TASK 4

ECS Fargate and RDS MySQL WordPress Deployment with Terraform

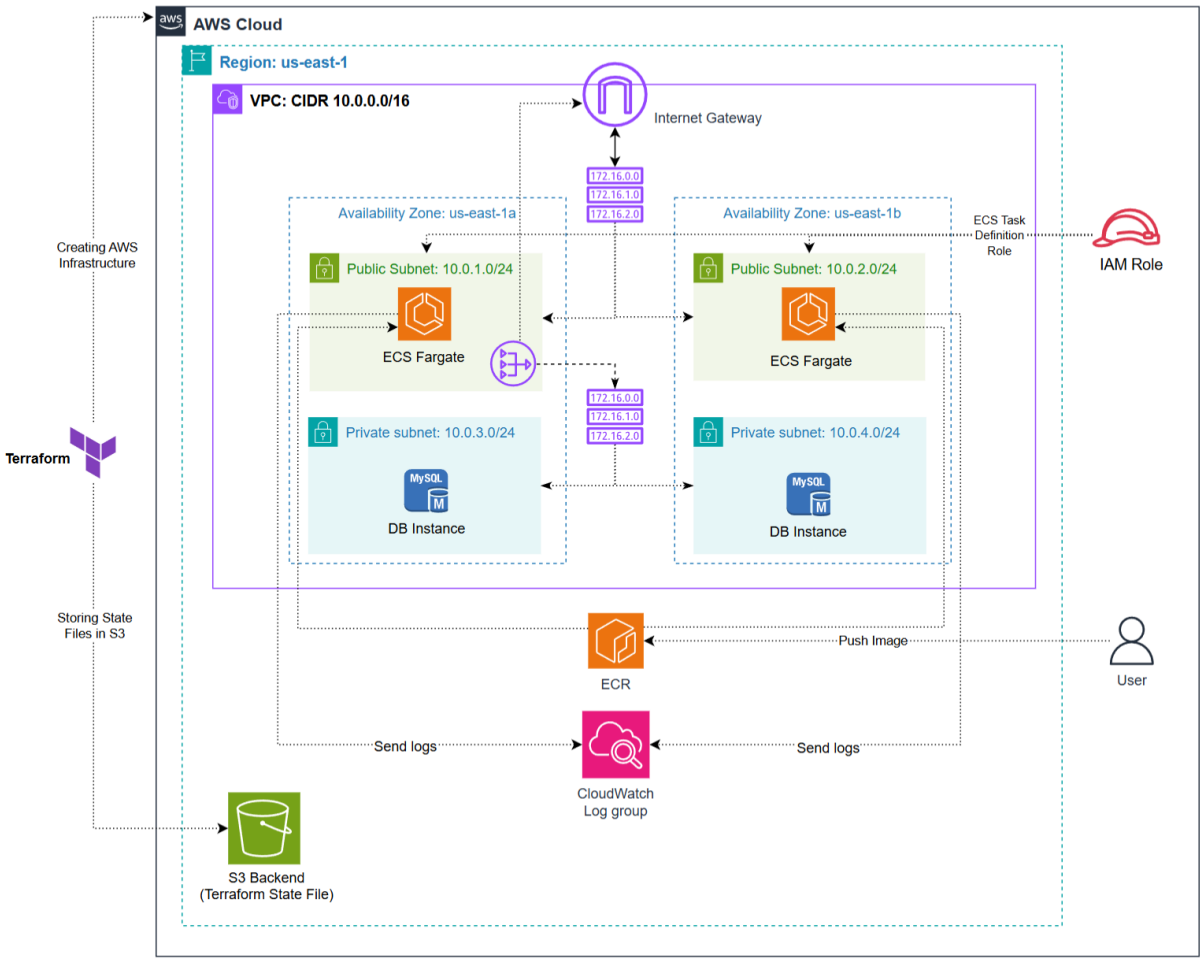
Umar Satti

Table of Contents

Task Description

Set up a WordPress application running on Amazon ECS Fargate using Terraform. Create a VPC with public and private subnets, multiple security groups, task definition, ECR repository, ECS cluster, ECS service, and RDS MySQL with minimal configurations.

Architecture Diagram



Task 1.1: 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 1.2 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 VPC ID, ECR Private Registry URI for Wordpress after deployment.
6. Create a modules directory that contains 6 total modules including VPC, Security groups, Task definition, ECR, ECS, and RDS. Each module contains its own main.tf, variables.tf, and outputs.tf files.

The terraform project directory structure looks like this:

Task 1.2: Create S3 Bucket for Terraform Remote Backend

Steps:

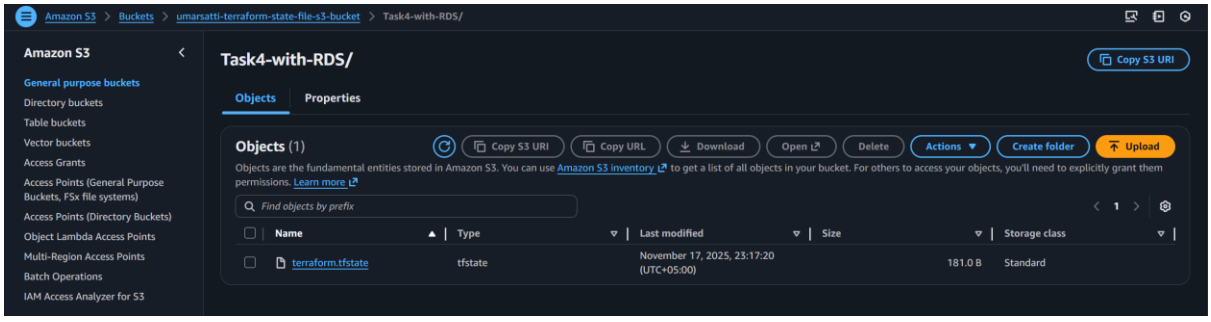
1. Log in to the AWS Management Console. Navigate to S3 using the search bar at the top of the console page.
2. Click on **Create Bucket** button.
3. Choose **General Purpose**, add a globally unique bucket name, and make sure the AWS Region is the same as Terraform.
4. Click **Create Bucket**.
5. Update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

```
backend "s3" {
  bucket         = "umarsatti-terraform-state-file-s3-bucket"
  key            = "task4/terraform.tfstate"
  region         = "us-east-1"
  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-state-file-s3-bucket** 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 > Task4-with-RDS > terraform.tfstate



This confirms that:

- Terraform successfully initialized the backend and wrote the state file to the S3 bucket.

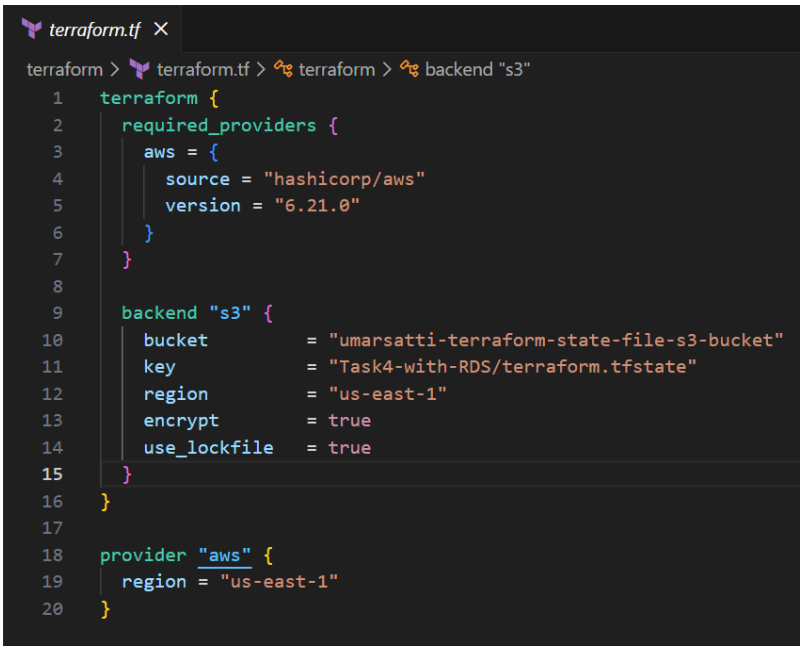
- The **terraform.tfstate** file contains metadata about all deployed AWS resources (VPC, Subnet, EC2, etc.).
- Every terraform plan, apply, or destroy operation reads and updates this file automatically.
- The locking mechanism (enabled by **use_lockfile = true**) ensures that no two processes modify the state simultaneously, preventing state corruption.

Note: The S3 bucket **umarsatti-terraform-state-file-s3-bucket** was already created in the prior projects. However, a new folder was created just for this project named **Task4-with-RDS repository** in which the **terraform.tfstate** file is stored.

Task 1.3: Root Directory Files

This section explains each Terraform configuration file in the root directory and how they work together to deploy the VPC, Security Groups, ECR Repository, RDS Database, ECS Task Definition, and ECS Fargate Service.

terraform.tf:



This file configures the AWS provider and sets up the remote backend used to store Terraform state securely in an S3 bucket.

- The terraform block specifies the AWS provider version (6.21.0) sourced from HashiCorp.
- The backend "s3" block stores the Terraform state file in the bucket **umarsatti-terraform-state-file-s3-bucket** under the path **Task4-with-RDS/terraform.tfstate**.
- Encryption is enabled, and use_lockfile = true ensures that only one Terraform process can update the state at a time.
- The provider "aws" block defines **us-east-1** as the region where all AWS resources will be created.

Using an S3 backend allows centralized, versioned, and protected state management—ideal for team use or working across multiple machines.

main.tf:

main.tf

terraform > main.tf > module "sg"

```hcl
module "vpc" {
  source   = "./modules/vpc"
  vpc_name = var.vpc_name
  vpc_cidr = var.vpc_cidr
}

module "sg" {
  source = "./modules/sg"
  vpc_id = module.vpc.vpc_id
}

module "ecr" {
  source             = "./modules/ecr"
  ecr_wordpress_name = var.ecr_wordpress_name
}

module "task_definition" {
  source            = "./modules/task_definition"
  ecr_uri_wordpress = module.ecr.ecr_uri_wordpress
  rds_endpoint      = module.rds.db_endpoint
  db_username       = var.db_username
  db_password       = var.db_password
  db_name           = var.db_name
}

module "ecs" {
  source             = "./modules/ecs"
  task_definition_arn = module.task_definition.task_definition_arn
  public_subnets      = module.vpc.public_subnets
  security_group_id   = module.sg.security_group_id
}

module "rds" {
  source                    = "./modules/rds"
  private_subnets           = module.vpc.private_subnets
  security_group_private_id = module.sg.security_group_private_id
  db_username               = var.db_username
  db_password               = var.db_password
  db_name                   = var.db_name
}

```

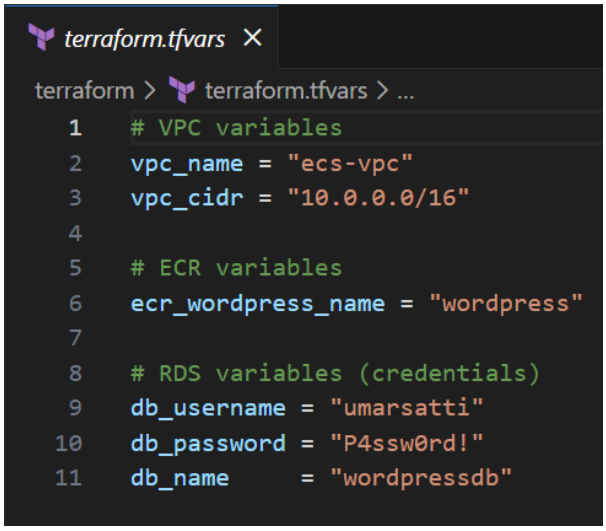
This file defines how Terraform assembles the entire WordPress + RDS architecture by wiring together all modules.

- **VPC module (./modules/vpc)**
  Creates all networking components including the VPC, public and private subnets, internet gateway, and route tables. It uses **vpc_name** and **vpc_cidr** from root variables.
- **Security Group module (./modules/sg)**
  Builds both the public security group (for ECS) and private security group (for RDS). It receives the VPC ID from the VPC module to ensure both security groups are associated with the correct network.
- **ECR module (./modules/ecr)**
  Creates the private ECR repository used to store the WordPress Docker image. The repository name is taken from the **ecr_wordpress_name** variable.

- **Task Definition module (./modules/task_definition)**
  Defines the WordPress ECS task that connects to the RDS database. This module uses:
    - The ECR WordPress image URI from the ECR module
    - RDS endpoint from the RDS module
    - Database username, password, and database name from root-level variables
- **ECS module (./modules/ecs)**
  Deploys the ECS Cluster and ECS Fargate Service using:
    - The Task Definition ARN from the Task Definition module
    - Public subnets from the VPC module
    - Security group ID from the SG module
- **RDS module (./modules/rds)**
  Provisions an RDS MySQL database inside the private subnets provided by the VPC module. It uses the private security group to restrict database access only to ECS tasks. Database credentials are passed through variables.

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

terraform.tfvars



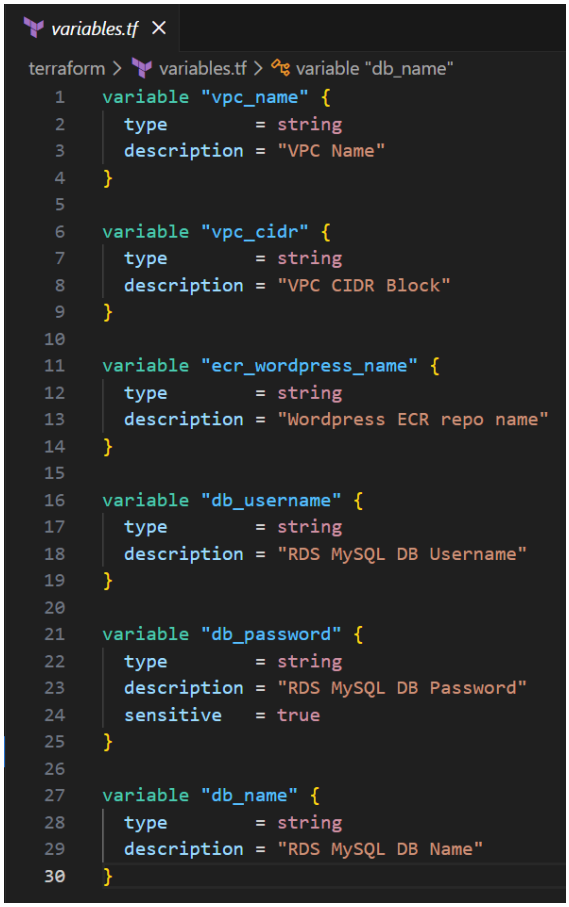
This file contains actual input values that Terraform uses during deployment.

- VPC settings (**vpc_name, vpc_cidr**) define the name and network range of the VPC.
- ECR repository name (**ecr_wordpress_name**) specifies the name of the WordPress image repository.

- RDS database credentials (**db_username, db_password, db_name**) configure the MySQL database that WordPress uses.

Storing values here keeps the configuration clean while allowing easy modification without touching the actual infrastructure code.

variables.tf:

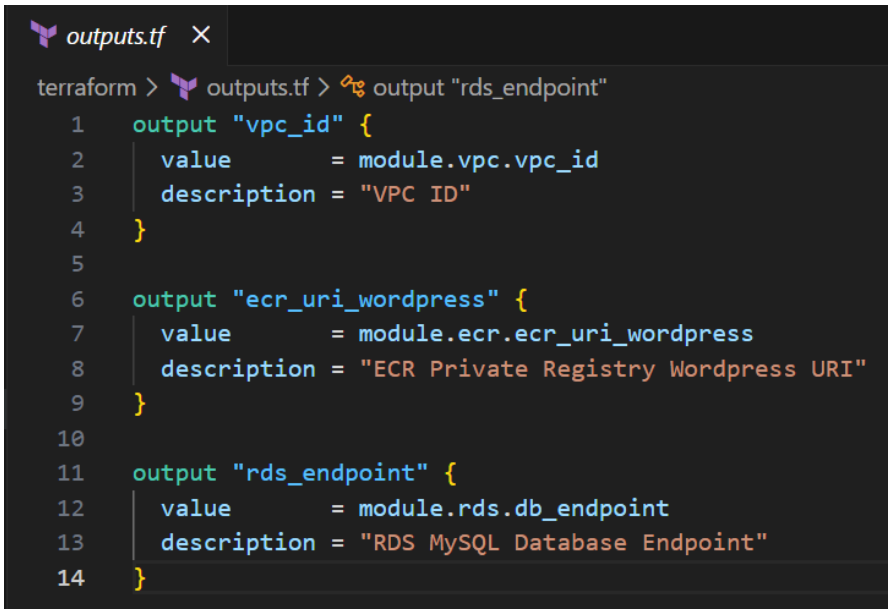


This file defines all the input variables required by the root module.

- VPC inputs (**vpc_name, vpc_cidr**) are used to build the networking layer.
- The **ecr_wordpress_name** variable allows the ECR module to create the WordPress repository dynamically.
- RDS variables (**db_username, db_password, db_name**) store database credentials. The password is marked **sensitive** so Terraform does not display it in logs or outputs.

This structure keeps the configuration modular and makes it clear what values the user must provide.

outputs.tf:



This file exposes key outputs from the root module after applying Terraform.

- **vpc_id** returns the VPC ID created by the VPC module.
- **ecr_uri_wordpress** outputs the full private ECR URI for the WordPress image.
- **rds_endpoint** returns the RDS MySQL endpoint used by WordPress to connect to the database.

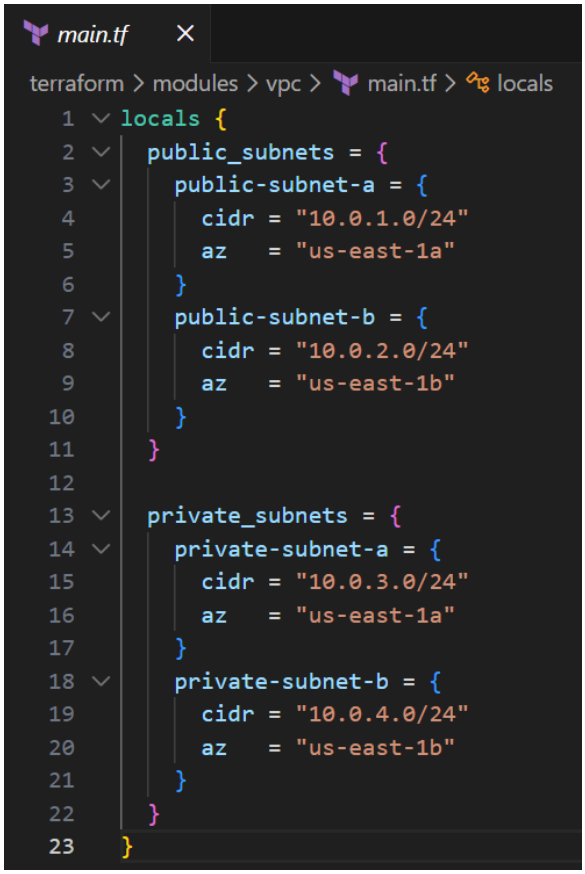
These outputs are helpful for verification and for referencing important resource details without manually navigating AWS.

Task 1.4: Configure VPC Module

This section explains each Terraform configuration file located in the VPC module (modules/vpc) and how they work together to build the networking layer used by the rest of the infrastructure.

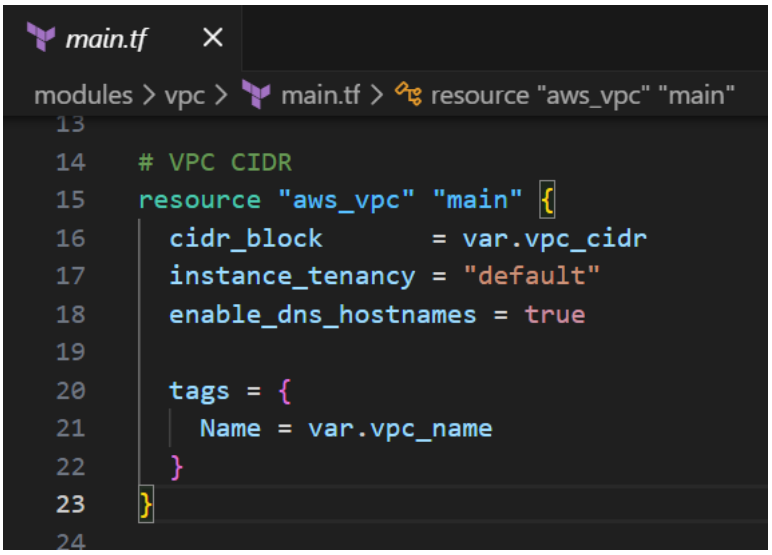
main.tf:

Local Variables



- The locals block defines two maps:
  - public_subnets
  - private_subnets
- Each map contains subnet names paired with their CIDR blocks and Availability Zones.
- This method allows the module to create multiple subnets dynamically using **for_each**, reducing repetitive code.
- Adding additional subnets in the future simply requires adding another entry to the map as no new resource blocks are needed.

Virtual Private Cloud



This resource creates the core VPC that all other resources will rely on.

- **cidr_block** is passed from root variables, enabling customizable network ranges.
- **enable_dns_hostnames = true** ensures that resources inside the VPC receive DNS names automatically.
- The VPC is tagged using **var.vpc_name**, allowing easy identification in the AWS console.

Public Subnets

```
# Subnets

# Public Subnets
resource "aws_subnet" "public_subnet" {
  for_each = local.public_subnets

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

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

This block creates public subnets using the map defined in **locals.public_subnets**.

- Created dynamically by iterating over **local.public_subnets**.
- Each subnet's CIDR block and AZ come from **each.value**.

- All subnets are automatically attached to the VPC through **vpc_id.**
- Each subnet is tagged using each.key, ensuring meaningful naming (e.g., *public-subnet-a*).

Private Subnets

```
# Private Subnets
resource "aws_subnet" "private_subnet" {
  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
  }
}
```

- Built the same way as public subnets but using **local.private_subnets**.
- These subnets will host private resources such as the RDS database.
- They do not have direct internet access and rely on the NAT Gateway for outbound connectivity.

Internet Gateway (IGW)

```
37  # Internet Gateway
38  resource "aws_internet_gateway" "igw" {
39    vpc_id = aws_vpc.main.id
40
41    tags = {
42      Name = "ecs-igw"
43    }
44  }
45
```

- Creates an Internet Gateway and attaches it to the VPC.
- Allows public subnets to send and receive internet traffic.
- Tagged as "**ecs-igw**" for consistent naming and visibility.

Elastic IP (EIP) for NAT

```
# EIP
resource "aws_eip" "nat_gateway_eip" {
  domain = "vpc"

  tags = {
    Name = "nat-gateway-eip"
  }

  depends_on = [aws_internet_gateway.igw]
}
```

- Allocates a static public IP address for the NAT Gateway.
- It depends on the IGW to ensure the VPC has internet connectivity before NAT is deployed.

NAT Gateway

```
# NAT Gateway
resource "aws_nat_gateway" "nat_gateway" {
  allocation_id = aws_eip.nat_gateway_eip.id
  subnet_id     = aws_subnet.public_subnet["public-subnet-a"].id

  tags = {
    Name = "nat-gateway"
  }

  depends_on = [aws_internet_gateway.igw]
}
```

- The NAT Gateway is deployed in **public-subnet-a**.
- Enables private subnets to reach the internet for updates (e.g., package downloads) without becoming publicly accessible.
- Depends on the Internet Gateway to ensure correct routing setup.

VPC Route Tables

Public Route Table:

```
# Route Tables

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

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

  tags = {
    Name = "Public-Route-Table"
  }
}
```

- Routes all outbound traffic (0.0.0.0/0) from public subnets to the IGW.
- Ensures ECS services placed in public subnets can reach the internet.

Private Route Table

```
# Private Route Table
resource "aws_route_table" "private_rt" {
  vpc_id = aws_vpc.main.id

  route {
    cidr_block = "0.0.0.0/0"
    nat_gateway_id = aws_nat_gateway.nat_gateway.id
  }

  tags = {
    Name = "Private-Route-Table"
  }
}
```

- Routes outbound traffic from private subnets through the NAT Gateway.
- Ensures RDS and other backend resources can reach the internet **without being exposed**.

Route Table Associations

Public Associations

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

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

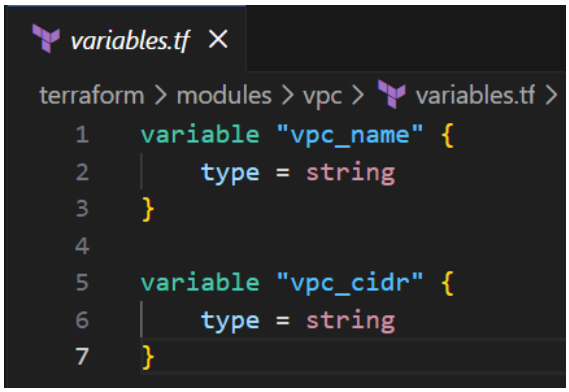
- Each public subnet is linked to the Public Route Table using an iterative for_each.
- Ensures public subnets use IGW for internet access.

Private Associations

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

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

- Each private subnet is associated with the Private Route Table.
- Ensures private subnets route outbound traffic through the NAT gateway only.

variables.tf:



- **vpc_name** defines the name tag applied to the VPC.
- **vpc_cidr** allows the VPC's CIDR block to be passed in dynamically from the root module.

These variables provide flexibility and reusability across different environments (dev, test, prod).

outputs.tf:

igw_name

- Outputs the Internet Gateway ID for reference by other modules if required.

vpc_id

- Provides the VPC ID to the root module so other services can attach resources inside this network.

public_subnets

- Returns a list of all public subnet IDs.
- Used by modules like ECS, which require public subnets for Fargate tasks.

private_subnets

- Returns a list of private subnet IDs.
- Used by the RDS module to deploy the database securely.

Task 1.5: Configure Security Group Module

This section explains each Terraform configuration file inside the Security Group module (modules/sg) and how they manage traffic flow between the ECS service and the RDS database.

main.tf:

Public Security Group

- This resource creates a security group inside the VPC (**vpc_id** passed from the root module).
- The purpose of this security group is to allow external HTTP traffic to reach the ECS Fargate service running WordPress.
- Ingress Rule:
    - Allows inbound TCP traffic on port 80 from anywhere (0.0.0.0/0).
    - This makes the ECS service publicly accessible for web requests.
- Egress Rule:
    - Allows all outbound traffic to any destination.
    - Ensures ECS tasks can reach external services such as ECR, package repositories, or the NAT Gateway.
- Tagged for clarity using Name, Application, and Service, making the resource easy to identify in the AWS Console.

Private Security Group (for RDS MySQL)

```
# Private Security Group used for RDS
resource "aws_security_group" "private_sg" {
  name        = "private-sg"
  description = "Allows MySQL traffic from ECS (public-sg) only"
  vpc_id      = var.vpc_id

  ingress {
    from_port       = 3306
    to_port         = 3306
    protocol        = "tcp"
    security_groups = [aws_security_group.public_sg.id]
  }

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

  tags = {
    Name        = "private-sg"
    Application = "RDS MySQL Database"
  }
}
```

- This security group is dedicated to the RDS MySQL database.
- It restricts incoming MySQL connections to only the ECS application by referencing the public security group.
- Ingress Rule:
    - Allows inbound traffic on port 3306 (MySQL) only from the **public_sg** security group.
    - This ensures the database is not exposed publicly and only accepts traffic from trusted ECS tasks.
- Egress Rule:
    - Allows all outbound traffic to support tasks like DNS resolution and updates.
- Tags clearly indicate that this SG belongs to the RDS database layer.

This setup ensures proper separation of concerns:

- The ECS service is accessible to the internet.
- The RDS database is isolated inside private subnets and only reachable by ECS.

variables.tf:

- **vpc_id** is the only required input variable.
- It ensures both security groups are created inside the correct VPC, making the module fully reusable across environments.

outputs.tf

security_group_id

- Outputs the public security group ID.
- Used by the ECS module to attach this SG to the Fargate service.

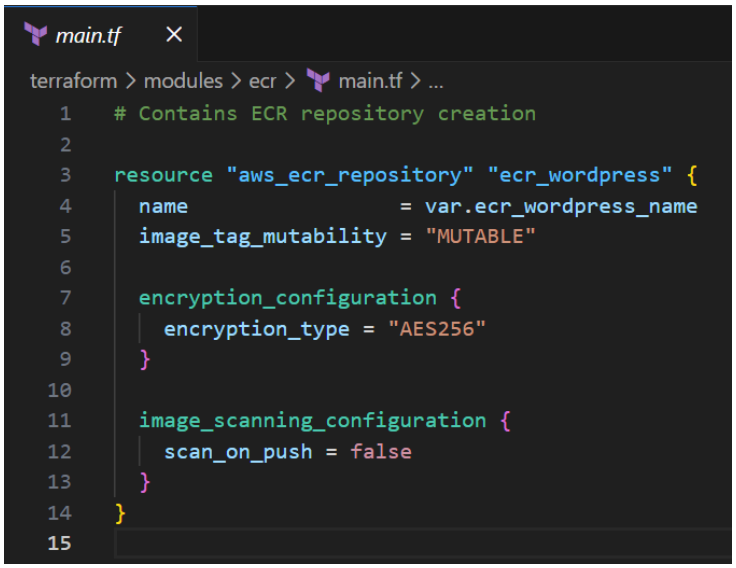
security_group_private_id

- Outputs the private RDS security group ID.
- Used by the RDS module to secure the database and restrict access to only the ECS application.

Task 1.6: Configure Elastic Container Registry (ECR) Module

This section explains the Terraform configuration files inside the **ECR module (modules/ecr)**, responsible for creating the private container registry used by ECS.

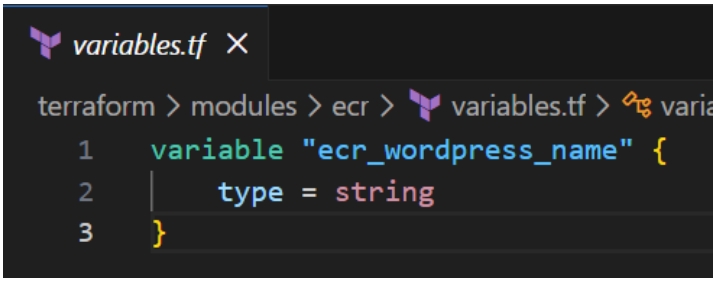
main.tf



- This module creates an Amazon ECR (Elastic Container Registry) repository to store the WordPress Docker image.
- name is dynamically passed from the root variable **ecr_wordpress_name**, allowing flexible naming across environments.
- **image_tag_mutability = "MUTABLE"** allows updating tags (e.g., overwriting latest), which simplifies redeployments.
- The repository is encrypted using **AES256** for secure storage.
- Image scanning on push is disabled (**scan_on_push = false**), though it can be enabled in production environments.

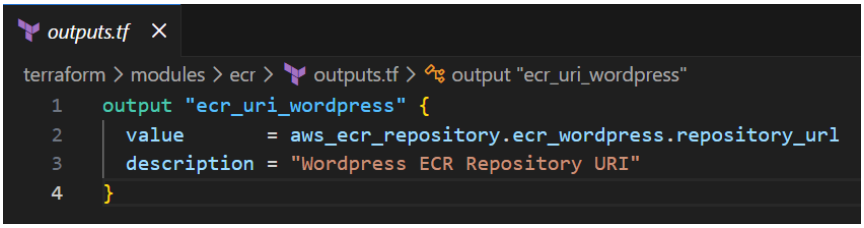
This registry will store the container image that is later pulled by ECS Fargate during application deployment.

variables.tf:



- Defines the variable **ecr_wordpress_name**, allowing the repository name to be supplied from **terraform.tfvars**.
- Keeping this variable external ensures the module can be reused for any application image in the future.

outputs.tf:



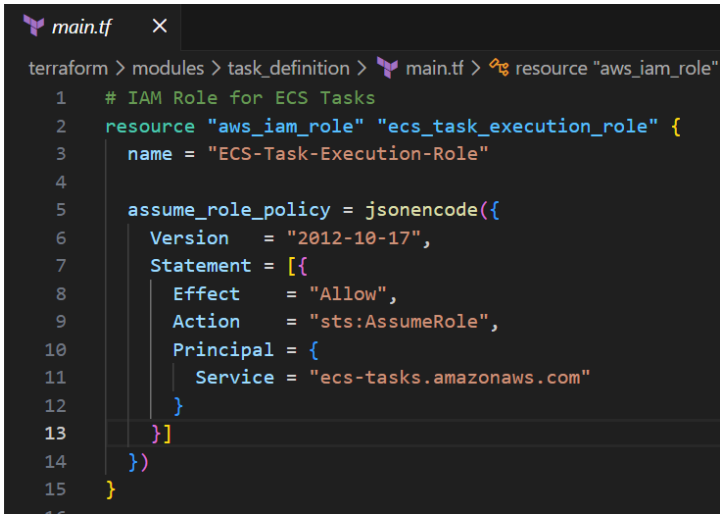
- Outputs **ecr_uri_wordpress**, which exposes the full repository URL (e.g., 123456789012.dkr.ecr.us-east-1.amazonaws.com/wordpress).
- This URL is passed to the Task Definition module so ECS knows where to pull the WordPress image from.

Task 1.7: Configure Task Definition Module

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

main.tf:

ECS Task Execution Role



- Creates an IAM Role that ECS tasks assume during execution.
- The **assume_role_policy** allows ECS to use this role.
- This role allows the task to:
    - Pull images from ECR
    - Send logs to CloudWatch
    - Access necessary AWS APIs during runtime

IAM Policy Attachment

```
# IAM Policy
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 AWS-managed policy **AmazonECSTaskExecutionRolePolicy.**
- This policy automatically provides:
    - ECR read access
    - CloudWatch Logs permissions
    - Basic ECS runtime permissions

This avoids creating custom IAM policies unless needed.

CloudWatch Log Group

```
# CloudWatch Log Group
resource "aws_cloudwatch_log_group" "ecs_logs" {
  name              = "/ecs/wordpress-app"
  retention_in_days = 7
}
```

- Creates a CloudWatch Logs group named /ecs/wordpress-app.
- All container logs from ECS will be stored here.
- Retention is set to 7 days, preventing unnecessary storage costs.

ECS Task Definition

```
# ECS Task Definition
resource "aws_ecs_task_definition" "task_definition" {
  family                   = "wordpress-app"
  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

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

      environment = [
        { name = "WORDPRESS_DB_HOST",     value = "${var.rds_endpoint}:3306" },
        { name = "WORDPRESS_DB_USER",     value = var.db_username },
        { name = "WORDPRESS_DB_PASSWORD", value = var.db_password },
        { name = "WORDPRESS_DB_NAME",     value = var.db_name }
      ]

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

      logConfiguration = {
        logDriver = "awslogs"
        options = {
          awslogs-group         = aws_cloudwatch_log_group.ecs_logs.name
          awslogs-region        = "us-east-1"
          awslogs-stream-prefix = "wordpress"
        }
      }
    }
  ])
```

This resource defines how the WordPress container runs on ECS Fargate. Key settings include:

General Settings

- family = "wordpress-app" groups revisions of the same task definition.
- network_mode = "awsvpc" is required for Fargate.
- CPU and memory are allocated (512 CPU units, 1024 MB memory).

Task and Execution Role

- Both execution_role_arn and task_role_arn use the ECS Task Execution Role created earlier.

Container Definitions

A single container is defined:

- Name: wordpress
- Image pulled from the ECR repository (**${var.ecr_uri_wordpress}:latest**)
- CPU and memory are defined for the container's share of task resources.

Environment Variables

These variables provide WordPress with the required database configuration:

- **WORDPRESS_DB_HOST**: RDS endpoint
- **WORDPRESS_DB_USER**: Database username
- **WORDPRESS_DB_PASSWORD**: Database password
- **WORDPRESS_DB_NAME**: Database name

This connects WordPress running on ECS to the MySQL database running on RDS.

Port Mapping

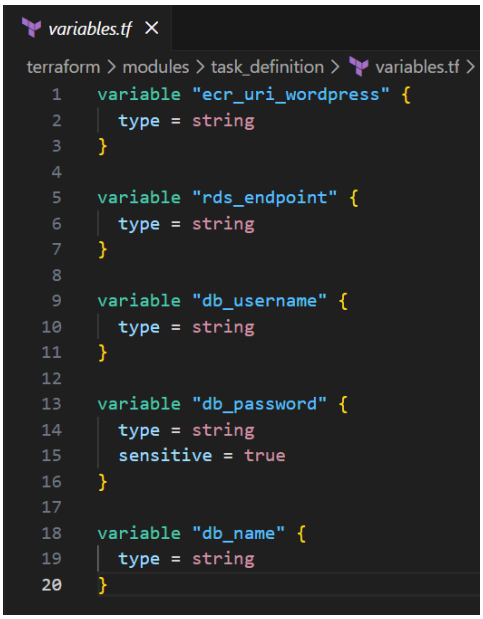
- Maps container port 80, allowing ECS to route traffic to the WordPress application.

Log Configuration

- Sends all logs to CloudWatch using:
    - log group: /ecs/wordpress-app
    - region: us-east-1
    - stream prefix: wordpress

This ensures observability for debugging and monitoring.

variables.tf:

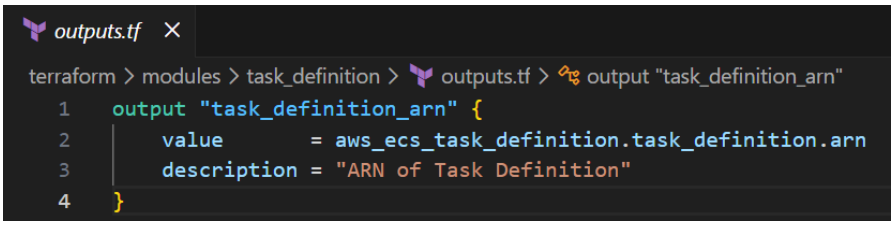


Defines inputs needed by the task definition:

- **ecr_uri_wordpress** is the Image location in ECR
- **rds_endpoint** is the Private RDS MySQL database endpoint
- **db_username, db_password, db_name** define credentials and DB name used by WordPress

The **db_password** variable is marked as sensitive, preventing accidental logging of secrets.

outputs.tf:



task_definition_arn

- Exposes the ARN of the ECS Task Definition.
- This ARN is required by the ECS module to deploy the service.

Task 1.8: Configure Elastic Container Service Module

This module provisions the ECS cluster and ECS service that run the WordPress container on AWS Fargate.

main.tf:

ECS Cluster

```
# Cluster
resource "aws_ecs_cluster" "ecs_cluster" {
  name = "wordpress-ecs-cluster"

  setting {
    name  = "containerInsights"
    value = "enhanced"
  }

  configuration {
    execute_command_configuration {
    logging = "DEFAULT"
    }
  }
}
```

- Creates an ECS cluster named **wordpress-ecs-cluster**.
- Enables Container Insights (enhanced) for improved performance monitoring.
- Configures ECS Exec, allowing secure command execution directly into running containers for debugging.

Cluster Capacity Providers

```
resource "aws_ecs_cluster_capacity_providers" "example" {
  cluster_name = aws_ecs_cluster.ecs_cluster.name

  capacity_providers = ["FARGATE"]

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

- Uses **FARGATE** as the capacity provider.
- Sets Fargate as the default with weight=100, ensuring all tasks run using serverless Fargate capacity.

ECS Service

```
resource "aws_ecs_service" "wordpress-service" {
  name                   = "wordpress-service"
  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
  }
}
```

- Creates the service **wordpress-service** running on Fargate.
- Uses the task definition passed in from the task definition module (**var.task_definition_arn**).
- Runs 2 containers for redundancy and high availability (desired_count = 2).
- Uses REPLICA scheduling strategy to evenly place tasks.
- Enables ECS Exec for troubleshooting.

Networking

- Uses:
    - **public_subnets** for task placement
    - **security_group_id** to control inbound/outbound traffic
    - public IP assignment enabled (**assign_public_ip = true**)

This ensures the WordPress application is reachable externally.

Deployment

- Uses a rolling deployment strategy, ensuring zero downtime when updating the service.

variables.tf:

Defines variables used to configure the service:

- **task_definition_arn:** The task definition to run
- **public_subnets:** List of subnets for Fargate tasks
- **security_group_id:** ECS service security group

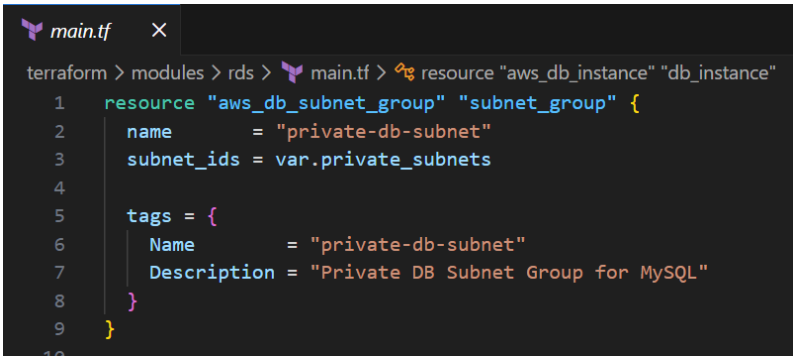
These inputs maintain modularity and allow you to plug in values from other modules.

Task 1.9: Configure Relation Database Service (RDS) Module

This module provisions the MySQL RDS database used by the WordPress ECS application. It creates a private subnet group for the DB and deploys the RDS instance inside secure, non-public subnets.

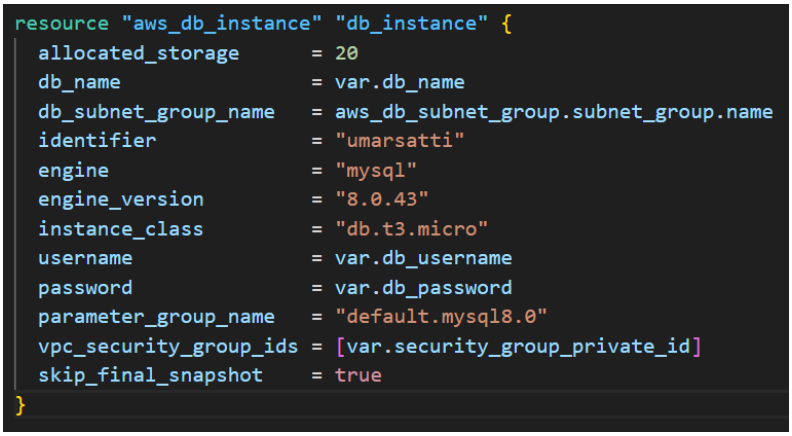
main.tf:

DB Subnet Group



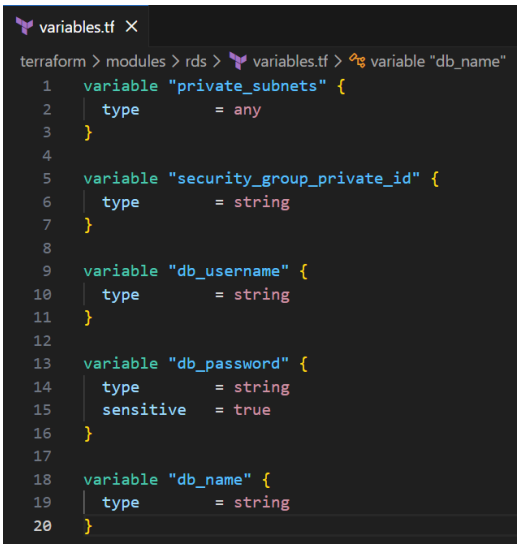
- Creates a subnet group named **private-db-subnet**.
- Uses the **private subnets** passed from the VPC module.
- Ensures the RDS instance is deployed in isolated subnets, not accessible from the internet.

RDS MySQL Instance



- Provisions a **MySQL 8.0** database using a small instance type (db.t3.micro) and 20GB storage.
- Uses credentials and database name passed in from variables.
- Associates the RDS instance with the **private security group** so only ECS containers (via public-sg → private-sg rule) can access it.
- skip_final_snapshot = true makes deletion faster during development.

variables.tf:

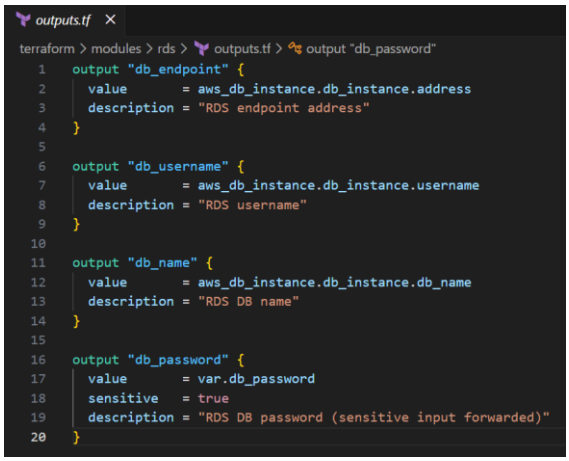


Defines inputs required to deploy RDS:

- **private_subnets**: Ensures DB is placed in private network tiers
- **security_group_private_id**: Restricts DB access to ECS tasks only
- **db_username, db_password**: Credentials for MySQL
- **db_name**: Name of the WordPress database

These values are passed down securely from the root module.

outputs.tf:



Exports key information for downstream modules:

- **db_endpoint** is the hostname used by the WordPress container
- **db_username, db_name** is used in task definition environment variables
- **db_password** is marked sensitive and forwarded securely

These outputs allow the ECS task definition module to connect WordPress to the database.

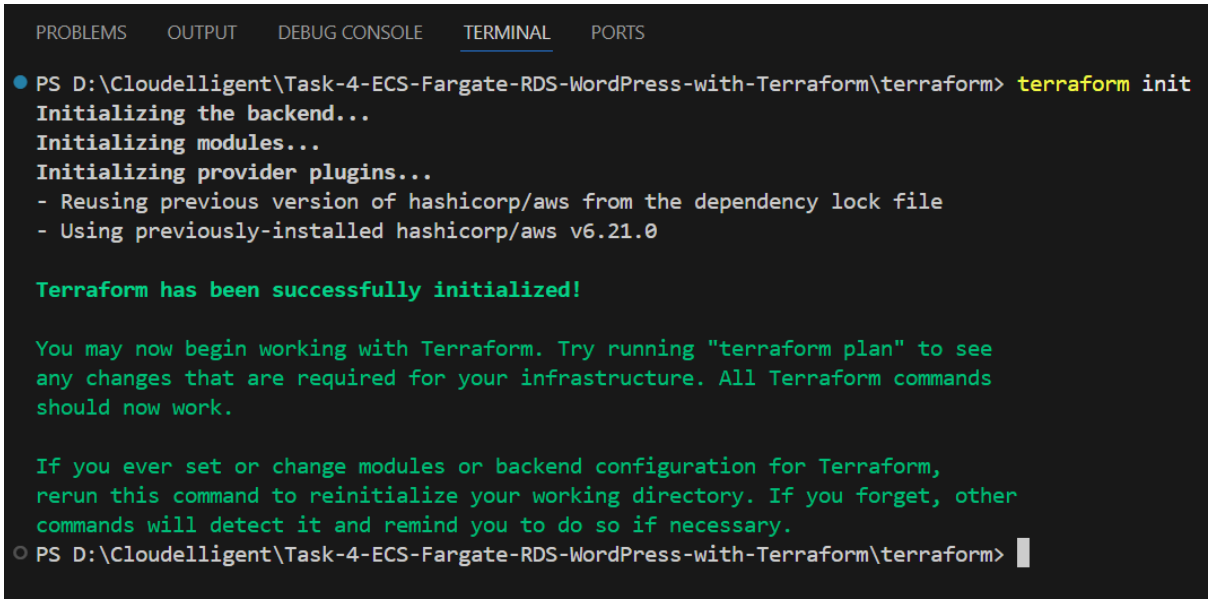
Task 1.10: Running Terraform Commands to Deploy Infrastructure

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

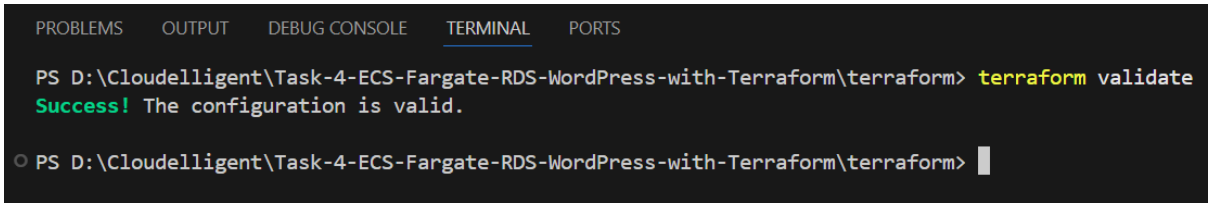
Initializes the working directory by downloading the required provider plugins and connecting them to the configured backend (S3).



By running this command, Terraform confirms initialization, backend setup, and provider readiness.

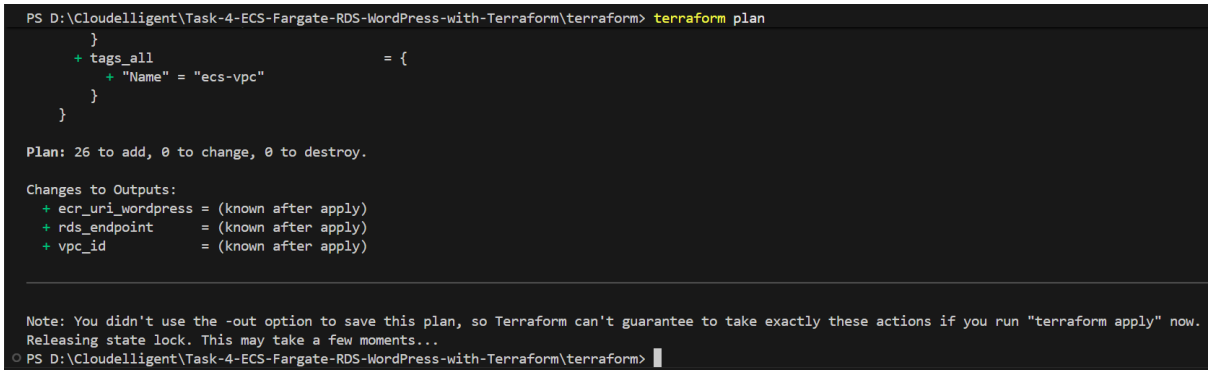
Step 2: terraform validate

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

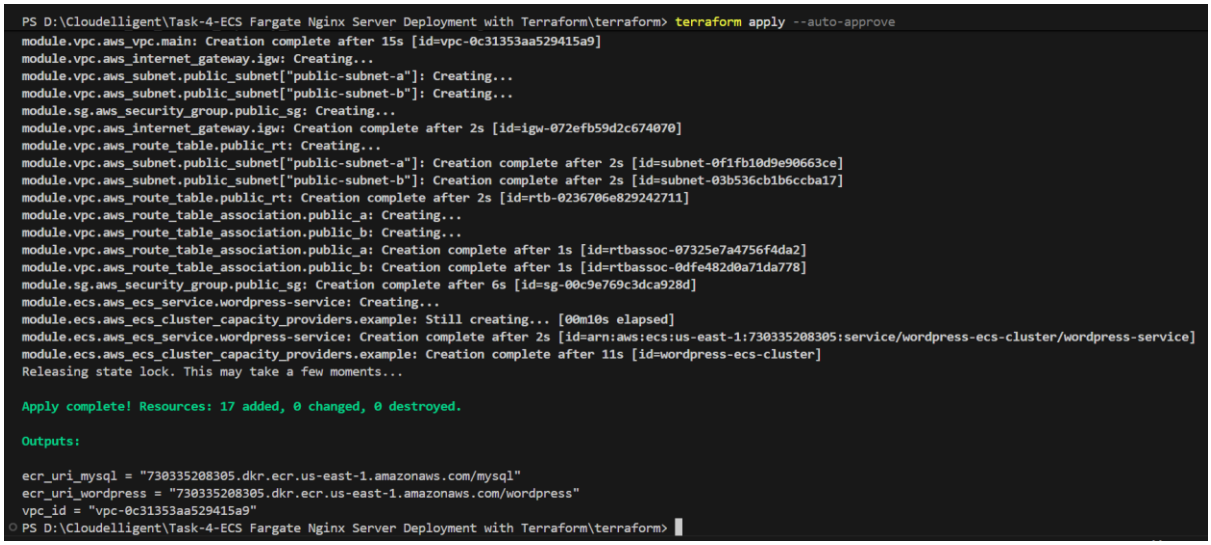
Generates an execution plan showing all actions Terraform will perform to reach the desired state (resource creation, updates, or deletions).



Plan shows **26 to add, 0 to change, 0 to destroy**, confirming all required AWS resources are queued for creation.

Step 4: terraform apply –auto-approve (terraform apply)

Executes the plan and provisions the resources defined in the Terraform configuration without manual confirmation.



Resources (VPC, Subnets, Route Tables, Security Group, ECR, ECS, RDS etc.) are created successfully, followed by VPC ID, ECR URI for WordPress, and RDS Endpoint.

Step 5: terraform destroy --auto-approve (or terraform destroy)

Destroys all resources previously created by Terraform, cleaning up the AWS environment.

```
PS D:\Cloudelligent\Task-4-ECS-Fargate-RDS-WordPress-with-Terraform\terraform> terraform destroy --auto-approve
module.sg.aws_security_group.private_sg: Destroying... [id=sg-055550042037788d0]
module.rds.aws_db_subnet_group.subnet_group: Destruction complete after 1s
module.vpc.aws_subnet.private_subnet["private-subnet-a"]: Destroying... [id=subnet-0ff1749746b583d5a]
module.vpc.aws_subnet.private_subnet["private-subnet-b"]: Destroying... [id=subnet-046c577b01ca4b010]
module.vpc.aws_subnet.private_subnet["private-subnet-b"]: Destruction complete after 1s
module.vpc.aws_subnet.private_subnet["private-subnet-a"]: Destruction complete after 1s
module.sg.aws_security_group.private_sg: Destruction complete after 2s
module.sg.aws_security_group.public_sg: Destroying... [id=sg-06e2b01e21cf91100]
module.sg.aws_security_group.public_sg: Destruction complete after 1s
module.vpc.aws_vpc.main: Destroying... [id=vpc-0e6ddc9cedfbc3776]
module.vpc.aws_vpc.main: Destruction complete after 1s
Releasing state lock. This may take a few moments...

Destroy complete! Resources: 26 destroyed.
PS D:\Cloudelligent\Task-4-ECS-Fargate-RDS-WordPress-with-Terraform\terraform>
```

Displays **Plan: 0 to add, 0 to change, 26 to destroy** and confirm complete deletion of all deployed resources.

The state file in S3 will also be updated automatically to reflect the destroyed state.

Task 1.11: Validate Infrastructure Deployment and WordPress Installation

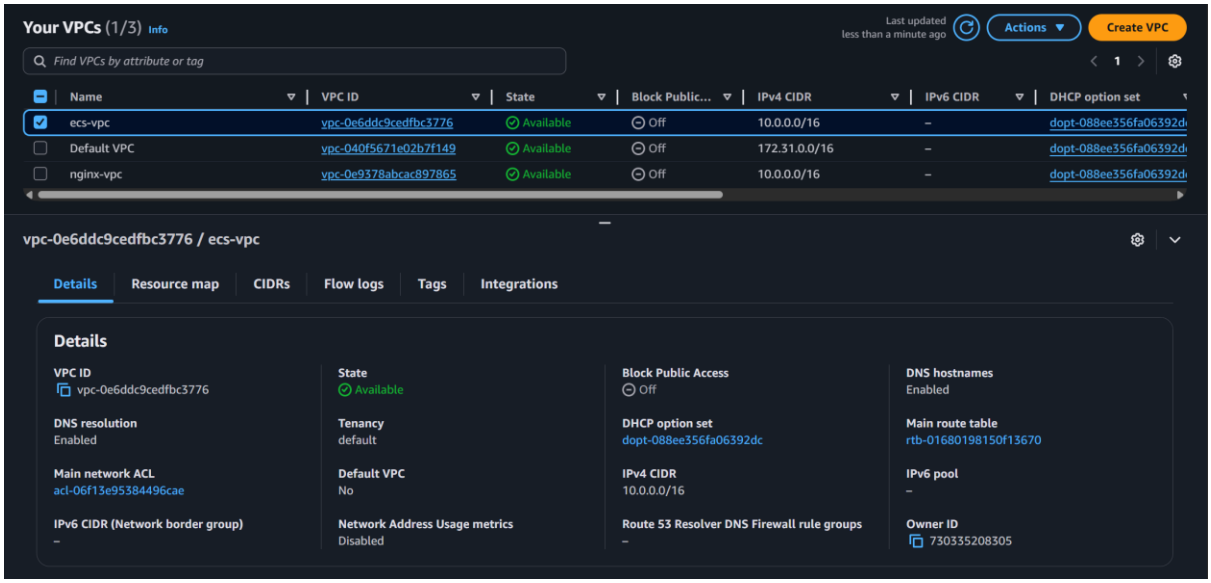
This task demonstrates the successful deployment and verification of all AWS and application-level components created using Terraform. It includes screenshots of the AWS infrastructure created such as VPC, Security groups, ECS, ECR etc.

Task 1.11.1: Verify Deployment of AWS Infrastructure

After running **terraform apply** command, Terraform provisions all the required networking and compute resources. In this step, confirm that these resources exist and are configured correctly.

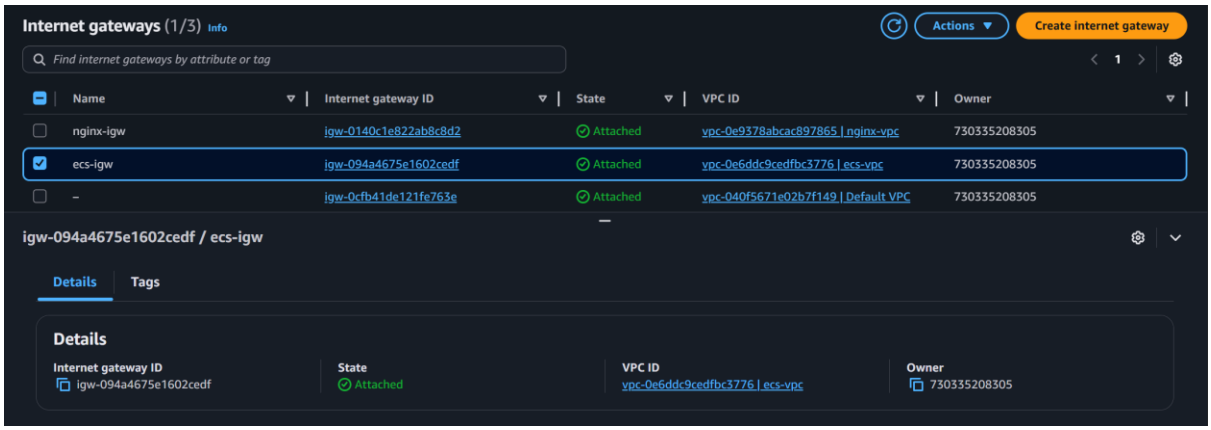
Verify VPC Creation

1. Sign in to the AWS Management Console.
2. Navigate to **VPC** service using the search bar at the top and then click **Your VPCs.**
3. Verify that a VPC named **ecs-vpc** has been created.
   - CIDR Block: 10.0.0.0/16
   - DNS Hostnames: Enabled
   - This VPC acts as the isolated network for the WordPress application environment.

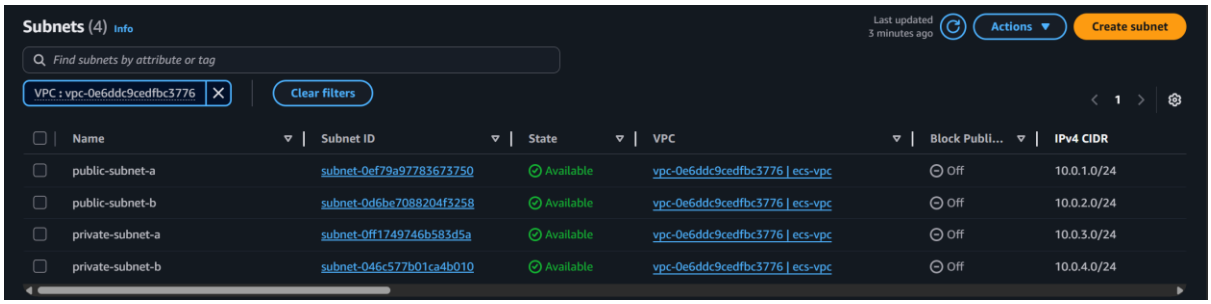


Verify Internet Gateway (IGW) Creation

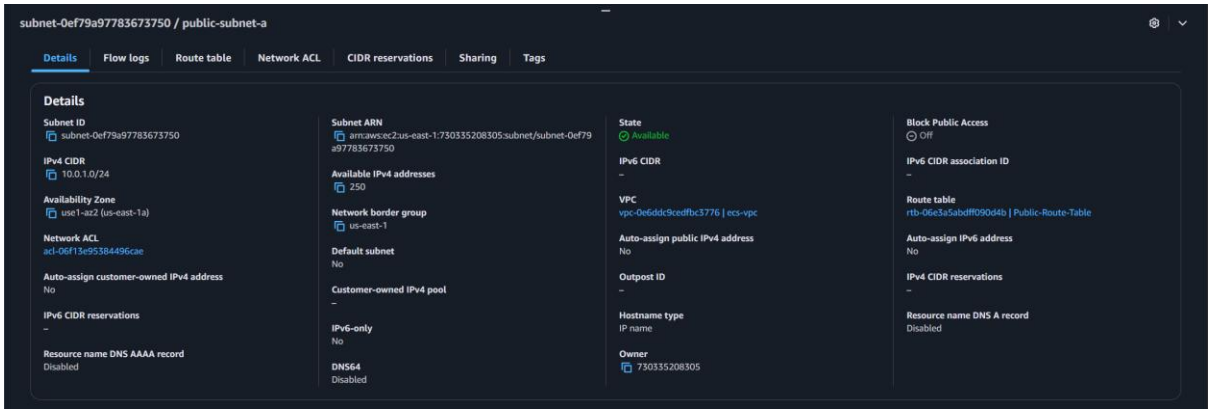
1. In the VPC console, choose **Internet Gateways**.
2. Confirm that an Internet Gateway named **ecs-igw** exists and is attached to your VPC. This gateway allows the ECS service to communicate with the internet.

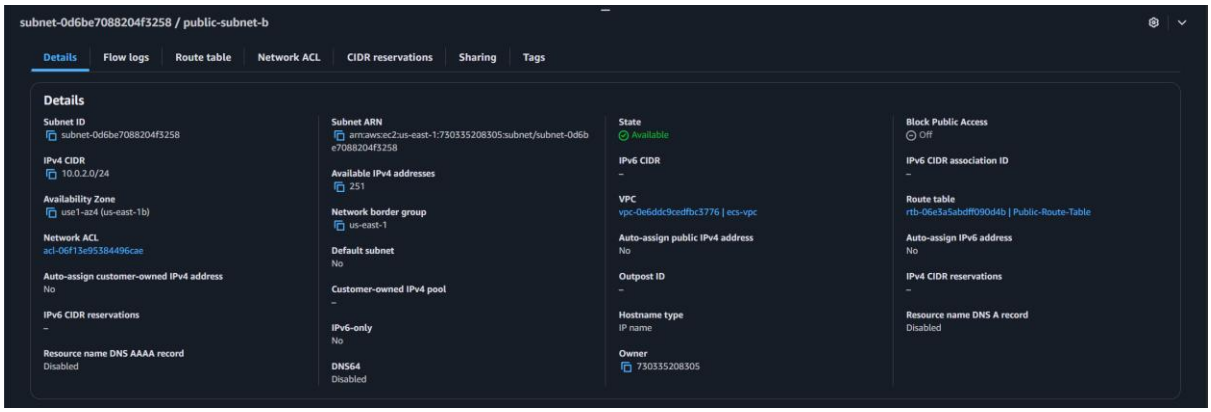


Verify Subnet Creation



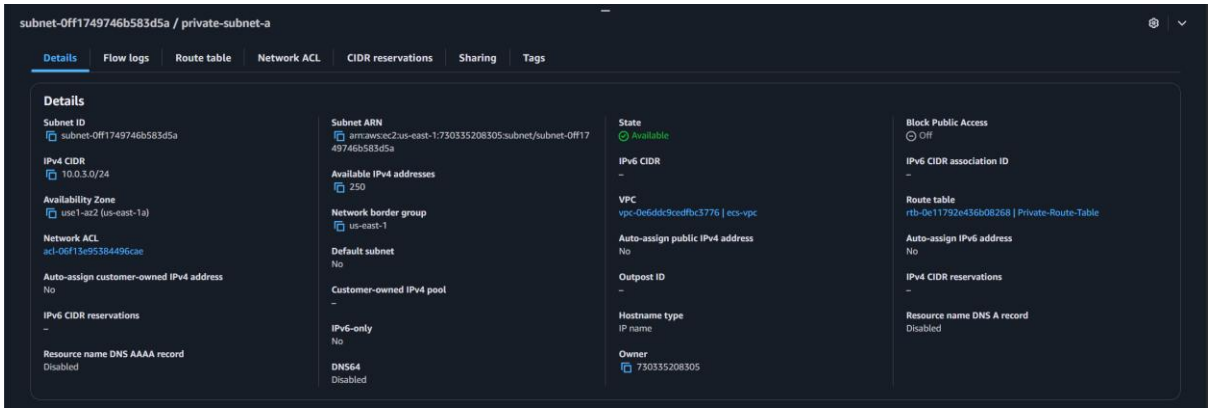
Public Subnets:

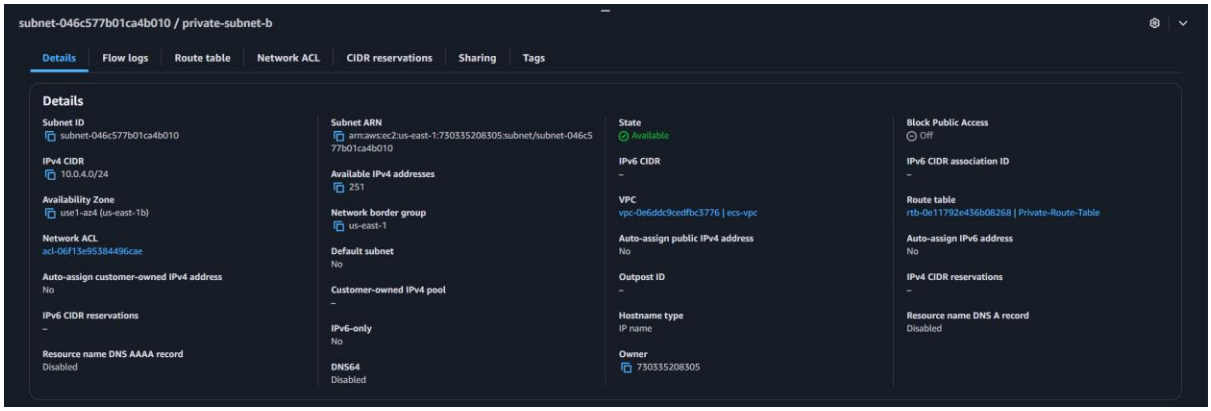




- In the VPC console, choose **Subnets** located on the left navigation panel.
- Confirm that two subnets named **public-subnet-a** and **public-subnet-b** exist.
- The subnets should have an IPv4 CIDR **10.0.1.0/24** and **10.0.2.0/24.**

Private Subnets:

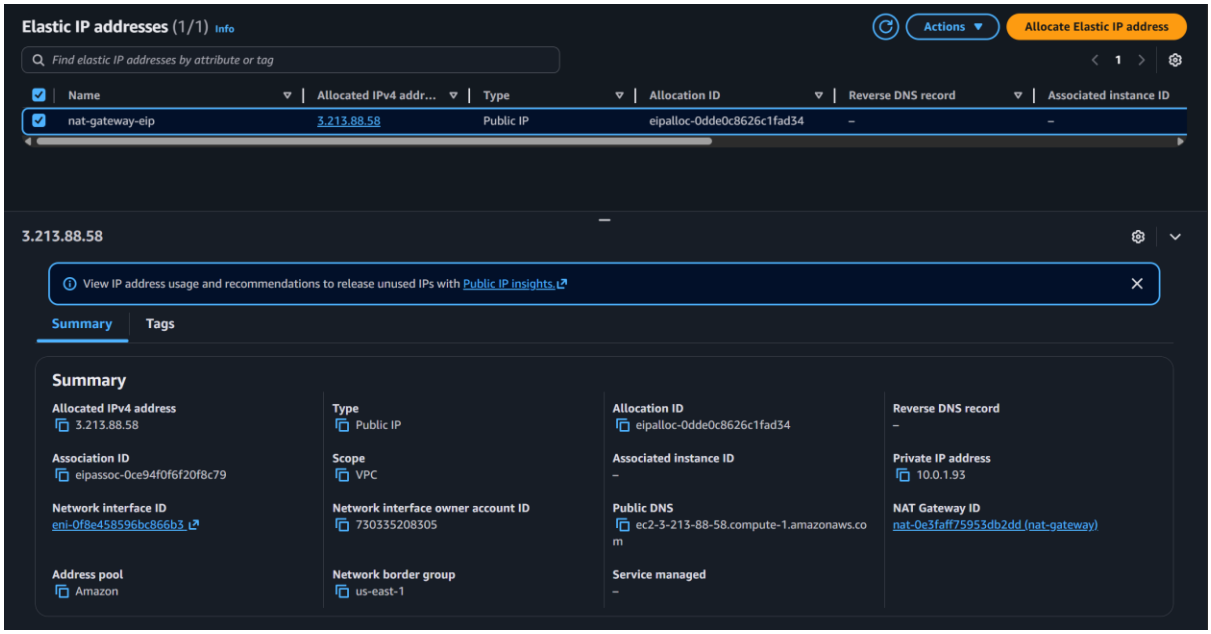




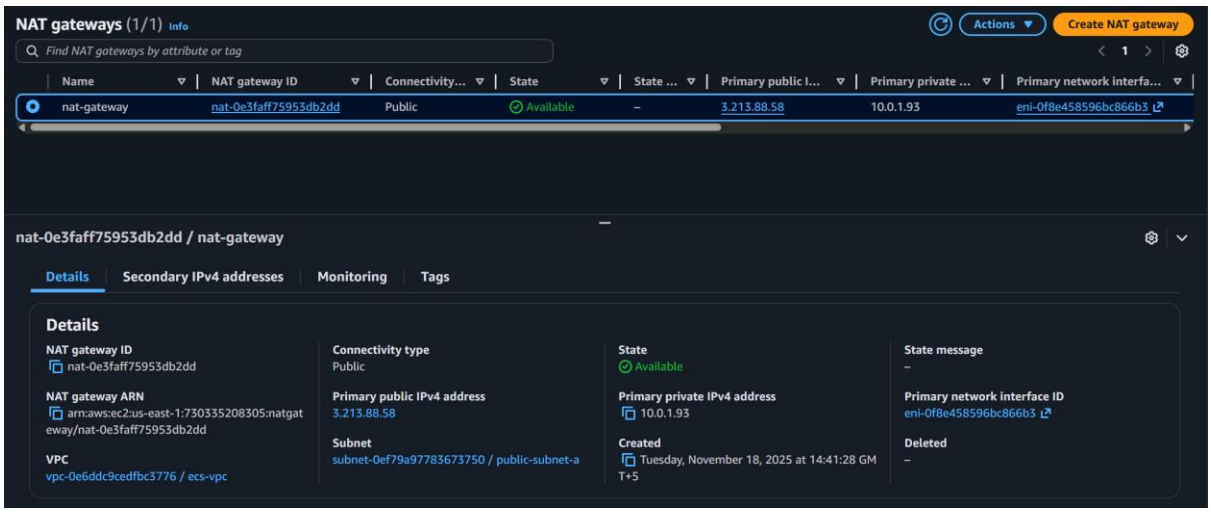
- In the VPC console, choose **Subnets** located on the left navigation panel.
- Confirm that two subnets named **private-subnet-a** and **private-subnet-b** exist.
- The subnets should have an IPv4 CIDR **10.0.3.0/24** and **10.0.4.0/24**.

NAT Gateway and Elastic IP

1. In the VPC console, choose **Elastic IPs** located on the left navigation panel.
2. Select the Elastic IP named **nat-gateway-eip**.
3. Confirm that it is allocated and has a Public IPv4 address.

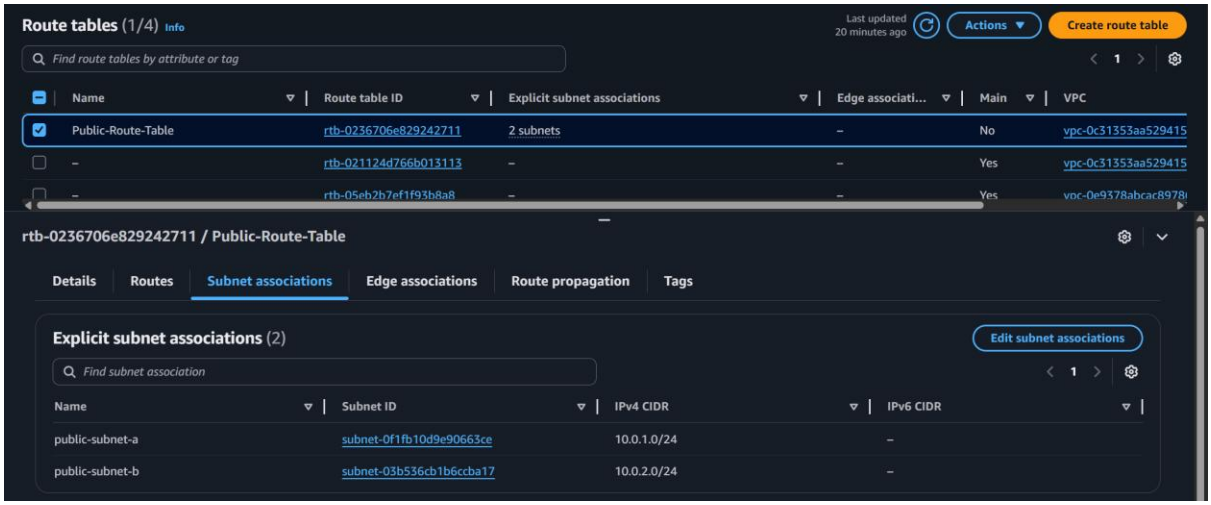


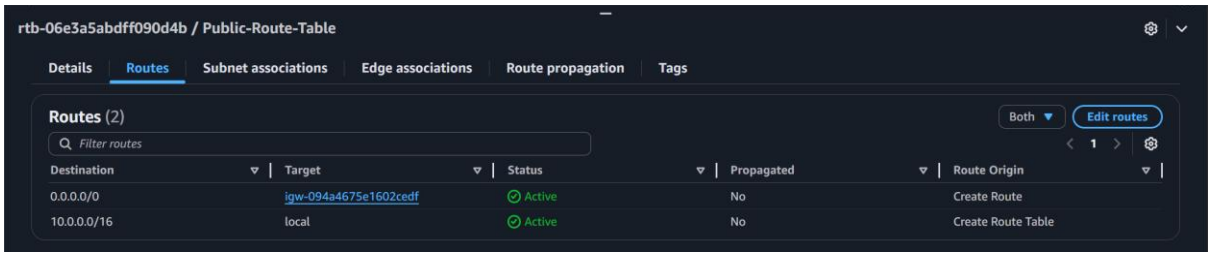
4. Scroll down and click **NAT gateways**.
5. Select the NAT Gateway named **nat-gateway**.
6. Confirm that it belongs to the same **VPC** (**ID: vpc-0e6ddc9cedfbc3776**) and has the same **Primary public IPv4 address** allocated to it (e.g. **3.213.88.58**).



Verify Public Route Table and Routes Creation

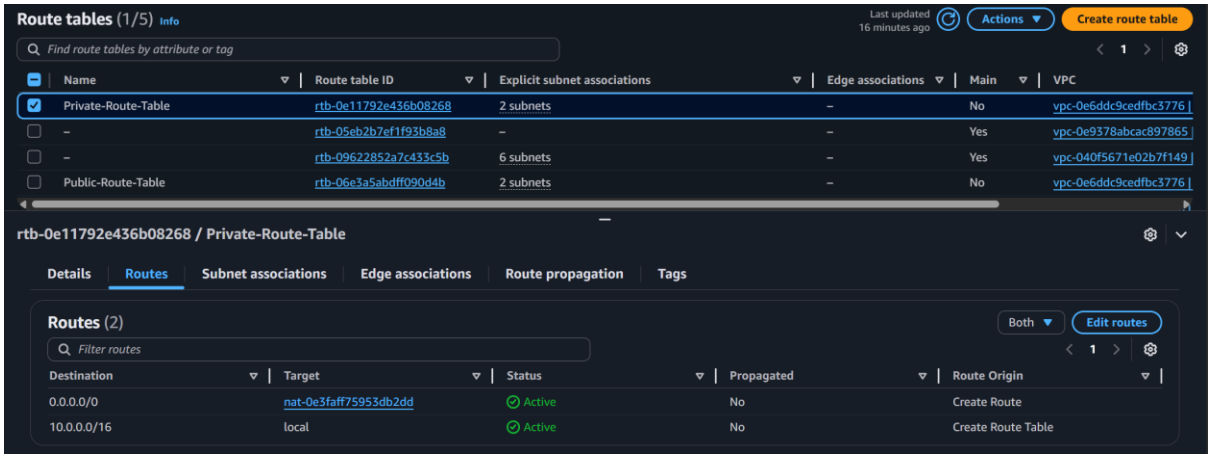
1. In the VPC console, choose **Route tables** located on the left navigation panel.
2. Select the route table named **Public-Route-Table**.
3. Check that it contains a route to the Internet Gateway with the destination **0.0.0.0/0**.
4. Under **Subnet Associations**, verify that **public-subnet-a**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

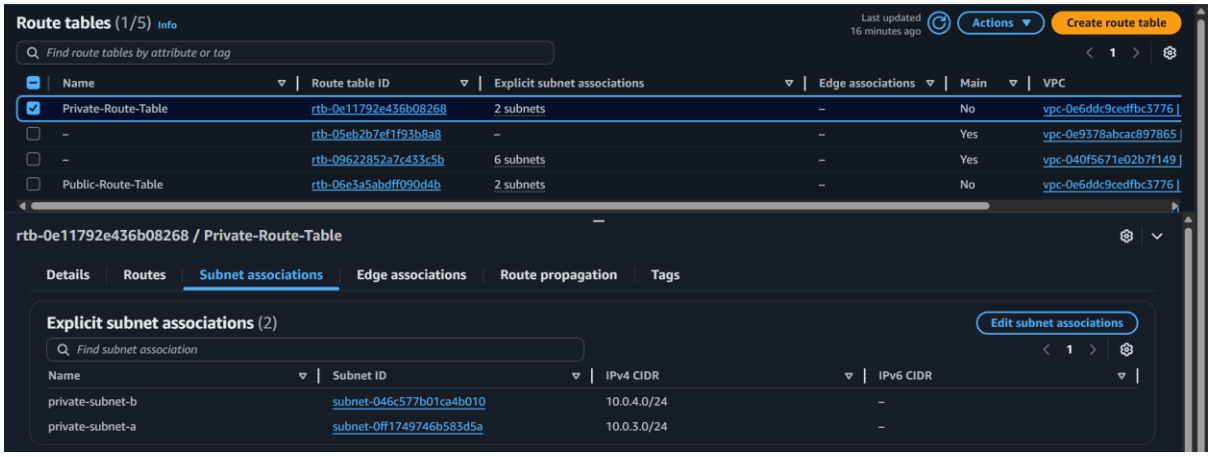




Verify Public Route Table and Routes Creation

1. In the VPC console, choose **Route tables** located on the left navigation panel.
2. Select the route table named **Private-Route-Table**.
3. Check that it contains 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 this route table as shown under **Explicit subnet associations (2)**.





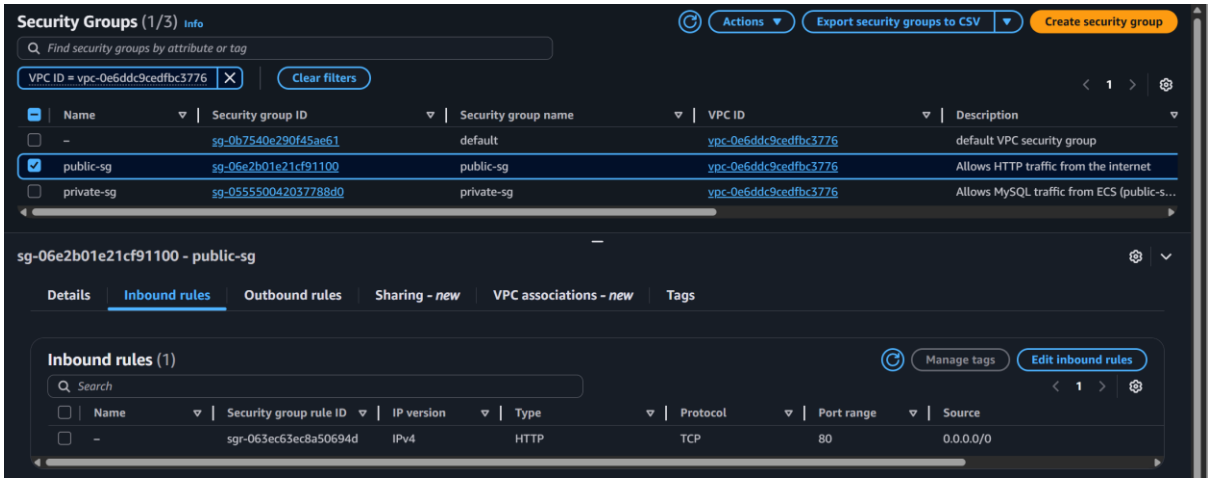
Verify Security Group Creation

Public Security Group

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

Type	Protocol	Port range	Source
HTTP	TCP	80	0.0.0.0/0

4. Outbound rules should allow all traffic.

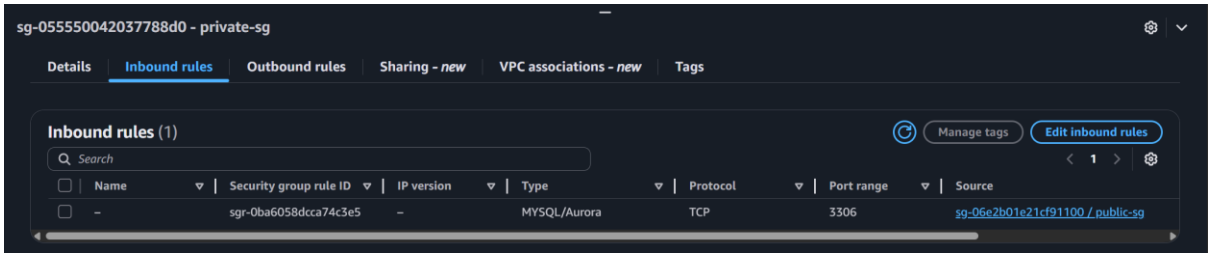


Private Security Group

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

Type	Protocol	Port range	Source
MySQL/Aurora	TCP	3306	public-sg

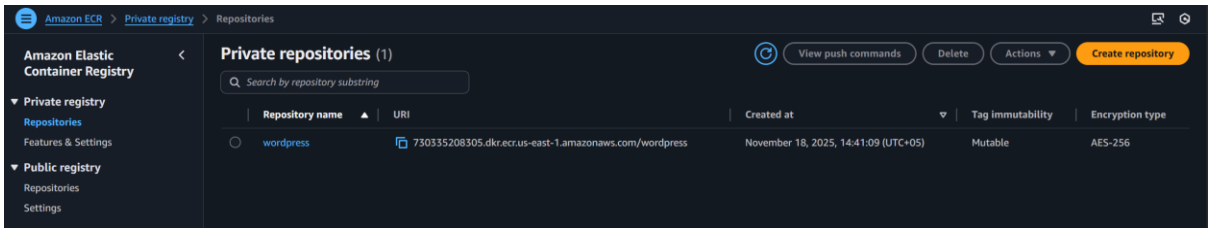
4. Outbound rules should allow all traffic.



At this point, VPC, Internet Gateway, NAT gateway, elastic IP, subnets, route tables, routes and security groups are confirmed as deployed and functional.

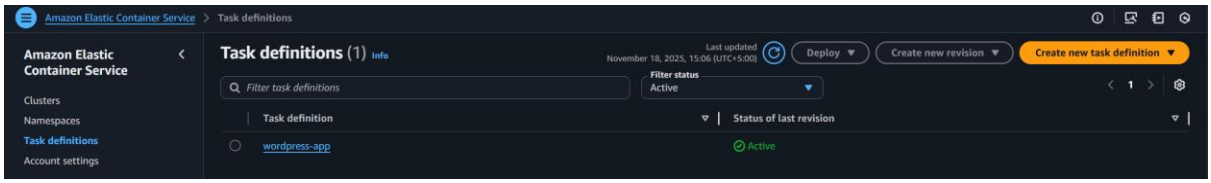
Verify ECR Creation

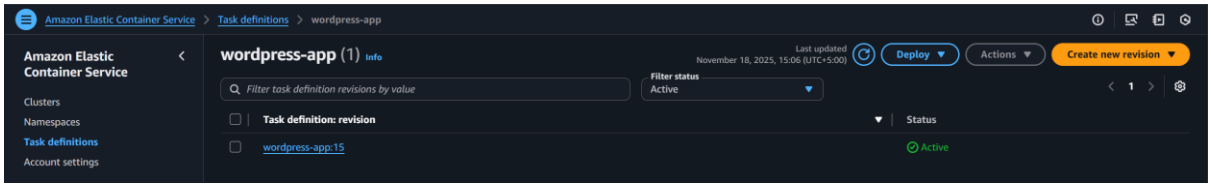
1. Sign in to the AWS Management Console.
2. Navigate to **ECR** service using the search bar at the top and then click **Repositories** under **Private registry** located in the left navigation bar.
3. Verify that the **wordpress** repository was created with a URI that matches the account ID.



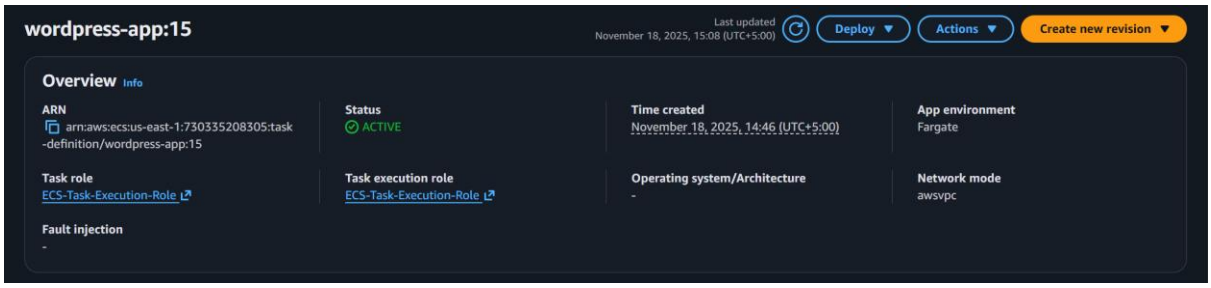
Verify Task Definition Creation

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 **wordpress-app** is created.

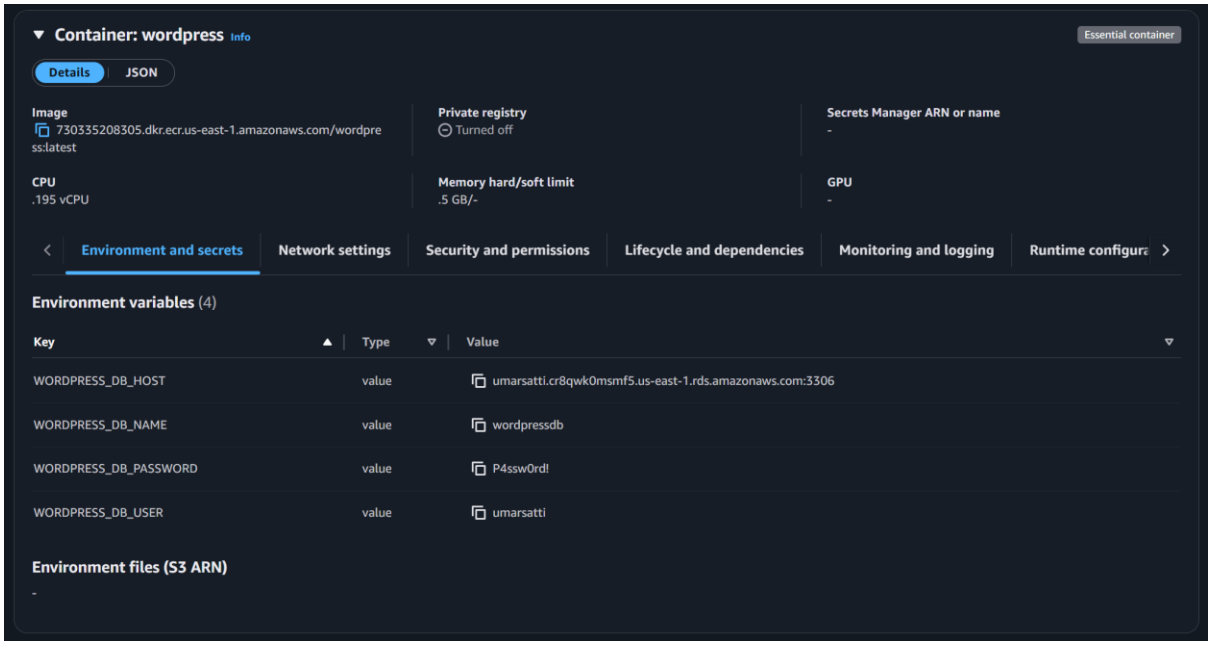




4. Click on **wordpress-app** then and then click on **wordpress-app** again to view details.



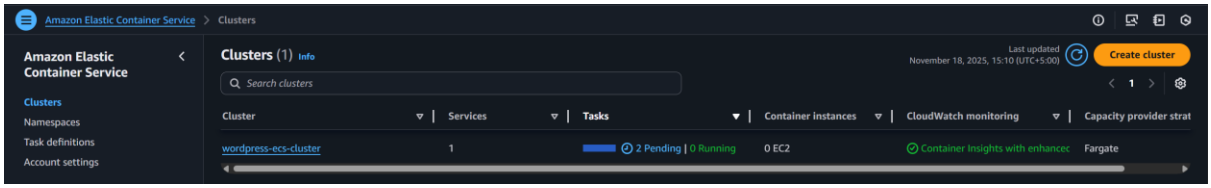
5. Confirm that it has an **Active** status.
6. Click **Containers** and scroll down to confirm that **wordpress** container has been defined with proper **Environment Variables** as shown below.



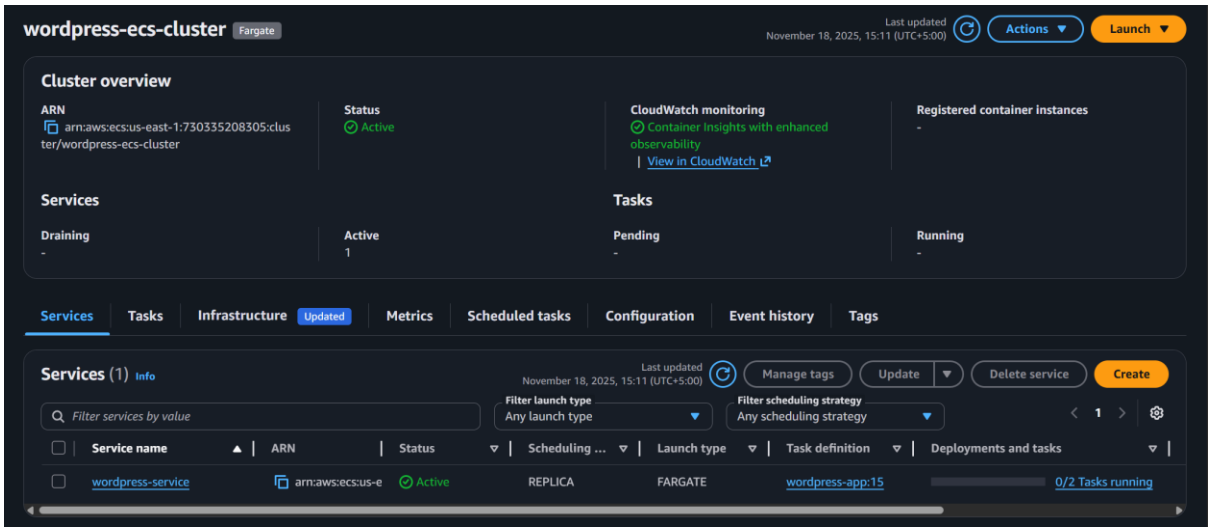
7. As shown in the image above, the environment variables value (as key-value pairs) correctly corresponds to RDS endpoint, database name, username, and password.

Verify ECS Cluster Creation

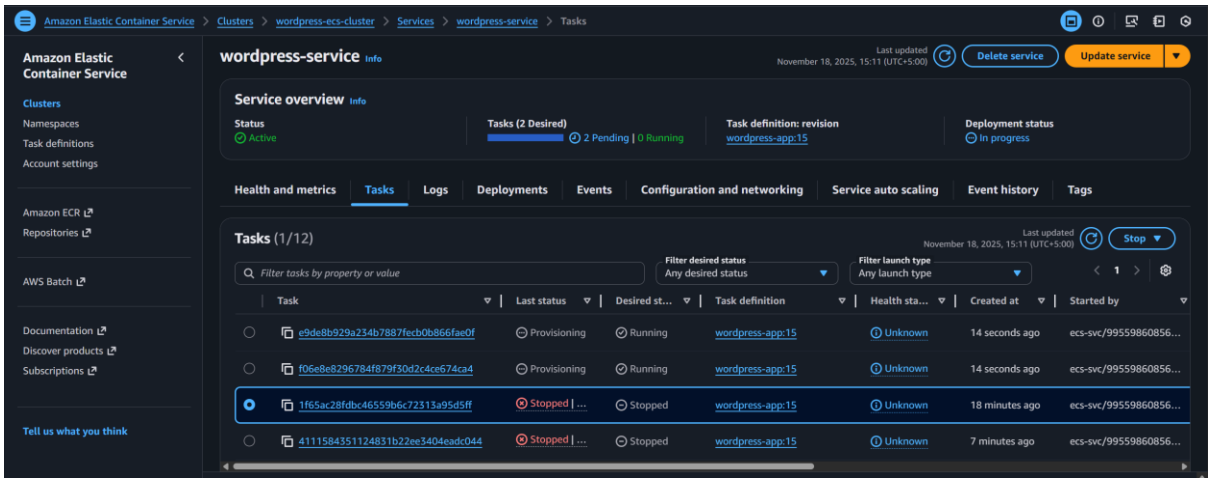
1. Sign in to the AWS Management Console.
2. Navigate to **ECS** service using the search bar at the top.
3. Click **Clusters** located in the left navigation bar.



4. Verify that a cluster named **wordpress-ecs-cluster** is already created.
5. Click on this **wordpress-ecs-cluster**.



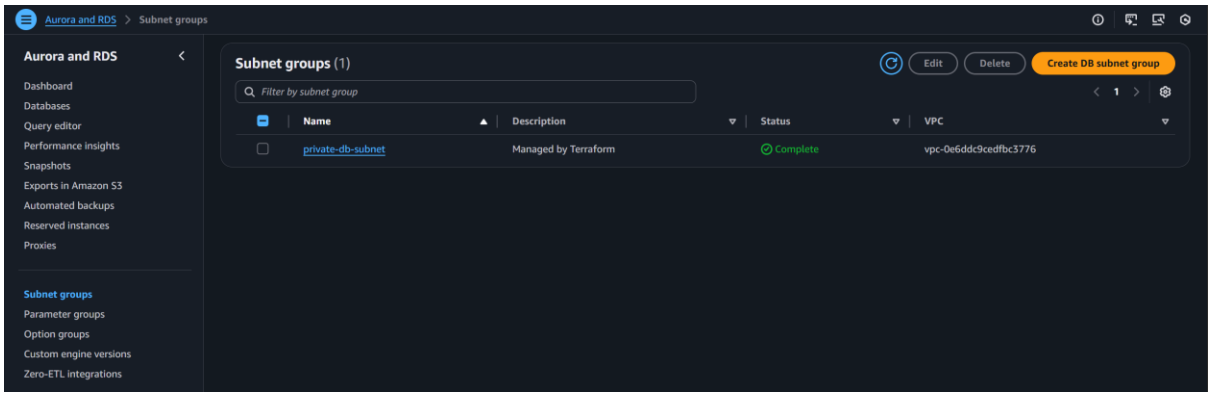
6. There should be a **Service** running with an **Active** status and **Deployments and tasks** showing **Tasks** running.
7. Click on **Tasks** tab. It should show Tasks in a **Stopped** state. This is because the **Tasks** are failing as there are **NO** images that have been **Pushed** to the ECR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



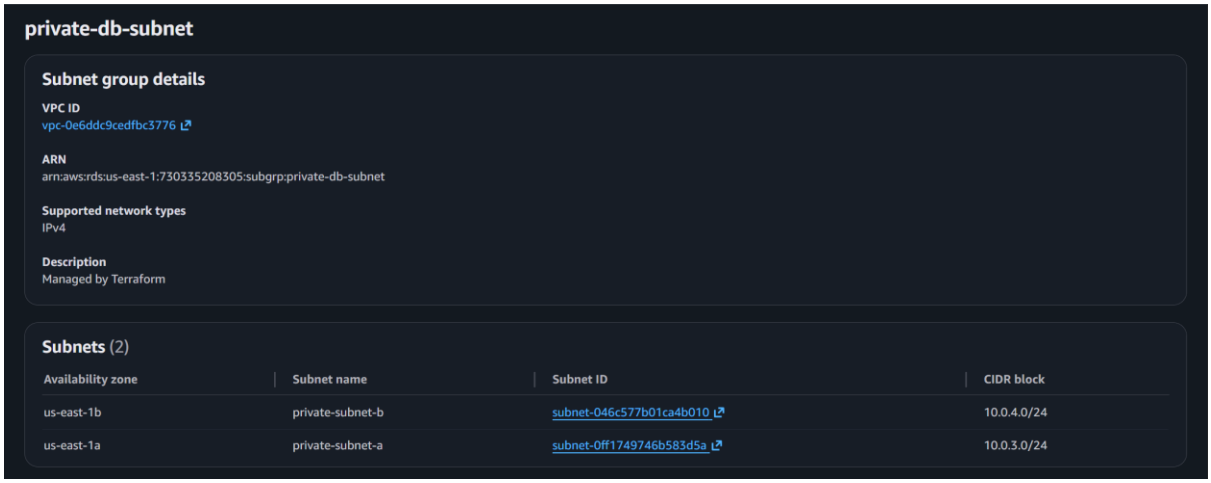
Verify RDS DB Subnet and Instance Creation

1. Sign in to the AWS Management Console.
2. Navigate to **RDS** service using the search bar at the top.
3. Click **Subnet groups** located in the left navigation bar.

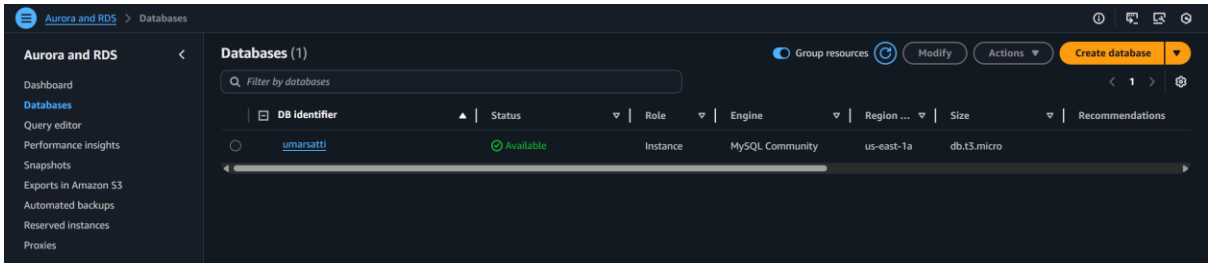
4. There should be a DB subnet group present that is named **private-db-subnet**.



5. Click on this DB subnet to view details.
6. It should have **private-subnet-a** and **private-subnet-b** as its subnets.



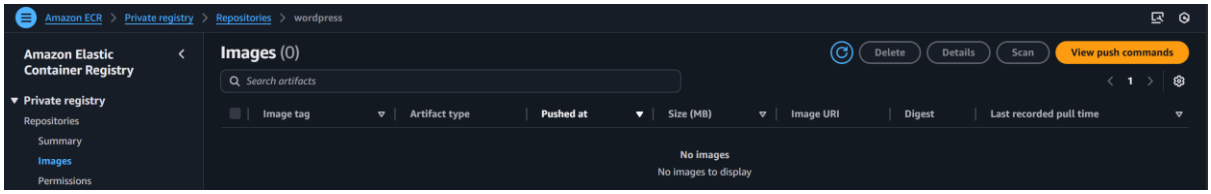
7. Go back to the RDS console and click **Databases**.
8. There should be an RDS DB Instance present that is named **umarsatti**.



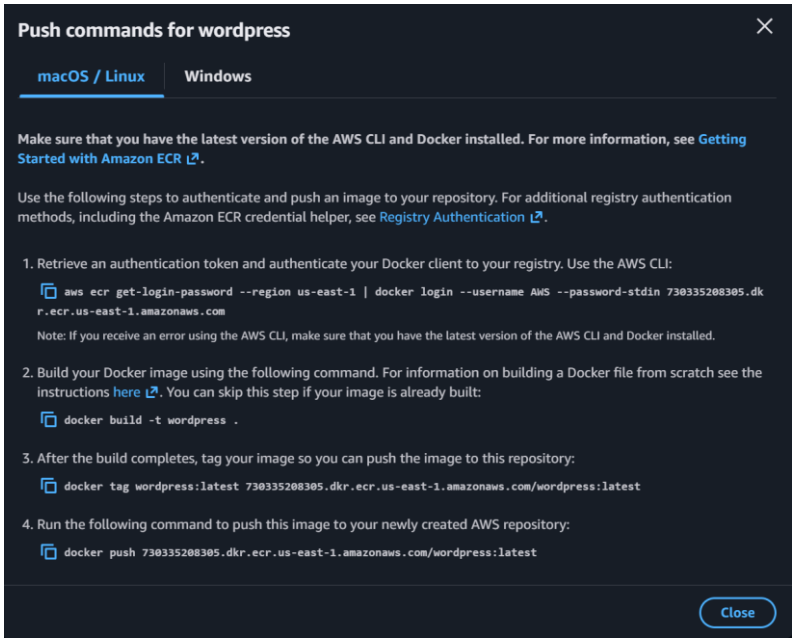
Task 1.10.2: Push WordPress Image to ECR

Push the images to ECR

- From the AWS Management Console, navigate to the **ECR** service using the search bar at the top
- Click **Repositories** located under **Private registry** in the left navigation bar.
- Perform the following:
    - Click **wordpress** repository (it should be empty).



    - Click on **View push commands** button on the top right.
    - Run the commands shown.



Note: Make sure Docker Desktop is installed and running for these commands to work. Additionally for step 2, run ***docker pull wordpress*** instead of ***docker build -t wordpress***. This pulls the official WordPress image from DockerHub.

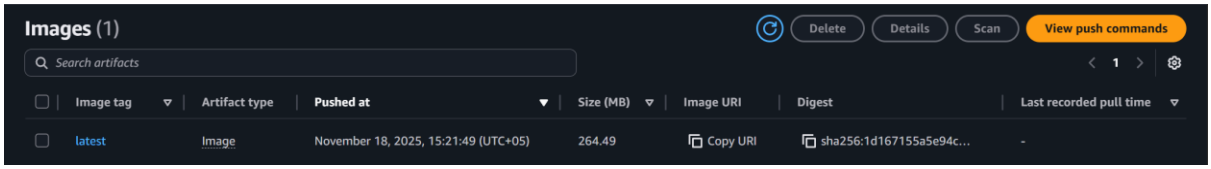
```
umars@Umar-Satti MINGW64 /d/Cloudelligent/Task-4-ECS-Fargate-RDS-WordPress-with-Terraform/terraform
$ aws ecr get-login-password --region us-east-1 | docker login --username AWS --password-stdin 730335208305.dkr.ecr.us-east-1.amazonaws.com
Login Succeeded

umars@Umar-Satti MINGW64 /d/Cloudelligent/Task-4-ECS-Fargate-RDS-WordPress-with-Terraform/terraform
$ docker pull wordpress:latest
latest: Pulling from library/wordpress
0e4bc2bd6656: Pull complete
46605a656915: Pull complete
f7c9fe45c578: Pull complete
8924dcffec96: Pull complete
6a307b08e6b5: Pull complete
bf66ebb0c9a9: Pull complete
e53b88f1a2a7: Pull complete
6067d09e211b: Pull complete
b59bc243be75: Pull complete
86a7cc0b1b7d: Pull complete
be2e241aa1de: Pull complete
02189dee7fe6: Pull complete
b5f62298dacb: Pull complete
e86e5731331c: Pull complete
4f4fb700ef54: Pull complete
4a2bea9f290c: Pull complete
d8cc4dd297ff: Pull complete
f4a7c4b52246: Pull complete
2c420b5256e3: Pull complete
557c18e28a88: Pull complete
db73bbb2ae8f: Pull complete
a5be4a15e86b: Pull complete
172fdd945699: Pull complete
4ebea7a66238: Pull complete
Digest: sha256:5dfda843a925a1da229eadfc5099cf9ce3d5481c6dd872fc7cdcf920951e663d
Status: Downloaded newer image for wordpress:latest
docker.io/library/wordpress:latest

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

umars@Umar-Satti MINGW64 /d/Cloudelligent/Task-4-ECS-Fargate-RDS-WordPress-with-Terraform/terraform
$
```

```
umars@Umar-Satti MINGW64 /d/Cloudelligent/Task-4-ECS-Fargate-RDS-WordPress-with-Terraform/terraform
$ docker tag wordpress:latest 730335208305.dkr.ecr.us-east-1.amazonaws.com/wordpress:latest
```

```
umars@Umar-Satti MINGW64 /d/Cloudelligent/Task-4-ECS-Fargate-RDS-WordPress-with-Terraform/terraform
$ docker push 730335208305.dkr.ecr.us-east-1.amazonaws.com/wordpress:latest
The push refers to repository [730335208305.dkr.ecr.us-east-1.amazonaws.com/wordpress]
93895b9c0c35: Pushed
7853a0439798: Pushed
fe8eca22a016: Pushed
07713e4f529e: Pushed
9780eeff63d5: Pushed
a40b04fb48e0: Pushed
3d10e3005795: Pushed
63035e698d41: Pushed
24fabd4e2fd9: Pushed
5f70bf18a086: Pushed
d57b785a1d6d: Pushed
98fc0bf9532f: Pushed
60193afa7423: Pushed
971f66171313: Pushed
c19583d1051e: Pushed
26b21ea4e330: Pushed
3a5c6176315b: Pushed
5220220157ff: Pushed
72dbab29b306: Pushed
efd9cff39906: Pushed
6c9577238c27: Pushed
2feb3f62c88e: Pushed
953c38baa70a: Pushed
70a290c5e58b: Pushed
latest: digest: sha256:1d167155a5e94c2a8ae7de9d419acd6ffa74a97a8201d7dde7b07726e7b6a580 size: 5330
```

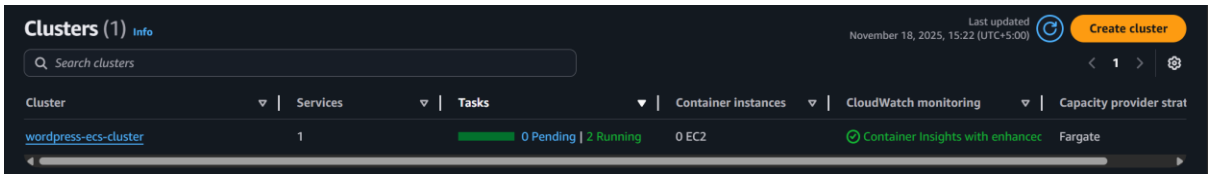
Verify the images have been pushed

1. Click the **wordpress** repository.
2. A **wordpress** image should be available with the **latest** tag.

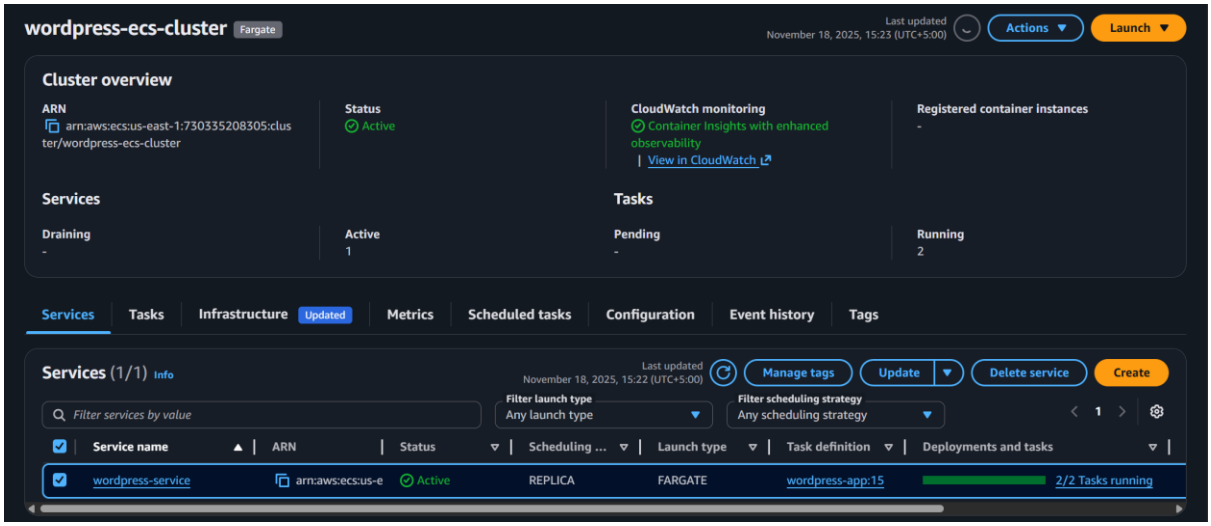


Verify the Service and Tasks are running

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 **Tasks** tab, it should display two tasks running successfully.



3. Click the **wordpress-ecs-cluster** and under **Services** tab, it should display again that 2/2 tasks are running successfully.

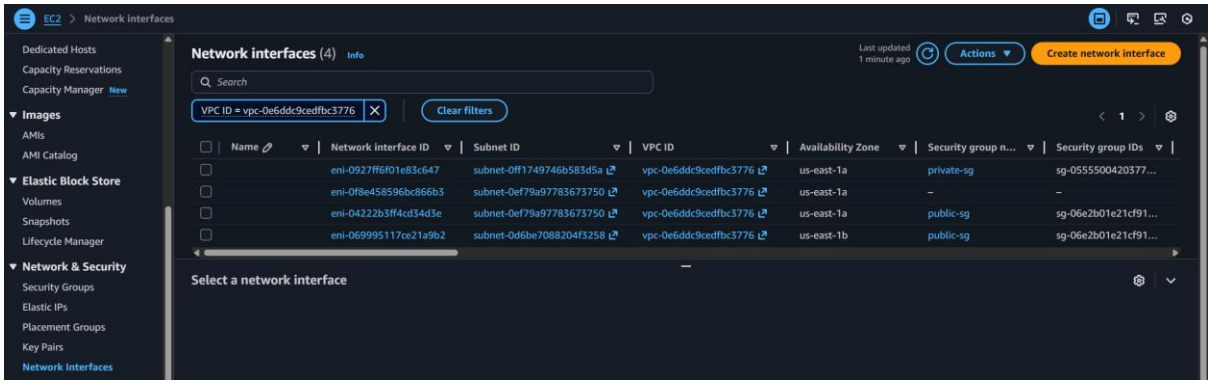


4. Optionally, explore this console for additional information such as Tasks, Logs, Health checks, Metrics etc.

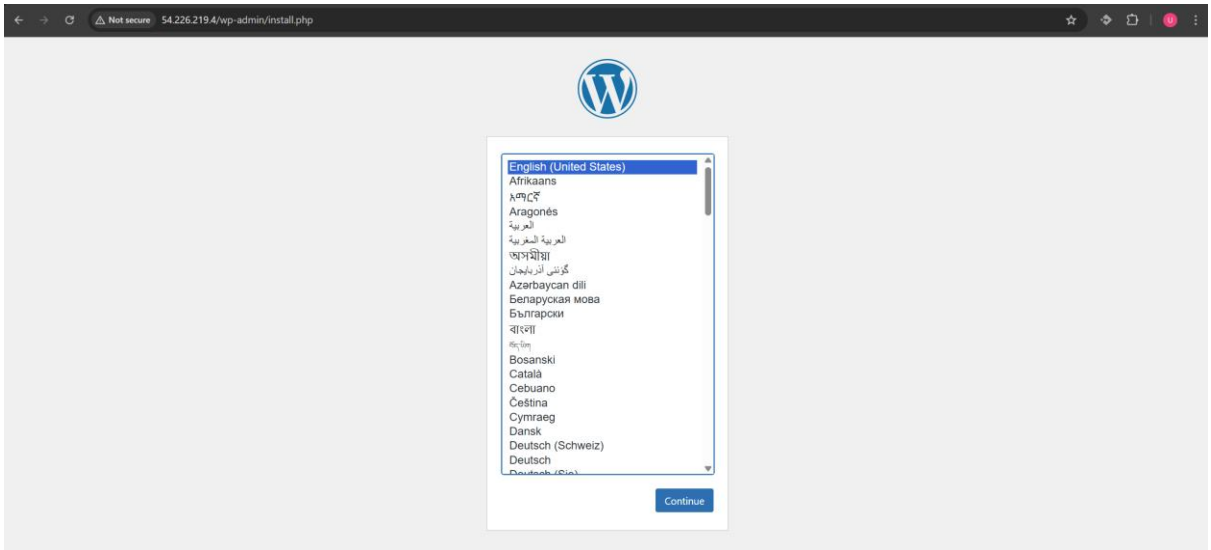
Task 1.10.3: Access the WordPress Application

Elastic Network Interface

- From the AWS Management Console, navigate to the **EC2** service using the search bar at the top.
- Scroll down and click **Network Interfaces** under **Network & Security**
- Two **ENIs** should be deployed with a **Public IPv4 address** which belong to **public-sg** security group and one **ENI** which belongs to **private-sg** security group.



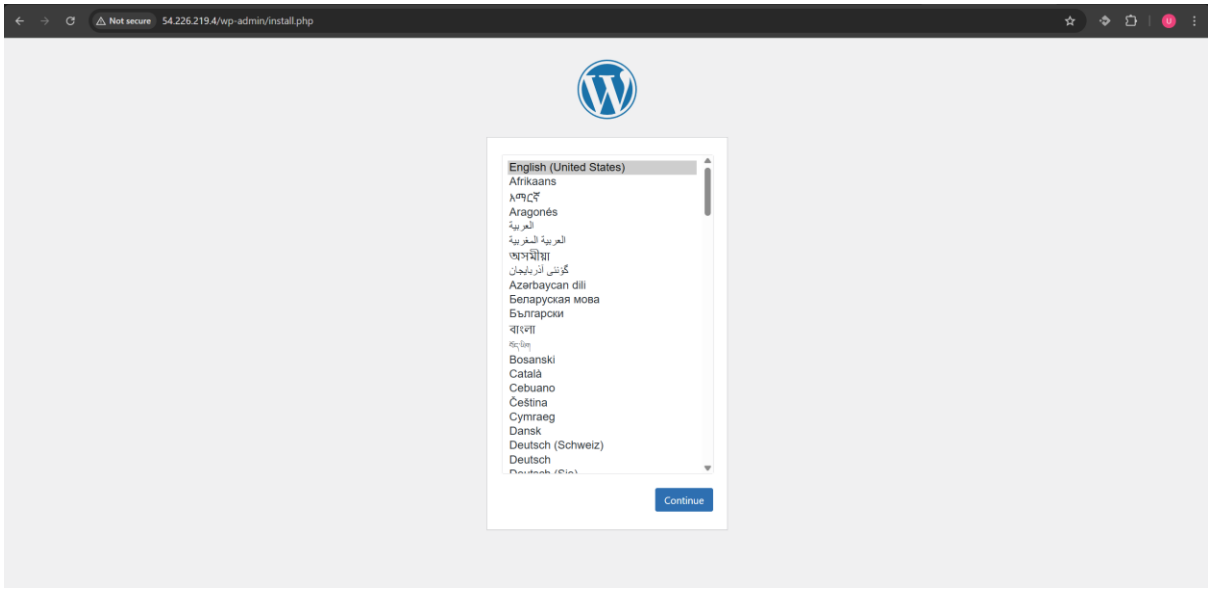
- Copy the IPv4 address from one of these **public-sg ENIs**.
- Open the application using this Public IP **http://<eni-public-IP>**
  - Example: **http://54.226.219.4**
  - This takes the user to the URL: **/wp-admin/install.php**
- Confirm that the WordPress setup page is opening successfully as shown below.



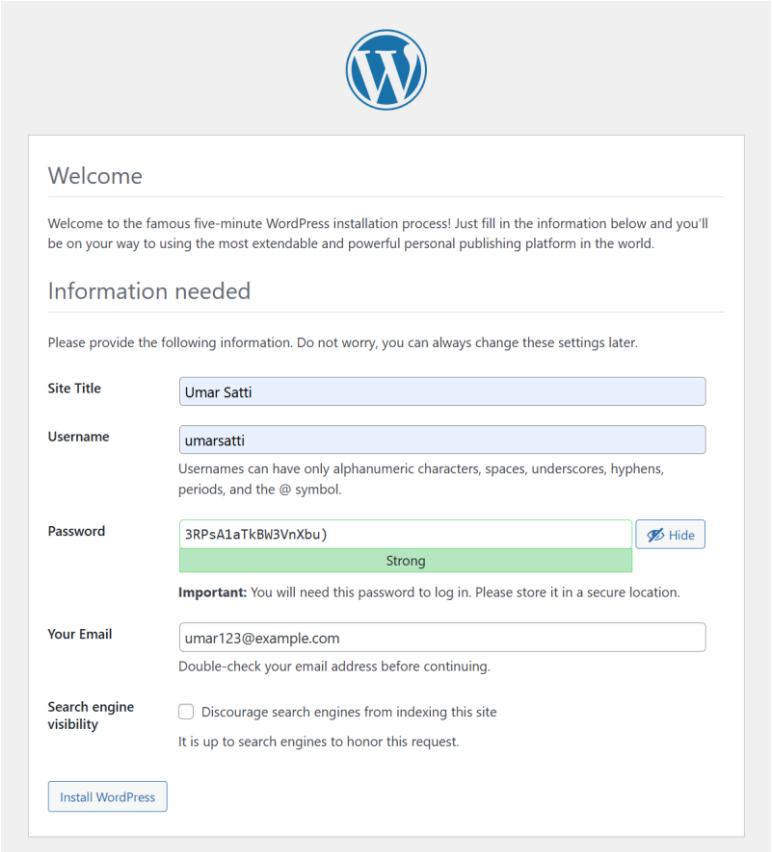
Task 1.10.4: Configure WordPress

Step 1: Open WordPress application

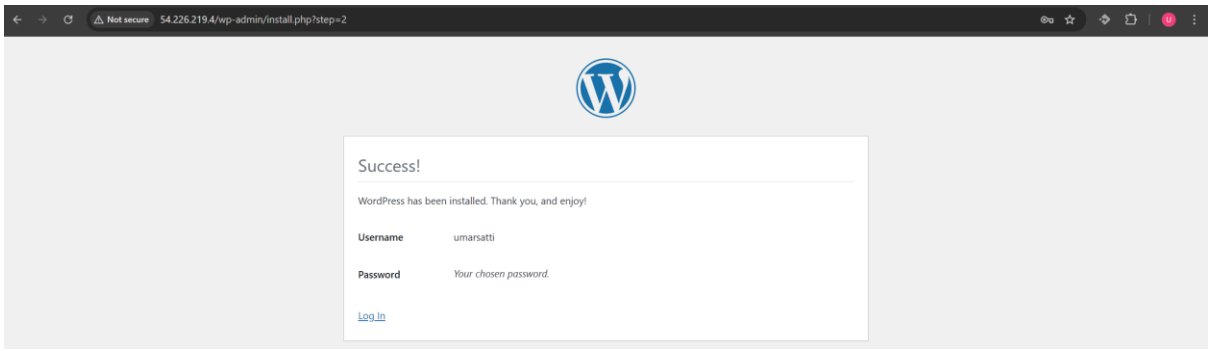
1. In browser, go to **http://<your-ec2-public-ip>/wp-admin/install.php**
    - Example: **http://54.226.219.4/wp-admin/install.php**



Step 2: Install WordPress

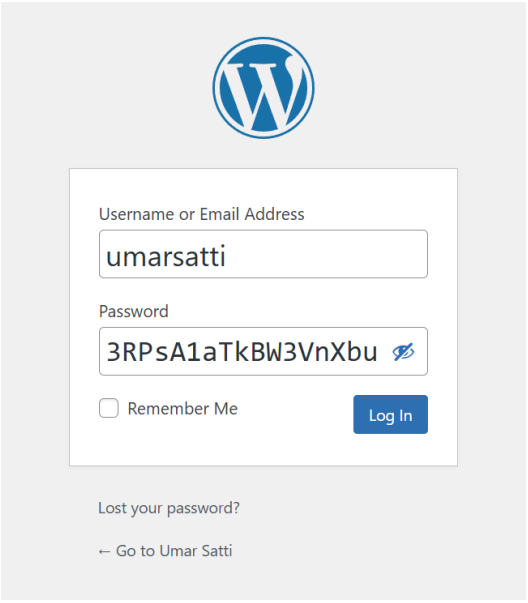


1. Once WordPress verifies database connection, it will display the installation page. To complete installation, fill out the required fields:
    - **Site Title:** Umar Satti
    - **Username:** umarsatti
    - **Password:** 3RPsA1aTkBW3VnXbu)
    - **Email:** umar123@example.com
2. Click **Install WordPress**.
3. After a few seconds, a success message is displayed:
    - "Success! WordPress has been installed. Thank you and enjoy!"

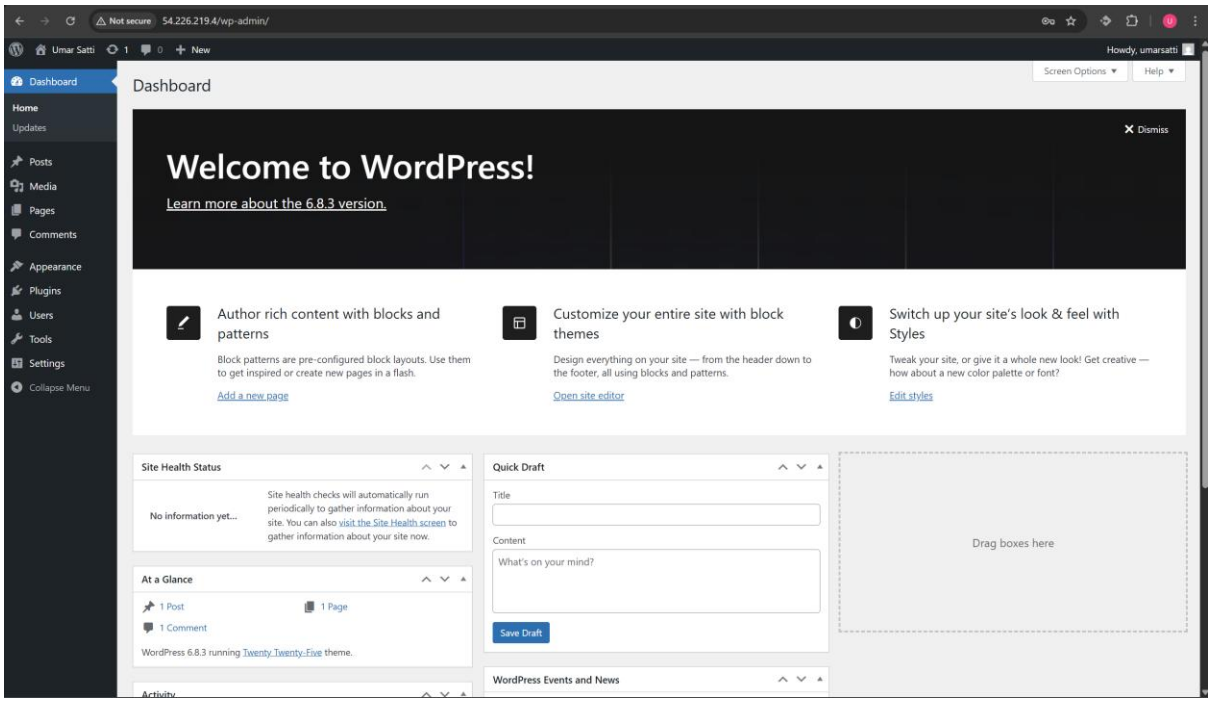


Step 3: Log In to the WordPress Admin Dashboard

1. After installation is completed, click **Log In** and use the credentials provided earlier.
2. Go to **http://<your-ec2-public-ip>/wp-login.php**
    - Example: http://13.220.176.179/wp-login.php
3. Enter the username (umarsatti) and password (P@ssw0rd).
4. Click **Log In.**



The page should automatically redirect to the WordPress Admin Dashboard, confirming that the website is now fully functional.



The AWS infrastructure, provisioned by Terraform, is now complete and verified. The deployment and initial configuration of the WordPress application were successful, meaning the application is healthy and accessible. The components defined in Terraform (VPC, Security groups, Task definition, ECR, ECS, and RDS) are used to create a live WordPress environment. This showcases how infrastructure-as-code simplifies complex deployments.

Task 1.11: Clean Up

Note: Make sure to manually delete images that were pushed to the ECR wordpress and mysql repositories by going into the ECR console before performing **terraform destroy** command.

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

Task 1.12: Troubleshooting

Issue 1: Difficulty Referencing the RDS Endpoint Correctly

Problem Description:
In a similar ECS-based project using Amazon RDS for MySQL, the RDS endpoint was not being referenced properly inside Terraform. This caused failures when ECS containers attempted to connect to the database.

Root Cause:
There was confusion about which RDS attribute exposed the actual endpoint address. Attempting to use incorrect attributes or incomplete references resulted in connection errors.

Solution:
Using the Terraform AWS provider documentation clarified that the correct attribute to expose the endpoint is **aws_db_instance.db_instance.address**. This value was then exported through an output variable, making it available to other modules such as the ECS Task Definition.

Issue 2: Incorrect Use of RDS Endpoint Inside the ECS Task Definition

Problem Description:
Even after correctly exporting the RDS endpoint, the value was incorrectly used within the ECS container environment variables. The port number (3306) was missing, causing WordPress to fail database connections.

Root Cause:
The WORDPRESS_DB_HOST environment variable must include both the RDS endpoint and the port number (default MySQL port is 3306). Using only **${var.rds_endpoint}** results in an incomplete host string.

Solution:
The environment variable was updated to append the port properly using:

- ${var.rds_endpoint}:3306

This ensured the application had the complete database host information required for a successful MySQL connection.

Conclusion

This project successfully demonstrated how to automate infrastructure provisioning and WordPress deployment on AWS using ECS Fargate and RDS with Terraform. By following this documentation, users can create the same setup to launch WordPress using AWS.