Sardar Noor Ul Hassan
Cloud Intern

Deploying a Three-Tier Application on AWS ECS (EC2) using CodePipeline & Terraform

Project Overview

This project implements a **production-grade three-tier application** (Frontend, Backend, Database) on **AWS ECS using EC2 launch type**, fully automated with **Terraform for Infrastructure as Code (IaC)** and **AWS CodePipeline for CI/CD**.

The solution includes:

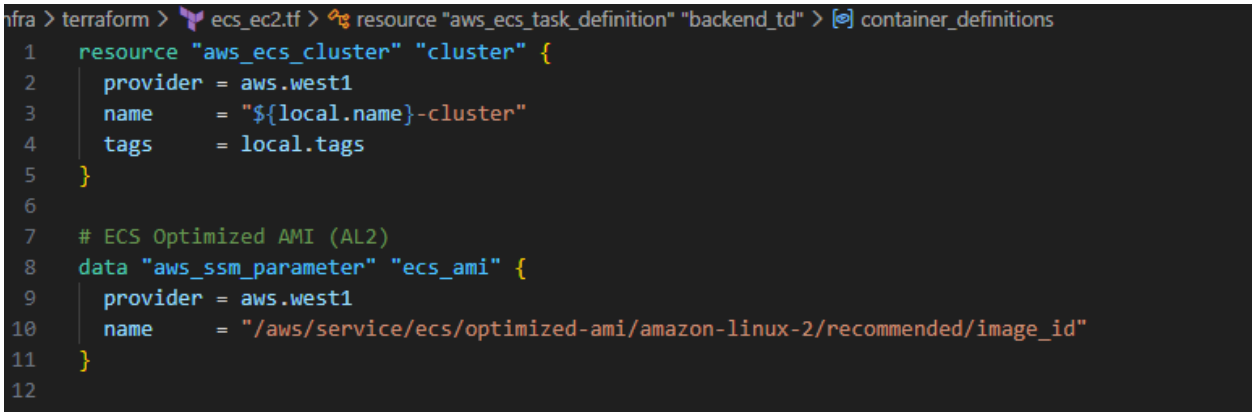
- ECS cluster backed by EC2 Auto Scaling Group
- Application Load Balancer (ALB)
- Separate ECS services for frontend and backend
- Amazon ECR for Docker images
- AWS CodePipeline + CodeBuild for CI/CD
- Amazon DocumentDB as managed database
- Secure networking using VPC, private subnets, and security groups

1. Define ECS Cluster with EC2 Launch Type in Terraform

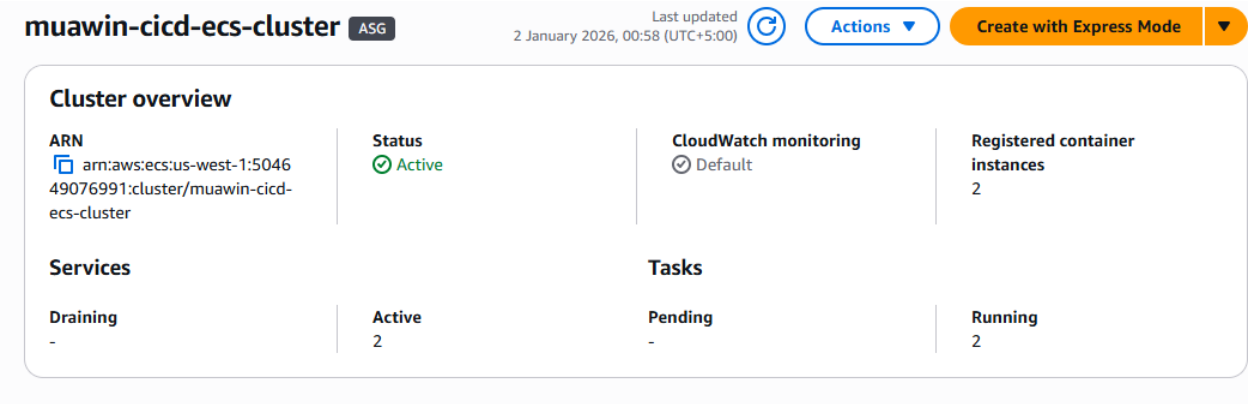
An ECS cluster was created to run containerized workloads using **EC2 launch type** instead of Fargate, allowing more control over compute capacity.

How it was implemented

ECS cluster resource was defined in Terraform.



```hcl
resource "aws_ecs_cluster_capacity_providers" "attach" {
  provider = aws.west1
  cluster_name = aws_ecs_cluster.cluster.name

  capacity_providers = [aws_ecs_capacity_provider.cp.name]

  default_capacity_provider_strategy {
    capacity_provider = aws_ecs_capacity_provider.cp.name
    weight            = 1
    base              = 1
  }
}
```



2. Configure EC2 Instances and Auto Scaling Group in Terraform

ECS-optimized EC2 instances were launched using an **Auto Scaling Group (ASG)**.

How it was implemented

ECS-optimized Amazon Linux 2 AMI fetched from SSM Parameter Store

```hcl
# ECS Optimized AMI (AL2)
data "aws_ssm_parameter" "ecs_ami" {
  provider = aws.west1
  name     = "/aws/service/ecs/optimized-ami/amazon-linux-2/recommended/image_id"
}
```

Launch Template defined with:

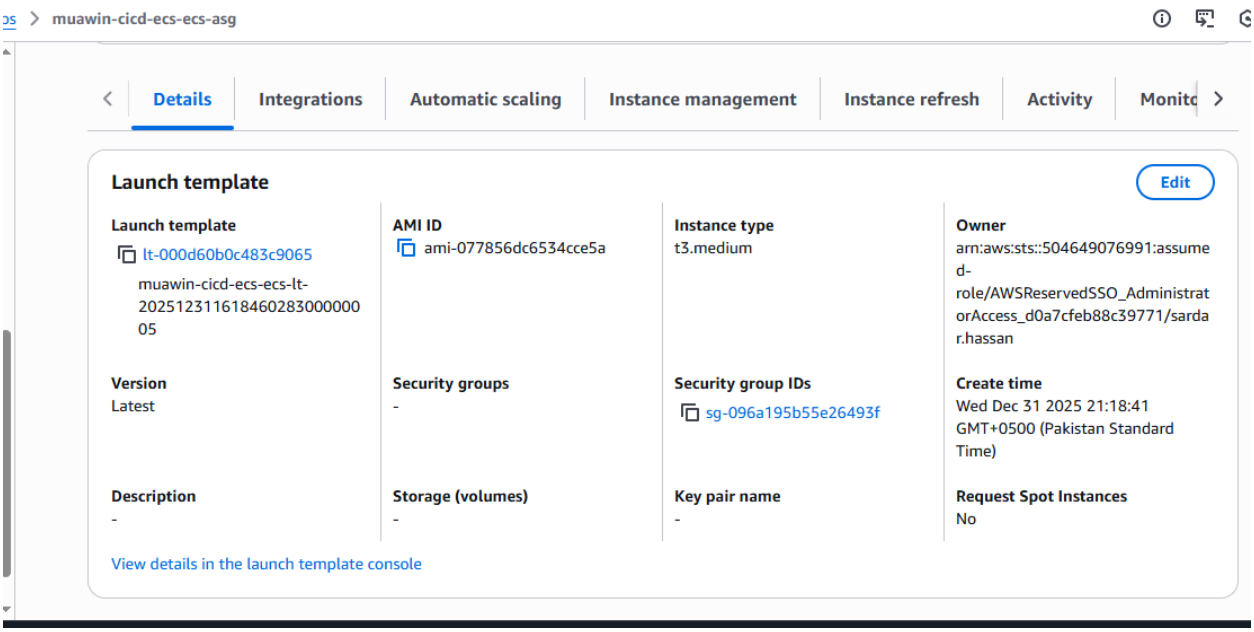
1. Instance type (t3.medium)
2. IAM Instance Profile
3. User-data to join ECS cluster

```
resource "aws_launch_template" "ecs_lt" {
  provider      = aws.west1
  name_prefix   = "${local.name}-ecs-lt-"
  image_id      = data.aws_ssm_parameter.ecs_ami.value
  instance_type = var.instance_type

  iam_instance_profile { name = aws_iam_instance_profile.ecs_instance_profile.name }
  vpc_security_group_ids = [aws_security_group.ecs_instances_sg.id]

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

Auto Scaling Group configured with:

1. Minimum, desired, and maximum capacity
2. Private subnets

```
resource "aws_autoscaling_group" "ecs_asg" {
  provider            = aws.west1
  name                = "${local.name}-ecs-asg"
  min_size            = var.asg_min
  desired_capacity    = var.asg_desired
  max_size            = var.asg_max
  vpc_zone_identifier = [for s in aws_subnet.private_west1 : s.id]

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

  tag {
    key                 = "Name"
    value               = "${local.name}-ecs-node"
    propagate_at_launch = true
  }
}
```



3. Create and Configure Security Groups for ECS and EC2 Instances

Separate security groups were created for:

1. ALB
2. ECS tasks
3. ECS EC2 instances

```
resource "aws_security_group" "alb_sg" {
  provider = aws.west1
  name     = "${local.name}-alb-sg"
  vpc_id   = aws_vpc.west1.id

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

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

```
# Security group for ECS tasks (allow from ALB)
resource "aws_security_group" "ecs_tasks_sg" {
  provider = aws.west1
  name     = "${local.name}-ecs-tasks-sg"
  vpc_id   = aws_vpc.west1.id

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

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

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

```
resource "aws_security_group" "ecs_instances_sg" {
  provider = aws.west1
  name     = "${local.name}-ecs-instances-sg"
  vpc_id   = aws_vpc.west1.id

  ingress {
  from_port   = 0
  to_port     = 0
  protocol    = "-1"
  cidr_blocks = [var.vpc_cidr_west1]
}

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

  tags = local.tags
}
```

4. Define an Application Load Balancer (ALB) and Target Groups in Terraform

A public Application Load Balancer was created to route traffic.

How it was implemented

ALB placed in public subnets

```
resource "aws_lb" "alb" {
  provider           = aws.west1
  name               = "${local.name}-alb"
  load_balancer_type = "application"
  security_groups    = [aws_security_group.alb_sg.id]
  subnets            = [for s in aws_subnet.public_west1 : s.id]
  tags               = local.tags
}
```

Two target groups:

Frontend target group (port 80)

```
resource "aws_lb_target_group" "frontend_tg" {
  provider    = aws.west1
  name        = "${local.name}-frontend-tg"
  port        = var.frontend_port
  protocol    = "HTTP"
  vpc_id      = aws_vpc.west1.id
  target_type = "ip"

  health_check {
    path = "/"
    matcher = "200-399"
  }

  tags = local.tags
}
```

Backend target group (port 5000)

```
resource "aws_lb_target_group" "backend_tg" {
  provider    = aws.west1
  name        = "${local.name}-backend-tg"
  port        = var.backend_port
  protocol    = "HTTP"
  vpc_id      = aws_vpc.west1.id
  target_type = "ip"

  health_check {
    path = "/api"
    matcher = "200-399"
  }

  tags = local.tags
```

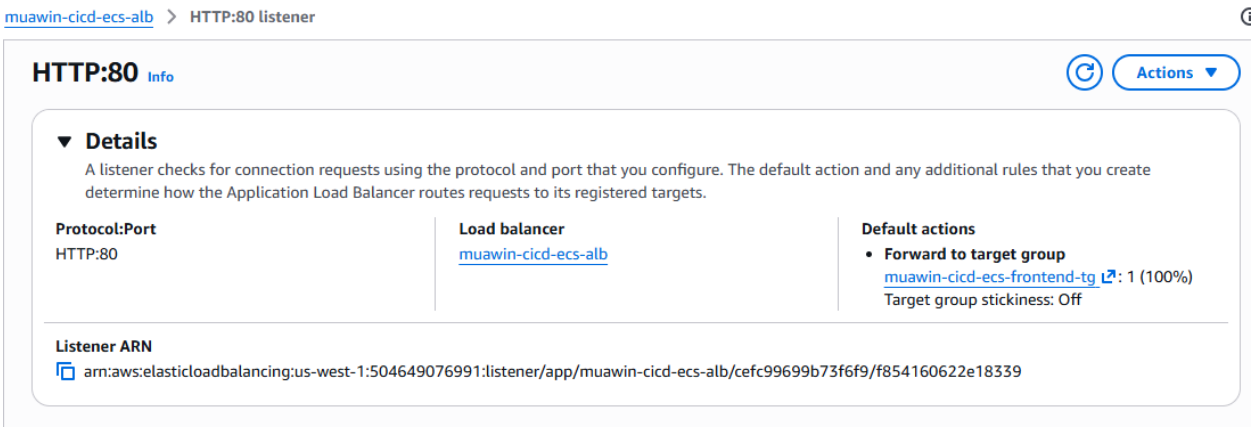
Listener rules configured:

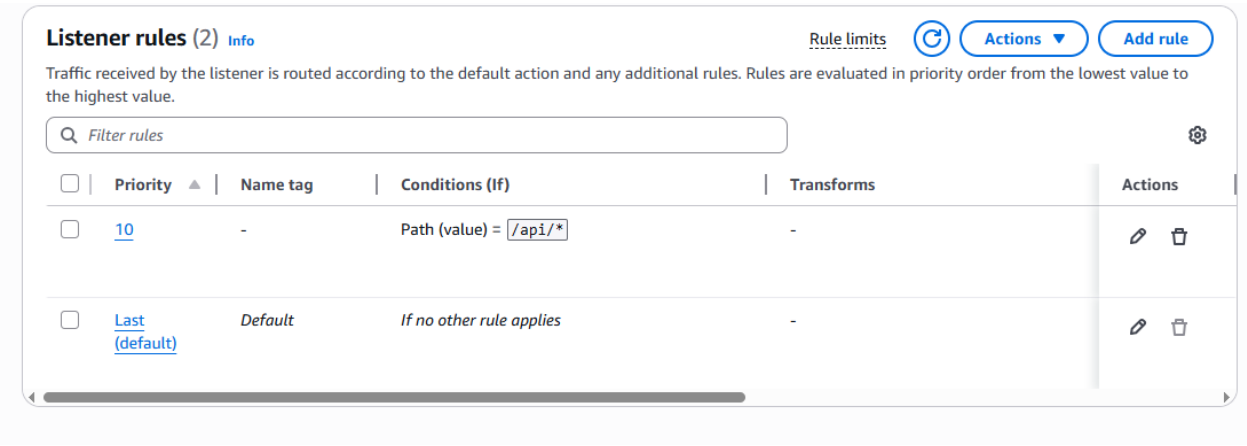
/api/* → backend

```
resource "aws_lb_listener_rule" "api_rule" {
  provider     = aws.west1
  listener_arn = aws_lb_listener.http.arn
  priority     = 10

  action {
    type             = "forward"
    target_group_arn = aws_lb_target_group.backend_tg.arn
  }

  condition {
    path_pattern { values = ["/api/*"] }
  }
}
```

/ → frontend

```
resource "aws_lb_listener" "http" {
  provider          = aws.west1
  load_balancer_arn = aws_lb.alb.arn
  port              = 80
  protocol          = "HTTP"

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



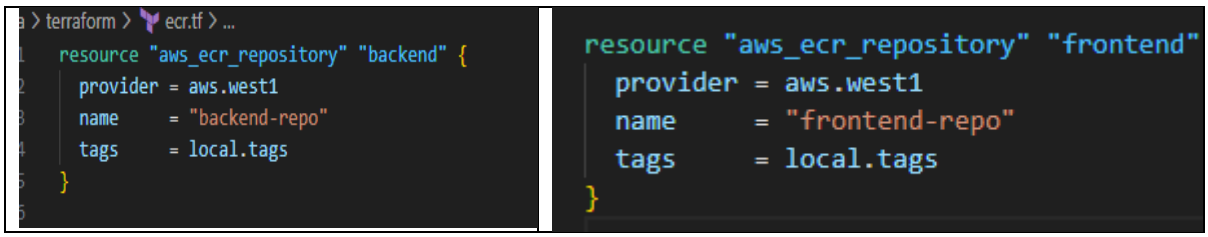


5. Create ECR Repository to Store Docker Images

Amazon ECR repositories were created for frontend and backend images.

How it was implemented

Separate ECR repos defined via Terraform



1. Repositories were used by CodeBuild to push images
2. ECS pulls images from ECR at runtime

○	backend-repo	504649076991.dkr.ecr.us-west-1.amazonaws.com/backend-repo	31 December 2025, 21:18:15 (UTC+05)	Mutable	AES-256
○	frontend-repo	504649076991.dkr.ecr.us-west-1.amazonaws.com/frontend-repo	31 December 2025, 21:18:15 (UTC+05)	Mutable	AES-256

6. Write Terraform Configuration for ECS Task Definition Referencing ECR Image

Task definitions were created for frontend and backend services.

How it was implemented

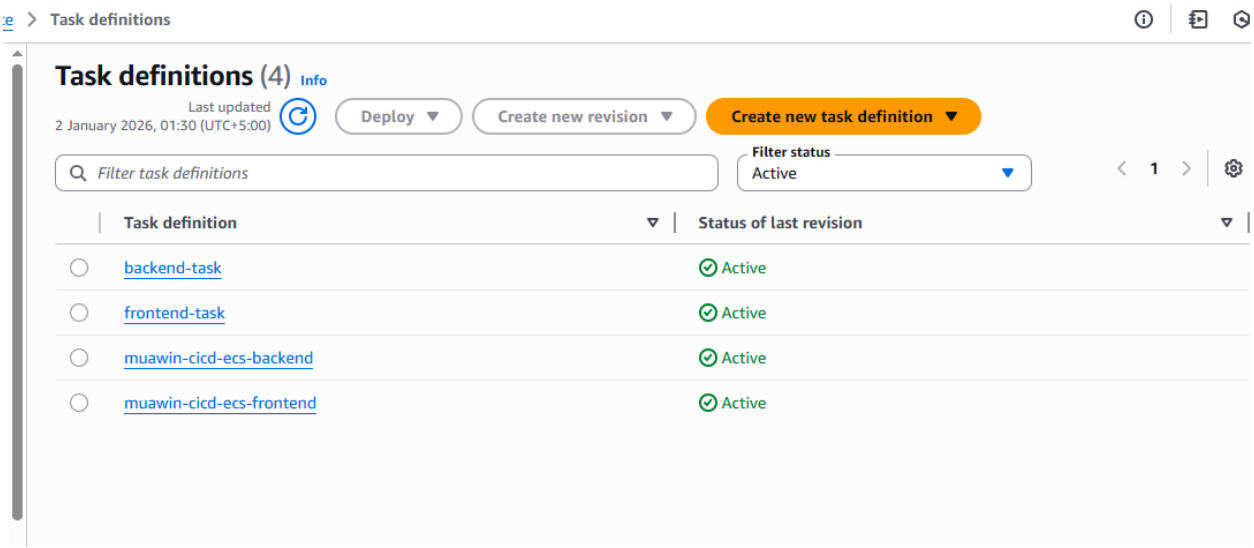
Task definitions specify:

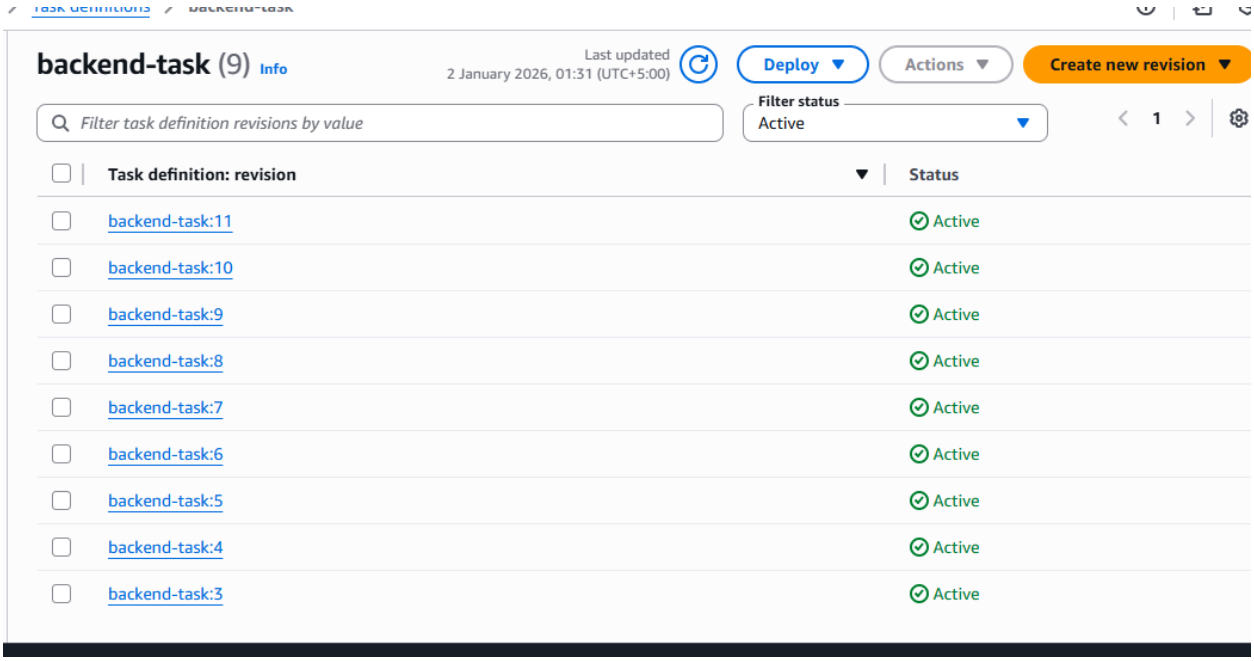
1. Container image from ECR
2. CPU and memory
3. Port mappings
4. Environment variables (e.g., MONGODB_URI)
5. CloudWatch logging

```
resource "aws_ecs_task_definition" "backend_td" {
  provider                 = aws.west1
  family                   = "${local.name}-backend"
  network_mode             = "awsvpc"
  requires_compatibilities = ["EC2"]
  cpu                      = "256"
  memory                   = "512"
  execution_role_arn       = aws_iam_role.ecs_task_execution.arn

  container_definitions = jsonencode([
    {
      name  = "backend"
      image = "${var.account_id}.dkr.ecr.${var.region_compute}.amazonaws.com/backend-repo:latest"
      essential = true
      portMappings = [{ containerPort = var.backend_port, protocol = "tcp" }]

     environment = [
  { name = "PORT",       value = tostring(var.backend_port) },
  { name = "NODE_ENV",   value = "production" },

  # app reads this
  { name = "MONGODB_URI", value = var.mongodb_uri },


  { name = "DOCDB_USER", value = var.docdb_username },
  { name = "DOCDB_PASS", value = var.docdb_password }
]
```

```
    logConfiguration = {
      logDriver = "awslogs",
      options = {
        awslogs-group         = aws_cloudwatch_log_group.backend_lg.name,
        awslogs-region        = var.region_compute,
        awslogs-stream-prefix = "ecs"
      }
    }
  }
)
```

```hcl
resource "aws_ecs_task_definition" "frontend_td" {
  provider                 = aws.west1
  family                   = "${local.name}-frontend"
  network_mode             = "awsvpc"
  requires_compatibilities = ["EC2"]
  cpu                      = "256"
  memory                   = "512"
  execution_role_arn       = aws_iam_role.ecs_task_execution.arn

  container_definitions = jsonencode([
    {
      name  = "frontend"
      image = "${var.account_id}.dkr.ecr.${var.region_compute}.amazonaws.com/frontend-repo:latest"
      essential = true
      portMappings = [{ containerPort = 80, protocol = "tcp" }]

      logConfiguration = {
        logDriver = "awslogs",
        options = {
          awslogs-group         = aws_cloudwatch_log_group.frontend_lg.name,
          awslogs-region        = var.region_compute,
          awslogs-stream-prefix = "ecs"
        }
      }
    }
  ])
}
```





7. Define ECS Service Linked to the ECS Cluster and ALB in Terraform

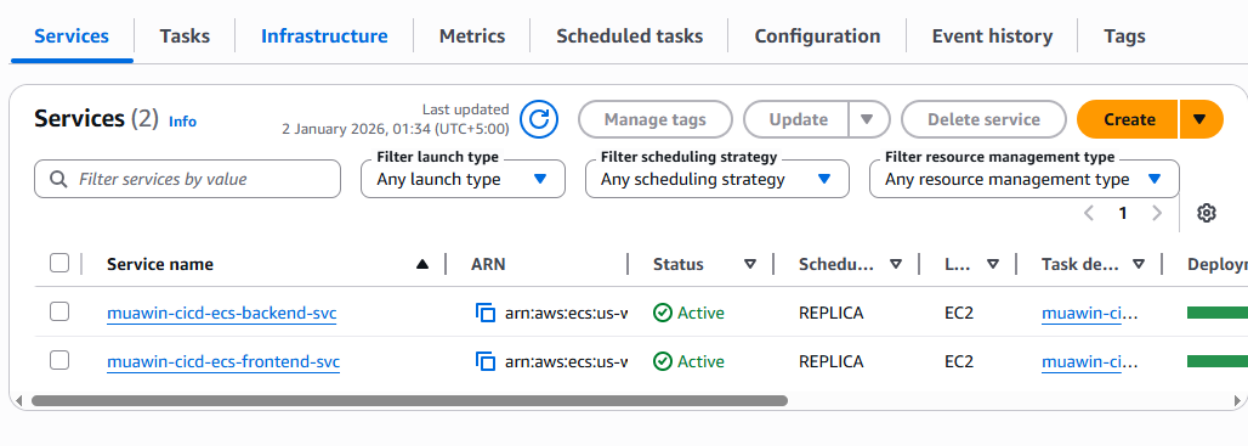
Separate ECS services were created for frontend and backend.

How it was implemented

1. Services reference:
    a. ECS cluster
    b. Task definition
    c. Target group
2. Desired count set to 1 (can scale later)
3. Network mode awsvpc used

```
# ECS services
resource "aws_ecs_service" "backend_svc" {
  provider        = aws.west1
  name            = "${local.name}-backend-svc"
  cluster         = aws_ecs_cluster.cluster.id
  task_definition = aws_ecs_task_definition.backend_td.arn
  desired_count   = 1
  launch_type     = "EC2"

  network_configuration {
    subnets         = [for s in aws_subnet.private_west1 : s.id]
    security_groups = [aws_security_group.ecs_tasks_sg.id]
  }

  load_balancer {
    target_group_arn = aws_lb_target_group.backend_tg.arn
    container_name   = "backend"
    container_port   = var.backend_port
  }

  depends_on = [aws_lb_listener_rule.api_rule]
}
```

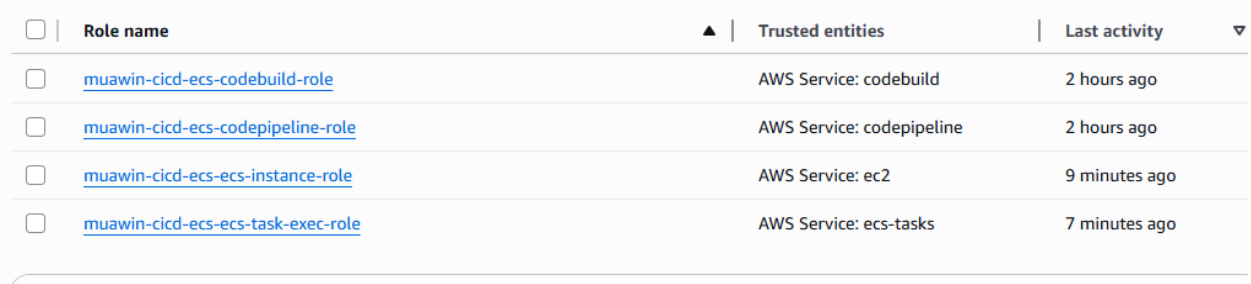
```hcl
resource "aws_ecs_service" "frontend_svc" {
  provider        = aws.west1
  name            = "${local.name}-frontend-svc"
  cluster         = aws_ecs_cluster.cluster.id
  task_definition = aws_ecs_task_definition.frontend_td.arn
  desired_count   = 1
  launch_type     = "EC2"

  network_configuration {
    subnets         = [for s in aws_subnet.private_west1 : s.id]
    security_groups = [aws_security_group.ecs_tasks_sg.id]
  }

  load_balancer {
    target_group_arn = aws_lb_target_group.frontend_tg.arn
    container_name   = "frontend"
    container_port   = 80
  }

  depends_on = [aws_lb_listener.http]
}
```



8. Create IAM Roles and Policies for ECS, CodeBuild, and CodePipeline

IAM roles were created for:

1. ECS task execution
2. ECS EC2 instances
3. CodeBuild
4. CodePipeline



Role name	Trusted entities	Last activity
muawin-cicd-ecs-codebuild-role	AWS Service: codebuild	2 hours ago
muawin-cicd-ecs-codepipeline-role	AWS Service: codepipeline	2 hours ago
muawin-cicd-ecs-ecs-instance-role	AWS Service: ec2	9 minutes ago
muawin-cicd-ecs-ecs-task-exec-role	AWS Service: ecs-tasks	7 minutes ago

9. Configure CodeBuild Project in Terraform for Building and Pushing Docker Images

A CodeBuild project was configured to build Docker images.

```hcl
resource "aws_codebuild_project" "build" {
  provider = aws.west1
  name     = "${local.name}-build"
  service_role = aws_iam_role.codebuild_role.arn

  artifacts { type = "CODEPIPELINE" }

  environment {
    compute_type                = "BUILD_GENERAL1_MEDIUM"
    image                       = "aws/codebuild/standard:7.0"
    type                        = "LINUX_CONTAINER"
    privileged_mode             = true
    environment_variable {
      name  = "AWS_REGION"
      value = var.region_compute
    }
  }

  source {
    type      = "CODEPIPELINE"
    buildspec = "buildspec.yml"
  }

  tags = local.tags
}
```

How it was implemented

1. Buildspec file defines:
    a. Login to ECR

```yaml
phases:
  pre_build:
    commands:
      - echo "Login to ECR"
      - aws --version
      - ACCOUNT_ID=$(aws sts get-caller-identity --query Account --output text)
      - aws ecr get-login-password --region $AWS_REGION | docker login --username AWS --password-stdin ${ACCOUNT_ID}.dkr.ecr.${AWS_
      - COMMIT_HASH=$(echo $CODEBUILD_RESOLVED_SOURCE_VERSION | cut -c 1-7)
      - IMAGE_TAG=${COMMIT_HASH:-latest}

      # Downloads DocumentDB CA bundle for TLS connection
      - mkdir -p backend/certs
      - curl -sS -o backend/certs/global-bundle.pem https://truststore.pki.rds.amazonaws.com/global/global-bundle.pem
```

    b. Build Docker images

```
build:
  commands:
    - echo "Build backend image"
    - docker build -t backend:$IMAGE_TAG ./backend
    - docker tag backend:$IMAGE_TAG ${ACCOUNT_ID}.dkr.ecr.${AWS_REGION}.amazonaws.com/backend-repo:$IMAGE_TAG

    - echo "Build frontend image"
    - docker build -t frontend:$IMAGE_TAG ./frontend
    - docker tag frontend:$IMAGE_TAG ${ACCOUNT_ID}.dkr.ecr.${AWS_REGION}.amazonaws.com/frontend-repo:$IMAGE_TAG
```

c. Push images to ECR

```
post_build:
  commands:
    - echo "Push images"
    - docker push ${ACCOUNT_ID}.dkr.ecr.${AWS_REGION}.amazonaws.com/backend-repo:$IMAGE_TAG
    - docker push ${ACCOUNT_ID}.dkr.ecr.${AWS_REGION}.amazonaws.com/frontend-repo:$IMAGE_TAG
```

10. Set Up CodePipeline in Terraform with Source, Build, and Deploy Stages

A complete CI/CD pipeline was created.

How it was implemented

Stages:

Source *(GitHub via CodeStar connection)*

```
resource "aws_codepipeline" "pipeline" {
  provider = aws.west1
  name     = "${local.name}-pipeline"
  role_arn = aws_iam_role.codepipeline_role.arn

  artifact_store {
    location = aws_s3_bucket.artifacts.bucket
    type     = "S3"
  }

  stage {
    name = "Source"
    action {
      name             = "Source"
      category         = "Source"
      owner            = "AWS"
      provider         = "CodeStarSourceConnection"
      version          = "1"
      output_artifacts = ["source_output"]
      configuration = {
        ConnectionArn    =  var.github_repo_full_name string
        FullRepositoryId = var.github_repo_full_name
        BranchName       = var.github_branch
      }
    }
  }
```

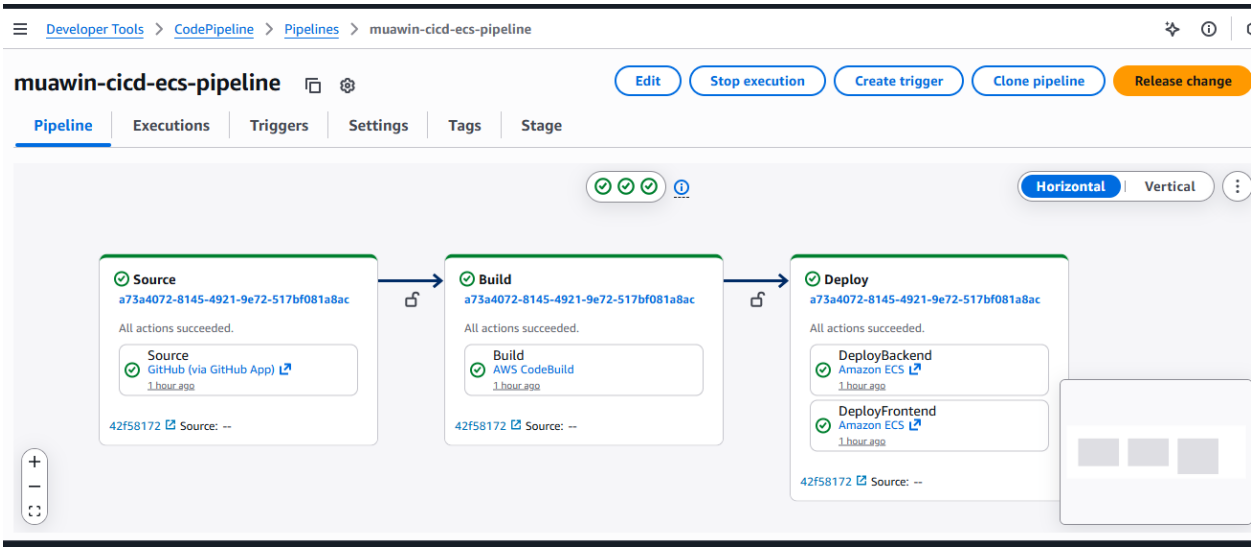
Build *(CodeBuild project)*

```
stage {
  name = "Build"
  action {
    name             = "Build"
    category         = "Build"
    owner            = "AWS"
    provider         = "CodeBuild"
    version          = "1"
    input_artifacts  = ["source_output"]
    output_artifacts = ["build_output"]
    configuration = {
      ProjectName = aws_codebuild_project.build.name
    }
  }
}
```

Deploy *(ECS service update)*

```
stage {
  name = "Deploy"
  action {
    name            = "DeployBackend"
    category        = "Deploy"
    owner           = "AWS"
    provider        = "ECS"
    version         = "1"
    input_artifacts = ["build_output"]
    configuration = {
      ClusterName = aws_ecs_cluster.cluster.name
      ServiceName = aws_ecs_service.backend_svc.name
      FileName    = "imagedefinitions-backend.json"
    }
  }
```

```
  action {
    name             = "DeployFrontend"
    category         = "Deploy"
    owner            = "AWS"
    provider         = "ECS"
    version          = "1"
    input_artifacts = ["build_output"]
    configuration = {
      ClusterName = aws_ecs_cluster.cluster.name
      ServiceName = aws_ecs_service.frontend_svc.name
      FileName    = "imagedefinitions-frontend.json"
    }
  }
}

tags = local.tags
```



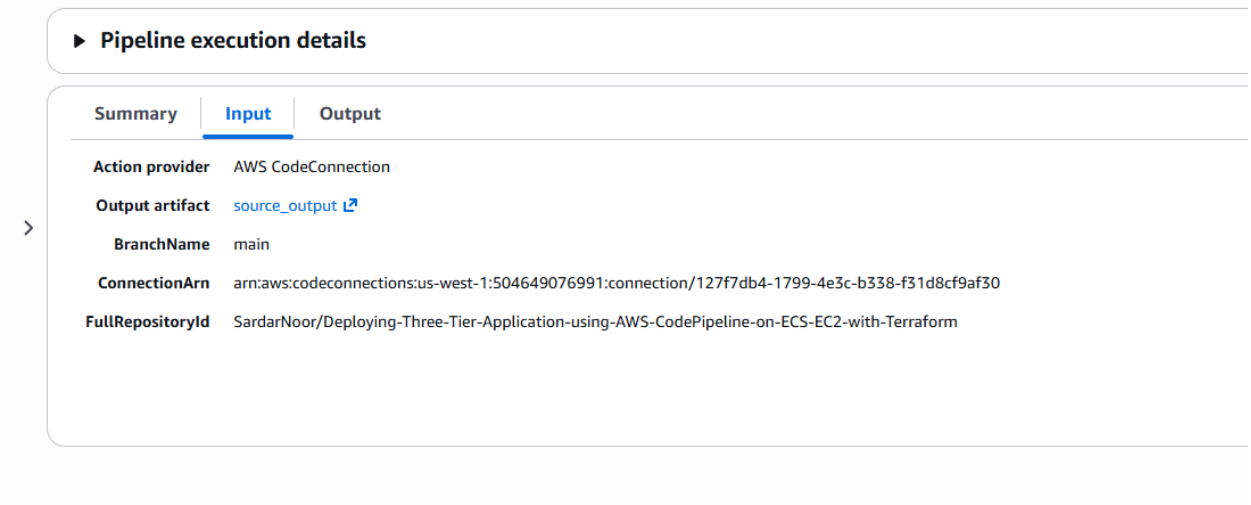
11. Connect Source Stage to Code Repository (GitHub)

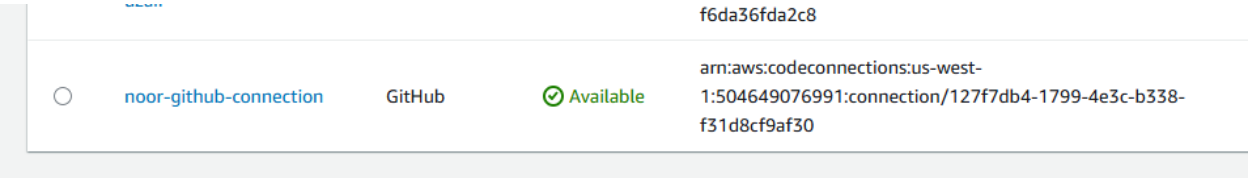
GitHub repository was connected securely.

How it was implemented

1. AWS CodeStar connection created

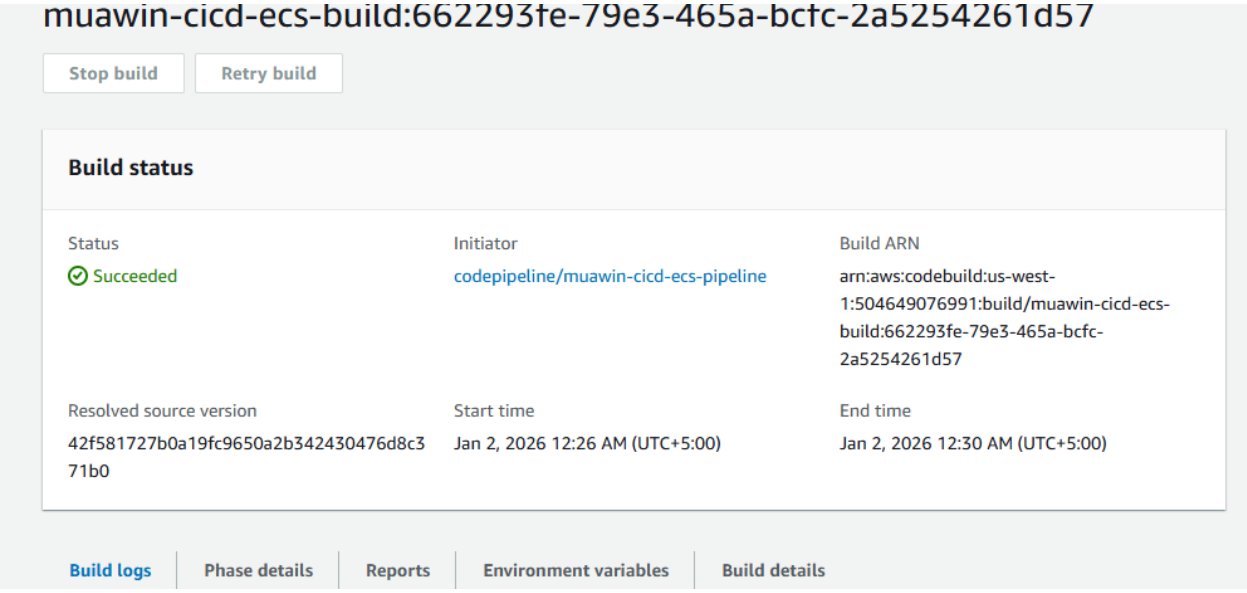
2. No personal access tokens used
3. Pipeline listens to repository changes

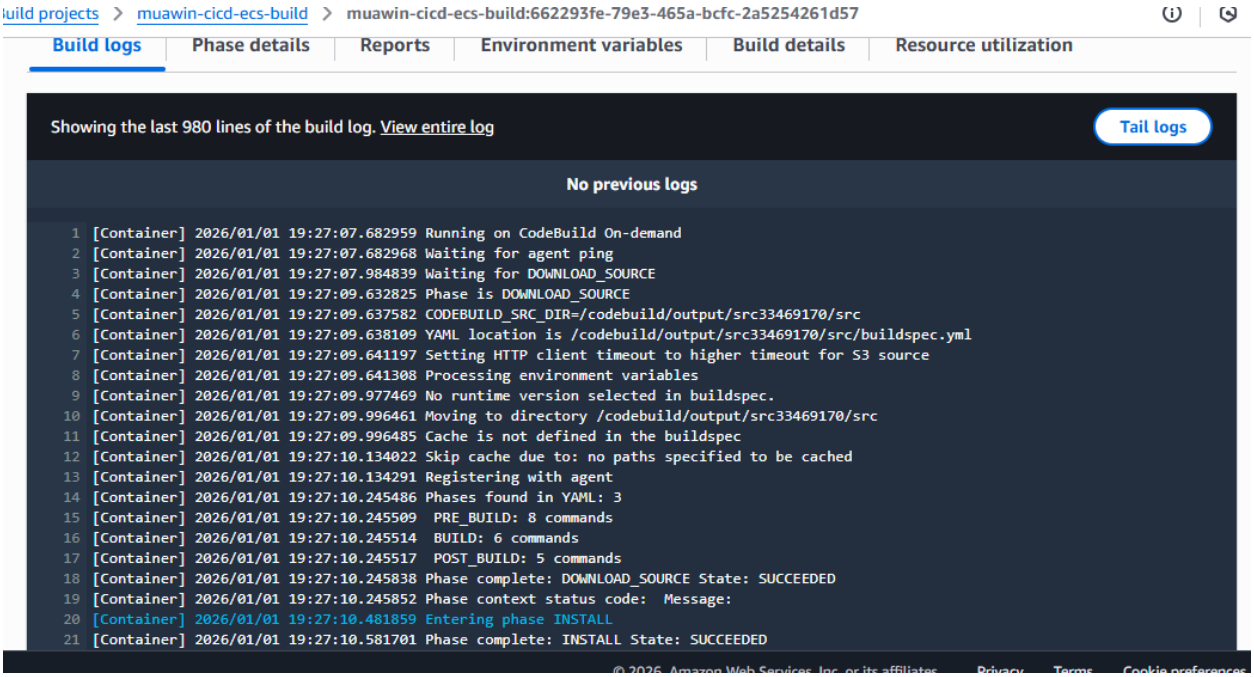




12. Configure Build Stage to Use CodeBuild Project for Docker Image Build

Pipeline invokes CodeBuild automatically.



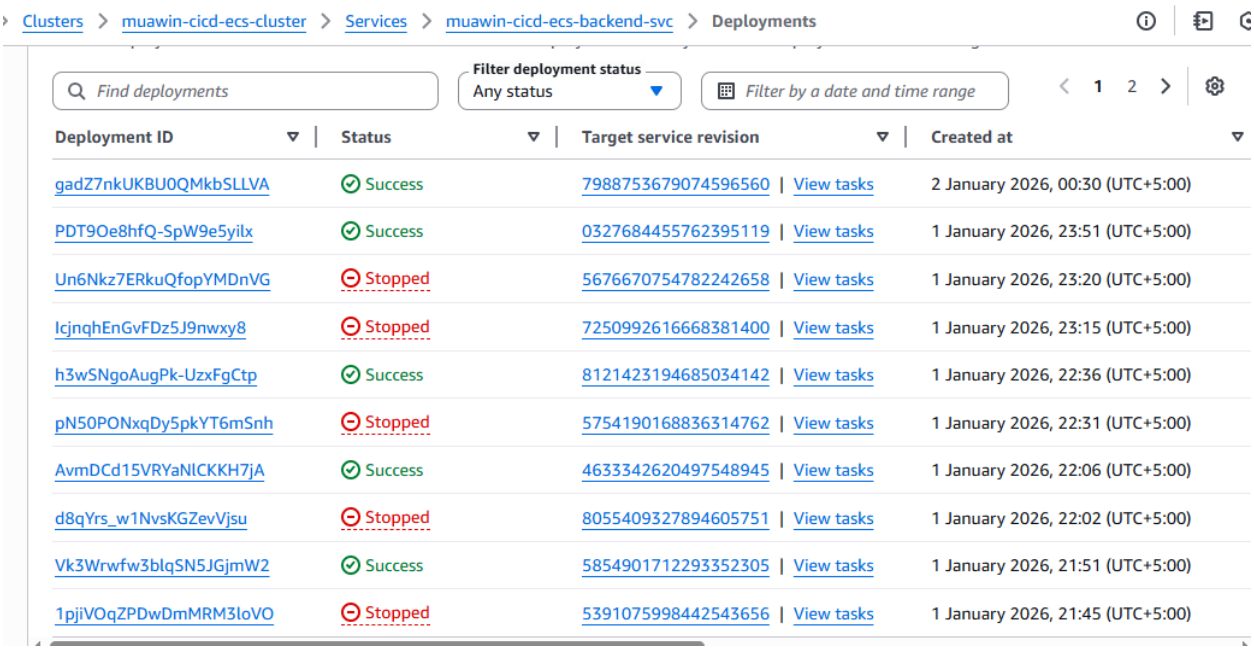


13. Configure Deploy Stage to Update ECS Service with New Task Definition

Deploy stage updates ECS service with new image.

How it was implemented

1. ECS deployment action configured
2. New task definition revision registered
3. ECS service performs rolling update



14. Apply Terraform Scripts to Provision All Resources

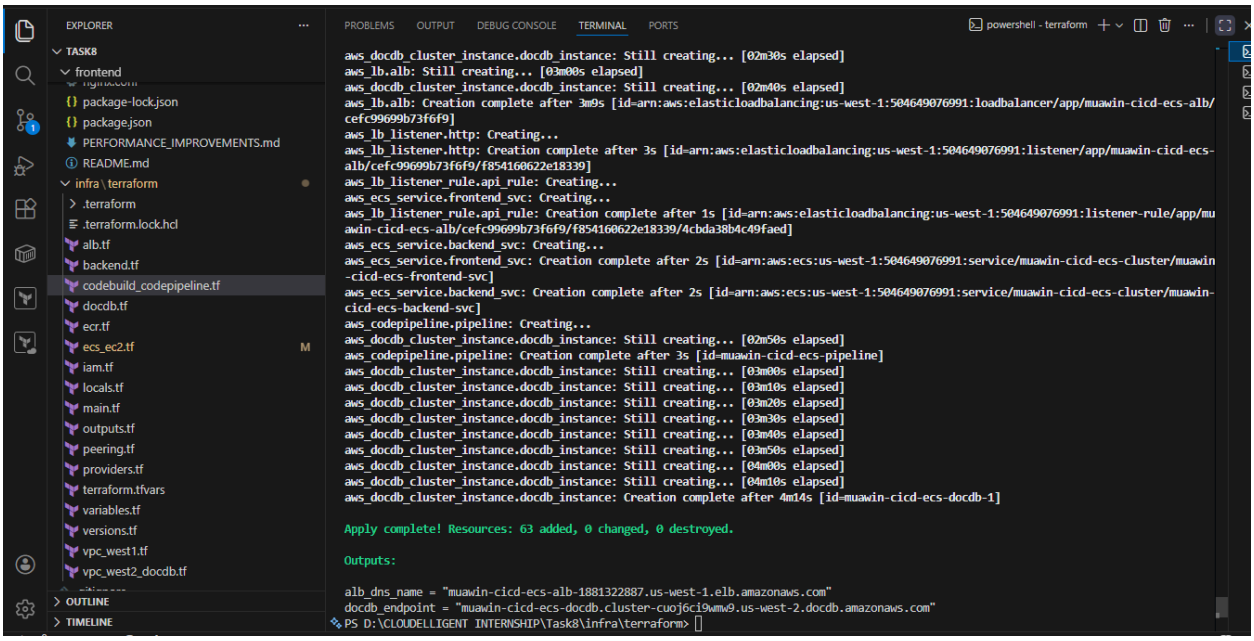
Infrastructure was provisioned using Terraform.

How it was implemented

terraform init

terraform plan

terraform apply

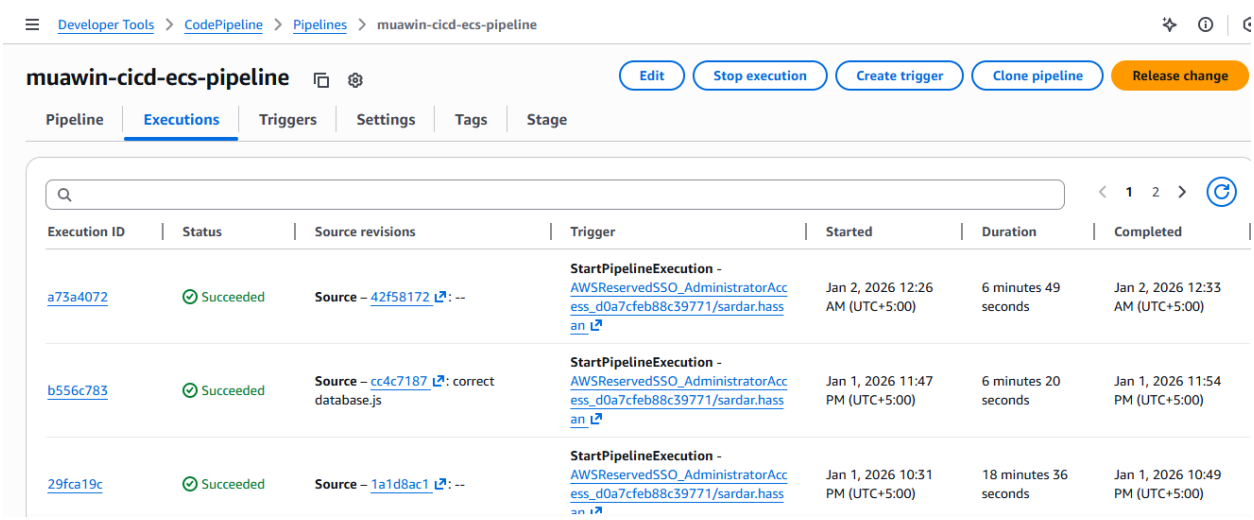


15. Trigger CodePipeline and Monitor Deployment Progress

Pipeline was triggered via code push.

How it was implemented

```
PS D:\CLOUDELLIGENT INTERNSHIP\Task8> git add .
PS D:\CLOUDELLIGENT INTERNSHIP\Task8> git commit -m "--"
[main 42f5817] --
 1 file changed, 1 insertion(+), 16 deletions(-)
PS D:\CLOUDELLIGENT INTERNSHIP\Task8> git push origin main
Enumerating objects: 7, done.
Counting objects: 100% (7/7), done.
Delta compression using up to 4 threads
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 473 bytes | 94.00 KiB/s, done.
Total 4 (delta 3), reused 0 (delta 0), pack-reused 0 (from 0)
remote: Resolving deltas: 100% (3/3), completed with 3 local objects.
To https://github.com/SardarNoor/Deploying-Three-Tier-Application-using-AWS-CodePipeline-on-ECS-EC2-with-Terraform.git
   cc4c718..42f5817  main -> main
PS D:\CLOUDELLIGENT INTERNSHIP\Task8>
```

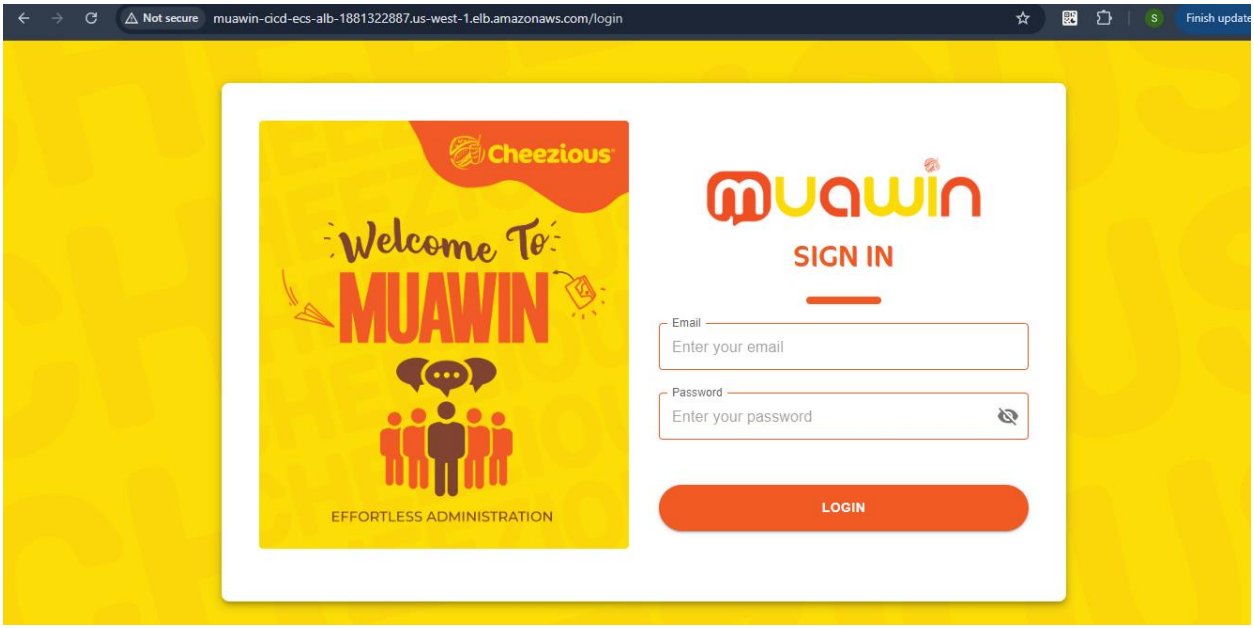


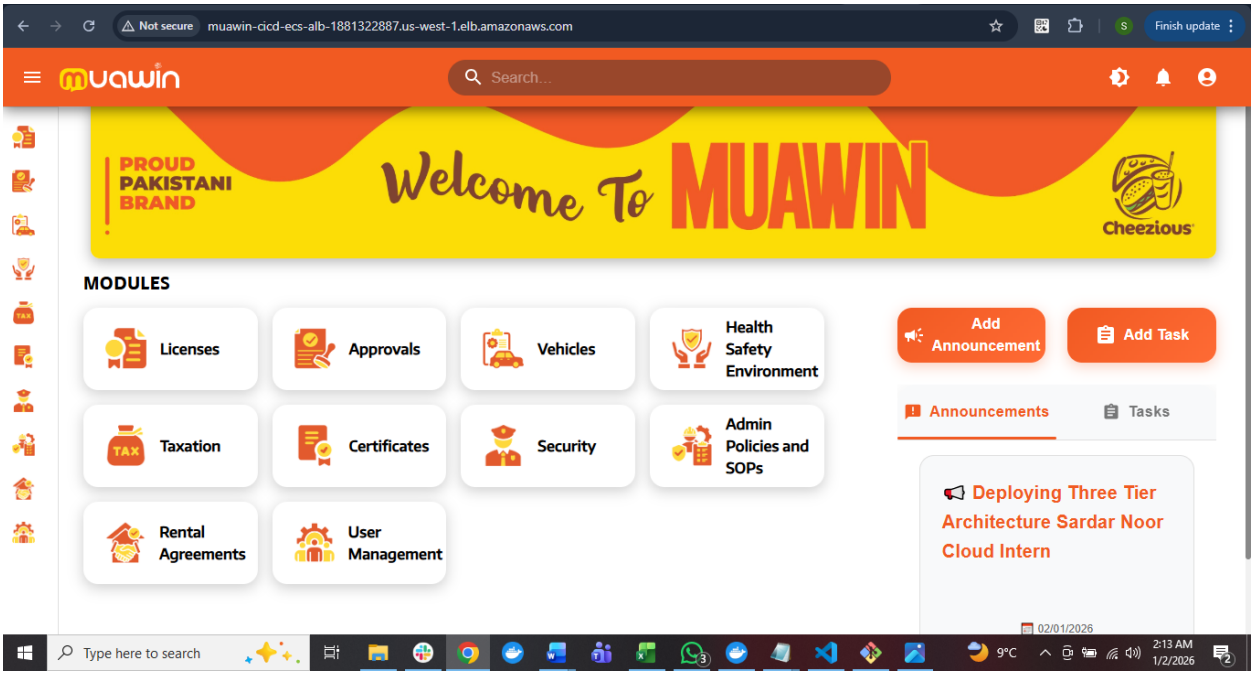
16. Validate Application Is Accessible Through the ALB After Deployment

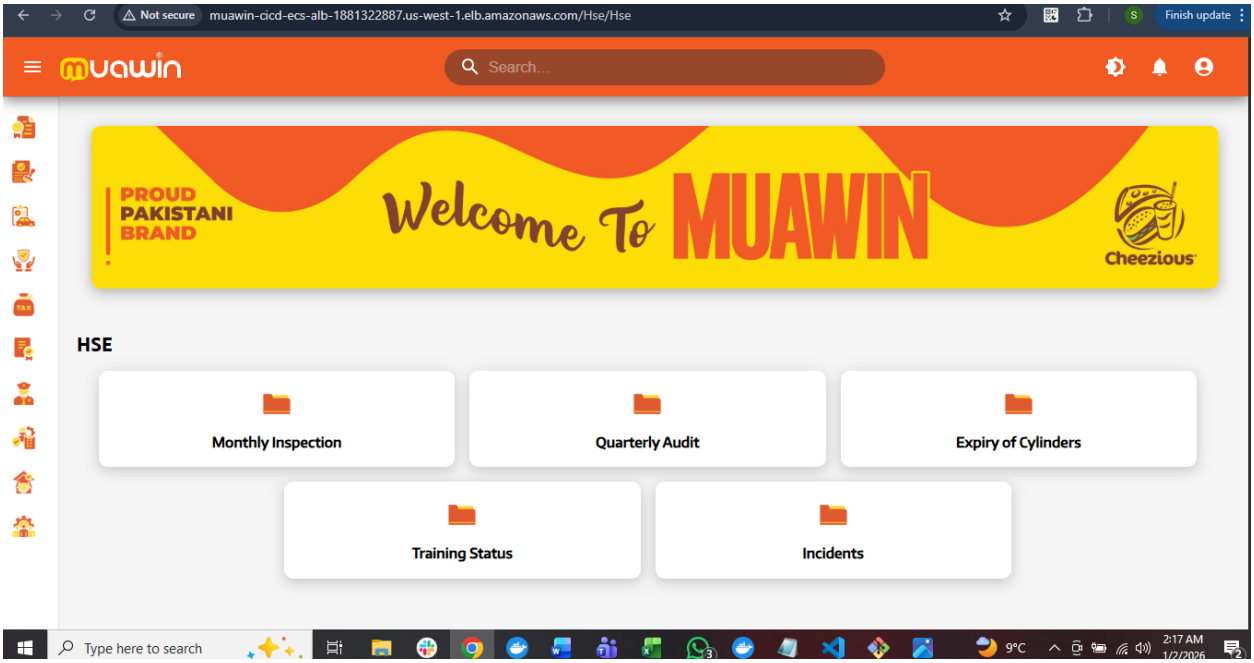
Application was tested using ALB DNS.

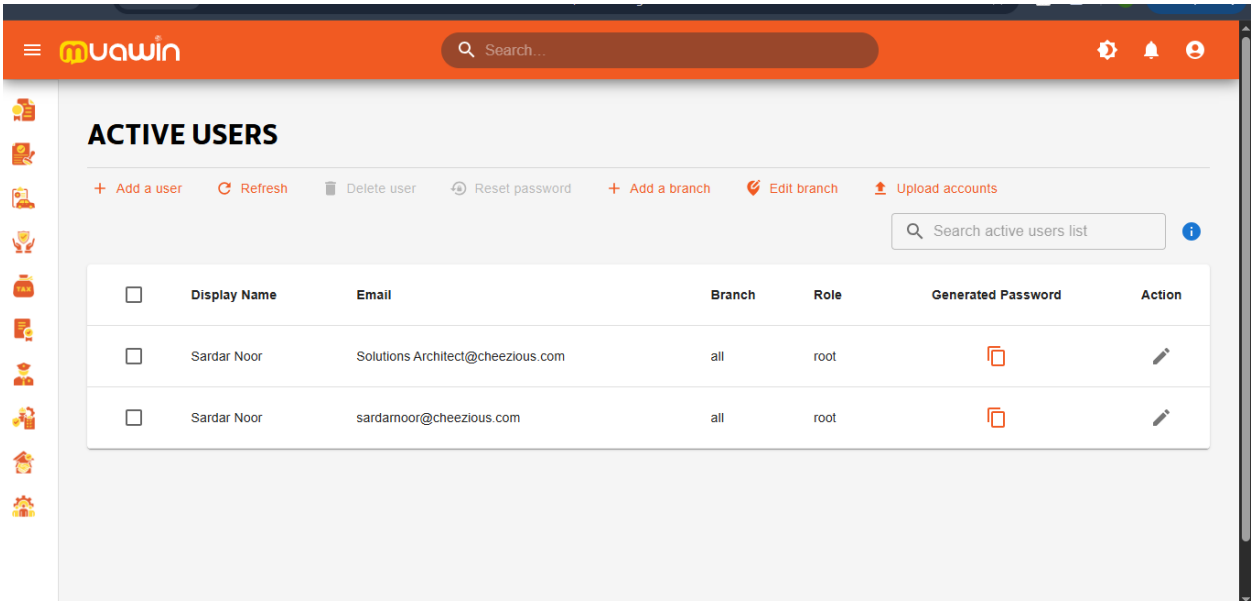
How it was implemented

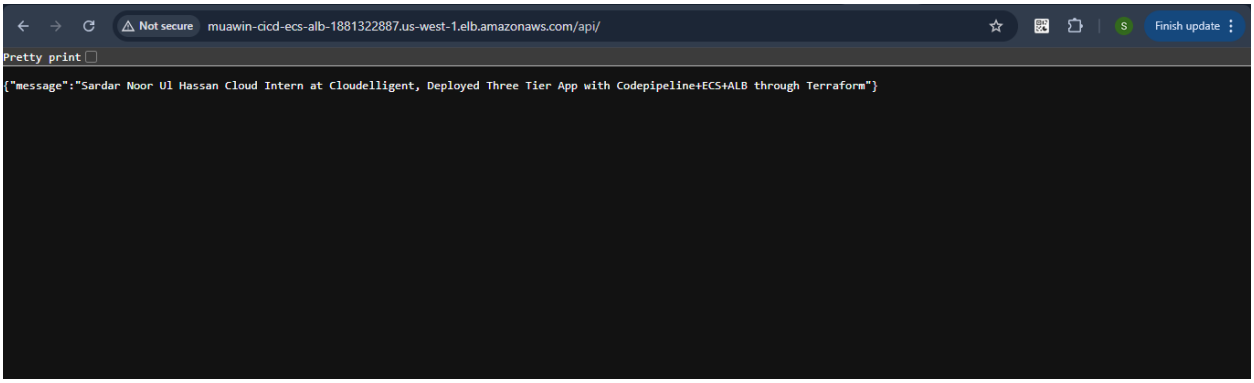
- Open ALB DNS in browser

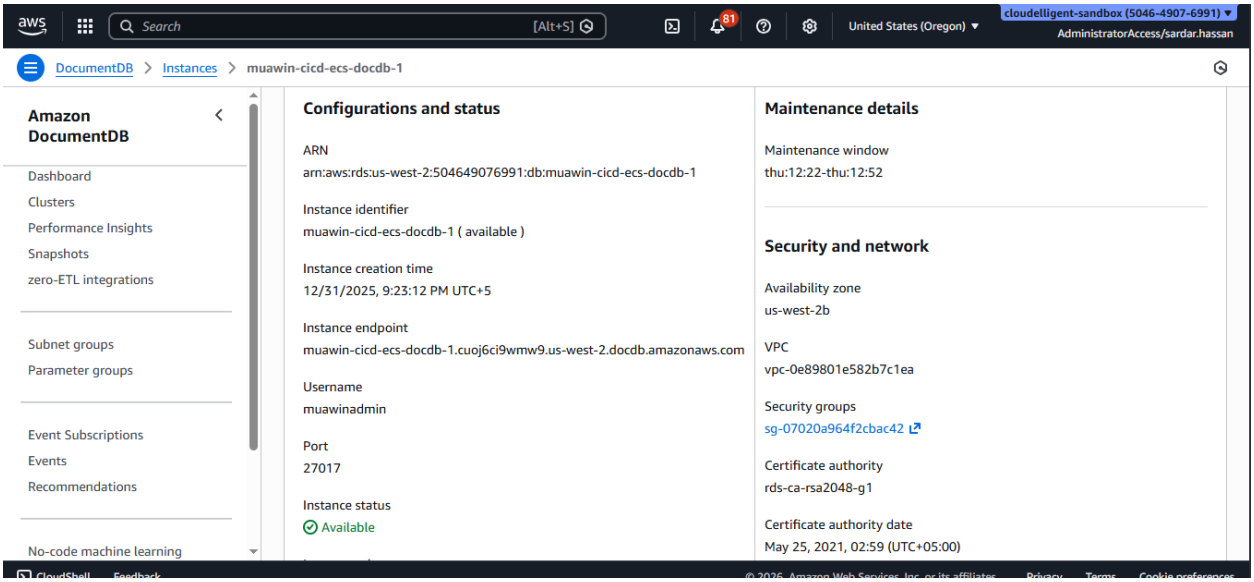

Not secure
muawin-cicd-ecs-alb-1881322887.us-west-1.elb.amazonaws.com/login
Finish update
Cheezious
Welcome To
MUAWIN
EFFORTLESS ADMINISTRATION
muawin
SIGN IN
Email
Enter your email
Password
Enter your password
LOGIN


Not secure
muawin-cicd-ecs-alb-1881322887.us-west-1.elb.amazonaws.com
Finish update
muawin
Search...
PROUD PAKISTANI BRAND
Welcome To MUAWIN
Cheezious
MODULES
Licenses
Approvals
Vehicles
Health Safety Environment
Taxation
Certificates
Security
Admin Policies and SOPs
Rental Agreements
User Management
Add Announcement
Add Task
Announcements
Tasks
Deploying Three Tier Architecture Sardar Noor Cloud Intern
02/01/2026
Type here to search
9°C
2:13 AM
1/2/2026


Not secure
muawin-cicd-ecs-alb-1881322887.us-west-1.elb.amazonaws.com/Hse/Hse
Finish update
muawin
Search...
PROUD PAKISTANI BRAND
Welcome To MUAWIN
Cheezious
HSE
Monthly Inspection
Quarterly Audit
Expiry of Cylinders
Training Status
Incidents
Type here to search
9°C
2:17 AM
1/2/2026


muawin
Search...
ACTIVE USERS
Add a user
Refresh
Delete user
Reset password
Add a branch
Edit branch
Upload accounts
Search active users list
Display Name
Email
Branch
Role
Generated Password
Action
Sardar Noor
Solutions Architect@cheezious.com
all
root
Sardar Noor
sardarnoor@cheezious.com
all
root





Problems Faced and How They Were Resolved

During implementation, multiple technical challenges were encountered across infrastructure, CI/CD, and application layers.

1. ECS Tasks Failing to Start (Backend Service)

Problem

The backend ECS service repeatedly showed:

1. Task failed to start
2. ECS tasks entering **STOPPED** state immediately after launch
3. ALB returning **503 Service Temporarily Unavailable**

Root Cause

Initial ECS task definitions were missing correct environment variable injection. Additionally, backend application code was falling back to localhost:27017 due to incorrect MongoDB connection handling.

Resolution

- Inspected ECS **Task Definition → Environment Variables** to verify runtime configuration
- Ensured MONGODB_URI was correctly injected via Terraform

2. MongoDB Connection Errors with Amazon DocumentDB

Problem

Backend logs consistently showed:

ECONNREFUSED 127.0.0.1:27017

even though Amazon DocumentDB cluster was running and accessible.

Root Cause

Amazon DocumentDB requires:

1. TLS enabled
2. CA certificate
3. retryWrites=false

These were missing in the Mongoose connection configuration.

Resolution

Updated database connection logic to include:

1. tls: true
2. tlsCAFile: /app/certs/global-bundle.pem
3. retryWrites: false

3. CodePipeline Source Stage Permission Failure

Problem

CodePipeline failed at Source stage with:

Unable to use Connection: Insufficient permissions

Root Cause

The CodePipeline IAM role lacked permission to use the CodeStar GitHub connection.

Resolution

Added the permission of **codestar-connections:UseConnection** and re-applied terraform.

4.Multi-Region Architecture Complexity (DocumentDB)

Problem

Amazon DocumentDB was not available in the primary region (us-west-1) where ECS was deployed.

Resolution

1. Deployed DocumentDB in a supported region
2. Configured VPC peering between regions
3. Updated security groups and routing to allow cross-region communication

Conclusion

This project successfully demonstrated the design and implementation of a production-grade three-tier application deployment on AWS, using Terraform for infrastructure automation and CodePipeline for continuous delivery.