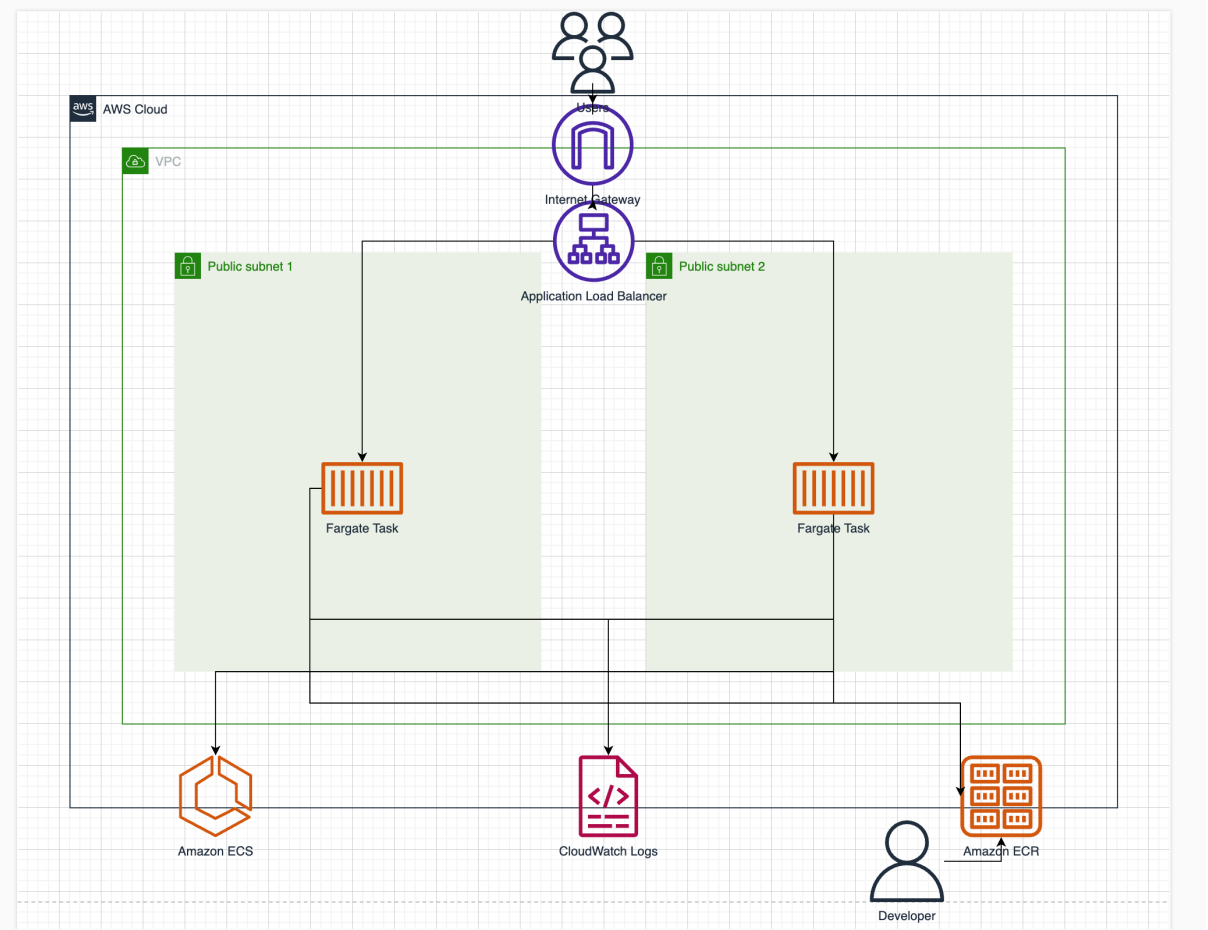
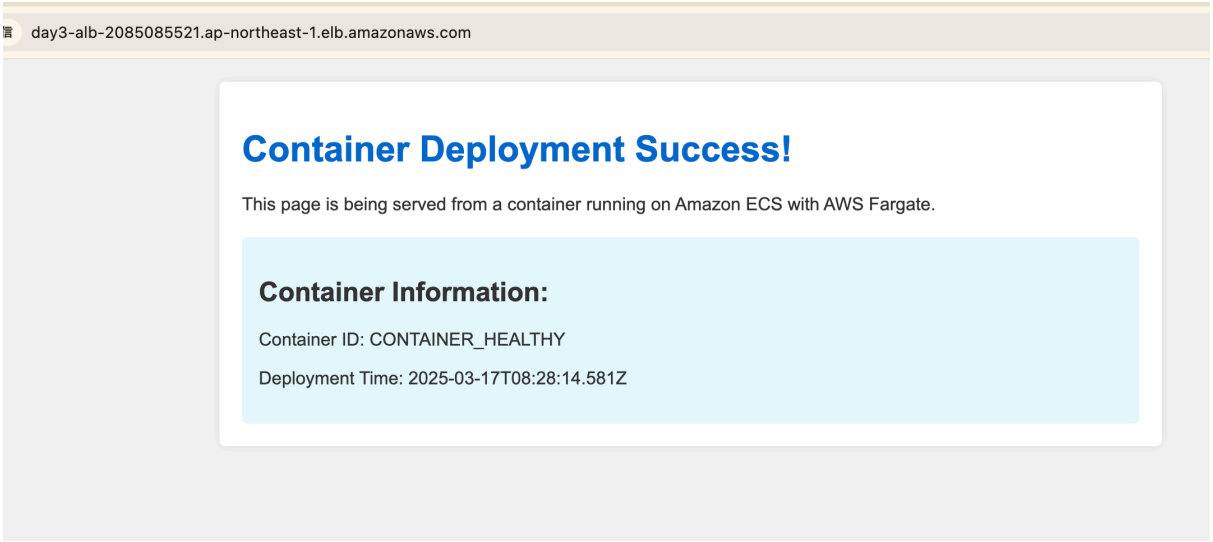
# Day 3 Real folw






# Mermaid Diagrams for Day 3 Challenge

Here are several mermaid diagrams to visualize different aspects of today's container orchestration challenge:

## 1. Overall Architecture Diagram

```
graph TD
    Developer[Developer] -->|Push Code & Docker Image| ECR[Amazon EC
R]
    ECR -->|Pull Image| ECS[Amazon ECS]
    ECS -->|Run Containers| Fargate[AWS Fargate]
    Fargate -->|Host Containers| Container[Container Instances]
    ALB[Application Load Balancer] -->|Route Traffic| Container
    User[End User] -->|HTTP Request| ALB
    CloudWatch[CloudWatch Logs] <-->|Monitor| Container

    subgraph "Container Registry"
        ECR
    end

    subgraph "Container Orchestration"
        ECS
        Fargate
```

```
    Container
  end

  subgraph "Load Balancing"
    ALB
  end

  subgraph "Monitoring"
    CloudWatch
  end
```

## 2. Deployment Workflow

```
sequenceDiagram
  participant Dev as Developer
  participant Docker as Docker CLI
  participant ECR as Amazon ECR
  participant ECS as Amazon ECS
  participant Fargate as AWS Fargate
  participant ALB as Application Load Balancer

  Dev->>Docker: docker build --platform linux/amd64 -t day3-app .
  Docker->>Docker: Build container image
  Dev->>Docker: docker tag day3-app:latest [ECR_URI]:latest
  Dev->>ECR: docker push [ECR_URI]:latest
  ECR->>ECR: Store container image
  Dev->>ECS: Create/update Task Definition
  Dev->>ECS: Create Service
  ECS->>Fargate: Schedule Tasks
  Fargate->>Fargate: Provision compute
  Fargate->>ECR: Pull container image
  Fargate->>Fargate: Start containers
  ALB->>Fargate: Health checks
  ALB->>Fargate: Route traffic to healthy containers
```

## 3. Networking Architecture

```
graph TD
    Internet((Internet)) ←→|HTTP/80| ALB[Application Load Balancer]

    subgraph "VPC"
        subgraph "Public Subnet 1"
            ALB
            Fargate1[Fargate Task]
        end

        subgraph "Public Subnet 2"
            Fargate2[Fargate Task]
        end
    end

    ALB →|Target Group| Fargate1
    ALB →|Target Group| Fargate2

    Fargate1 →|Pull Images| ECR[Amazon ECR]
    Fargate2 →|Pull Images| ECR

    Fargate1 →|Logs| CloudWatch[CloudWatch Logs]
    Fargate2 →|Logs| CloudWatch
```

## 4. Container Architecture

```
graph TD
    subgraph "Fargate Task"
        subgraph "Container"
            Nginx[NGINX Web Server]
            HTML[Static HTML Files]
            Health[Health Check Endpoint]
        end
    end

    ALB[Application Load Balancer] →|"/"| Nginx
    ALB →|"/health.html"| Health
    Nginx → HTML
```

```
User((User)) →|HTTP Request| ALB
```

## 5. Troubleshooting Flow

```
flowchart TD
    A[Service Creation Fails] → B{Check Task Status}
    B →|Stopped| C[Check CloudWatch Logs]
    B →|Pending| D[Check Network Configuration]
    B →|Running but Unhealthy| E[Check Health Checks]

    C → F{Error Type?}
    F →|Exec Format Error| G[Architecture Mismatch]
    F →|Container Exited| H[Application Error]
    F →|Permission Denied| I[IAM Role Issue]

    G → J[Rebuild Image with --platform linux/amd64]
    H → K[Fix Application Code]
    I → L[Update Task Execution Role]

    D → M[Verify Public Subnets & Security Groups]
    E → N[Verify Health Check Path Exists]

    J → O[Create New Task Definition Revision]
    K → O
    L → O
    M → P[Update Service Network Configuration]
    N → Q[Update Target Group Settings]

    O → R[Create New Service]
    P → R
    Q → R
```

## 6. Cost Optimization Diagram

```
graph TD
    Cost[Cost Optimization] → Compute[Compute Costs]
```

```
    Cost --> Network[Network Costs]
    Cost --> Storage[Storage Costs]

    Compute --> Fargate[Fargate Pricing]
    Fargate --> TaskSize[Right-size Task CPU/Memory]
    Fargate --> ARM[Consider ARM64 for Lower Cost]

    Network --> ALB[Load Balancer Costs]
    ALB --> SharedALB[Share ALB Across Services]

    Storage --> ECR[ECR Repository]
    ECR --> Lifecycle[Image Lifecycle Policies]
    ECR --> ImageSize[Optimize Image Size]

    TaskSize --> AutoScale[Auto Scaling Based on Demand]
    SharedALB --> ReduceEndpoints[Consolidate Endpoints]
    Lifecycle --> RetainLimit[Limit Retained Images]
```

These diagrams provide a comprehensive visual representation of the container orchestration challenge, covering architecture, workflow, networking, troubleshooting, and cost optimization aspects.

## Step 1: Create a Simple Web Application

1. Create the application directory structure:

```
mkdir -p Day3/app
cd Day3/app
```

2. Create a simple HTML file (`index.html`):

```
<!DOCTYPE html>
<html>
<head>
    <title>AWS 14-Day Challenge: Day 3</title>
    <style>
```

```
        body {
            font-family: Arial, sans-serif;
            margin: 0;
            padding: 20px;
            background-color: #f4f4f4;
            color: #333;
        }
        .container {
            max-width: 800px;
            margin: 0 auto;
            background-color: white;
            padding: 20px;
            border-radius: 5px;
            box-shadow: 0 0 10px rgba(0,0,0,0.1);
        }
        h1 {
            color: #0066cc;
        }
        .info {
            background-color: #e6f7ff;
            padding: 15px;
            border-radius: 5px;
            margin-top: 20px;
        }
    </style>
</head>
<body>
    <div class="container">
        <h1>Container Deployment Success!</h1>
        <p>This page is being served from a container running on Amazon ECS with AWS Fargate.</p>

        <div class="info">
            <h2>Container Information:</h2>
            <p>Container ID: <span id="container-id">Loading...</span></p>
            <p>Deployment Time: <span id="timestamp"></span></p>
        </div>
```

```
    </div>

    <script>
        // Display current timestamp
        document.getElementById('timestamp').textContent = new Date().t
olISOString();

        // Fetch container ID from health endpoint
        fetch('/health.html')
            .then(response => response.text())
            .then(data => {
                document.getElementById('container-id').textContent = data.t
rim();
            })
            .catch(error => {
                document.getElementById('container-id').textContent = "Error
fetching container ID";
            });
    </script>
</body>
</html>
```

3. Create a health check file ( `health.html` ):

```
CONTAINER_HEALTHY
```

## Step 2: Containerize the Application

1. Create a Dockerfile:

```
FROM nginx:alpine

COPY index.html /usr/share/nginx/html/
COPY health.html /usr/share/nginx/html/

EXPOSE 80

CMD ["nginx", "-g", "daemon off;"]
```

2. Build the Docker image locally:

```
docker build -t day3-app .
```

```
# Build the image specifically for AMD64 platform
docker build --platform linux/amd64 -t day3-app .
```

3. Test the container locally:

```
docker run -p 8080:80 day3-app
```

4. Open your browser to `http://localhost:8080` to verify it works

# Part 2: Setting Up AWS Infrastructure Manually

## Step 1: Create an ECR Repository

1. Open the AWS Management Console
2. Navigate to Amazon ECR
3. Click "Create repository"
4. Name: `day3-app-repo`
5. Click "Create repository"

# Part 2: Setting Up AWS Infrastructure Manually (Updated for Console)

## Step 2: Push Your Image to ECR (Using AWS Console and CLI)

1. **Get your AWS account ID and region**:

```
# Get your AWS account ID
AWS_ACCOUNT_ID=$(aws sts get-caller-identity --query Account --ou
tput text)
```

```
# Get your AWS region
AWS_REGION=$(aws configure get region)

# Verify the values
echo "AWS Account ID: $AWS_ACCOUNT_ID"
echo "AWS Region: $AWS_REGION"
```

2. **Open the ECR Console**:
   - Go to AWS Management Console
   - Search for "ECR" and click on "Elastic Container Registry"
   - Click on your repository (day3-app-repo)
   - Click the "View push commands" button in the top right
3. **Follow the push commands displayed in the console**:
   - The console will show you customized commands for your repository
   - These commands will include:

```
# Log in to ECR
aws ecr get-login-password --region $AWS_REGION | docker login
--username AWS --password-stdin $AWS_ACCOUNT_ID.dkr.ecr.$A
WS_REGION.amazonaws.com



# Tag your image
docker tag day3-app-repo:latest $AWS_ACCOUNT_ID.dkr.ecr.$AW
S_REGION.amazonaws.com/day3-app-repo:latest

# Push your image
docker push $AWS_ACCOUNT_ID.dkr.ecr.$AWS_REGION.amazona
ws.com/day3-app-repo:latest
```

4. **Verify the image was pushed successfully**:
   - In the ECR console, refresh the repository page
   - You should see your image listed with the "latest" tag

- Note the image URI - you'll need this for the ECS task definition

## Step 3: Create an ECS Cluster (Using AWS Console)

1. Navigate to Amazon ECS in the AWS Console
2. Click "Create Cluster"
3. Select "Networking only" (Fargate)
4. Fill in the following details:
    - Cluster name: `day3-cluster`
    - Create VPC: Leave unchecked (we'll use the default VPC)
    - Tags: Add a tag with Key=Purpose, Value=Training
5. Click "Create"
6. Wait for the cluster to be created (should take less than a minute)
7. You'll see a success message - click "View Cluster"

## Step 4: Create a Task Definition (Using AWS Console)

1. In the ECS console, click on "Task Definitions" in the left sidebar
2. Click "Create new Task Definition"
3. Select "Fargate" as the launch type compatibility
4. Click "Next step"
5. Fill in the following details:
    - Task Definition Name: `day3-task`
    - Task Role: `None` (or select an appropriate role if needed)
    - Operating system family: `Linux`
    - Task execution role: `Create new role` (this will create the necessary permissions)
    - Task size:
        - Task memory: `0.5GB`
        - Task CPU: `0.25 vCPU`
    - Container Definitions:

- Click "Add container"
- Container name: `app-container`
- Image: Paste the ECR image URI from Step 2 (should look like: `$AWS_ACCOUNT_ID.dkr.ecr.$AWS_REGION.amazonaws.com/day3-app-repo:latest`)
- Memory Limits: Leave as default
- Port mappings: `80`
- Environment: Leave as default
- Storage and Logging:
  - Check "Auto-configure CloudWatch Logs"
- Click "Add"

6. Review the settings and click "Create"
7. You'll see a success message - your task definition is now registered

## Step 5: Create a Service (Using AWS Console)

1. Go back to your ECS cluster (click "Clusters" in the left sidebar, then select `day3-cluster`)
2. In the "Services" tab, click "Create"
3. Configure the service:
   - Launch type: `FARGATE`
   - Task Definition: Select `day3-task` (select the latest revision)
   - Cluster: Ensure `day3-cluster` is selected
   - Service name: `day3-service`
   - Service type: `REPLICA`
   - Number of tasks: 1
   - Minimum healthy percent: `50`
   - Maximum percent: `200`
   - Deployment circuit breaker: Check "Enable"
   - Click "Next step"
4. Configure the network:

- Cluster VPC: Select your default VPC
- Subnets: Select at least two public subnets
- Security groups:
  - Click "Create new security group"
  - Security group name: `day3-ecs-sg`
  - Description: `Security group for Day 3 ECS tasks`
  - Add a rule: Type=HTTP, Source=Anywhere
  - Click "Create"
- Auto-assign public IP: `ENABLED`
- Click "Next step"

5. Configure the load balancer:
   - Load balancer type: `Application Load Balancer`
   - Load balancer name:
     - Select "Create a new load balancer"
     - Name: `day3-alb`
   - Container to load balance:
     - Select "app-container:80:80"
     - Click "Add to load balancer"
   - Target group name:
     - Create new target group: `day3-tg`
   - Path pattern: `/`
   - Health check path: `/health.html`
   - Click "Next step"
6. Configure Auto Scaling (optional):
   - For this exercise, select "Do not adjust the service's desired count"
   - Click "Next step"
7. Review your settings and click "Create Service"

8. Click "View Service" to monitor the deployment

## Step 6: Test Your Deployment (Using AWS Console)

1. Wait for the service to be created and tasks to start running
   - In the ECS console, monitor the "Tasks" tab of your service
   - Wait until at least one task shows "RUNNING" status (may take 1-2 minutes)
2. Find your load balancer DNS name:
   - In the ECS service details, click on the "Load Balancing" tab
   - Click on the Target Group name
   - In the Target Group details, click on the "Load Balancer" tab
   - Click on the Load Balancer name
   - In the Load Balancer details, copy the "DNS name" from the Description tab
3. Open the DNS name in your browser
   - Example: `http://day3-alb-123456789.us-east-1.elb.amazonaws.com`
   - You should see your containerized application running
4. Verify health checks are passing:
   - In the EC2 console, go to "Target Groups"
   - Select your target group (`day3-tg`)
   - Check the "Targets" tab
   - Verify that your targets show "healthy" status

## Step 7: Monitor Your Application (Using AWS Console)

1. View container logs:
   - Go to CloudWatch console
   - Click on "Log groups" in the left sidebar
   - Find and click on the log group for your ECS service (should be named `/ecs/day3-task`)
   - Click on a log stream to view container logs

2. Monitor ECS service metrics:
    - In the ECS console, select your cluster and service
    - Click on the "Metrics" tab to view service metrics
    - You can see CPU and memory utilization
3. Set up a CloudWatch alarm:
    - In the CloudWatch console, click "Alarms" → "Create alarm"
    - Click "Select metric"
    - Navigate to ECS → ClusterName, ServiceName
    - Select the CPUUtilization metric for your service
    - Click "Select metric"
    - Set threshold to 80%
    - Click "Next"
    - Create a new SNS topic and add your email
    - Click "Create topic"
    - Click "Next"
    - Name the alarm "Day3-High-CPU"
    - Click "Next"
    - Click "Create alarm"

## Step 8: Make Changes to Your Application

1. Update your local `index.html` file with a new message or design
2. Rebuild and push the Docker image:

```
# Build the updated image
docker build -t day3-app:v2 .

# Tag the new image
docker tag day3-app:v2 $AWS_ACCOUNT_ID.dkr.ecr.$AWS_REGION.a
mazonaws.com/day3-app-repo:v2

# Push the new image
```

```
docker push $AWS_ACCOUNT_ID.dkr.ecr.$AWS_REGION.amazonaws.com/day3-app-repo:v2
```

3. Update the ECS service:
    - In the ECS console, go to "Task Definitions"
    - Select your task definition family
    - Click "Create new revision"
    - Click "Add container"
    - Update the image URI to point to your new image (with `:v2` tag)
    - Click "Update"
    - Click "Create"
    - Go to your ECS service
    - Click "Update"
    - Select the new task definition revision
    - Click "Skip to review"
    - Click "Update Service"
4. Monitor the deployment:
    - Watch the "Tasks" tab to see new tasks being created and old ones stopping
    - Once complete, refresh your application URL to see the changes

## Step 9: Clean Up Resources (Using AWS Console)

When you're done with the challenge:

1. Delete the ECS service:
    - Go to the ECS console → Clusters → day3-cluster
    - Select the service
    - Click "Delete"
    - Confirm by typing "delete"
    - Click "Delete"

2. Delete the Load Balancer:
    - Go to EC2 console → Load Balancers
    - Select your ALB
    - Click "Actions" → "Delete"
    - Confirm deletion
3. Delete the Target Group:
    - Go to EC2 console → Target Groups
    - Select your target group
    - Click "Actions" → "Delete"
    - Confirm deletion
4. Delete the Task Definition:
    - Go to ECS console → Task Definitions
    - Select your task definition
    - Select all revisions
    - Click "Actions" → "Deregister"
    - Confirm deregistration
5. Delete the ECS Cluster:
    - Go to ECS console → Clusters
    - Select your cluster
    - Click "Delete Cluster"
    - Confirm deletion
6. Delete the ECR Repository:
    - Go to ECR console
    - Select your repository
    - Click "Delete"
    - Type "delete" to confirm
    - Click "Delete"

7. Delete CloudWatch Log Groups:
    - Go to CloudWatch console → Log Groups
    - Select the log group for your ECS tasks
    - Click "Actions" → "Delete log group"
    - Confirm deletion
8. Delete CloudWatch Alarms:
    - Go to CloudWatch console → Alarms
    - Select any alarms you created
    - Click "Actions" → "Delete"
    - Confirm deletion