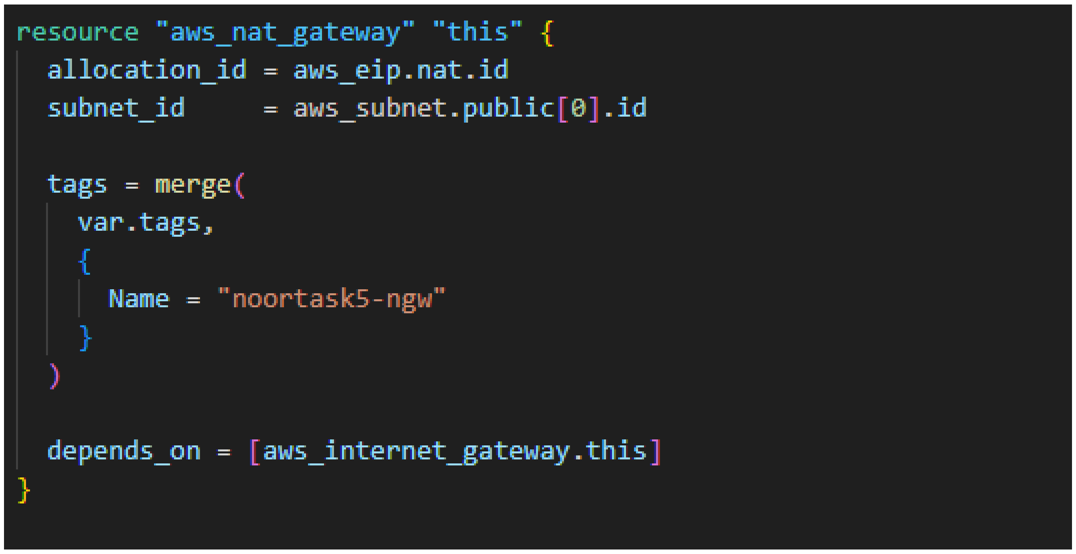
This allows instances in private subnets to:

Download OS packages

Communicate with AWS APIs

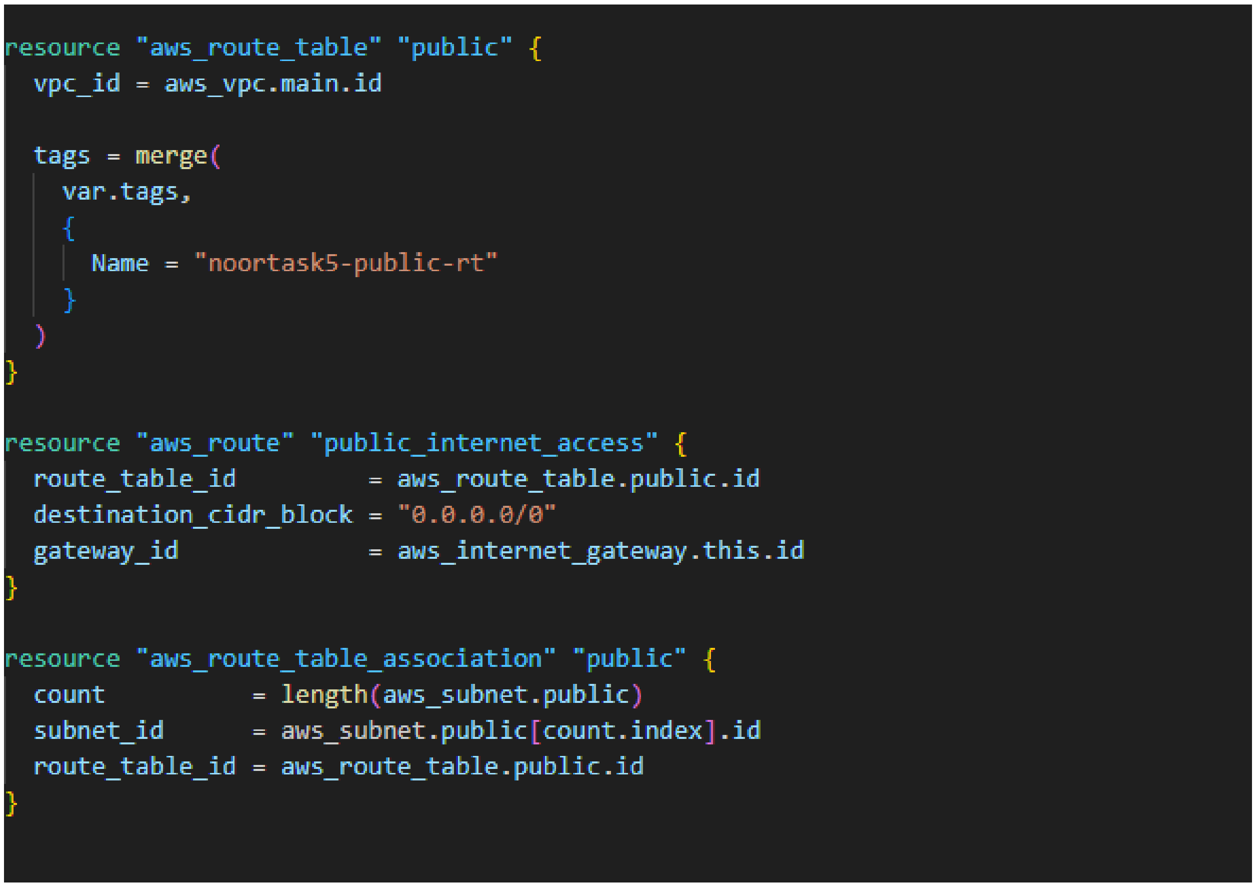
Access ECR/ECS endpoints

Reach the internet securely without being publicly exposed

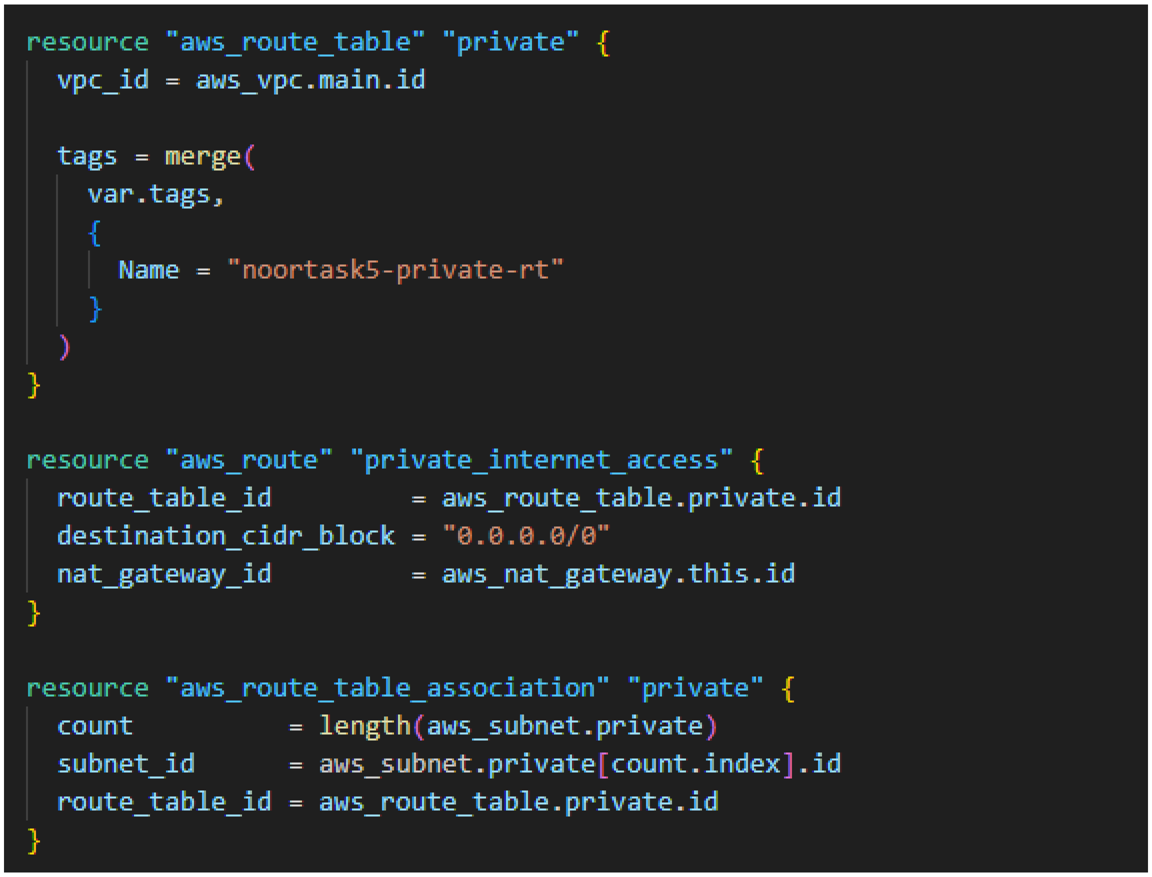


```
resource "aws_nat_gateway" "this" {
  allocation_id = aws_eip.nat.id
  subnet_id     = aws_subnet.public[0].id

  tags = merge(
    var.tags,
    {
      Name = "noortask5-ngw"
    }
  )

  depends_on = [aws_internet_gateway.this]
}
```

- Route Tables



```
resource "aws_route_table" "public" {
  vpc_id = aws_vpc.main.id

  tags = merge(
    var.tags,
    {
      Name = "noortask5-public-rt"
    }
  )
}

resource "aws_route" "public_internet_access" {
  route_table_id         = aws_route_table.public.id
  destination_cidr_block = "0.0.0.0/0"
  gateway_id             = aws_internet_gateway.this.id
}

resource "aws_route_table_association" "public" {
  count          = length(aws_subnet.public)
  subnet_id      = aws_subnet.public[count.index].id
  route_table_id = aws_route_table.public.id
}
```



```
resource "aws_route_table" "private" {
  vpc_id = aws_vpc.main.id

  tags = merge(
    var.tags,
    {
      Name = "noortask5-private-rt"
    }
  )
}

resource "aws_route" "private_internet_access" {
  route_table_id         = aws_route_table.private.id
  destination_cidr_block = "0.0.0.0/0"
  nat_gateway_id         = aws_nat_gateway.this.id
}

resource "aws_route_table_association" "private" {
  count          = length(aws_subnet.private)
  subnet_id      = aws_subnet.private[count.index].id
  route_table_id = aws_route_table.private.id
}
```

3. Configure security groups for ALB, ECS tasks, and EFS access

3.1 What I wanted to achieve

I wanted clear separation between:

- External traffic from the internet.
- Internal traffic from ALB to ECS tasks.
- Storage traffic between ECS tasks and EFS.

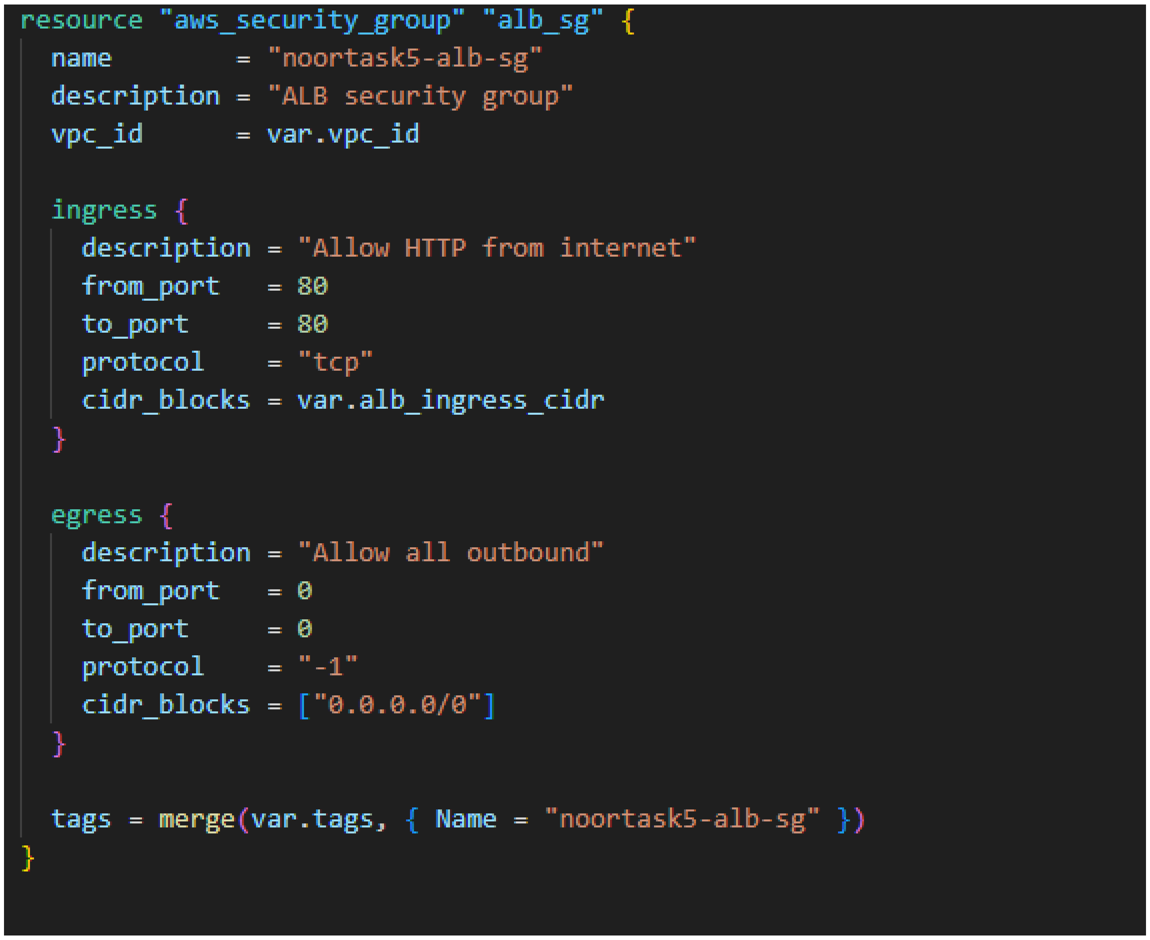
The rule was: nothing should be "open to the world" unless absolutely required.

3.2 Design decisions

I created three main Security Groups:

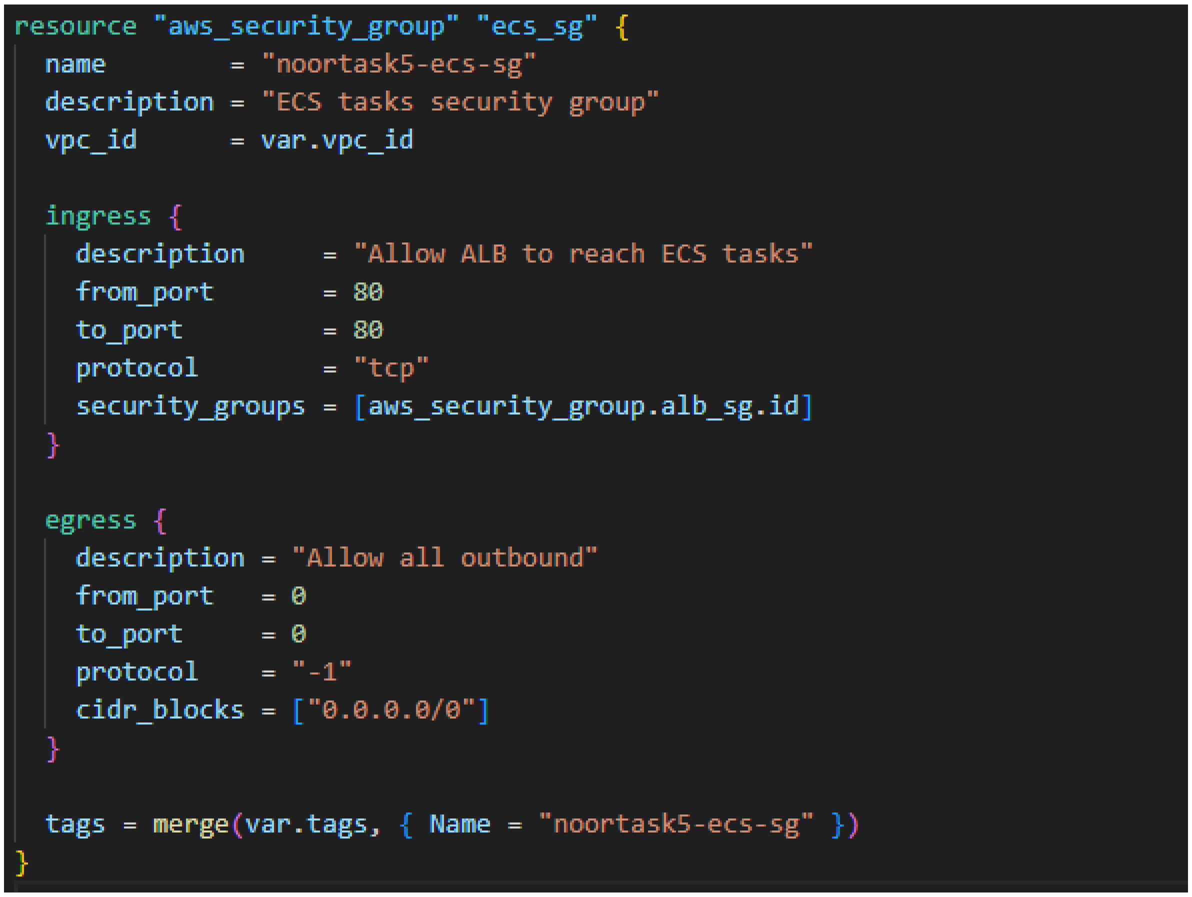
1. **ALB Security Group**

- Inbound:
  - HTTP (80) from 0.0.0.0/0 so users can reach the application.
- Outbound:
  - Allowed to all (0.0.0.0/0) to reach ECS tasks and perform health checks.

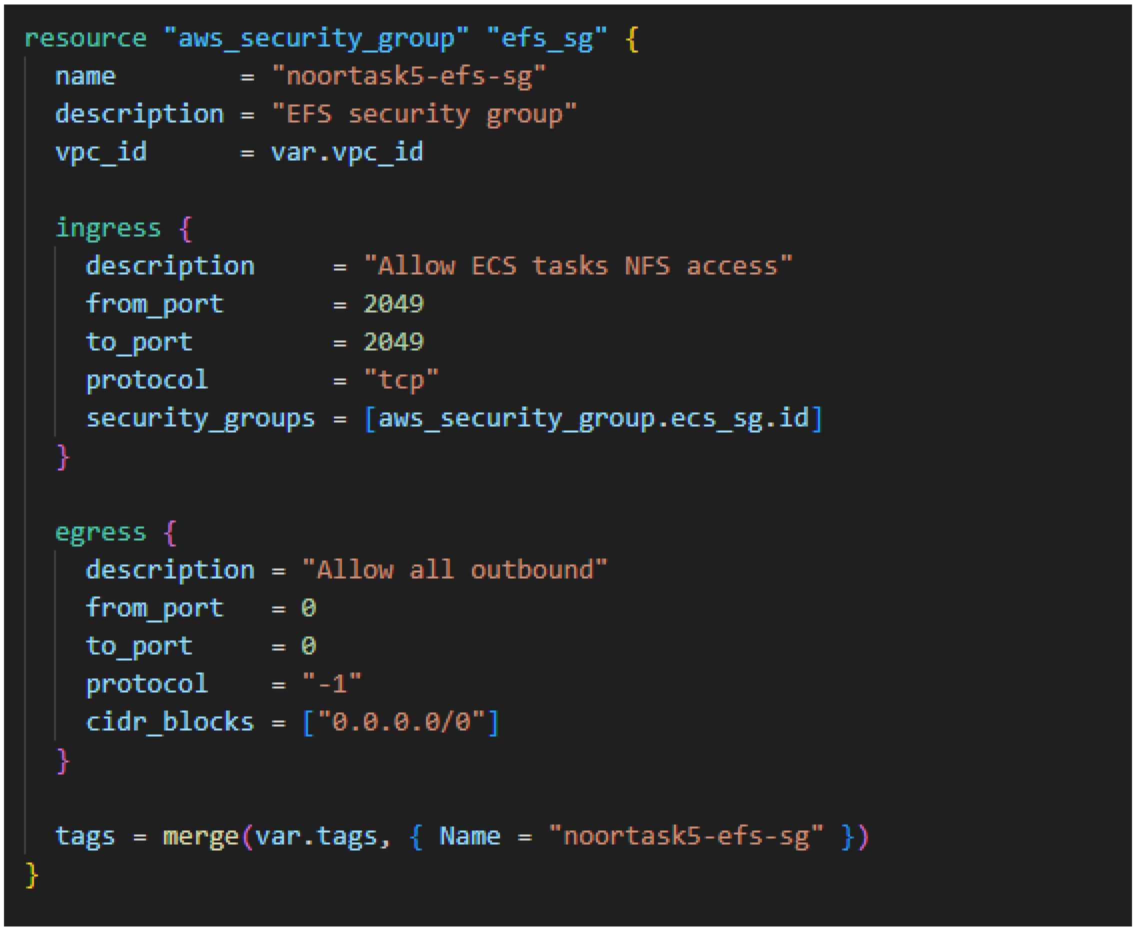


```
resource "aws_security_group" "alb_sg" {
  name        = "noortask5-alb-sg"
  description = "ALB security group"
  vpc_id      = var.vpc_id

  ingress {
    description = "Allow HTTP from internet"
    from_port   = 80
    to_port     = 80
    protocol    = "tcp"
    cidr_blocks = var.alb_ingress_cidr
  }

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

  tags = merge(var.tags, { Name = "noortask5-alb-sg" })
}
```

2. **ECS Tasks Security Group**
   - Inbound:
     - HTTP (80) only from the **ALB Security Group**, not from the whole internet.
   - Outbound:
     - Allowed to all so containers can pull image layers, talk to EFS, etc.


```
resource "aws_security_group" "ecs_sg" {
  name        = "noortask5-ecs-sg"
  description = "ECS tasks security group"
  vpc_id      = var.vpc_id

  ingress {
    description     = "Allow ALB to reach ECS tasks"
    from_port       = 80
    to_port         = 80
    protocol        = "tcp"
    security_groups = [aws_security_group.alb_sg.id]
  }

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

  tags = merge(var.tags, { Name = "noortask5-ecs-sg" })
}
```

3. **EFS Security Group**
   - Inbound:
     - NFS (2049) only from the **ECS Tasks Security Group**.
   - Outbound:
     - Allowed to all (default) which is fine for EFS.


```
resource "aws_security_group" "efs_sg" {
  name        = "noortask5-efs-sg"
  description = "EFS security group"
  vpc_id      = var.vpc_id

  ingress {
    description     = "Allow ECS tasks NFS access"
    from_port       = 2049
    to_port         = 2049
    protocol        = "tcp"
    security_groups = [aws_security_group.ecs_sg.id]
  }

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

  tags = merge(var.tags, { Name = "noortask5-efs-sg" })
}
```

This chained approach makes the path clear:
Internet → ALB-SG → ECS-SG → EFS-SG

---

4. Create an ECS cluster with Terraform

4.1 Goal

Provide a logical group where my Fargate services and tasks will run.

4.2 Design decisions

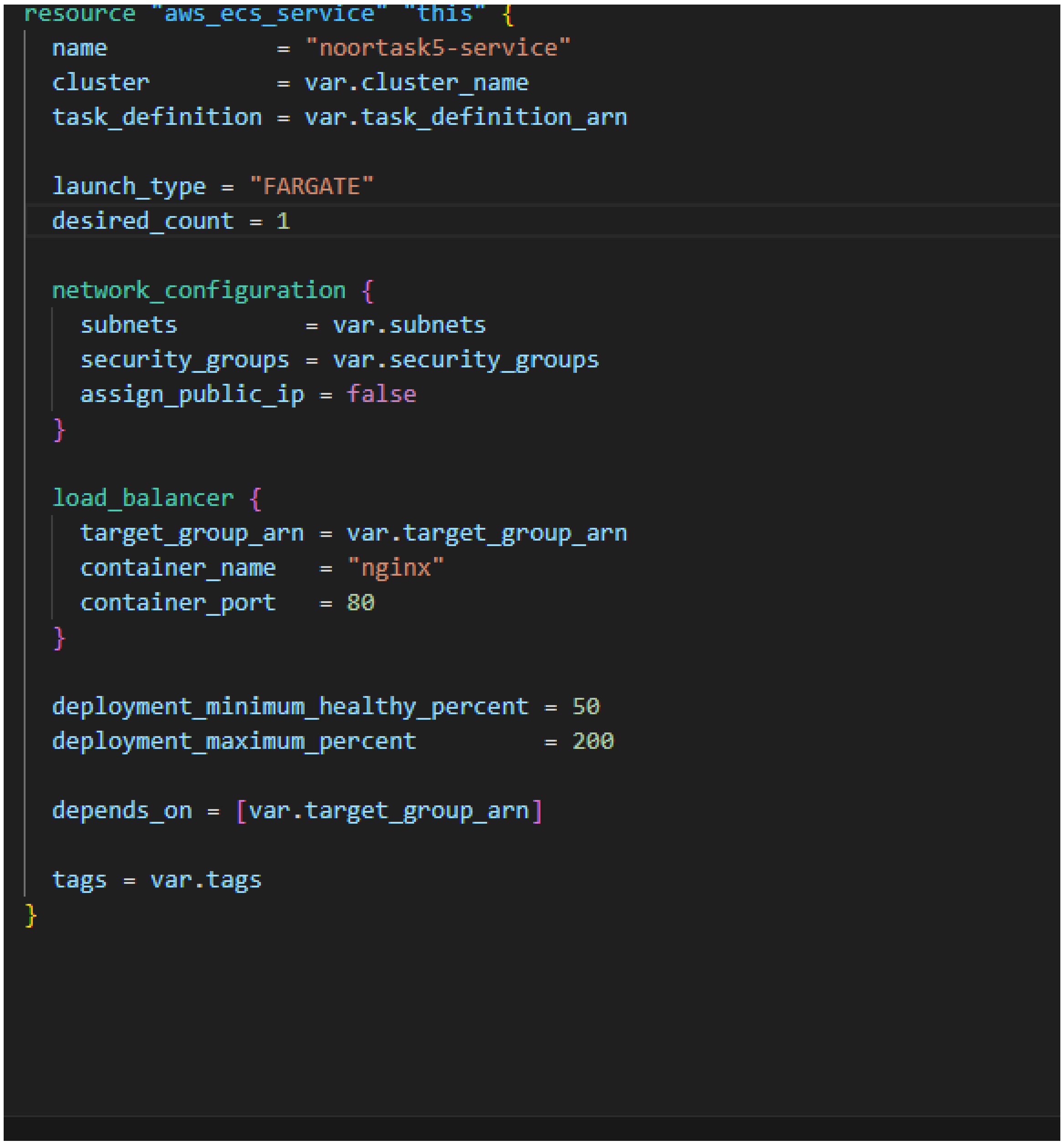
- I used **Fargate** launch type so I don't manage EC2 instances at all.
- I kept the cluster simple: one ECS cluster specifically for this project, which makes monitoring and cleanup easier.

4.3 Implementation

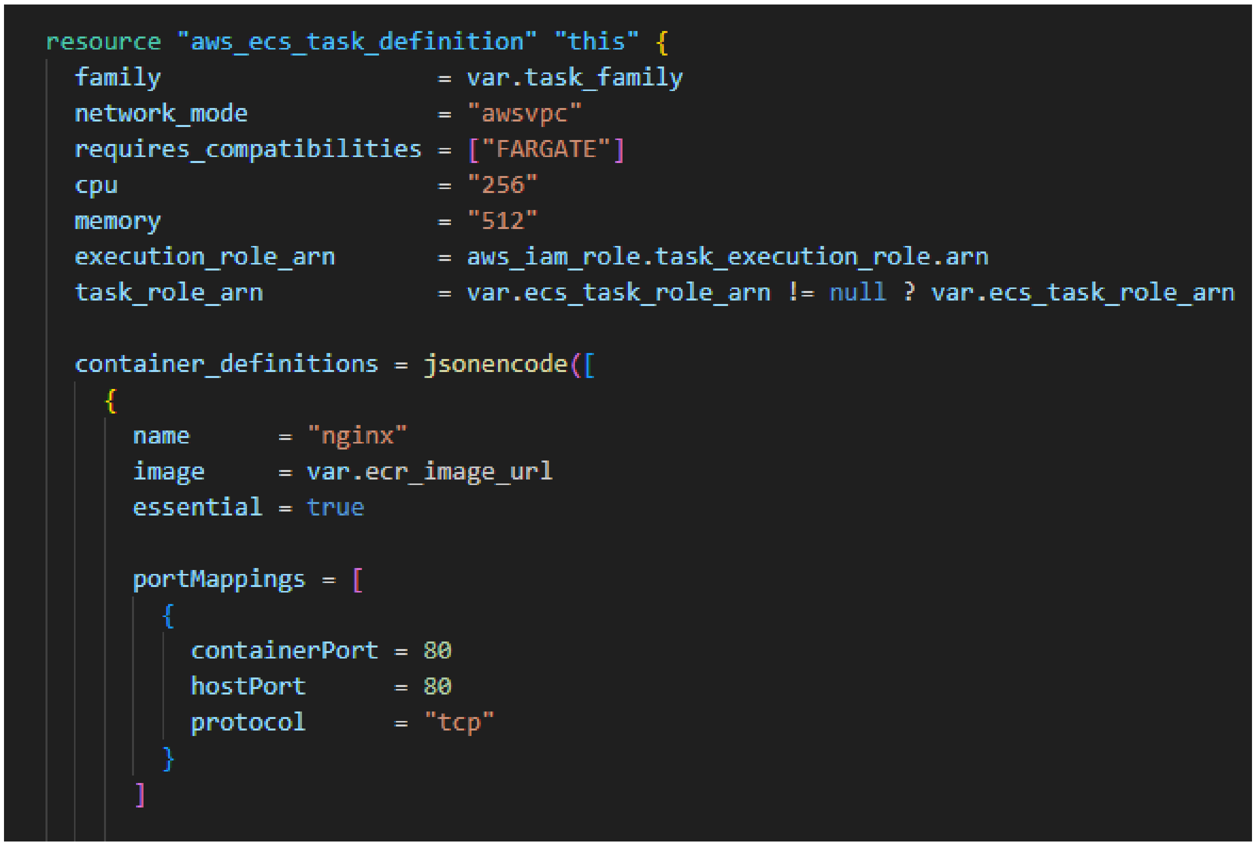
In a dedicated ECS cluster module I defined an aws_ecs_cluster resource. Key points:

- Set a readable name like "noortask5-ecs-cluster".
- Enabled container insights (optional but helpful for CloudWatch metrics).
- No capacity providers explicitly; I kept it basic and used launch_type = "FARGATE" in the service.

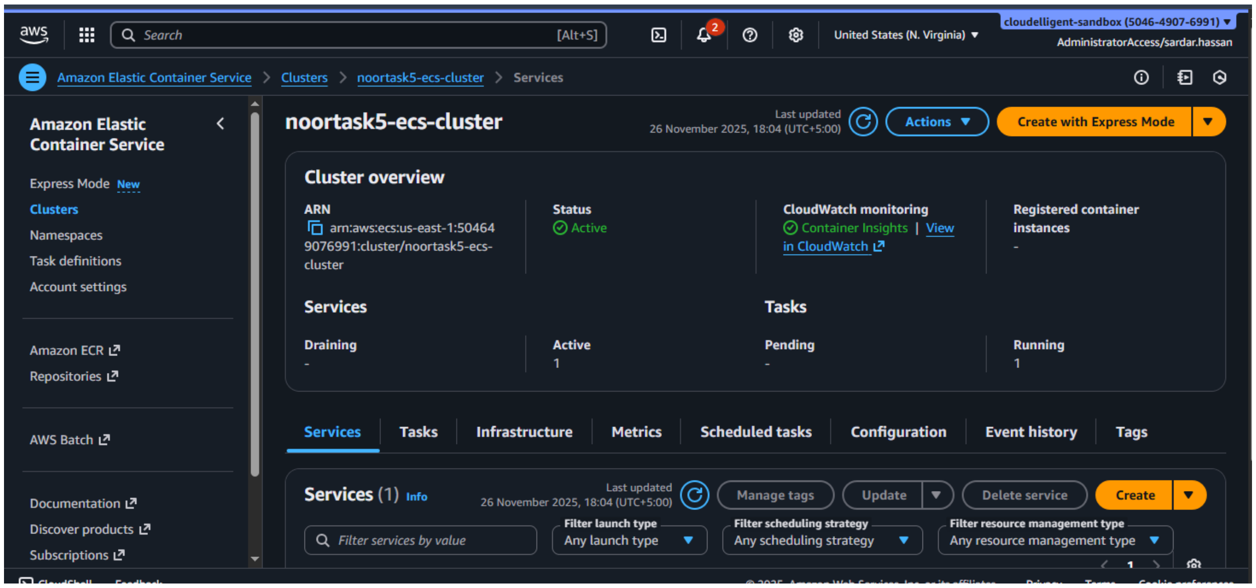
Terraform outputs the cluster name/ARN, which I later used when defining the ECS service.



```
resource "aws_ecs_service" "this" {
  name            = "noortask5-service"
  cluster         = var.cluster_name
  task_definition = var.task_definition_arn

  launch_type = "FARGATE"
  desired_count = 1

  network_configuration {
    subnets         = var.subnets
    security_groups = var.security_groups
    assign_public_ip = false
  }

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

  deployment_minimum_healthy_percent = 50
  deployment_maximum_percent         = 200

  depends_on = [var.target_group_arn]

  tags = var.tags
}
```



```
resource "aws_ecs_task_definition" "this" {
  family                   = var.task_family
  network_mode             = "awsvpc"
  requires_compatibilities = ["FARGATE"]
  cpu                      = "256"
  memory                   = "512"
  execution_role_arn       = aws_iam_role.task_execution_role.arn
  task_role_arn            = var.ecs_task_role_arn != null ? var.ecs_task_role_arn

  container_definitions = jsonencode([
    {
      name      = "nginx"
      image     = var.ecr_image_url
      essential = true

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



noortask5-ecs-cluster

Last updated
26 November 2025, 18:04 (UTC+5:00)

Cluster overview

ARN
arn:aws:ecs:us-east-1:504649076991:cluster/noortask5-ecs-cluster

Status
Active

CloudWatch monitoring
Container Insights | View in CloudWatch

Registered container instances
-

Services

Draining
-

Active
1

Tasks

Pending
-

Running
1

Services (1)

Last updated
26 November 2025, 18:04 (UTC+5:00)

Filter services by value

Filter launch type
Any launch type

Filter scheduling strategy
Any scheduling strategy

Filter resource management type
Any resource management type

5. Build and push a custom Docker image to a container registry (ECR)

5.1 What I wanted to achieve

I didn't want to use the default Nginx image directly. I needed:

- My own **HTML landing page** with my name and project title.
- A small startup script to handle EFS logic before Nginx launches.

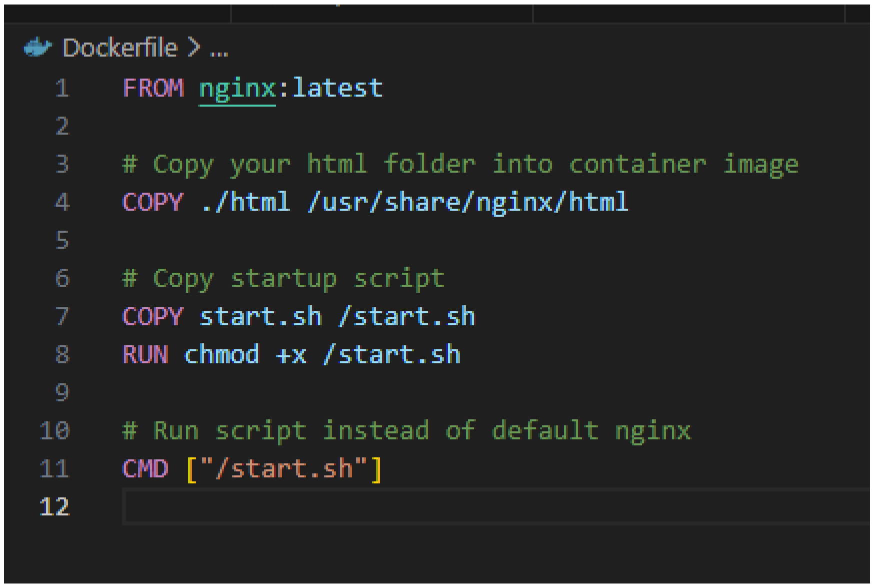
So I built a custom Docker image and pushed it to **Amazon ECR**.

5.2 Application content

- **index.html** contains a simple but customized page: dark background, neon-style text, and a message "Welcome to Noor's ECS Fargate(ALB+EFS) Nginx Project".
- I used a **start.sh** script as the container's entrypoint to handle EFS logic (described in more detail later).

5.3 Dockerfile logic

The Dockerfile:


Dockerfile > ...

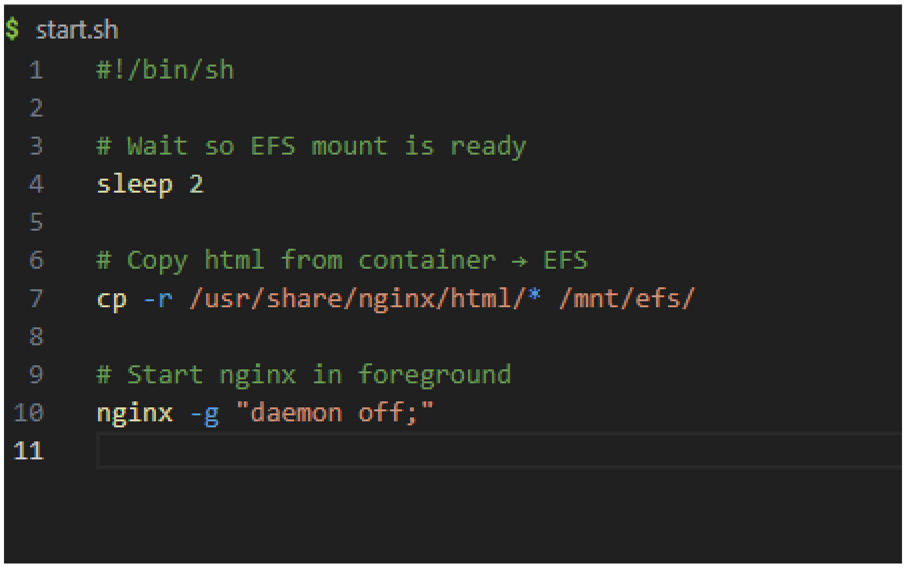
```
FROM nginx:latest

# Copy your html folder into container image
COPY ./html /usr/share/nginx/html

# Copy startup script
COPY start.sh /start.sh
RUN chmod +x /start.sh

# Run script instead of default nginx
CMD ["/start.sh"]

```

- Uses an Nginx base image.
- Copies my index.html into Nginx's web root (e.g. /usr/share/nginx/html/index.html).
- Copies start.sh into the image and makes it executable.

- Sets the container's command/entrypoint to run start.sh.

start.sh itself does three things:

1. **Sleep briefly** so that the EFS mount inside the task has time to be ready.
2. **Copy HTML content** from the container path /usr/share/nginx/html/ into the EFS mount path /mnt/efs/.
3. **Start Nginx in the foreground** with nginx -g "daemon off;".



```
$ start.sh
#!/bin/sh

# Wait so EFS mount is ready
sleep 2

# Copy html from container → EFS
cp -r /usr/share/nginx/html/* /mnt/efs/

# Start nginx in foreground
nginx -g "daemon off;"
```

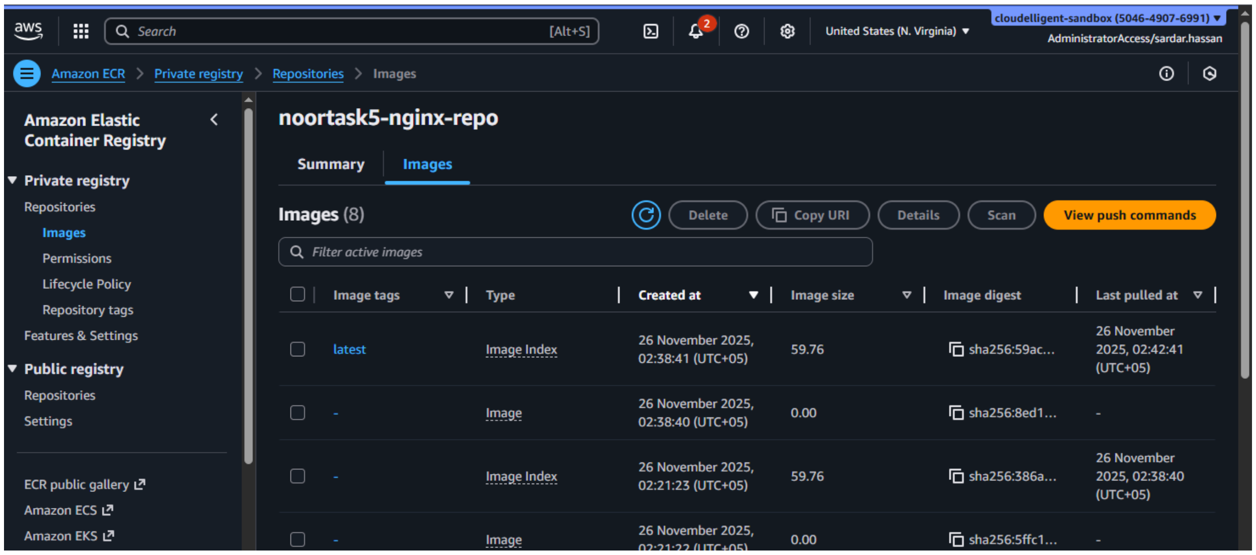
This approach ensures that the first time a task runs, my custom HTML gets copied onto EFS, so future tasks share the same persistent web content.

5.4 Pushing to ECR

Steps I followed:

1. Created an **ECR repository** (e.g., noor-task5-nginx).
2. Built the image:
3. docker build -t noor-task5-nginx .
4. Tagged it for ECR:
5. docker tag noor-task5-nginx:latest <account-id>.dkr.ecr.<region>.amazonaws.com/noor-task5-nginx:latest
6. Logged in to ECR using the AWS CLI.
7. Pushed the image:

8. docker push <account-id>.dkr.ecr.<region>.amazonaws.com/noor-task5-nginx:latest



noortask5-nginx-repo

Images (8)

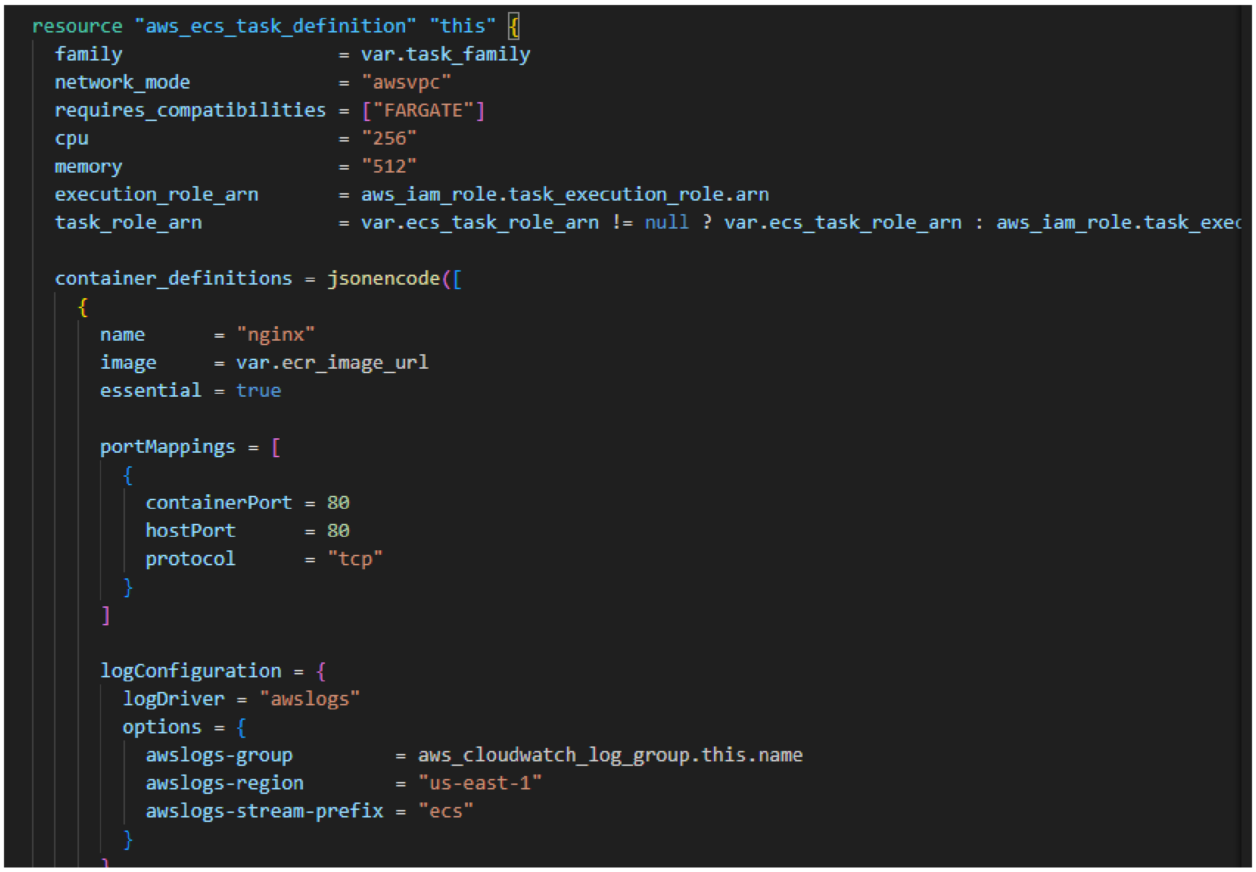
Image tags	Type	Created at	Image size	Image digest	Last pulled at
latest	Image Index	26 November 2025, 02:38:41 (UTC+05)	59.76	sha256:59ac...	26 November 2025, 02:42:41 (UTC+05)
-	Image	26 November 2025, 02:38:40 (UTC+05)	0.00	sha256:8ed1...	-
-	Image Index	26 November 2025, 02:21:23 (UTC+05)	59.76	sha256:386a...	26 November 2025, 02:38:40 (UTC+05)
-	Image	26 November 2025,	0.00	sha256:5ffc1...	-

6. Define ECS task definition to use the custom Docker image and mount EFS volumes

6.1 Goal

Tell ECS exactly how to run my container: image, CPU, memory, ports, and how to attach the EFS volume.

6.2 Design decisions

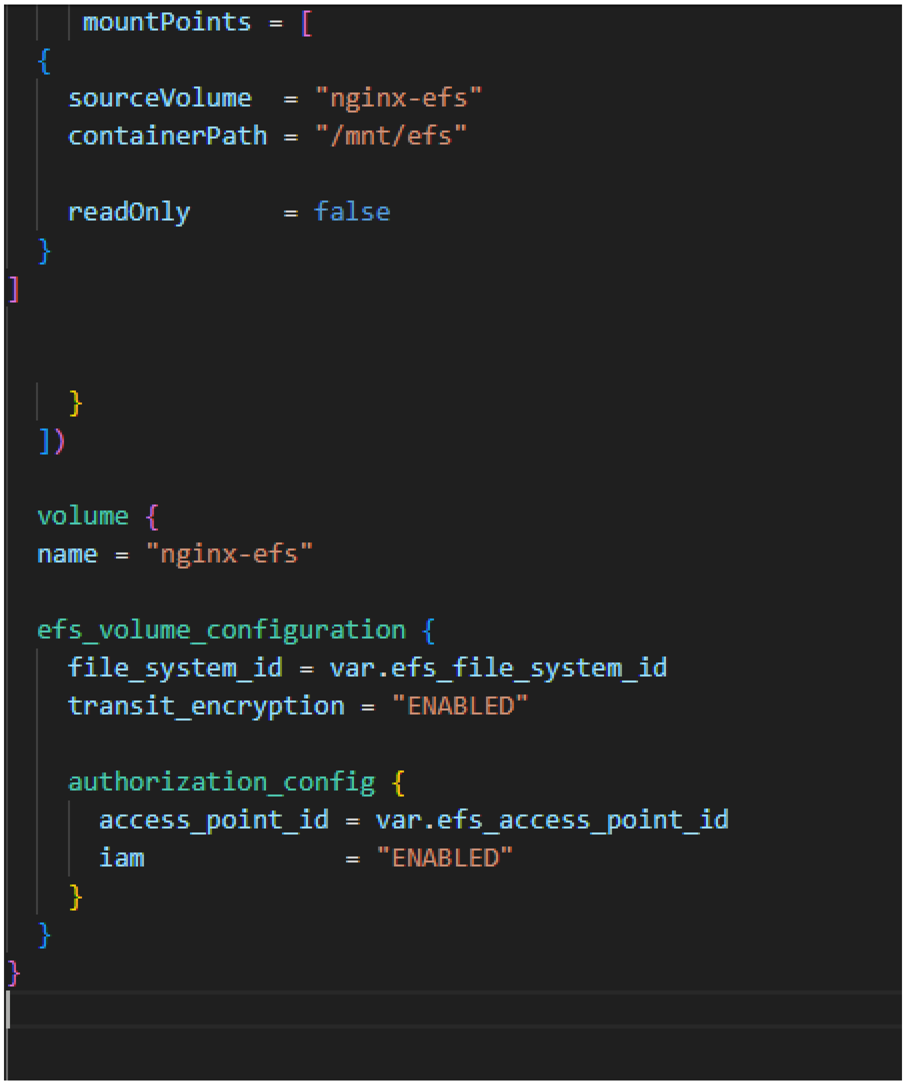


```
resource "aws_ecs_task_definition" "this" {
  family                   = var.task_family
  network_mode             = "awsvpc"
  requires_compatibilities = ["FARGATE"]
  cpu                      = "256"
  memory                   = "512"
  execution_role_arn       = aws_iam_role.task_execution_role.arn
  task_role_arn            = var.ecs_task_role_arn != null ? var.ecs_task_role_arn : aws_iam_role.task_exec

  container_definitions = jsonencode([
    {
      name      = "nginx"
      image     = var.ecr_image_url
      essential = true

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

      logConfiguration = {
        logDriver = "awslogs"
        options = {
          awslogs-group         = aws_cloudwatch_log_group.this.name
          awslogs-region        = "us-east-1"
          awslogs-stream-prefix = "ecs"
        }
```

- **Launch type:** Fargate.
- **Network mode:** awsvpc (required by Fargate).
- **Container port:** 80, because Nginx serves HTTP on port 80.
- **EFS volume name:** A logical name like "nginx-efs", which is referenced inside the container definition.
- **Mount path inside container:** I used /mnt/efs as the EFS mount path. start.sh copies HTML into this path, and Nginx can be configured to serve from there if needed.

6.3 EFS volume configuration

In the task definition's volume block, EFS is defined with:



```
      mountPoints = [
  {
    sourceVolume  = "nginx-efs"
    containerPath = "/mnt/efs"

    readOnly      = false
  }
]


    }
  ])

  volume {
  name = "nginx-efs"

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

    authorization_config {
      access_point_id = var.efs_access_point_id
      iam             = "ENABLED"
    }
  }
}
```

- file_system_id – ID of the EFS filesystem.
- transit_encryption = "ENABLED" – to encrypt traffic between tasks and EFS.
- authorization_config:
    - access_point_id – EFS access point, which controls directory path and POSIX permissions.
    - iam = "ENABLED" – EFS authorizes access based on IAM role.

Inside container_definitions:

- I mounted that volume to /mnt/efs.
- Set portMappings to map container port 80.
- Defined logging to CloudWatch so I can see Nginx logs.

7. Create an Application Load Balancer (ALB) with proper listeners and target groups

7.1 Goal

Expose the application to the internet and distribute traffic across running Fargate tasks.

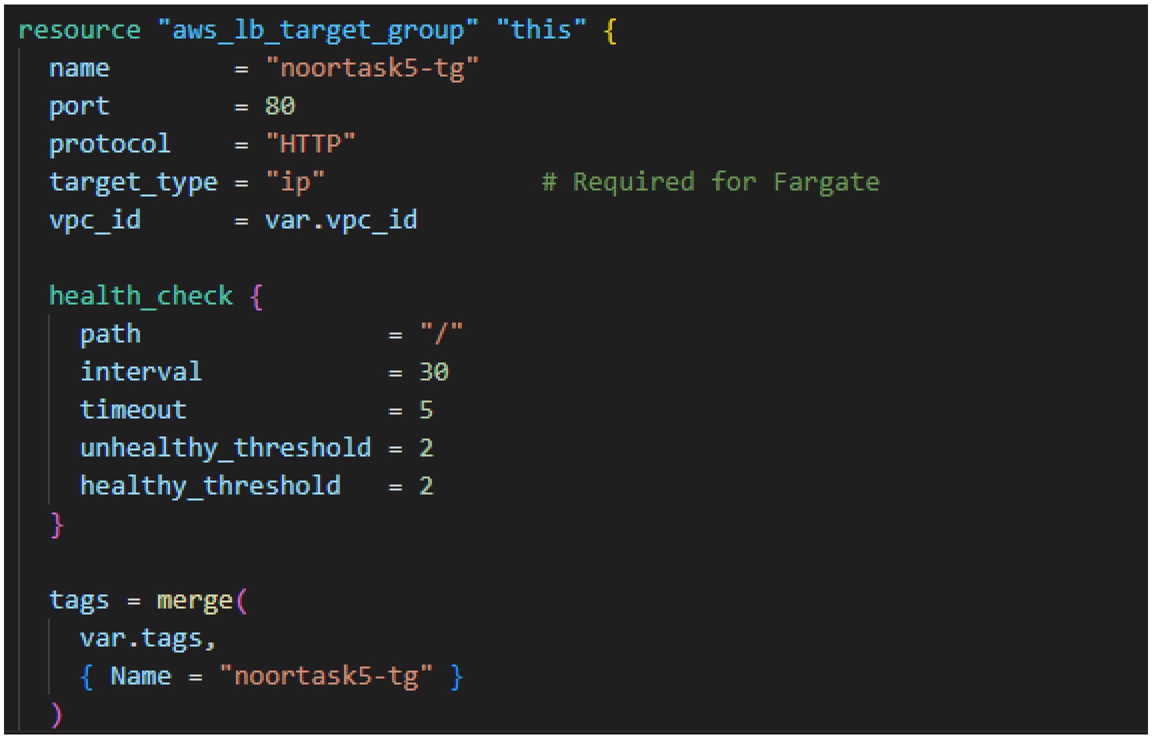
7.2 Design decisions

- **Type:** Application Load Balancer (layer 7) because I'm serving HTTP and might add path-based rules in future.
- **Scheme:** Internet-facing.
- **Subnets:** Both public subnets across two AZs for high availability.
- **Security Group:** The ALB SG created earlier, which allows inbound 80 from the internet.

7.3 Target group

- **Target type:** ip (required when using Fargate).
- **Port:** 80.
- **Health check path:** / (simple check to see if Nginx responds).

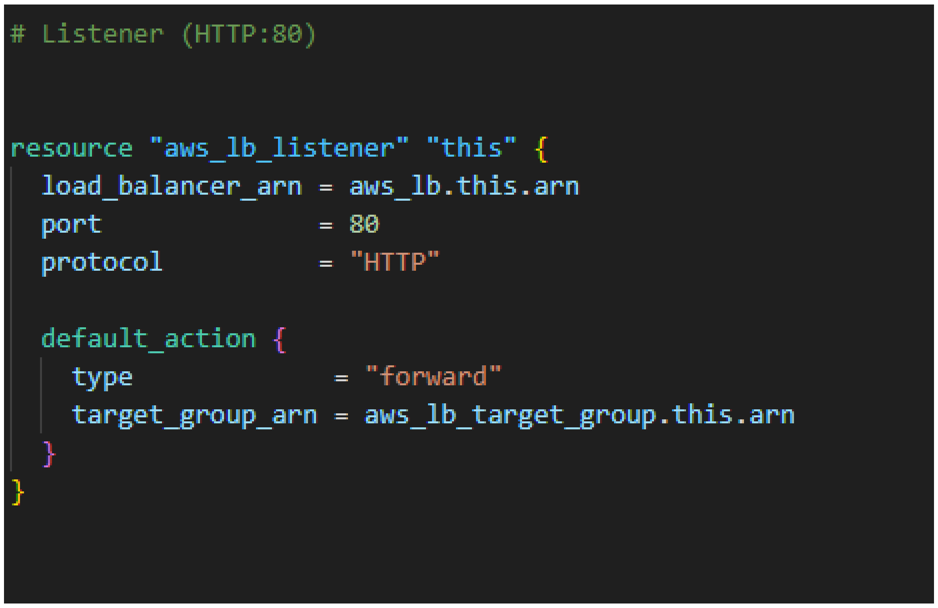
The ECS service later registers Fargate tasks with this target group automatically.



```
resource "aws_lb_target_group" "this" {
  name        = "noortask5-tg"
  port        = 80
  protocol    = "HTTP"
  target_type = "ip"              # Required for Fargate
  vpc_id      = var.vpc_id

  health_check {
    path                = "/"
    interval            = 30
    timeout             = 5
    unhealthy_threshold = 2
    healthy_threshold   = 2
  }

  tags = merge(
    var.tags,
    { Name = "noortask5-tg" }
  )
```

7.4 Listener

- Created an HTTP listener on port 80 that forwards to the target group.
- For now I did not configure HTTPS; that could be an improvement using ACM in future.



```
# Listener (HTTP:80)


resource "aws_lb_listener" "this" {
  load_balancer_arn = aws_lb.this.arn
  port              = 80
  protocol          = "HTTP"

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

8. Configure ECS service to use Fargate with ALB integration for load balancing

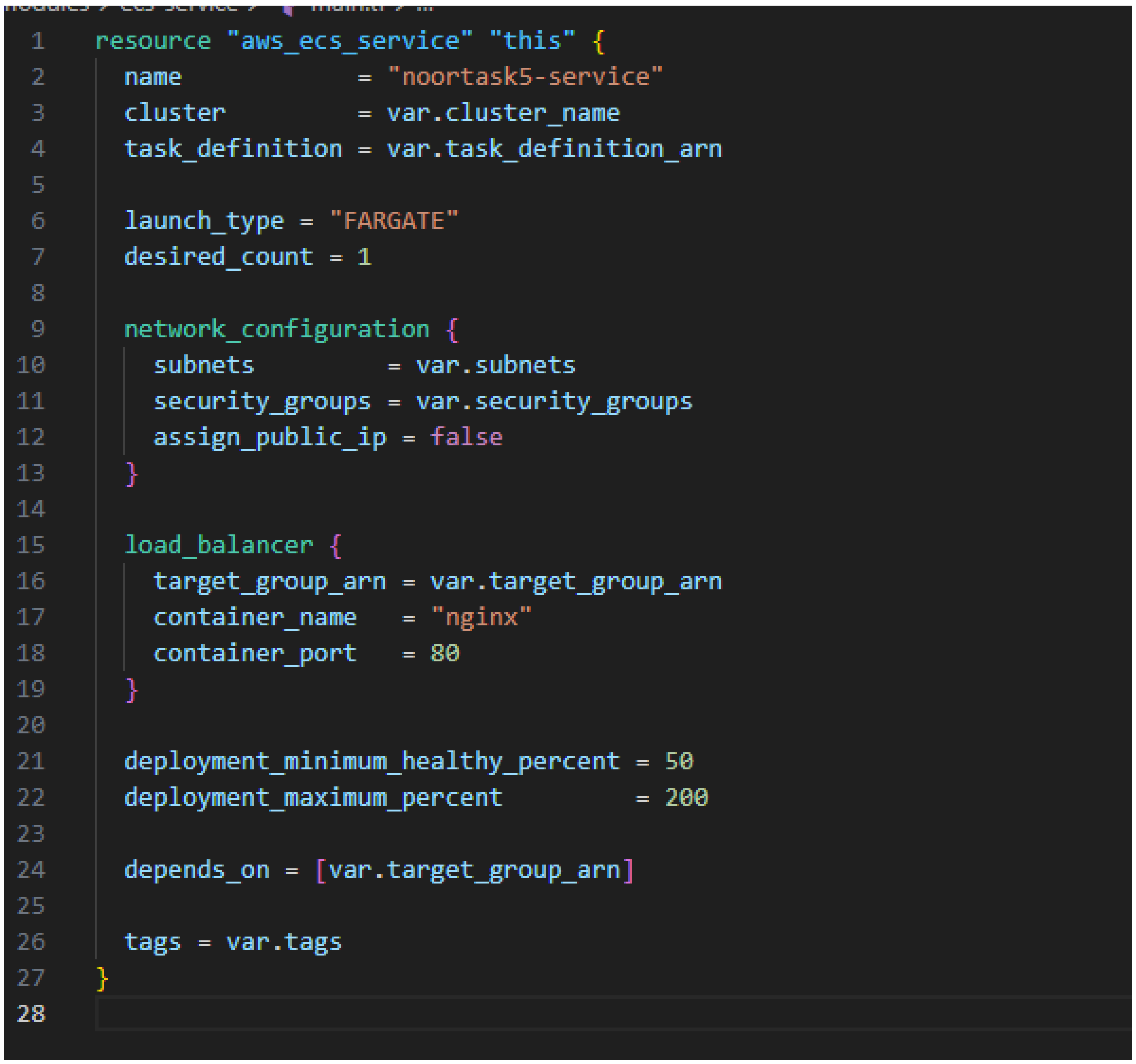
8.1 Goal

Run and manage a scalable set of tasks based on my task definition and attach them to the ALB target group.

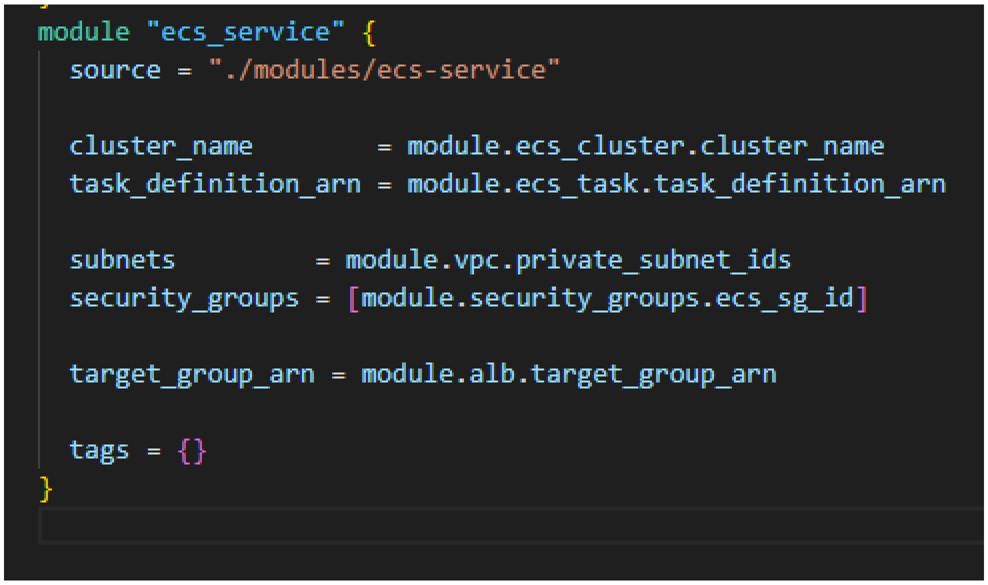
8.2 Design decisions

- **Launch type:** Fargate.
- **Desired count:** For the demo I started with 1, but the service can be scaled to more tasks anytime.
- **Subnets:** Same public subnets used by the ALB, so tasks receive routable IPs and can reach the internet.
- **Security Group:** ECS Tasks SG which only allows HTTP from ALB.

8.3 Terraform implementation



```hcl
resource "aws_ecs_service" "this" {
  name            = "noortask5-service"
  cluster         = var.cluster_name
  task_definition = var.task_definition_arn

  launch_type = "FARGATE"
  desired_count = 1

  network_configuration {
    subnets         = var.subnets
    security_groups = var.security_groups
    assign_public_ip = false
  }

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

  deployment_minimum_healthy_percent = 50
  deployment_maximum_percent         = 200

  depends_on = [var.target_group_arn]

  tags = var.tags
}

```



```
module "ecs_service" {
  source = "./modules/ecs-service"

  cluster_name        = module.ecs_cluster.cluster_name
  task_definition_arn = module.ecs_task.task_definition_arn

  subnets         = module.vpc.private_subnet_ids
  security_groups = [module.security_groups.ecs_sg_id]

  target_group_arn = module.alb.target_group_arn

  tags = {}
}
```

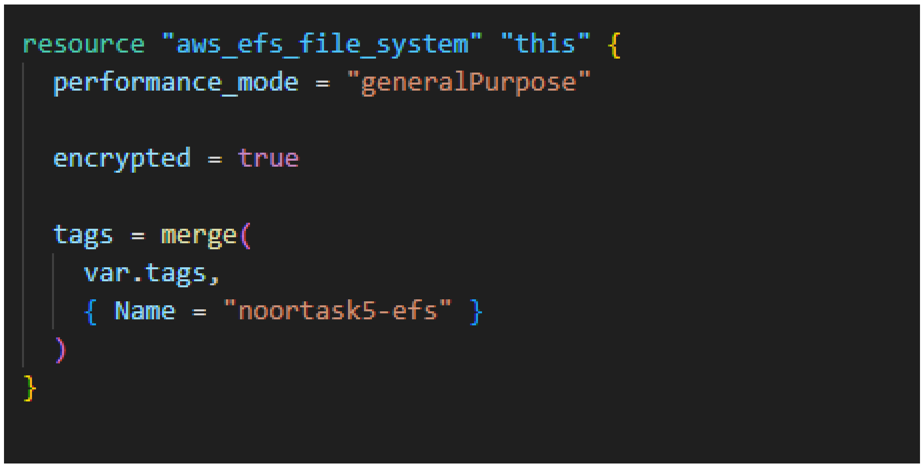
9. Set up and mount EFS for persistent storage in ECS tasks

9.1 Goal

Ensure that static web content survives task restarts and that multiple tasks share the same files.

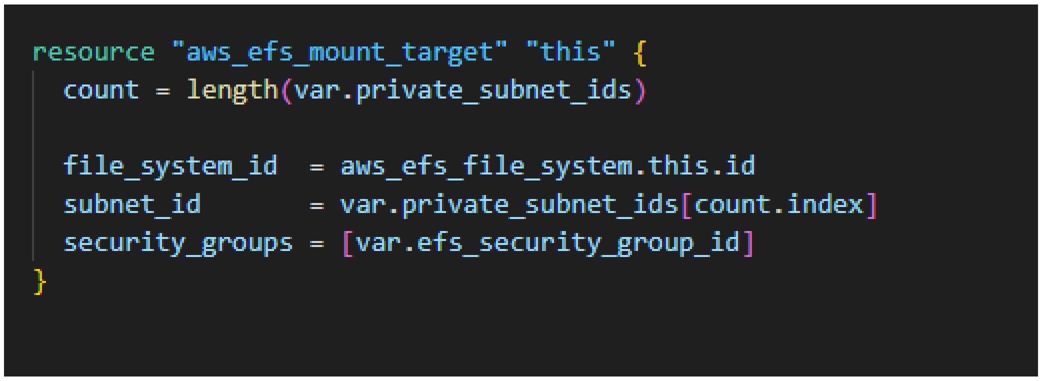
9.2 Design decisions

- **One EFS filesystem** for the web application data.



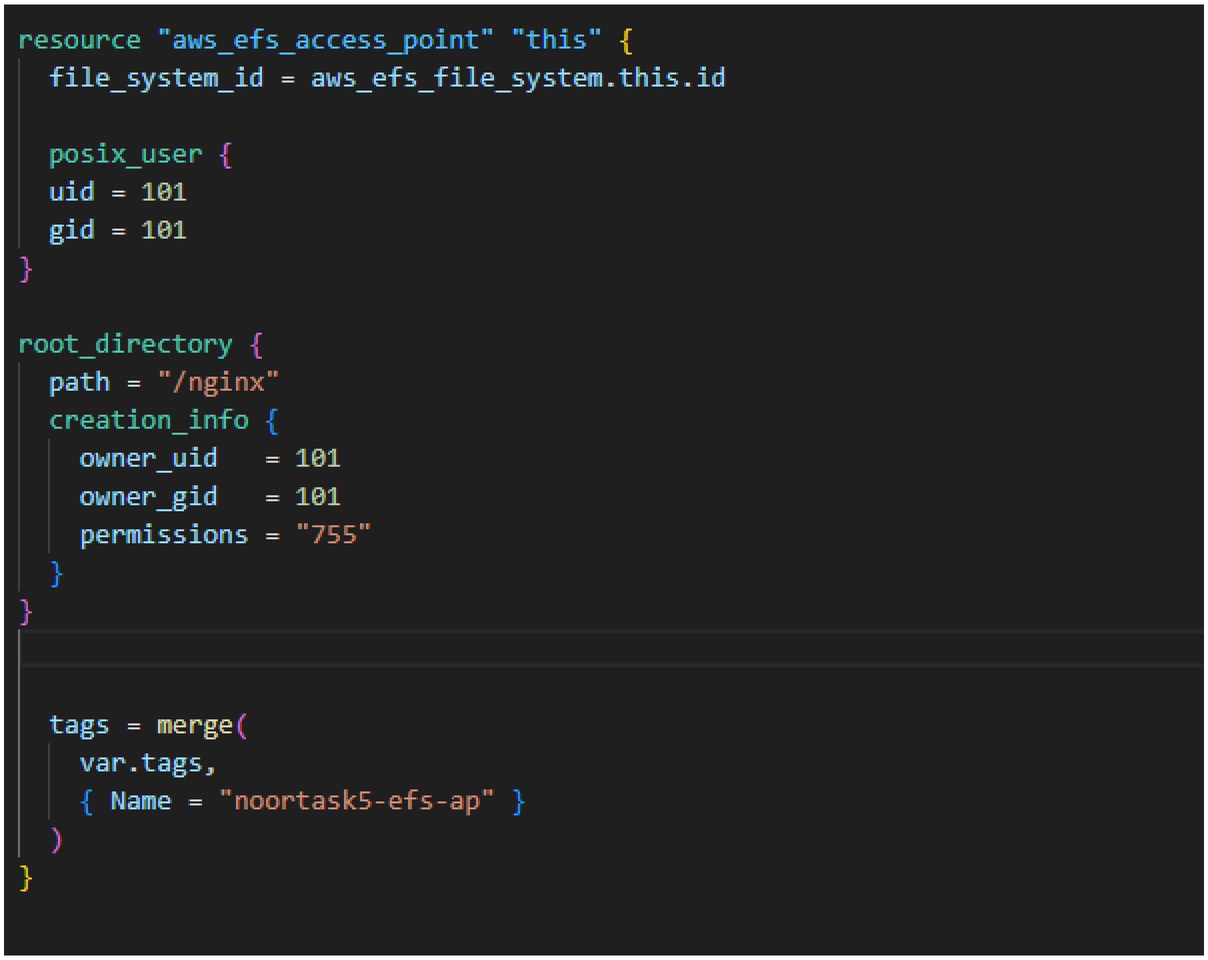
```
resource "aws_efs_file_system" "this" {
  performance_mode = "generalPurpose"

  encrypted = true

  tags = merge(
    var.tags,
    { Name = "noortask5-efs" }
  )
}
```

- **Mount targets** created in each AZ where my tasks can run, attached to the same EFS SG.



```
resource "aws_efs_mount_target" "this" {
  count = length(var.private_subnet_ids)

  file_system_id  = aws_efs_file_system.this.id
  subnet_id       = var.private_subnet_ids[count.index]
  security_groups = [var.efs_security_group_id]
}
```

- **Access point** to define a specific directory path and POSIX permissions used by the tasks.



```
resource "aws_efs_access_point" "this" {
  file_system_id = aws_efs_file_system.this.id

  posix_user {
  uid = 101
  gid = 101
}

root_directory {
  path = "/nginx"
  creation_info {
    owner_uid   = 101
    owner_gid   = 101
    permissions = "755"
  }
}


  tags = merge(
    var.tags,
    { Name = "noortask5-efs-ap" }
  )
}
```

9.3 How it works together with the container

- In the task definition, the EFS volume is mounted to /mnt/efs.
- When the container starts, start.sh:

    - Waits 2 seconds to give time for the EFS mount.
    - Copies everything from /usr/share/nginx/html/ (which contains my index.html) into /mnt/efs/.
    - Starts Nginx in the foreground so ECS can monitor the process.

Once copied, the HTML files live on EFS. If I scale to 2 or 3 tasks, each task mounts the same EFS and serves the exact same content.

9.4 Security

- EFS is locked down by security groups so only ECS tasks can connect.
- Transit encryption is enabled.
- IAM authorization with an access point further restricts who can mount and how.

10. Attach IAM roles and policies for ECS tasks and EFS access

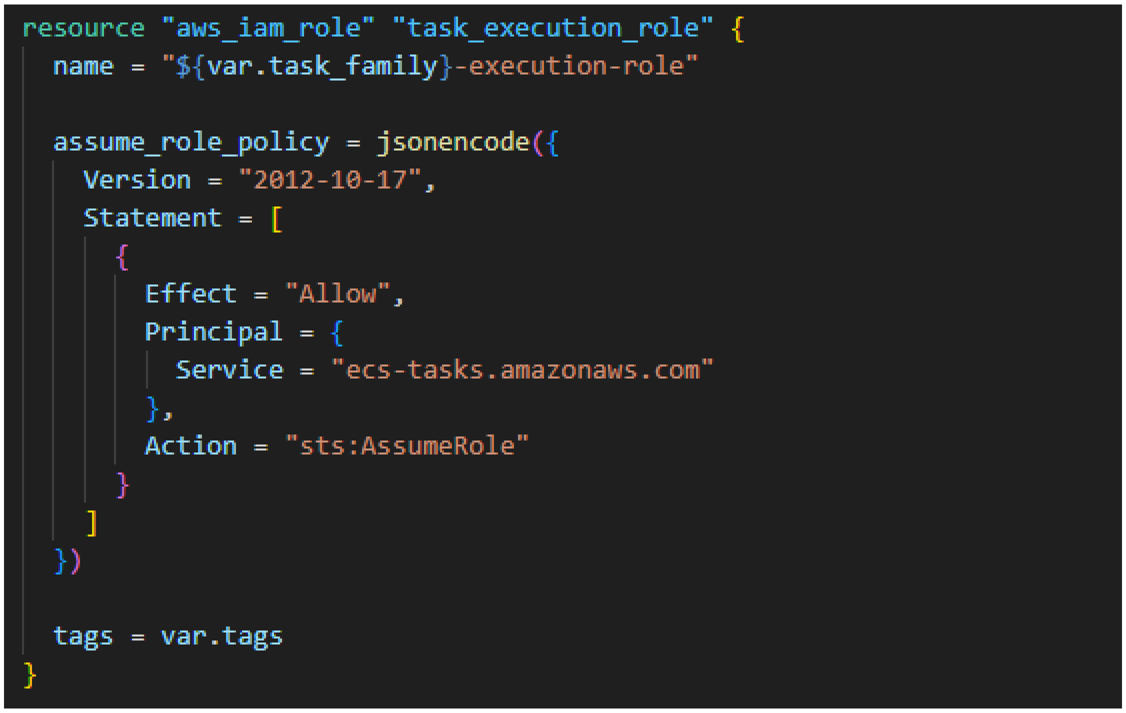
10.1 Goal

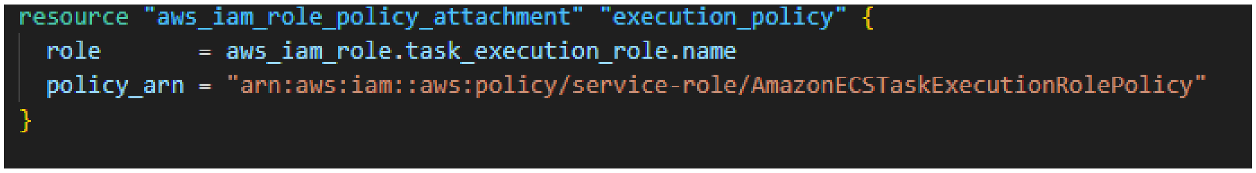
Give ECS just enough permissions to:

- Pull images from ECR.
- Write logs to CloudWatch.
- Access EFS using the access point.

10.2 Roles

1. **Task Execution Role (ecsTaskExecutionRole)**
    - Has the AWS managed policy AmazonECSTaskExecutionRolePolicy.
    - Allows ECS agents to pull images from ECR, read SSM parameters if needed, and push logs to CloudWatch.


```
resource "aws_iam_role" "task_execution_role" {
  name = "${var.task_family}-execution-role"

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

  tags = var.tags
}
```


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

---

11. Deploy infrastructure using Terraform and verify scalability and availability

11.1 Terraform backend and workflow

I used a **remote S3 backend** so that Terraform state is not stored locally. That allows:

- Safe collaboration.
- Easier recovery if my local machine is lost.

Typical deployment flow:

1. terraform init , to initialize backend and download providers/modules.
2. terraform validate , to catch syntax errors.
3. terraform plan ,to see what AWS resources will be created or changed.

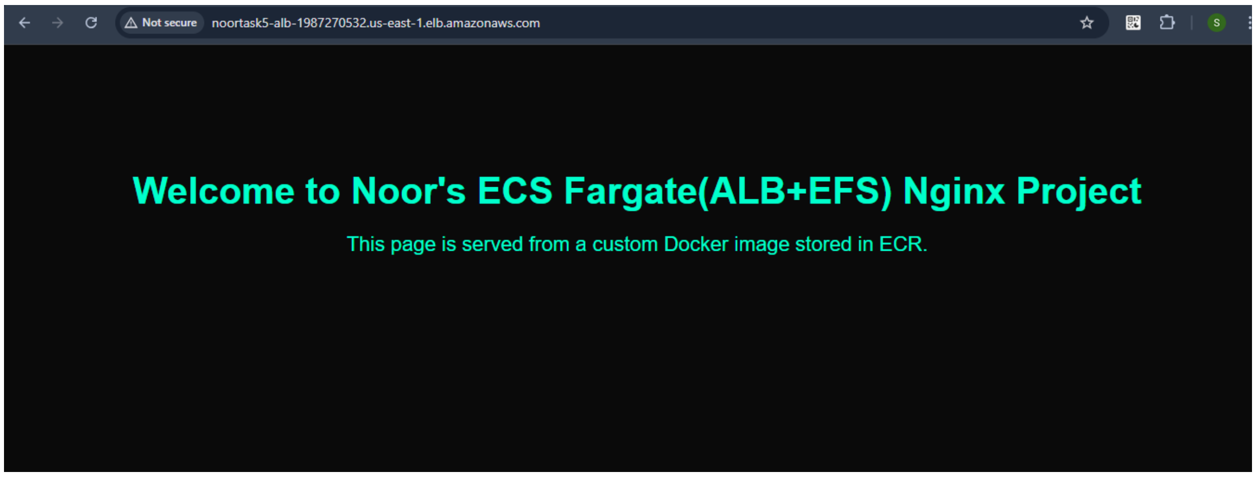
4. terraform apply , to actually create the infrastructure.

After apply completes, Terraform outputs:

- ALB DNS name.
- Possibly ECS cluster/service names.
- EFS filesystem ID, etc.

11.2 Verifying the deployment

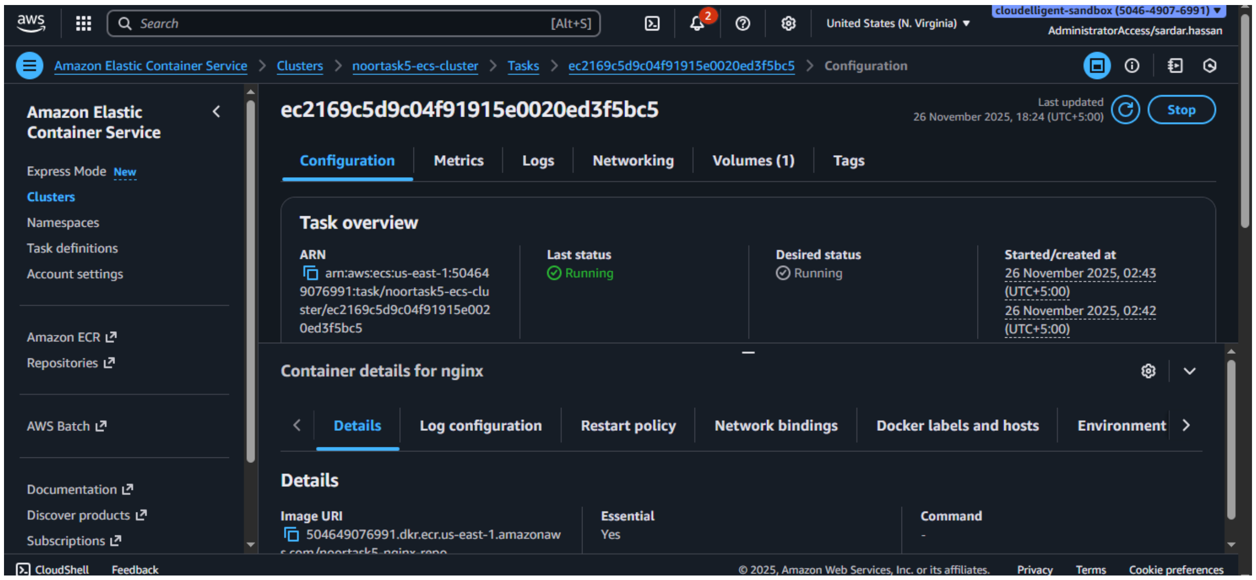
1. **Open ALB DNS in browser**
    - Copied the DNS name from the ALB details page.
    - Pasted it into the browser.
    - I saw my custom HTML page with the message “Welcome to Noor's ECS Fargate(ALB+EFS) Nginx Project…”.



Welcome to Noor's ECS Fargate(ALB+EFS) Nginx Project

This page is served from a custom Docker image stored in ECR.

2. **Check ECS tasks health**
    - In ECS console, all tasks were in RUNNING status.
    - In Target Group, health check status was healthy for all IP targets.



ec2169c5d9c04f91915e0020ed3f5bc5

Last updated 26 November 2025, 18:24 (UTC+5:00)

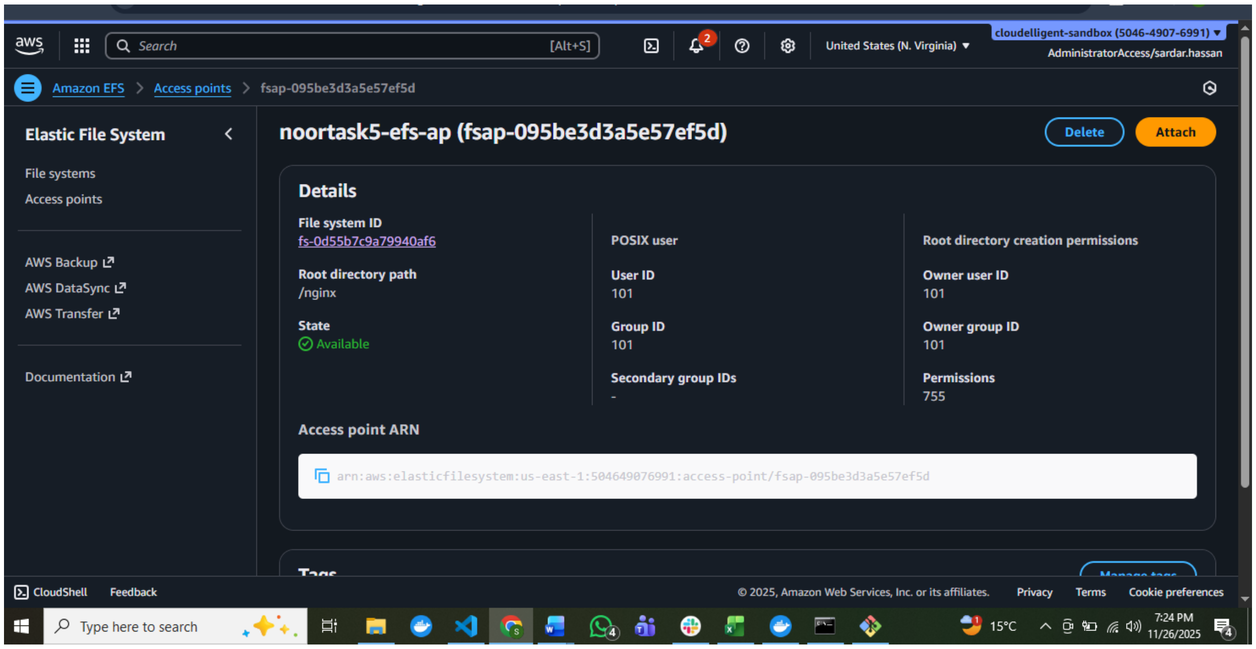
Task overview

ARN	Last status	Desired status	Started/created at
arn:aws:ecs:us-east-1:504649076991:task/noortask5-ecs-cluster/ec2169c5d9c04f91915e0020ed3f5bc5	Running	Running	26 November 2025, 02:43 (UTC+5:00) 26 November 2025, 02:42 (UTC+5:00)

Container details for nginx

Details

Image URI	Essential	Command
504649076991.dkr.ecr.us-east-1.amazonaws.com/noortask5-nginx-repo	Yes	-


Amazon EFS > Access points > fsap-095be3d3a5e57ef5d

noortask5-efs-ap (fsap-095be3d3a5e57ef5d)

Details

File system ID
fs-0d55b7c9a79940af6

Root directory path
/nginx

State
Available

POSIX user

User ID
101

Group ID
101

Secondary group IDs
-

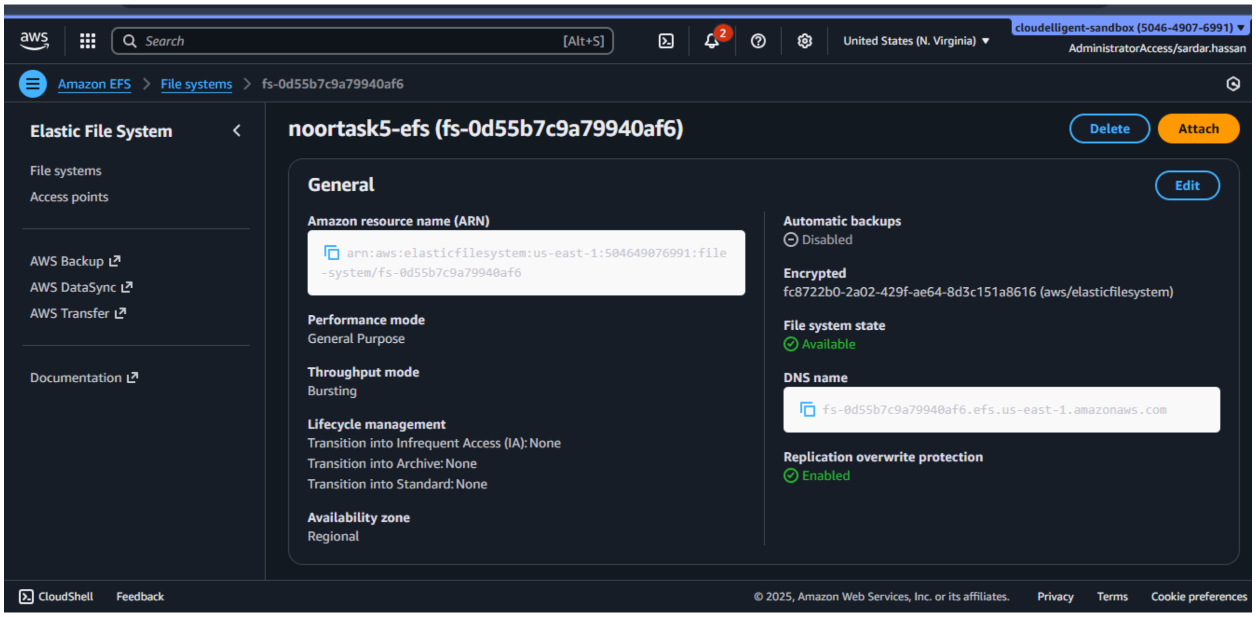
Root directory creation permissions

Owner user ID
101

Owner group ID
101

Permissions
755

Access point ARN
```
arn:aws:elasticfilesystem:us-east-1:504649076991:access-point/fsap-095be3d3a5e57ef5d
```


Amazon EFS > File systems > fs-0d55b7c9a79940af6

noortask5-efs (fs-0d55b7c9a79940af6)

General

Amazon resource name (ARN)
```
arn:aws:elasticfilesystem:us-east-1:504649076991:file-system/fs-0d55b7c9a79940af6
```

Performance mode
General Purpose

Throughput mode
Bursting

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

Availability zone
Regional

Automatic backups
Disabled

Encrypted
fc8722b0-2a02-429f-ae64-8d3c151a8616 (aws/elasticfilesystem)

File system state
Available

DNS name
```
fs-0d55b7c9a79940af6.efs.us-east-1.amazonaws.com
```

Replication overwrite protection
Enabled

12. Troubleshooting and issues faced

This section is for actual problems that can happen in this architecture and how I would fix them. These are the typical ones you face when using ECS Fargate + ALB + EFS.

12.1 Issue: ALB shows “503 Service Unavailable”

Symptoms

- ALB DNS loads but shows 503.

- Target group shows all targets as unhealthy.

Root cause

- Usually happens when health check path is wrong or ECS security group doesn't allow traffic from ALB.

What I checked and fixed

1. Verified health check path is / and port is 80 on the target group.
2. Checked ECS tasks security group:
    - Confirmed an inbound rule 80 from the **ALB security group**, not from some random SG.
3. Checked container logs in CloudWatch to ensure Nginx started without errors.

After fixing the security group , the targets became healthy and 503 disappeared.

12.2 Issue: EFS mount fails or Nginx can't serve from EFS

Symptoms

- Container restarts repeatedly.
- Logs show NFS or permission errors.
- EFS console shows little or no traffic.

Root cause

- NFS port 2049 blocked between ECS tasks and EFS.
- Wrong EFS access point or IAM permissions.
- EFS mount not ready when Nginx starts.

What I checked and fixed

1. Confirmed EFS Security Group allows inbound 2049 from ECS tasks SG.
2. Confirmed tasks are in subnets that have network path to EFS mount targets (same VPC).
3. Ensured the access point ID in the task definition matches the actual EFS access point.

4. Used the start.sh script to **wait a bit (sleep 2) before starting Nginx**, so that the EFS mount has time to be ready.

After these fixes, EFS mounted correctly and Nginx served content from the shared filesystem.

---

13. Conclusion

This project takes a full path from networking to container deployment:

- I designed a VPC and subnets for high availability.
- I locked down network paths with separate security groups for ALB, ECS, and EFS.
- I built a custom Nginx image with my own HTML and a small startup script to integrate with EFS.
- I deployed everything on ECS Fargate behind an ALB, using Terraform with a remote backend.
- I verified that the application is accessible, scalable, and that data persists across task restarts thanks to EFS.