AWS ECS Fargate Infrastructure with Terraform and GitHub Actions

Table of Contents

Overview

This project implements a production-grade infrastructure for deploying containerized Node.js applications on AWS ECS Fargate. The infrastructure is fully automated using Terraform and integrates with GitHub Actions for continuous deployment using OIDC authentication.

Key Components:

- Multi-AZ VPC with public and private subnets
- Application Load Balancer for traffic distribution
- ECS Fargate for serverless container orchestration
- ECR for container image storage
- GitHub OIDC for secure, keyless CI/CD authentication
- IAM roles with least privilege access

Prerequisites

AWS Account Configuration:

- Active AWS account with relevant access
- AWS CLI configured with appropriate credentials
- Terraform backend configured (S3)

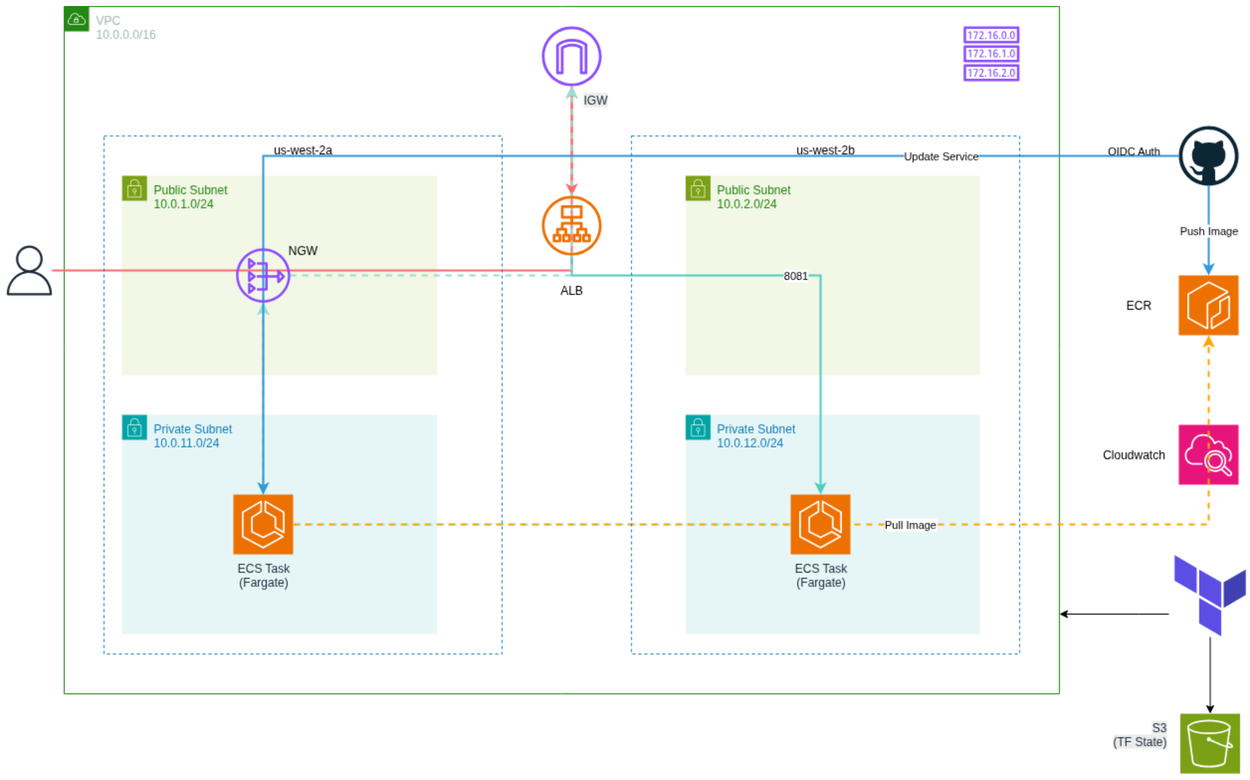
Container Image Preparation:

- ECR repository created manually
- Initial Docker image built and pushed to ECR
- Image URI noted for Terraform variables

Development Tools:

- Terraform >= 1.0 installed
- GitHub repository with Actions enabled
- Docker installed for local image building

Architecture Diagram



Infrastructure Modules

VPC Module

The VPC module creates the foundational network infrastructure spanning two availability zones for high availability.

Network Configuration:

• VPC with configurable CIDR block (default: 10.0.0.0/16)

• DNS hostnames and DNS support enabled for service discovery

• Two availability zones (us-west-2a and us-west-2b)

• Four subnets total: two public (10.0.1.0/24, 10.0.2.0/24) and two private (10.0.11.0/24, 10.0.12.0/24)

Public Subnet Configuration:

• Hosts the Application Load Balancer

• Automatic public IP assignment enabled

• Internet Gateway provides direct internet connectivity

• Route table directs all outbound traffic (0.0.0.0/0) to Internet Gateway

Private Subnet Configuration:

• Hosts ECS Fargate tasks
• No public IP assignment for enhanced security
• Single NAT Gateway deployed in first availability zone (cost optimization)
• Route table directs all outbound traffic through NAT Gateway

Connectivity Workflow:

Internet traffic reaches the ALB in public subnets through the Internet Gateway. The ALB forwards requests to ECS tasks in private subnets. ECS tasks access the internet (for pulling images, external APIs) through the NAT Gateway, maintaining security by never exposing tasks directly to the internet.

Security Module

The security module implements network-level access controls using security groups, following the principle of least privilege.

ALB Security Group:

• Ingress: Allows HTTP (port 80) from anywhere (0.0.0.0/0)
• Egress: Allows all outbound traffic for health checks and forwarding to targets

ECS Tasks Security Group:

• Ingress: Only accepts traffic from ALB security group on container port 8081
• Egress: Allows all outbound traffic for ECR pulls, CloudWatch logs, and external APIs

Security Workflow:

The security groups create a secure communication path where only the ALB can reach ECS tasks, and ECS tasks are completely isolated from direct internet access. This prevents unauthorized access while allowing necessary outbound connectivity through the NAT Gateway.

Application Load Balancer Module

The ALB module provides internet-facing load balancing and health checking for the containerized application.

Load Balancer Configuration:

• Internet-facing scheme for public accessibility
• Deployed across both public subnets for high availability
• Application layer (Layer 7) load balancing
• Associated with ALB security group

Target Group Configuration:

• Target type: IP (required for Fargate tasks)
• Protocol: HTTP on port 8081
• Health check path: / (root endpoint)
• Health check interval: 30 seconds
• Healthy threshold: 2 consecutive successful checks
• Unhealthy threshold: 3 consecutive failed checks

Listener Configuration:

• Listens on port 80 (HTTP)
• Forwards all traffic to the target group

Traffic Workflow:

Users access the application via the ALB DNS name on port 80. The ALB performs health checks on ECS tasks and distributes incoming requests only to healthy targets. Traffic is forwarded to the container port 8081 where the Node.js application listens.

IAM Module

The IAM module is critical for secure authentication and authorization. It implements three distinct roles with specific purposes and permissions, following AWS security best practices.

GitHub Actions Role (OIDC)

This role enables GitHub Actions to authenticate with AWS without storing long-lived credentials. Instead of using access keys, it leverages OpenID Connect for temporary, scoped credentials.

OIDC Provider Configuration:

• Provider URL: https://token.actions.githubusercontent.com
• Audience: sts.amazonaws.com
• Thumbprint: GitHub's certificate fingerprint for validation

Trust Policy:

• Action: sts:AssumeRoleWithWebIdentity
• Principal: Federated identity provider (GitHub OIDC)
• Condition: Restricted to specific GitHub repository using sub claim

Permissions Granted:

• ECR Authentication: GetAuthorizationToken (account-wide)
• ECR Image Operations: Push, pull, and manage layers (repository-specific)
• ECS Task Definition: Register and describe task definitions (account-wide)

• ECS Service: Update and describe the specific service

• IAM PassRole: Pass ECS execution and task roles to new task definitions

ECS Task Execution Role

This role is used by the ECS agent and Fargate runtime to prepare the container environment. It operates before the application container starts.

Trust Policy:

• Service: ecs-tasks.amazonaws.com can assume this role

Permissions:

• AWS Managed Policy: AmazonECSTaskExecutionRolePolicy

• ECR Image Pull: Authenticate and download container images

• CloudWatch Logs: Create log streams and write container logs

• Secrets Manager: Retrieve secrets if configured (not used in this project)

ECS Task Role

This role is used by the running application container itself. The application inherits these permissions during runtime for accessing AWS services.

Trust Policy:

• Service: ecs-tasks.amazonaws.com can assume this role

Permissions:

• CloudWatch Logs: Write application logs from within the container

• Custom permissions can be added here for application-specific AWS service access (S3, DynamoDB, etc.)

IAM Workflow:

When GitHub Actions runs, it requests temporary credentials from AWS STS using the OIDC token. AWS validates the token against the trust policy and issues short-lived credentials. These credentials allow the pipeline to push images to ECR and update the ECS service. When ECS starts a task, it assumes the execution role to pull the image and set up logging, then assumes the task role before starting the application container.

OpenID Connect (OIDC) Authentication

OIDC provides keyless authentication between GitHub Actions and AWS, eliminating the need to store AWS access keys in GitHub secrets.

How OIDC Works:

GitHub generates a signed JSON Web Token (JWT) for each workflow run. This token contains claims about the workflow identity including the repository name, branch, and commit SHA. AWS validates the token's signature using GitHub's public keys and checks the claims against the IAM role's trust policy. If validation succeeds, AWS issues temporary credentials (valid for 1 hour) that the workflow uses for AWS API calls.

Security Benefits:

• No long-lived credentials stored in GitHub
• Credentials automatically expire after workflow completion
• Fine-grained control through trust policy conditions
• Audit trail via CloudTrail with workflow identity
• Reduced risk of credential leakage or compromise

Implementation in This Project:

• OIDC provider registered in IAM with GitHub's identity URL
• Trust policy restricts access to specific repository
• GitHub Actions workflow uses configure-aws-credentials action
• Workflow receives temporary credentials for each deployment

ECS Module

The ECS module orchestrates container deployment using AWS Fargate, a serverless compute engine that eliminates the need to manage EC2 instances.

ECS Cluster:

• Logical grouping for services and tasks
• Fargate-only configuration (no EC2 capacity providers)

Task Definition:

• Launch type: Fargate
• Network mode: awsvpc (each task gets its own ENI)
• CPU: 256 units (0.25 vCPU)
• Memory: 512 MB
• Container image: Pulled from ECR
• Container port: 8081
• Execution role: For ECS agent operations
• Task role: For application runtime permissions
• Logging: CloudWatch Logs with 7-day retention

ECS Service:

• Desired count: 2 tasks for high availability
• Deployment in private subnets across both availability zones
• No public IP assignment (traffic routes through ALB)
• Load balancer integration with target group

• Automatic task replacement on failure

Container Lifecycle:

ECS service maintains the desired count of tasks. When a deployment occurs, ECS registers a new task definition, starts new tasks with the updated image, waits for them to pass health checks, then drains and stops old tasks. The ALB continuously monitors task health and only sends traffic to healthy targets.

CI/CD Pipeline

The GitHub Actions pipeline automates the build, push, and deployment process. It triggers on every push to the main branch and executes a series of steps to update the running application.

Pipeline Configuration

Trigger:

• Activated on push events to the main branch (automatic)

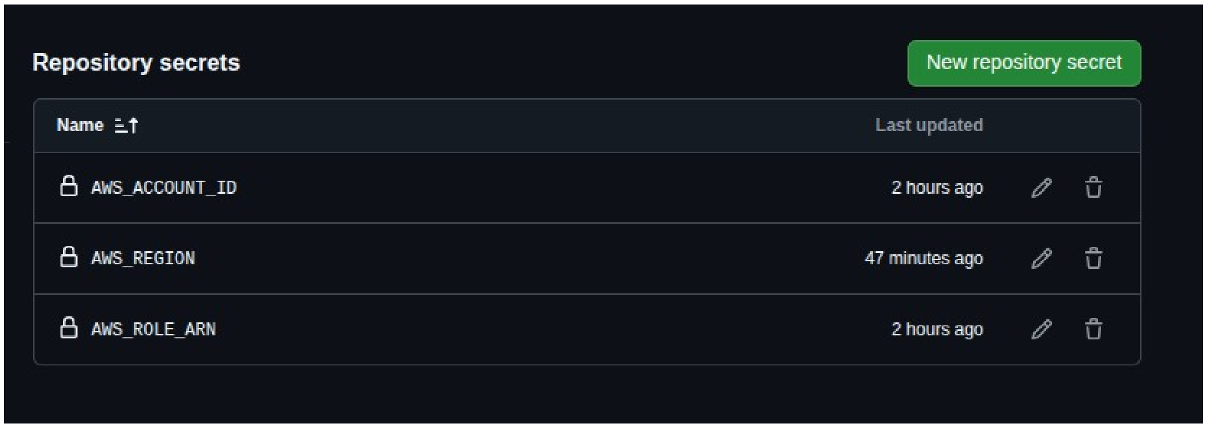
• Activated manually using the workflow_dispatch method (to avoid accidental deployment to environment)

Permissions:

• id-token: write (required for OIDC authentication)
• contents: read (required to checkout repository)

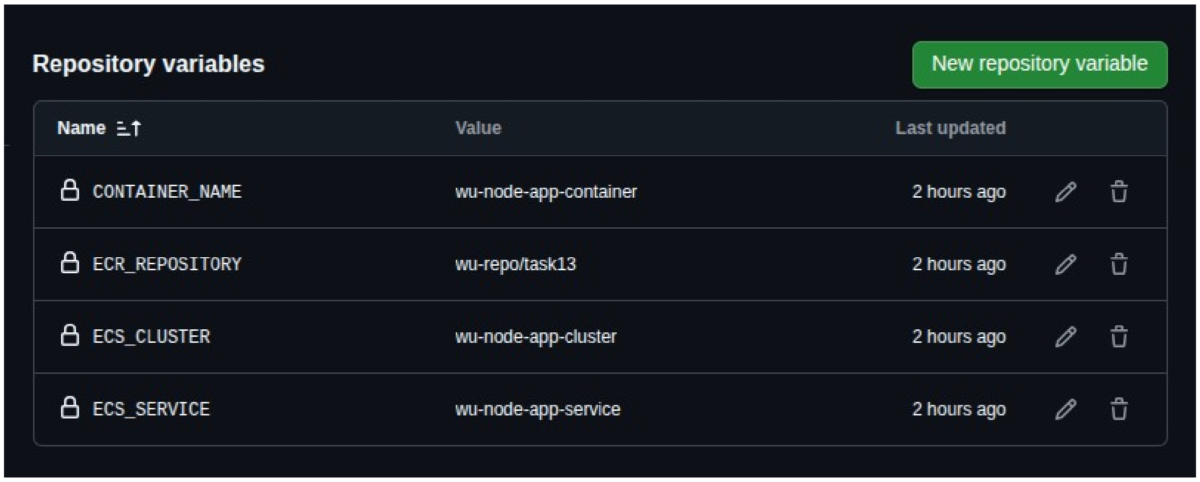
Environment Variables:

Repository secrets:
• AWS_REGION: Target deployment region
• AWS_ACCOUNT_ID: AWS account identifier
• AWS_ROLE_ARN: github action role



Name	Last updated
AWS_ACCOUNT_ID	2 hours ago
AWS_REGION	47 minutes ago
AWS_ROLE_ARN	2 hours ago

Repository Variables:
• ECR_REPOSITORY: Container registry repository name
• ECS_CLUSTER: Target ECS cluster name
• ECS_SERVICE: Target ECS service name
• CONTAINER_NAME: Container name in task definition


Repository variables

Name	Value	Last updated
CONTAINER_NAME	wu-node-app-container	2 hours ago
ECR_REPOSITORY	wu-repo/task13	2 hours ago
ECS_CLUSTER	wu-node-app-cluster	2 hours ago
ECS_SERVICE	wu-node-app-service	2 hours ago

Pipeline Stages

Stage 1: Source Checkout

Checks out the repository code including the Dockerfile and application source.

Stage 2: AWS Authentication

Uses the configure-aws-credentials action to authenticate with AWS using OIDC. The action requests a token from GitHub, exchanges it with AWS STS for temporary credentials, and configures the AWS CLI.

Required inputs: IAM role ARN and AWS region.

Stage 3: ECR Login

Authenticates Docker with Amazon ECR using the temporary credentials from the previous step. This enables pushing images to the private registry.

Stage 4: Build and Push Image

Builds the Docker image with a tag based on the Git commit SHA. This ensures each deployment has a unique, traceable image version. The image is tagged with both the commit SHA and latest tag, then pushed to ECR.

The image URI is exported as an output for subsequent steps.

Stage 5: Download Current Task Definition

Retrieves the task definition currently used by the ECS service. This ensures all existing configuration (CPU, memory, environment variables, IAM roles) is preserved during the update.

Stage 6: Clean Task Definition

Removes AWS-managed fields that cannot be included when registering a new task definition revision. Fields removed include taskDefinitionArn, revision, status, requiresAttributes, compatibilities, registeredAt, registeredBy, and enableFaultInjection.

Uses jq to process the JSON and create a clean task definition.

Stage 7: Render New Task Definition

Updates the container image field in the task definition with the newly built image URI. All other configuration remains unchanged.

Stage 8: Deploy to ECS

Registers the new task definition with ECS and updates the service to use it. ECS performs a rolling deployment, starting new tasks with the updated image while maintaining the desired count.

The pipeline waits for service stability before completing, ensuring the deployment succeeded and new tasks are healthy.

Deployment Workflow:

A code push to main triggers the pipeline. The application is built into a container image tagged with the commit SHA. This image is pushed to ECR. The current ECS task definition is downloaded and cleaned. The image field is updated with the new image URI. The updated task definition is registered with ECS. The ECS service is updated to use the new task definition. ECS starts new tasks, waits for them to be healthy, then stops old tasks. The ALB routes traffic only to healthy tasks throughout the deployment.

Testing and Validation

Infrastructure Validation

Terraform Validation:

• Run terraform validate to check configuration syntax

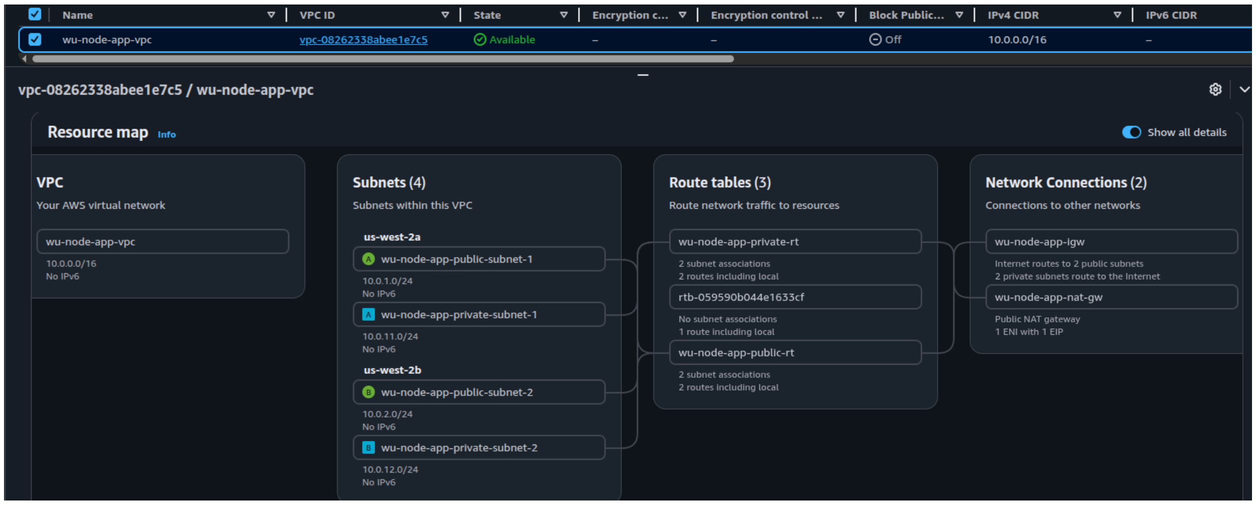
• Execute terraform plan to preview resource creation

• Verify outputs match expected values (VPC ID, subnet IDs, security group IDs)

Network Connectivity:

• Confirm Internet Gateway attached to VPC

• Verify NAT Gateway has Elastic IP and routes traffic
• Check route tables point to correct gateways
• Validate security group rules allow expected traffic



Name	VPC ID	State	Encryption c...	Encryption control ...	Block Public...	IPv4 CIDR	IPv6 CIDR
wu-node-app-vpc	vpc-08262338abee1e7c5	Available	–	–	Off	10.0.0.0/16	–

vpc-08262338abee1e7c5 / wu-node-app-vpc

Resource map Info

Show all details

VPC
Your AWS virtual network

wu-node-app-vpc
10.0.0.0/16
No IPv6

Subnets (4)
Subnets within this VPC

us-west-2a
wu-node-app-public-subnet-1
10.0.1.0/24
No IPv6
wu-node-app-private-subnet-1
10.0.11.0/24
No IPv6

us-west-2b
wu-node-app-public-subnet-2
10.0.2.0/24
No IPv6
wu-node-app-private-subnet-2
10.0.12.0/24
No IPv6

Route tables (3)
Route network traffic to resources

wu-node-app-private-rt
2 subnet associations
2 routes including local
rtb-059590b044e1633cf
No subnet associations
1 route including local
wu-node-app-public-rt
2 subnet associations
2 routes including local

Network Connections (2)
Connections to other networks

wu-node-app-igw
Internet routes to 2 public subnets
2 private subnets route to the Internet
wu-node-app-nat-gw
Public NAT gateway
1 ENI with 1 EIP

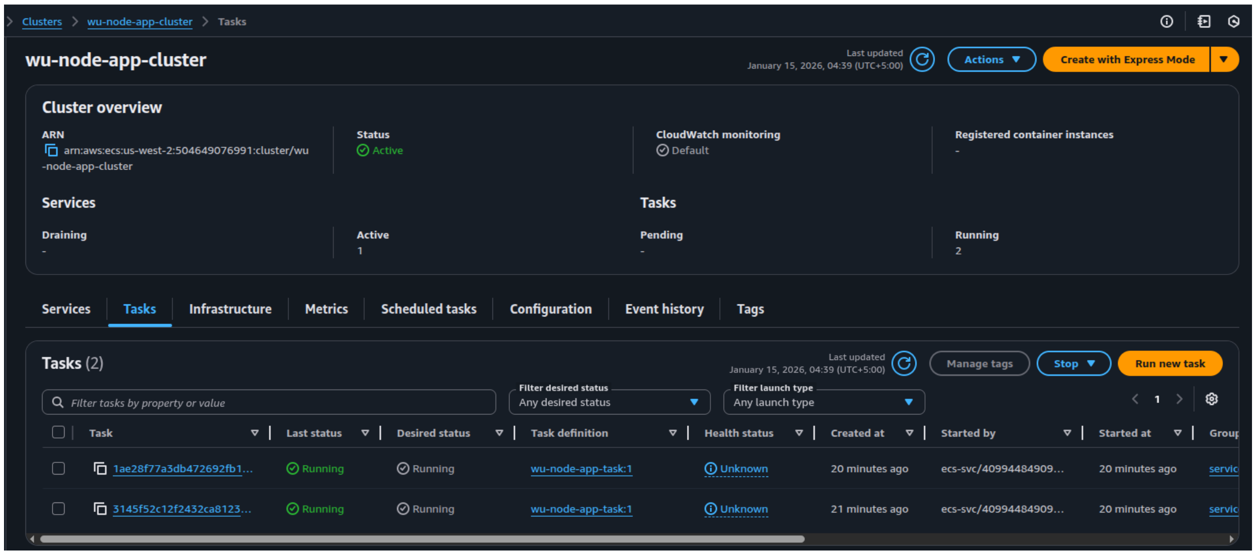
IAM Configuration:

• Verify OIDC provider is registered in IAM
• Check GitHub Actions role has correct trust policy
• Confirm ECS roles have appropriate permissions
• Test role assumption from GitHub Actions

Application Validation

ECS Service Health:

• Navigate to ECS console and verify cluster is active
• Check service shows desired task count running
• Verify tasks are in RUNNING state
• Confirm tasks pass ALB health checks


Clusters > wu-node-app-cluster > Tasks

wu-node-app-cluster

Last updated
January 15, 2026, 04:39 (UTC+5:00)

Actions
Create with Express Mode

Cluster overview

ARN
arn:aws:ecs:us-west-2:504649076991:cluster/wu-node-app-cluster

Status
Active

CloudWatch monitoring
Default

Registered container instances
-

Services

Draining
-

Active
1

Tasks

Pending
-

Running
2

Services | Tasks | Infrastructure | Metrics | Scheduled tasks | Configuration | Event history | Tags

Tasks (2)

Last updated
January 15, 2026, 04:39 (UTC+5:00)

Manage tags
Stop
Run new task
Filter tasks by property or value

Filter desired status: Any desired status
Filter launch type: Any launch type

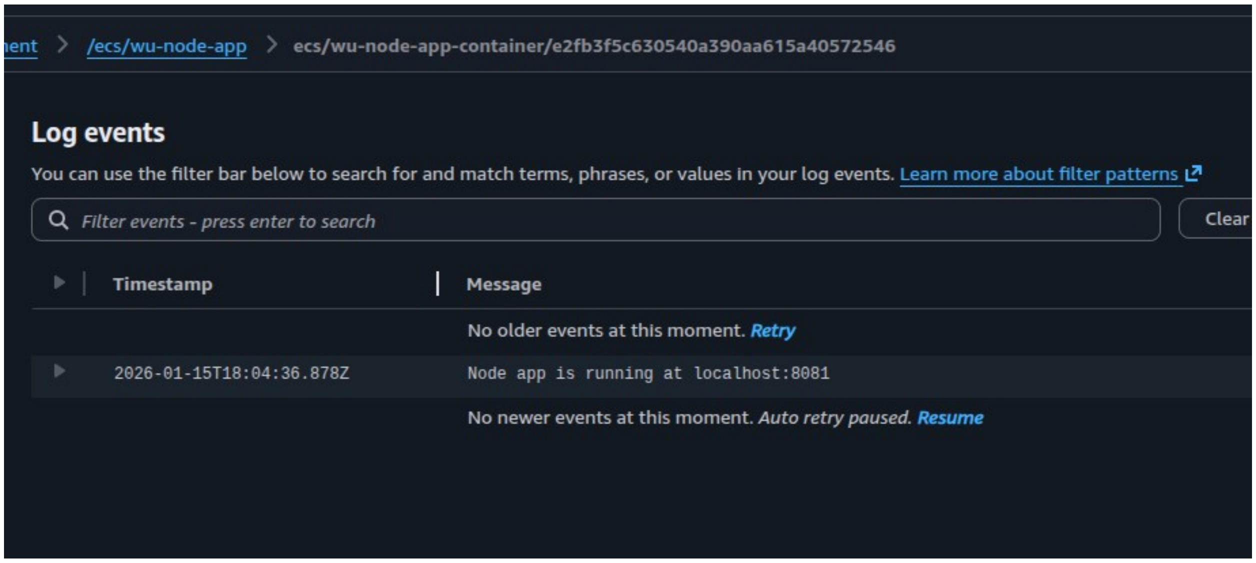
Task	Last status	Desired status	Task definition	Health status	Created at	Started by	Started at	Group
1ae28f77a3db472692fb1...	Running	Running	wu-node-app-task:1	Unknown	20 minutes ago	ecs-svc/40994484909...	20 minutes ago	servic
3145f52c12f2432ca8123...	Running	Running	wu-node-app-task:1	Unknown	21 minutes ago	ecs-svc/40994484909...	20 minutes ago	servic

Load Balancer Testing:

- Access ALB DNS name in browser
- Verify application responds correctly
- Check target group shows healthy targets
- Test multiple requests to verify load distribution

Logging Verification:

- Navigate to CloudWatch Logs console
- Find log group: /ecs/wu-node-app
- Verify log streams exist for each task
- Check application logs are being written


ient > /ecs/wu-node-app > ecs/wu-node-app-container/e2fb3f5c630540a390aa615a40572546

Log events

You can use the filter bar below to search for and match terms, phrases, or values in your log events. Learn more about filter patterns

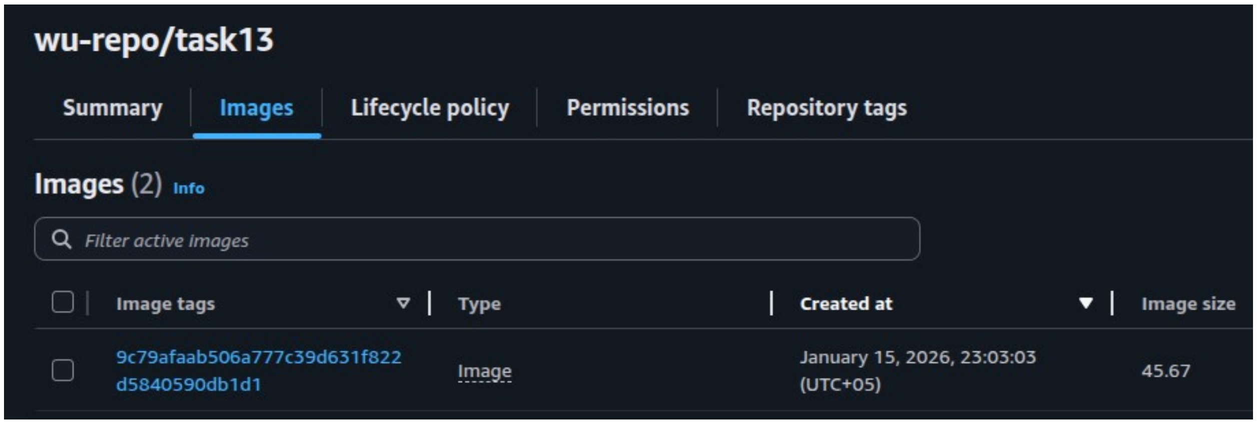
Filter events - press enter to search
Clear

```
Timestamp                  Message
                           No older events at this moment. Retry
2026-01-15T18:04:36.878Z   Node app is running at localhost:8081
                           No newer events at this moment. Auto retry paused. Resume
```

CI/CD Pipeline Testing

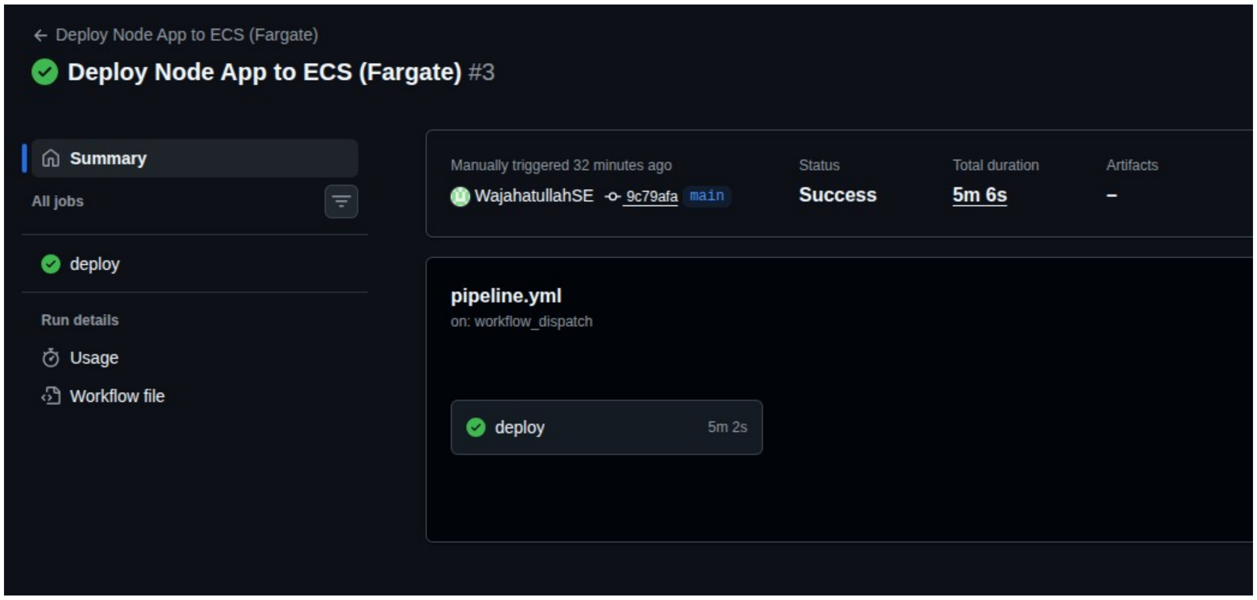
Initial Deployment:

• Make a small code change and commit to main branch
• Monitor GitHub Actions workflow execution
• Verify each pipeline stage completes successfully
• Confirm new image appears in ECR with commit SHA tag



Images (2) Info

Image tags	Type	Created at	Image size
9c79afaab506a777c39d631f822d5840590db1d1	Image	January 15, 2026, 23:03:03 (UTC+05)	45.67



Deploy Node App to ECS (Fargate) #3

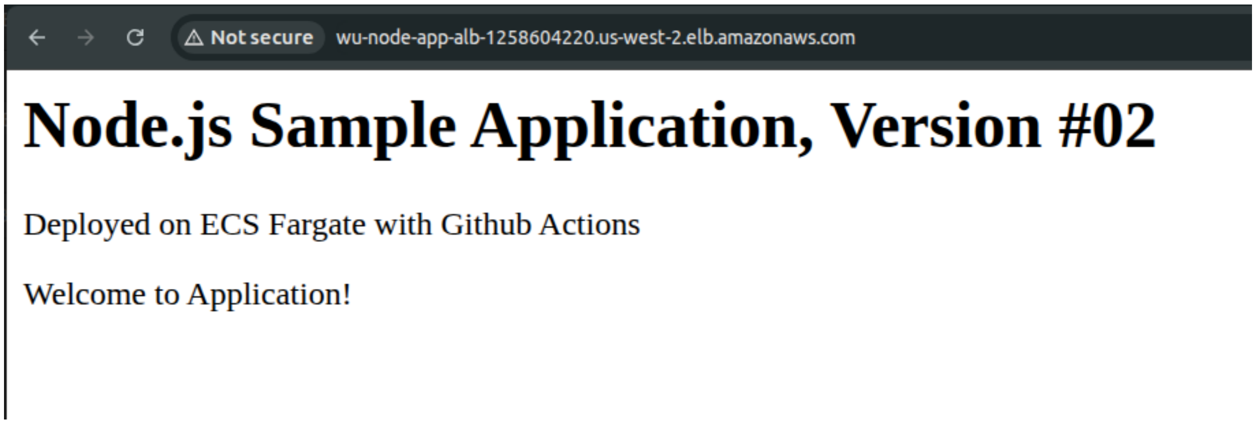
Manually triggered 32 minutes ago	Status	Total duration	Artifacts
WajahatullahSE 9c79afa main	Success	5m 6s	–

pipeline.yml
on: workflow_dispatch

deploy 5m 2s

Deployment Verification:

- Check ECS service events for deployment status
- Verify new task definition revision is registered
- Confirm new tasks start and old tasks are replaced
- Test application reflects the code changes


Not secure wu-node-app-alb-1258604220.us-west-2.elb.amazonaws.com

Node.js Sample Application, Version #02

Deployed on ECS Fargate with Github Actions

Welcome to Application!

ECS Deployment Approaches

Approach 1: Download → Modify → Register → Deploy

How it works

- Fetch currently running task definition from ECS
- Remove AWS-managed / read-only fields
- Inject new container image
- Register a new task definition revision
- Update ECS service to the new revision

Characteristics

- Immutable deployments
- New task definition per release
- ECS remains the source of truth
- CI/CD modifies only the image

Why this is better

- Prevents configuration drift
- Full deployment history via task definition revisions
- Safe rollback by reverting to older revision
- Clean separation:
    - Terraform → infrastructure
    - CI/CD → releases

Approach 2: Static Task Definition JSON in Repository

How it works

- Maintain `task-def.json` in repo
- CI/CD updates image field
- Registers task definition from static file
- Updates ECS service

Drawbacks

- Configuration drift between Terraform and JSON

- Duplicate ownership of task definition
- Risk of overwriting infra changes
- Requires maintaining roles, CPU, memory, logging in multiple places
- Poor long-term maintainability
- Breaks Terraform’s state consistency

Use only when

- Small demo
- No Terraform
- Single environment

Approach 3: CLI Force Deployment (`--force-new-deployment`)

How it works

- No new task definition
- ECS restarts tasks using existing task definition
- Image must be mutable (e.g., `latest`)

Drawbacks

- No versioning or audit trail
- No rollback capability
- Non-deterministic deployments
- Relies on image cache behavior
- Breaks immutable deployment principles
- Not suitable for production systems

Use only when

- Dev-only environments
- Temporary testing
- Manual troubleshooting

Troubleshooting

Issue 1: OIDC Provider Already Exists

Cause:

The GitHub OIDC provider already exists in the AWS account, typically created by another team member or previous project in shared sandbox environments.

Solution:

Use a data source instead of creating a new resource. In modules/iam/main.tf, replace the resource block:

Replace:

```
resource "aws_iam_openid_connect_provider" "github" { ... }
```

With:

```
data "aws_iam_openid_connect_provider" "github" { url = "https://token.actions.githubusercontent.com" }
```

Update the reference in the trust policy (same file):

Change:

```
identifiers = [aws_iam_openid_connect_provider.github.arn]
```

To:

```
identifiers = [data.aws_iam_openid_connect_provider.github.arn]
```

After making changes, re-run terraform plan and apply.