# Deploying a Containerized Application on Amazon ECS EC2 with AWS CI/CD

## Table of Contents

# 1. Prerequisites

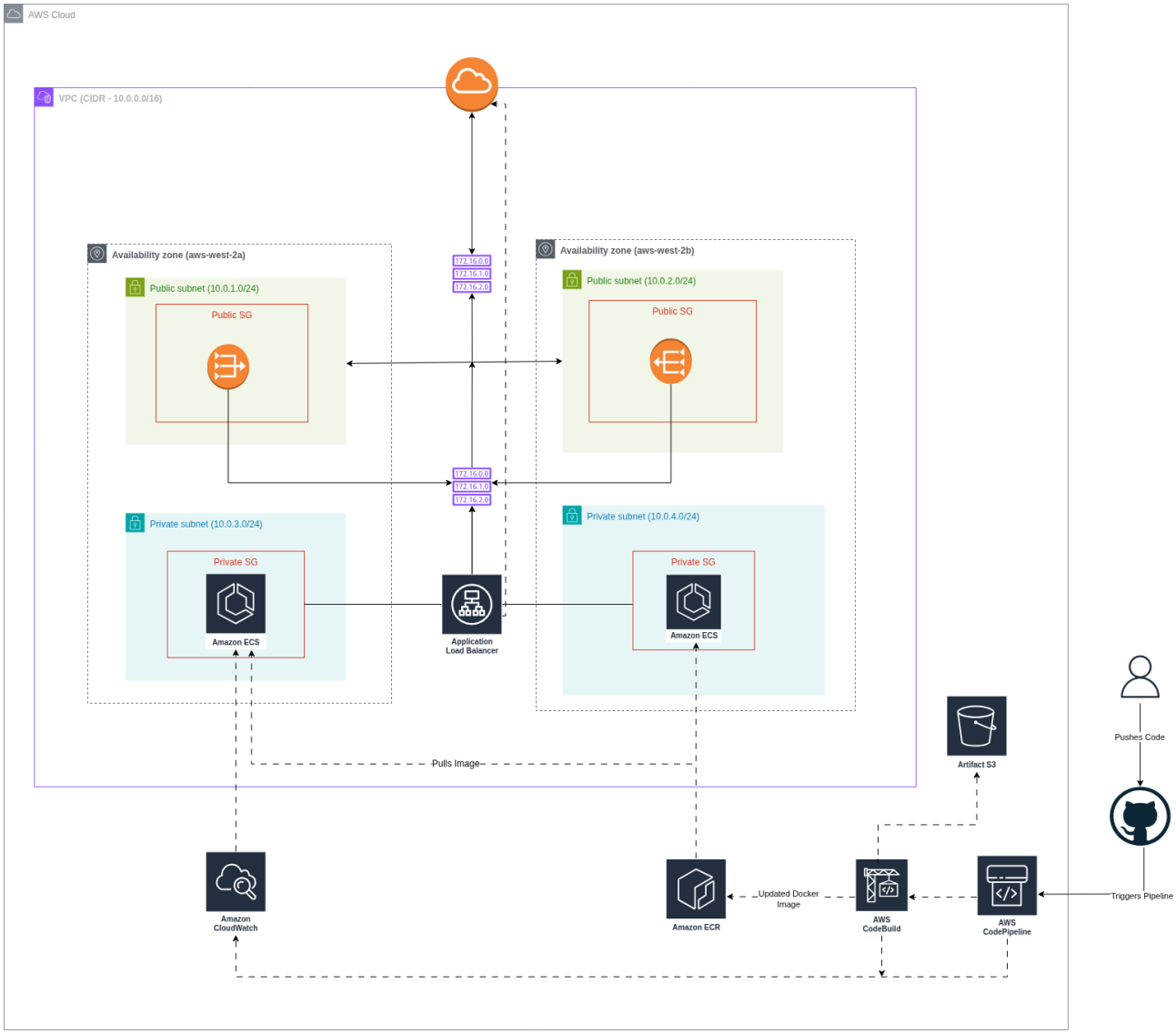
- An **AWS account** with administrative privileges to create and manage resources.

# 2. Objective

The goal of this guide is to set up a **CI/CD pipeline** for deploying a containerized application on **Amazon ECS (EC2 launch type)** using:

- **ECS Cluster** with Auto Scaling.
- **Application Load Balancer (ALB)** for traffic distribution.
- **ECR** for Docker image storage.
- **CodePipeline and CodeBuild** for automation (optional, covered in later sections).

# 3. Architecture Overview



# 4. Implementation (Section A: Network & ECS Config)

## 4.1 Create the VPC and Networking

**1. Create VPC**

- In AWS Console → VPC → Your VPCs → Create VPC.
    - Name tag: `wu2-vpc` (or `wu-vpc` per your naming scheme).
    - IPv4 CIDR block: `10.44.0.0/16`
    - IPv6: none
    - Enable DNS resolution: **Yes**
    - Enable DNS hostnames: **Yes**
- Create and note the VPC ID.

**2. Create public subnets (2 AZs)**

- VPC → Subnets → Create subnet (do twice).
    - Subnet 1:
        - Name tag: `wu2-public-sn-1`
        - Availability Zone: `us-west-2a`
        - CIDR block: `10.44.1.0/24`
        - Auto-assign public IPv4: **Enable**
    - Subnet 2:
        - Name tag: `wu2-public-sn-2`
        - Availability Zone: `us-west-2b`
        - CIDR block: `10.44.2.0/24`
        - Auto-assign public IPv4: **Enable**
- Save subnet IDs.

**3. Create private subnets (2 AZs)**

- VPC → Subnets → Create subnet (twice).
    - Private Subnet 1:
        - Name: `wu2-private-sn-1`
        - AZ: `us-west-2a`
        - CIDR: `10.44.3.0/24`

        - Auto-assign public IPv4: **Disable**
    - Private Subnet 2:
        - Name: `wu2-private-sn-2`
        - AZ: `us-west-2b`
        - CIDR: `10.44.4.0/24`
        - Auto-assign public IPv4: **Disable**
- Save private subnet IDs.

**4. Create Internet Gateway**

- VPC → Internet Gateways → Create Internet Gateway.
    - Name tag: `wu2-igw`
    - Attach to your VPC.

**5. Create NAT Gateway**

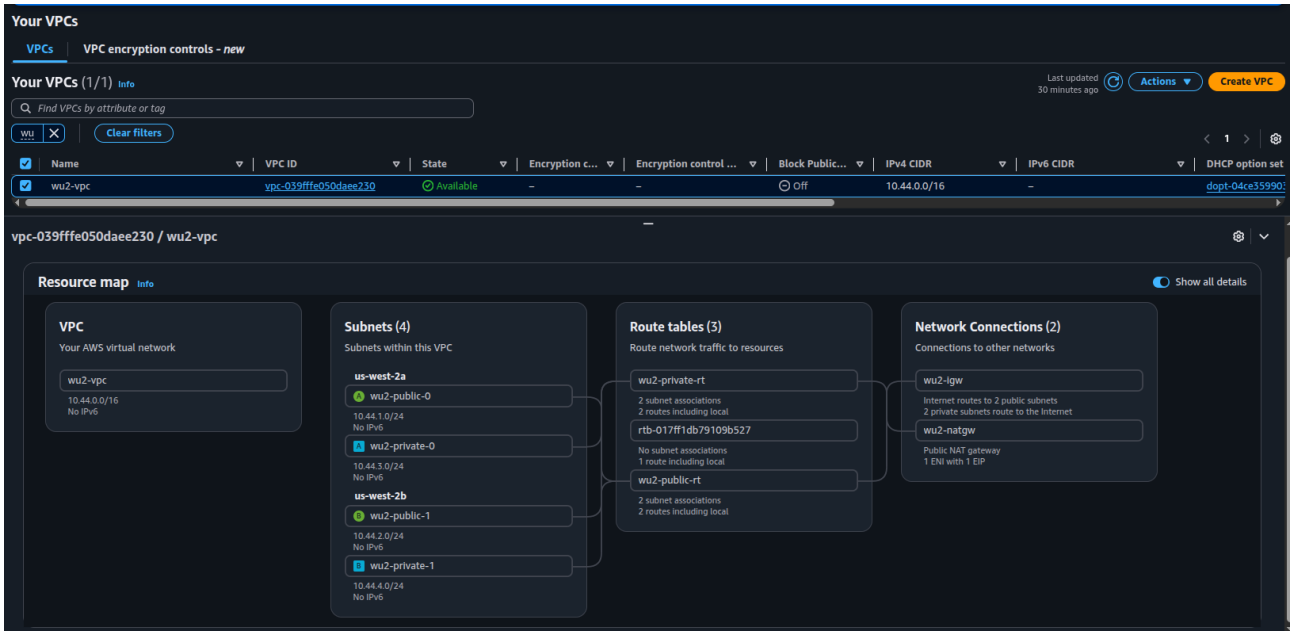
- VPC → NAT Gateways → Create NAT Gateway.
    - Subnet: choose one public subnet (e.g., `wu2-public-sn-1`).
    - Allocate new Elastic IP (or choose existing EIP).
    - Name tag: `wu2-natgw`
- Wait for NAT Gateway to become **Available**.

**6. Create Route Tables**

- Public Route Table:
    - VPC → Route Tables → Create route table.
        - Name: `wu2-public-rt`
        - VPC: select your VPC
    - Edit routes:
        - `0.0.0.0/0` → Target: Internet Gateway (`wu2-igw`)
    - Associate route table:
        - Associate with both public subnets (`wu2-public-sn-1`, `wu2-public-sn-2`).
- Private Route Table:
    - Create route table:
        - Name: `wu2-private-rt`
        - VPC: select your VPC
    - Edit routes:

    - `0.0.0.0/0` → Target: NAT Gateway (`wu2-natgw`)
- Associate with both private subnets.

**Why this matters:** public subnets host ALB and NAT; private subnets host ECS instances and use NAT for outbound ECR/CloudWatch access.




# 4.2 Security Groups (Least privilege)

### 7. Create ALB security group

- EC2 → Security Groups → Create security group
    - Name: `wu2-alb-sg`
    - VPC: your VPC
    - Inbound rules:
        - HTTP (TCP 80) — Source: `0.0.0.0/0` (public)
    - Outbound rules:
        - All traffic — Destination: `0.0.0.0/0`
    - Tag Name: `wu2-alb-sg`

### 8. Create ECS instances security group

- Create SG:
    - Name: `wu2-ecs-sg`
    - VPC: your VPC
    - Inbound rules:

        - Custom TCP — Port range: `32768-65535` — Source: Security group `wu2-alb-sg`
            - Reason: dynamic host ports (Docker ephemeral range) used in bridge mode.
        - (Optional) SSH (TCP 22) — Source: *Admin CIDR* (only if you need SSH; otherwise rely on SSM).
    - Outbound rules:
        - All traffic — Destination: `0.0.0.0/0`
    - Tag Name: `wu2-ecs-sg`

**Confirm security group linkage:** ALB should be allowed to reach ECS instances on ephemeral ports; ECS instances must be able to reach ECR/CloudWatch via NAT.

# 4.3 Application Load Balancer

**9. Create ALB**

- EC2 → Load Balancing → Load Balancers → Create Load Balancer → Application Load Balancer
    - Name: `wu2-alb`
    - Scheme: Internet-facing
    - IP address type: IPv4
    - VPC: your VPC
    - Subnets: select both public subnets (`wu2-public-sn-1` and `wu2-public-sn-2`)
    - Security group: `wu2-alb-sg`
    - Name tag: `wu2-alb`

**10. Create Target Group**

- Target Groups → Create target group
    - Target type: **Instance**
    - Protocol: HTTP
    - Port: `8081` (reference container port)
    - VPC: your VPC
    - Health checks:
        - Protocol: HTTP
        - Path: `/` (or your app health endpoint)

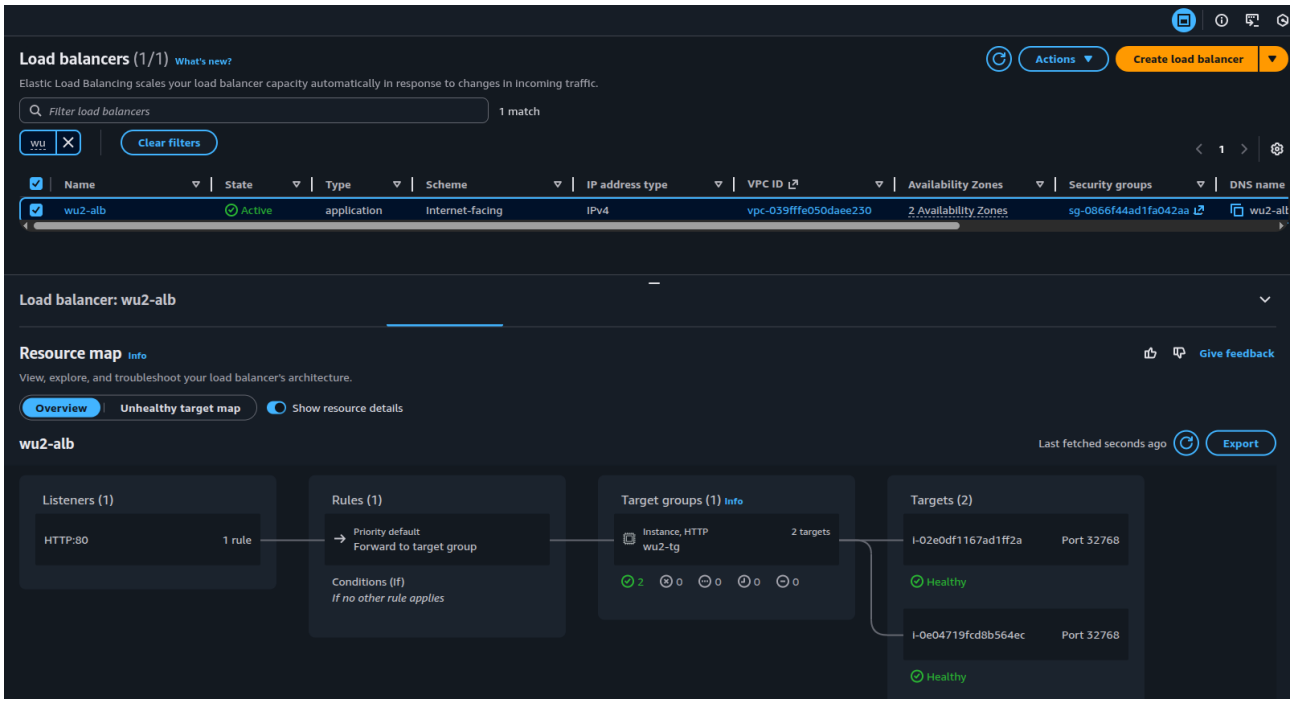
        - Port: traffic-port
        - Healthy thresholds: 3
        - Interval: 30s
        - Timeout: 5s
        - Success codes: `200-399`
    - Name tag: `wu2-tg`

**11. Create Listener**

- On ALB → Listeners → Create listener
    - Protocol: HTTP
    - Port: 80
    - Default action: Forward to `wu2-tg`

**Notes about dynamic host port discovery**

- When tasks use hostPort = 0 (dynamic), ALB uses ECS service registration to discover actual host port. Target group port is the reference; ALB will route to the discovered host port on the instance.




# 4.4 ECR (reference existing repository)

**12. Reference existing ECR repo (no creation required)**

- Confirm ECR repository `wu-repo/task07` exists and note its URI:

    - e.g., `504649076991.dkr.ecr.us-west-2.amazonaws.com/wu-repo/task07`
- Build/push workflow (local machine or CI):
    - `docker build -t wu-repo:latest .`
    - Tag:
        - `docker tag wu-repo:latest 504649076991.dkr.ecr.us-west-2.amazonaws.com/wu-repo/task07:latest`
    - Login to ECR and push:
        - `aws ecr get-login-password --region us-west-2 | docker login --username AWS --password-stdin 504649076991.dkr.ecr.us-west-2.amazonaws.com`
        - `docker push 504649076991.dkr.ecr.us-west-2.amazonaws.com/wu-repo/task07:latest`

# 4.5 IAM Roles — EC2 Instance Role and Instance Profile

**13. Create IAM Role for EC2 instances**

- IAM → Roles → Create role
    - Trusted entity: **AWS service** → EC2
    - Attach managed policies:
        - `AmazonEC2ContainerServiceforEC2Role`
        - `AmazonSSMManagedInstanceCore` (optional but recommended for SSM Session Manager)
    - Role name: `wu2-ec2-instance-role`
    - Tag: `Name=wu2-ec2-instance-role`

**14. Create Instance Profile**

- IAM → Instance Profiles (Roles page): the role is automatically usable as an instance profile when attached to launch template; if needed, create instance profile:
    - Instance profile name: `wu2-ec2-instance-profile`
    - Attach the `wu2-ec2-instance-role`

**Why this matters:** the EC2 instances assume this role so the ECS agent can register with ECS, pull images from ECR, and send logs to CloudWatch. Without it, instances start but never join the cluster.

# 4.6 CloudWatch Log Group

**15. Create CloudWatch Log Group**

- CloudWatch → Log groups → Create log group:
    - Name: `/ecs/wu2-ecs-cluster`
    - Retention: 7 days
    - Tag: `Name=/ecs/wu2-ecs-cluster`
- This group will receive container logs via the ecs task execution role and awslogs driver.

# 4.7 ECS Cluster and Launch Template

**16. Create ECS cluster**

- ECS → Clusters → Create cluster → EC2 Linux + Networking
    - Cluster name: `wu2-ecs-cluster`
    - Container insights: **Optional** (enable if you want enhanced metrics)
    - Create (no EC2 instances here — we will use an Auto Scaling group)

**17. Create Launch Template**

- EC2 → Instances → Launch Templates → Create launch template
    - Name: `wu2-lt`
    - AMI: ECS-optimized Amazon Linux 2023 image (use SSM parameter or console to get the latest ECS-optimized AMI). Example AMI (confirm current): `ami-0a55b150592945e7d` (verify before using).
    - Instance type: `t3.medium`
    - Key pair: `wu-key` (if you need SSH; prefer SSM otherwise)
    - Network interfaces:
        - Security groups: `wu2-ecs-sg`
        - Associate public IP address: **No** (instances live in private subnets)
    - IAM instance profile:
        - Name: `wu2-ec2-instance-profile`
    - Block device mapping:
        - Root volume: 30 GiB gp3 (as required)
    - User data (base64 or plain script): minimal, set ECS cluster name:
    - `#!/bin/bash`

```
echo "ECS_CLUSTER=wu2-ecs-cluster" > /etc/ecs/ecs.config
```

    - Tag template: Name = `wu2-ecs-instance`
- Save template version.

**Why:** the launch template ensures instances boot with the ECS agent configured to join `wu2-ecs-cluster` and with correct IAM profile.

# 4.8 Auto Scaling Group + Capacity Provider

**18. Create Auto Scaling Group (ASG)**

- EC2 → Auto Scaling → Create Auto Scaling group
    - Name: `wu2-ecs-asg`
    - Launch template: `wu2-lt` (select latest version)
    - VPC: your VPC
    - Subnets: select both **private** subnets (`wu2-private-sn-1`, `wu2-private-sn-2`)
    - Group size:
        - Min: `2`
        - Desired: `2`
        - Max: `2`
    - Health check type: `EC2`
    - Instance protection / scale-in protection: **Enable** for instances if you want ECS to manage draining.
    - Tags: Name = `wu2-ecs-instance` (propagate at launch)
- Create ASG.

**19. Create ECS Capacity Provider**

- ECS → Capacity providers → Create capacity provider
    - Name: `wu2-ecs-cp`
    - Auto Scaling Group: select `wu2-ecs-asg`
    - Managed scaling: Enabled (optional)
        - Target capacity: `10%` (or your desired)
    - Managed termination protection: **ENABLED** (recommended)
- Associate capacity provider with the ECS cluster:
    - Go to clusters → `wu2-ecs-cluster` → Capacity providers → Manage capacity provider
    - Add `wu2-ecs-cp` with weight 1 and base 0.

**Why:** capacity provider allows ECS to request ASG to scale and manages safe draining when scale-in occurs.

# 4.9 IAM Task Execution Role and Task Role

**20. Create Task Execution Role**

- IAM → Roles → Create role
    - Trusted entity: AWS service → `ecs-tasks.amazonaws.com`
    - Attach managed policy: `AmazonECSTaskExecutionRolePolicy`
    - Role name: `wu2-task-execution-role`
    - Tag: `Name=wu2-task-execution-role`

**21. (Optional) Create Task Role (application role)**

- If your app needs AWS permissions (S3, DynamoDB), create a role:
    - Trusted entity: `ecs-tasks.amazonaws.com`
    - Attach minimal custom policies needed.
    - Role name: `wu2-task-role` (if needed)

# 4.10 ECS Task Definition (bridge mode, dynamic port mapping)

**22. Create Task Definition**

- ECS → Task Definitions → Create new task definition
    - Launch type compatibility: **EC2**
    - Task definition name: `wu2-taskdef`
    - Network mode: **bridge**
    - Task role: (optional) `wu2-task-role`
    - Task execution role: `wu2-task-execution-role`
    - Task CPU / Memory: e.g., CPU 256, Memory 512
    - Container definitions:
        - Container name: `wu-app`
        - Image: `504649076991.dkr.ecr.us-west-2.amazonaws.com/wu-repo/task07:latest`
        - Memory limits: 480 MiB
        - CPU: 200
        - Port mappings:
            - Container port: `8081`

        - Host port: `0` (dynamic host port)
        - Protocol: `tcp`
    - Logging:
        - Log driver: `awslogs`
        - Log group: `/ecs/wu2-ecs-cluster` (created earlier)
        - Region: `us-west-2`
        - Stream prefix: `wu-app`
    - Environment variables: (add if required)
- Register the task definition.

**Important application note:** ensure the Node.js app binds to `0.0.0.0:8081` (not `localhost`) so Docker host forwarding works.

# 4.11 ECS Service

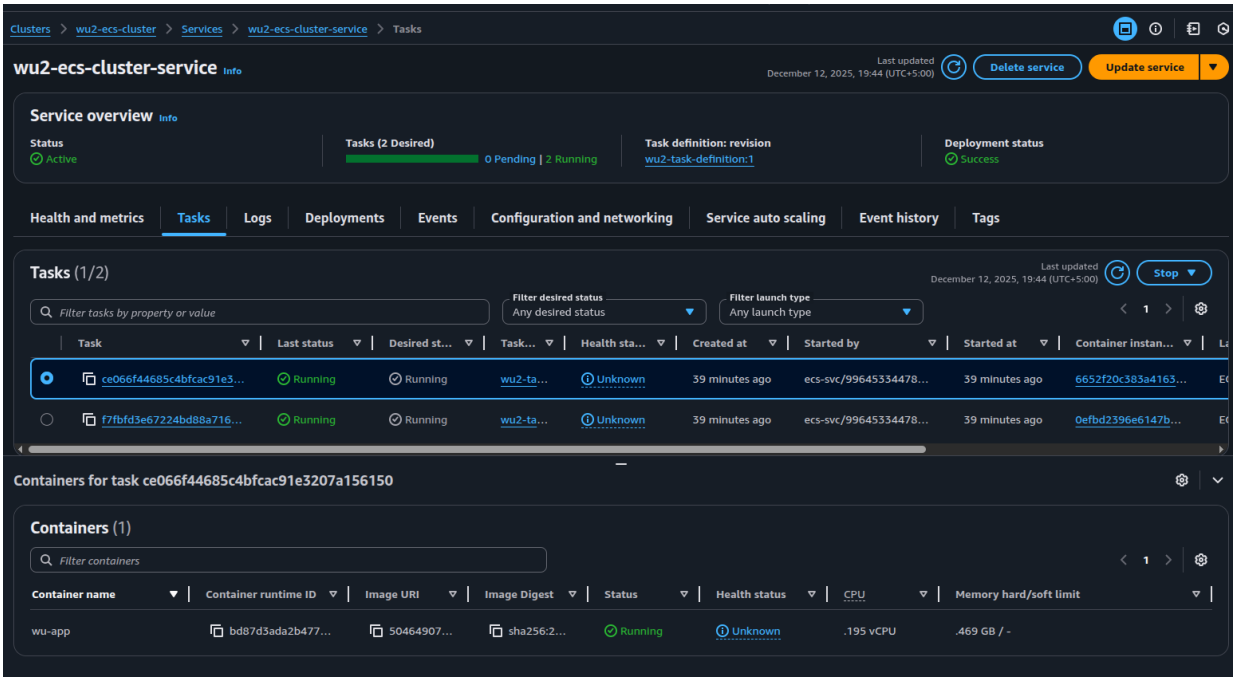
**23. Create ECS service**

- ECS → Clusters → `wu2-ecs-cluster` → Services → Create
    - Launch type: **EC2**
    - Service name: `wu2-service`
    - Task definition: select the `wu2-taskdef` revision
    - Number of tasks: `2`
    - Deployment options:
        - Minimum healthy percent: `50%`
        - Maximum percent: `100%`
    - Capacity provider strategy: choose `wu2-ecs-cp` (if using capacity provider) OR use ASG-managed capacity — do not specify both launch type and capacity provider simultaneously in API calls (console sets correctly when you choose EC2 + capacity provider)
    - Load balancing:
        - Enable service discovery with ALB: **Yes**
        - Load balancer type: **Application Load Balancer**
        - Select ALB: `wu2-alb`
        - Target group: `wu2-tg`
        - Container name: `wu-app`
        - Container port: `8081`

- Health check grace period: set (e.g., 60s) to allow container startup
- Start service

**Notes:**

- The service instructs ECS to register the instance:hostport mapping for each task with the target group so ALB can route traffic.




# 4.12 Post-deployment Verification

### 24. Confirm EC2 instances are registered with ECS

- ECS → Clusters → `wu2-ecs-cluster` → Infrastructure
  - Verify `Registered container instances`: should show 2.
- If not:
  - SSH (or SSM) into an instance:
    - `sudo cat /etc/ecs/ecs.config` → should show `ECS_CLUSTER=wu2-ecs-cluster`
    - `sudo systemctl status ecs` → check agent is running
    - Check logs: `sudo journalctl -u ecs -n 200 --no-pager`
  - Confirm IAM instance profile attached to the instance.

### 25. Confirm tasks are RUNNING

- ECS → Clusters → Services → `wu2-service` → Tasks
  - Should be in RUNNING state.

- Note the host port assigned (ECS console for each task shows the host port mapped from the ephemeral range).

**26. Confirm ALB target group health**

- Load Balancers → Target Groups → `wu2-tg` → Targets
    - Registered targets should show instance IDs and assigned port (e.g., `i-abc123:32768`)
    - Health status: `healthy` (3/3 checks passing)
    - If unhealthy: check health check path, container logs, and security groups.

**27. Test the application**

- Browser: open ALB DNS name (from ALB → Description → DNS name).
- Example: `http://<wu2-alb-dns>`
    - You should see the Node.js sample page (Version text from the app).


# Operational notes & best practices

- Keep `hostPort` mapping dynamic (`0`) if you want multiple tasks per instance. Ensure `wu2-ecs-sg` allows the ephemeral range from ALB.
- Prefer SSM for instance access instead of opening SSH ports.
- Use IAM instance profile on launch template — otherwise instances will not register.
- Set CloudWatch log retention and use `awslogs` driver so you can inspect container logs centrally.
- Use capacity providers and ASG tags to allow ECS to manage scaling safely (enable managed termination protection).

# 5. Implementation (Section B: CI/CD Pipeline with CodeBuild and CodePipeline)

## 5.1 IAM Config for CodeBuild & CodePipeline

### 1. CodeBuild Service Role

**Role Name:** `CodeBuild-ECSEC2-ServiceRole-wu`

**Purpose:** Allows CodeBuild to access ECR, S3, CloudWatch Logs, and VPC resources.

**Attached Policies:**

| Policy Name | Type | Purpose |
|---|---|---|
| `AmazonEC2ContainerRegistryPowerUser` | AWS Managed | Push/pull Docker images to/from ECR |
| `AmazonS3FullAccess` | AWS Managed | Store build artifacts and cache |
| `AWSCodeBuildAdminAccess` | AWS Managed | Full CodeBuild permissions |
| `CloudWatchLogsFullAccess` | AWS Managed | Write build logs to CloudWatch |
| `CodeBuildBasePolicy-*` | Customer Managed | Base permissions for CodeBuild project |
| `CodeBuildCloudWatchLogsPolicy-*` | Customer Managed | CloudWatch Logs write access |
| `CodeBuildCodeConnectionsSourceCredentialsPolicy-*` | Customer Managed | Access GitHub via CodeConnections |
| `CodeBuildVpcPolicy-wu` | Customer Managed | Access VPC resources (for private subnets) |

### 2. CodePipeline Service Role

**Role Name:** `AWSCodePipelineServiceRole-us-west-2-wu-codepipeline`

**Purpose:** Allows CodePipeline to orchestrate source, build, and deploy stages.

**Attached Policies:**

| Policy Name | Type | Purpose |
|---|---|---|
| `AmazonECS_FullAccess` | AWS Managed | Deploy to ECS (update services, task definitions) |
| `AWSCodePipelineServiceRole-*` | Customer Managed | Core pipeline permissions |

| Policy Name | Type | Purpose |
|---|---|---|
| `CodePipeline-CodeBuild-*` | Customer Managed | Trigger CodeBuild projects |
| `CodePipeline-CodeConnections-*` | Customer Managed | Connect to GitHub |
| `CodePipeline-ECSDeploy-*` | Customer Managed | Deploy to ECS service |
| `CodePipeline-ECSDeployWithPassRoles-*` | Customer Managed | Pass IAM roles to ECS tasks |

# 5.2 Main CodeBuild Configuration

Before configuration, make sure you create a builspec.yml file inside the project root directory with necessary stages configuration of the pipeline. Also connect your Code source (Github, Gitlab etc) with your aws account from codebuild console.

## Project Configuration

**Project Name:** `wu-nodejs-app`

**Source Configuration:**

- **Provider:** GitHub
- **Repository:** `WajahatullahSE/NodeJS-Deployment-on-AWS-ECS-EC2-using-CodePipeline`
- **Branch:** `main`
- **Git Clone Depth:** 1 (shallow clone for faster builds)
- **Webhook:** Disabled (triggered by CodePipeline instead)

## Environment Configuration

**Build Environment:**

- **Image:** `aws/codebuild/amazonlinux-x86_64-standard:5.0`
- **Environment Type:** Linux Container
- **Compute:** EC2 (2 vCPUs, 4 GiB memory)
- **Privileged Mode: Enabled** (required for Docker builds)
- **Service Role:** `CodeBuild-ECSEC2-ServiceRole-wu`

**Why Privileged Mode:** Required to run Docker daemon inside CodeBuild container for building and pushing images.

## VPC Configuration

**Purpose:** Build within VPC to access private resources (if needed) and use same network as ECS cluster.

- **VPC:** `vpc-0120e4676180eac99` (wu2-vpc)
- **Subnets:** Both private subnets where ECS instances are hosted
  - `subnet-0e68acf734959ff77` (wu2-private-sn-1)
  - `subnet-03448c8e2944c9267` (wu2-private-sn-2)

- **Security Group:** `sg-031913a7b453ff86b` (wu2-ecs-sg)

**Note:** VPC configuration allows CodeBuild to access resources in private subnets if needed, though not strictly required for ECR push operations.

### Environment Variables

| Name | Value | Type |
|---|---|---|
| `ACCOUNT_ID` | 504649076991 | Plaintext |
| `REGION` | us-west-2 | Plaintext |
| `REPOSITORY_NAME` | wu-repo/task07 | Plaintext |

**Usage:** These variables are used in `buildspec.yml` to construct ECR repository URI and authenticate to ECR.

### Build Specification

**Buildspec:** Uses `buildspec.yml` from repository root directory.

**Key Actions Performed:**

- Authenticate to Amazon ECR
- Build Docker image from Dockerfile
- Tag image with commit SHA and `latest`
- Push both tags to ECR
- Generate `imagedefinitions.json` for ECS deployment

### Artifacts

**Type:** Amazon S3 (implicit via CodePipeline)

**Output:** `imagedefinitions.json` file. ECS uses this file to determine which Docker image to deploy.

## 5.3 CodePipeline Configuration

### Pipeline Settings

**Pipeline Name:** `wu-codepipeline`

**Pipeline Type:** V2

**Execution Mode:** QUEUED (sequential execution)

**Artifact Location:** S3 bucket `codepipeline-us-west-2-*` (auto-created)

**Service Role:** `AWSCodePipelineServiceRole-us-west-2-wu-codepipeline`

### Stage 1: Source

**Purpose:** Retrieve latest code from GitHub repository.

**Configuration:**

| Parameter | Value |
|---|---|
| **Provider** | GitHub (via GitHub App) |
| **Connection** | `arn:aws:codeconnections:...:connection/03860c98-...` |

| Parameter | Value |
|---|---|
| **Repository** | `WajahatullahSE/NodeJS-Deployment-on-AWS-ECS-EC2-using-CodePipeline` |
| **Branch** | `main` |
| **Output Format** | `CODE_ZIP` |
| **Change Detection** | Enabled (automatic trigger on push) |
| **Retry on Failure** | Enabled |

**Trigger Configuration:**

- **Type:** No filter (triggers on all pushes to `main` branch)

**Output Artifact:** Source code archive passed to Build stage.

## Stage 2: Build

**Purpose:** Build Docker image, push to ECR, and generate deployment artifact.

**Configuration:**

| Parameter | Value |
|---|---|
| **Provider** | AWS CodeBuild |
| **Project** | `wu-nodejs-app` |
| **Input Artifact** | Source code from Stage 1 |
| **Output Artifact** | `imagedefinitions.json` |
| **Retry on Failure** | Enabled |

**Build Process:**

1. Authenticate to ECR
2. Build Docker image: `docker build -t <repo-uri>:latest .`
3. Tag image with commit SHA
4. Push both tags to ECR
5. Generate `imagedefinitions.json`

**Build Duration:** Typically 2-5 minutes depending on image size and dependencies.

## Stage 3: Deploy

**Purpose:** Deploy new Docker image to ECS service with rolling update.

**Configuration:**

| Parameter | Value |
|---|---|
| **Provider** | Amazon ECS |
| **Cluster** | `wu2-ecs-cluster` |
| **Service** | `wu2-ecs-cluster-service` |
| **Input Artifact** | `imagedefinitions.json` from Build stage |
| **Automatic Rollback** | Enabled on stage failure |
| **Retry on Failure** | Disabled |

**Deployment Process:**

1. CodePipeline reads `imagedefinitions.json`
2. Creates new ECS task definition revision with updated image URI
3. Updates ECS service to use new task definition
4. ECS performs rolling deployment:
    - Starts new tasks with updated image
    - Waits for tasks to become healthy in ALB
    - Stops old tasks
    - Repeats until desired count reached

**Deployment Strategy:**

- **Minimum Healthy:** 50% (at least 1 task running during deployment)
- **Maximum:** 100% (no more than 2 tasks total during deployment)

**Rollback:** If deployment fails (e.g., unhealthy tasks, timeout), CodePipeline automatically rolls back to previous task definition revision.

# 6. Testing & Validation

## 1. Modify Application Code

**Objective:** Trigger automated deployment through code change.

**Steps:**

- Navigate to repository directory (already cloned)
- Edit `index.js` to update version and commit.

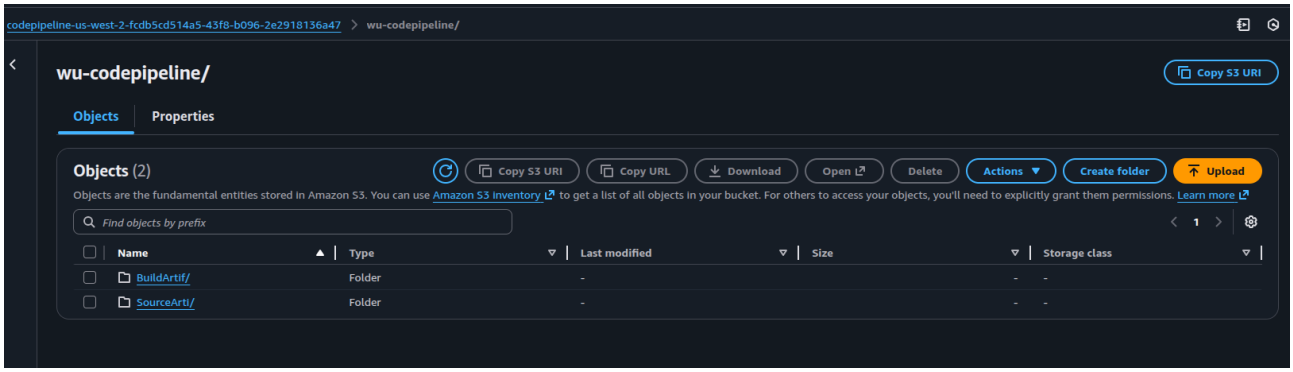
**Expected:** Pipeline automatically triggers within 30 seconds.

## 2. Monitor Pipeline Execution

**Navigate to:** AWS Console → CodePipeline → `wu-codepipeline`

### Source Stage

- **Status:** In Progress → Succeeded (green checkmark)
- **Duration:** ~10-30 seconds
- **Action:** Retrieved latest code from GitHub
- **Output:** Source artifact created



**Build Stage**

- **Status:** In Progress → Succeeded (green checkmark)
- **Duration:** ~2-5 minutes
- **Actions Performed:**
    - Logged in to Amazon ECR
    - Built Docker image from Dockerfile
    - Tagged image with commit SHA and `latest`
    - Pushed both tags to ECR
    - Generated `imagedefinitions.json`
- **Verification:** Check ECR console for new image with commit SHA tag

**Deploy Stage**

- **Status:** In Progress → Succeeded (green checkmark)
- **Duration:** ~3-5 minutes
- **Actions Performed:**
    - Created new ECS task definition revision
    - Updated ECS service with new task definition
    - Performed rolling deployment

**ECS Service Events:**

- Service started new tasks with updated image
- New tasks registered with ALB target group
- Health checks passed (3 consecutive successes)
- Old tasks deregistered and stopped
- Service reached steady state

## 3. Rolling Deployment Overview

**Configuration:**

- Desired Count: 2 tasks
- Minimum Healthy: 50% (keeps 1 task running)
- Maximum: 100% (no more than 2 tasks total)

**Deployment Flow:**

1. Start new task with updated image
2. Wait for ALB health checks to pass
3. Stop old task
4. Repeat for second task
5. Deployment complete

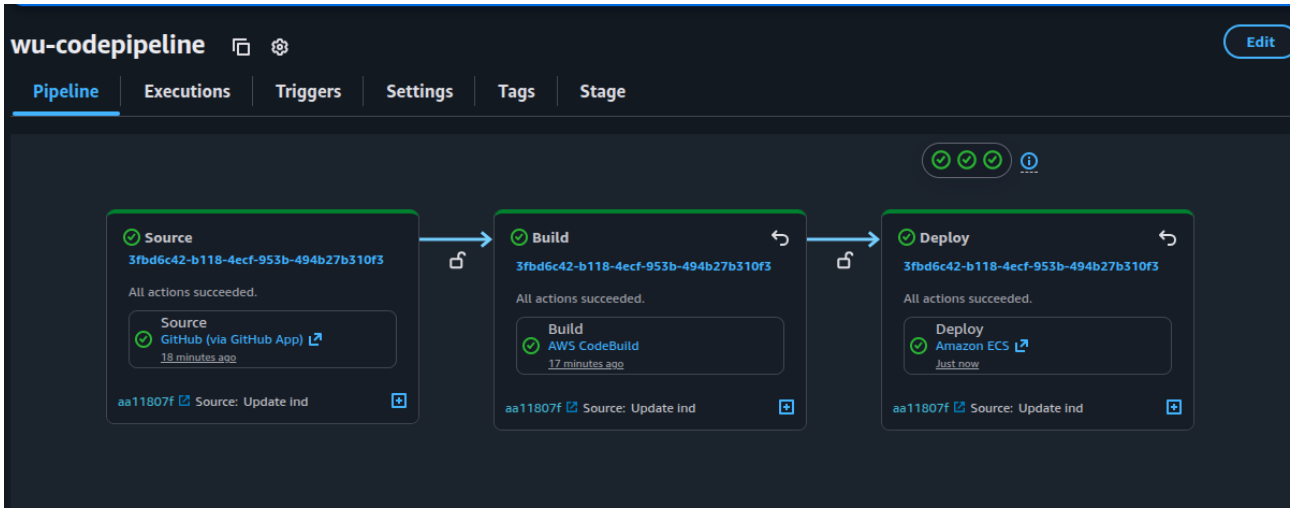
**Result:** Zero downtime - at least one healthy task always serving traffic.

## 4. Verify Deployment

**Access Application via ALB:**

- Navigate to EC2 → Load Balancers → wu2-alb
- Copy DNS name
- Open in browser: `http://<alb-dns-name>`

- **Expected:** Updated page showing "Version 2"





# 7. Troubleshooting Guide: Common Errors and Solutions

Below is a table of errors encountered during the deployment, their root causes, and the steps taken to resolve them.

Common Errors and Solutions

| Error | Root Cause | Solution |
|---|---|---|
| `EMPTY_CAPACITY_PROVIDER`<br><br>`Service was unable to place a task` | EC2 instances were not registered with the ECS cluster due to missing public IPv4 addresses. | 1. Edit the **Launch Template** and enable **Auto-assign Public IPv4** in the network settings.<br>2. Update the **Auto Scaling Group** to use the new launch template version.<br>3. Terminate existing instances to force a refresh. |
| **504 Bad Gateway**<br><br>ALB target group shows unhealthy targets | ECS instances were not registered with the target group, or the application was not running. | 1. Verify that ECS tasks are running (`ECS > Tasks`).<br>2. Check the **Target Group Health Checks** (ensure the path and port are correct). |

| Error | Root Cause | Solution |
| --- | --- | --- |
| | | 3. Verify the **Security Group** rules for ECS instances. |
| **502 Bad Gateway**<br><br>Incorrect inbound rule for ECS instances | The inbound rule for the ECS security group was misconfigured (e.g., HTTP:80 instead of TCP:8081). | 1. Update the **ECS Instance Security Group** to allow **TCP:8081** from the ALB security group (`wu-alb-sg`).<br><br>2. Ensure the **Target Group** is configured to use port **8081**. |
| `RESOURCE:MEMORY`<br><br>`TaskFailedToStart` | The `t3.small` instance type did not have sufficient memory for the task. | 1. Edit the **Launch Template** to use a larger instance type (e.g., `t3.medium`).<br><br>2. Update the **Auto Scaling Group** to use the new launch template version.<br><br>3. Terminate existing instances to force a refresh. |
| **ALB shows 504/502 errors**<br><br>Tasks not running | ECS service was not deploying tasks due to misconfiguration or insufficient capacity. | 1. Check **ECS Service Events** for errors.<br><br>2. Ensure the **Desired Task Count** is set correctly.<br><br>3. Force a **new deployment** in the ECS service. |
| **ECS tasks stuck in** `PENDING` **or** `PROVISIONING` | Insufficient resources (CPU/Memory) or misconfigured task definition. | 1. Verify the **Task Definition** (CPU, Memory, and Port Mappings).<br><br>2. Check **ECS Service Events** for detailed error messages.<br><br>3. Ensure the **ECS Cluster** has registered container instances. |
| **ALB health checks failing**<br><br>Targets remain unhealthy | The application was not responding to health check requests on the specified path/port. | 1. Verify the **Health Check Path** in the Target Group (e.g., `/`).<br><br>2. Check **container logs** for errors (`ECS > Tasks > Logs`).<br><br>3. Ensure the application is listening on the correct port (`8081`). |
| **ECS instances not registering with the cluster** | Incorrect **User Data** script or missing IAM permissions. | 1. Verify the **User Data** script in the Launch Template:<br><br>`#!/bin/bash`<br><br>`echo ECS_CLUSTER=wu-task7-cluster >> /etc/ecs/ecs.config`<br><br>2. Ensure the **IAM Instance Profile** (`ecsInstanceRole`) is attached. |
| **Docker image pull errors** | Incorrect ECR | 1. Verify the **ECR image URI** in the Task |

| Error | Root Cause | Solution |
| --- | --- | --- |
| Tasks fail to start | image URI or missing permissions for the ECS task execution role. | Definition.<br><br>2. Ensure the **ECS Task Execution Role** (`ecsTaskExecutionRole`) has the `AmazonECSTaskExecutionRolePolicy` policy attached. |
| **"Insufficient permissions to access ECS"** error in CodePipeline deploy stage | The **CodePipeline service role** (`AWSCodePipelineServiceRole-us-west-2-wu-codepipeline`) lacks permissions to interact with ECS. | Attach the necessary ECS permissions to the CodePipeline service role. Attach **ECSfullAccess** policy to the role. |