Skill Lab - Cloud Computing - CI/CD with GitHub Actions

Session - 1 - Introduction and Continuous Integration (CI)

Objectives

1. Understand the fundamentals of CI/CD
2. Learn how to create and push a Node.js Application
3. Setup a GitHub Actions workflow for CI (code build and run)
4. Test Workflow Automation using GitHub Actions

Introduction

CI/CD is a set of **automation practices** used in modern software engineering to **build, test, and deliver software faster and more reliably**. It forms the backbone of the **DevOps** culture, where development and operations teams work closely to shorten the software development lifecycle.

Continuous Integration

Definition:

Continuous Integration is the practice where **developers frequently integrate their code into a shared repository**, usually several times a day.

How It Works:

- Developers **commit code** to a version control system like Git.
- An **automated build process** is triggered.
- **Automated tests** (unit tests, integration tests) are executed.
- Developers get **immediate feedback** if the new code breaks anything.

Goal:

Catch bugs early, reduce integration problems, and improve software quality.

Example Tools:

GitHub Actions, Jenkins, GitLab CI, Travis CI, CircleCI

Continuous Delivery

Definition:

Continuous Delivery is the extension of CI where code changes are automatically **tested, built, and prepared for a manual or scheduled release** to production.

Key Features:

- Ensures the application is **always in a deployable state**.
- Requires **automated testing, packaging, and staging** environments.
- Final deployment can be triggered manually (e.g., by a release manager).

Goal:

Reduce deployment risk, shorten release cycles, and improve productivity.

Continuous Deployment

Definition:

Continuous Deployment takes Continuous Delivery a step further by **automatically deploying every code change that passes the CI pipeline** to production without manual intervention.

Benefits:

- Faster innovation and delivery to end-users.
- Immediate feedback from production usage.
- Requires very high-quality automated testing to avoid bugs reaching users.

CI/CD Pipeline - The WorkFlow

A typical CI/CD pipeline involves the following stages:

1. **Code Commit** – Developer pushes code to Git.
2. **Build Stage** – The code is compiled and built into artifacts.
3. **Test Stage** – Automated tests are executed (unit, integration, etc.).
4. **Package** – The tested code is packaged (e.g., Docker image).

5. **Deploy** – Code is deployed to a staging or production environment.
6. **Monitoring** – Logs and metrics are analyzed post-deployment.

Benefits of CI/CD

- **Faster Time-to-Market**: Software reaches users quickly.
- **Reduced Risk**: Frequent testing and deployment catch errors early.
- **Better Collaboration**: Teams work in shorter feedback loops.
- **Automation**: Reduces human errors and manual processes.
- **Customer Satisfaction**: Rapid delivery of features and fixes.

Popular CI/CD Tools

Purpose	**Tools**
CI/CD Pipelines	Jenkins, GitHub Actions, GitLab CI
Containerization	Docker
Orchestration	Kubernetes
Monitoring	Prometheus, Grafana, ELK Stack
Testing	JUnit, Selenium, Postman, Jest

Tools Used

- Git & GitHub
- Node.js & Express
- GitHub Actions for automation

I. GitHub and Git Setup

Step - 1 - Create a GitHub Account

- Open a browser and visit https://github.com
- Click **"Sign up"**.
- Enter your:
  - Email address
  - Password

    - Username

- Follow the on-screen instructions to verify your email address.
- Once verified, log in to your GitHub account.

Step - 2 - Create a Repository

- After logging into GitHub, click the **"+"** icon on the top-right corner and choose **"New repository"**.
- Fill in the repository details:
    - **Repository name**: `nodejs-ci-cd-demo`
    - **Description**: Optional
    - Select **Public** or **Private** depending on your need.
- Check **"Initialize this repository with a README"**.
- Click the **"Create repository"** button.

Step - 3 - Install Git

On Windows:

1. Download Git from https://git-scm.com/download/win
2. Run the installer and accept all default settings (make sure *Git Bash* is selected).
3. After installation, open **Git Bash** from the Start Menu.

On Linux:

```
sudo apt update
sudo apt install git
```

On MacOS:

Option 1 (recommended):

```
xcode-select --install
```

Option 2 (with Homebrew):

```
brew install git
```

Step - 4 - Configure Git

After installation, open your terminal (**Git Bash**, Linux Terminal, or macOS Terminal) and run:

```
git config --global user.name "Your Full Name"
git config --global user.email "your_email@example.com"
```

- This information will appear in your commits.
- The email should match your GitHub account to link commits properly.

Step - 5 - Clone Your Repository Locally

1. Go to your repository on GitHub:
   `https://github.com/YOUR_USERNAME/nodejs-ci-cd-demo`
2. Click the **"Code"** button → copy the **HTTPS URL**.
3. In terminal, run:

```
git clone https://github.com/YOUR_USERNAME/nodejs-ci-cd-demo.git
cd nodejs-ci-cd-demo
```

You're now inside your local project directory, ready to initialize your project and start version control.

II. Create a Node.js App

Step - 1 - Initialize the App

```
npm init -y
```

- This creates a default `package.json` file that defines your project's metadata (name, version, scripts, etc.).
- The `y` flag automatically accepts all default options.

```
npm install express
```

- Installs the Express.js library, which is used to build the web server.
- This adds `express` as a dependency in `package.json`.

Step - 2 - Create index.js file

Create a file named `index.js` and add the following code:

```
javascript
CopyEdit
const express = require('express');           // Import Express module
const app = express();                        // Create an Express app instance
const port = 3000;                            // Define the port to listen on

// Middleware to log request details
app.use((req, res, next) => {
  console.log(`[${new Date().toISOString()}] ${req.method} ${req.url}`);
  next(); // Call next middleware or route handler
});

app.use(express.json()); // Parse JSON bodies of incoming requests

// Define different routes
app.get('/', (req, res) => res.send('Welcome to the CI/CD Node.js App!'));

app.get('/about', (req, res) =>
  res.send('This is a sample Node.js app to demonstrate GitHub Actions CI/CD.'));

app.get('/status', (req, res) =>
  res.json({ status: 'OK', timestamp: new Date().toISOString() }));

app.get('/user/:username', (req, res) =>
  res.send(`Hello, ${req.params.username}!`));

app.post('/api/data', (req, res) => {
  const data = req.body;
```

```
    if (!data.name || !data.value) {
        return res.status(400).json({ error: 'Invalid data format' });
    }
    res.status(201).json({ message: 'Data received successfully', received: data });
});

app.get('/health', (req, res) => res.send('App is healthy and running!'));

// Start the server
app.listen(port, () => console.log(`Server running at http://localhost:${port}`));
```

Step - 3 - Configure Line Endings (CRLF vs LF)

Line endings differ between operating systems:

- **Windows** uses `CRLF`
- **Linux/macOS** uses `LF`

To ensure consistent behavior, especially for CI tools like GitHub Actions that run on Linux:

A. Use Git Config:

```
# On Windows
git config --global core.autocrlf true

# On Linux/macOS
git config --global core.autocrlf input
```

This ensures:

- Windows: LF → CRLF on checkout, CRLF → LF on commit
- Linux/macOS: Checkout as-is, convert CRLF → LF on commit

B. Use .gitattributes File:

Create a file named `.gitattributes` in the root directory:

```
* text=auto
*.js text eol=lf
```

This ensures:

- All `.js` files are stored in the repo with LF line endings
- Git handles line ending conversion based on system settings

Step - 4 - Update `package.json` Start Script

Edit the `package.json` file and add this under the `"scripts"` section:

```
"scripts": {
  "start": "node index.js"
}
```

This allows you to run the server using:

```
npm start
```

Instead of typing `node index.js` manually.

Step - 5 - Test App Locally

Start the application:

```
node index.js
```

Or, if you've added the script:

```
npm start
```

Visit in browser:

- http://localhost:3000/
- http://localhost:3000/about
- http://localhost:3000/status
- http://localhost:3000/user/YourName

- http://localhost:3000/health

III. Push to GitHub

```
git add .
```

- Stages all modified and new files for commit.

```
git commit -m "Initial Node.js app"
```

- Commits the staged changes with a meaningful message.

```
git push origin main
```

- Pushes your changes to the `main` branch on GitHub.

IV. Add GitHub Actions CI Workflow

Step - 1 - Create Workflow Directory

```
mkdir -p .github/workflows
```

- Creates the folder structure where GitHub Actions workflows are stored.

Step - 2 - Create CI Workflow File

Create a file at `.github/workflows/ci.yml` with the following contents:

```
name: Node.js CI

on:
  push:
    branches: [ main ]

jobs:
  build:
    runs-on: ubuntu-latest   # Use Linux environment
    steps:
    - name: Checkout Repository
      uses: actions/checkout@v3   # Checks out the source code

    - name: Setup Node.js
      uses: actions/setup-node@v3
      with:
        node-version: '16'       # Specify Node.js version

    - name: Install Dependencies
      run: npm install          # Installs all npm packages listed in package.json

    - name: Run the App (for testing only)
      run: npm start &          # Starts the server in the background
```

Step - 3 - Commit and Push Workflow

```
git add .github/workflows/ci.yml
git commit -m "Add GitHub Actions CI workflow"
git push origin main
```

V. View CI Pipeline on GitHub

- Go to your GitHub repository.
- Click the **Actions** tab.
- You'll see a workflow titled `Node.js CI` triggered by your push.
- Click it to view job steps like:
  - Code checkout
  - Node.js setup
  - Dependency installation
  - Application startup

Outcomes

- **Created and connected a GitHub repository** to manage their Node.js project source code using version control.
- **Developed a functional Node.js application** using Express.js with multiple API endpoints and middleware for request logging.
- **Configured a CI workflow using GitHub Actions**, enabling automated tasks (like dependency installation and app launch) to run on every code push to the main branch.

Session - 2 - Containerization and Continuous Delivery (CD)

Objectives

- Understand containerization using Docker
- Dockerize the Node.js app
- Set up AWS Elastic Container Registry (ECR)
- Create a GitHub Actions workflow to build and push Docker image to ECR

Introduction

1. Containerization

- **Definition:**

  Containerization is the process of packaging an application along with all its dependencies, libraries, configuration files, and runtime into a single **lightweight executable unit** called a **container**.

- **Tool Used:**

  Docker is the most widely-used containerization tool. It allows developers to build once and run anywhere, ensuring consistency across development, testing, and production environments.

- **Benefits:**
  - Environment consistency

  - Easy to replicate and scale
  - Lightweight compared to virtual machines
  - Fast boot/startup times

2. Orchestration

- **Definition:**

  Once you have multiple containers running (possibly across multiple servers), orchestration becomes necessary. **Container orchestration** is the automated management of containerized applications.

- **Tasks Handled:**
  - Deployment of containers
  - Scaling up/down based on load
  - Health monitoring and self-healing
  - Networking and service discovery
- **Popular Tools:**
  - **Kubernetes:** Open-source platform that automates container operations
  - **Amazon ECS (Elastic Container Service):** AWS's managed container orchestration service
  - **AWS Fargate:** Serverless compute engine for containers (you don't manage servers)

Workflow Diagram Overview: From Code to Deployment

```
Developer writes code (e.g., Node.js App)
        ↓
Dockerfile is written to define how the image is built
        ↓
GitHub Actions is configured to:
  - Build Docker image
  - Push image to AWS ECR (Elastic Container Registry)
        ↓
AWS ECS/Fargate pulls the image from ECR and deploys it
```

This pipeline ensures that whenever code is pushed to GitHub:

- Docker automatically builds a container image.
- That image is pushed to a central repository (ECR).
- AWS ECS or Fargate then deploys the new container to run in production.

I. Docker Installation

Windows:

1. **Download Docker Desktop**

- Visit: https://www.docker.com/products/docker-desktop
- Select the **Windows version** and download the installer.

2. **Install Docker**

- Run the downloaded `.exe` file.
- During installation, **enable the WSL2 backend**.
    - **WSL2 (Windows Subsystem for Linux v2)** enables high-performance Linux container support on Windows.
    - If WSL2 is not installed, Docker will **guide you** through the installation of:
        - WSL2 kernel update
        - Required Windows features
        - A default Linux distribution (e.g., Ubuntu)

3. **Complete Installation**

- Once setup is complete, **launch Docker Desktop**.
- The **Docker whale icon** 🐳 should appear in the system tray.
- Docker Desktop provides a GUI dashboard as well as CLI access via PowerShell or CMD.

4. **Verify Installation**

Open Command Prompt or PowerShell and type:

```
docker --version
```

Expected Output:

```
Docker version 24.0.x, build abc123
```

This confirms Docker is installed and the CLI is working.

MacOS:

Key Differences:

- No WSL2 is required.
- Docker Desktop for Mac runs Docker Engine using **Apple Hypervisor** (on Intel Macs) or **virtualization.framework** (on Apple Silicon).

Steps:

1. Download Docker Desktop for Mac from the same official site.
2. Run the `.dmg` installer and drag Docker into the Applications folder.
3. Launch Docker Desktop and allow required permissions.
4. Verify with:

   ```
   docker --version
   ```

Linux

Key Differences:

- No Docker Desktop GUI — only Docker Engine/CLI is installed.
- Installation is done via package manager.

Steps:

1. Update package list and install dependencies:

   ```
   sudo apt update
   sudo apt install ca-certificates curl gnupg
   ```

2. Add Docker’s GPG key:

   ```
   sudo install -m 0755 -d /etc/apt/keyrings
   curl -fsSL https://download.docker.com/linux/ubuntu/gpg | \
   ```

```
  sudo gpg --dearmor -o /etc/apt/keyrings/docker.gpg
```

3. Add Docker repository:

```
echo \
  "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/
docker.gpg] \
  https://download.docker.com/linux/ubuntu \
  $(lsb_release -cs) stable" | \
  sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
```

4. Install Docker Engine:

```
sudo apt update
sudo apt install docker-ce docker-ce-cli containerd.io
```

5. Verify:

```
docker --version
```

II. Dockerize Node.js App

Step - 1 - Create `Dockerfile`

A `Dockerfile` is a text document that contains all the commands to assemble an image. Here's a line-by-line breakdown of the Dockerfile content:

```
# 1. Use official Node image (based on Alpine Linux for minimal size)
FROM node:16-alpine
```

- **`FROM`**: Specifies the base image for your container.
- `node:16-alpine` uses Node.js v16 on Alpine Linux, a small, fast image that's great for production.

```
# 2. Set the working directory in the container
WORKDIR /app
```

- `WORKDIR` : All following commands (`COPY` , `RUN` , etc.) will be run in this directory.
- It isolates your app files to `/app` within the container.

```
# 3. Copy only package files first
COPY package*.json ./
RUN npm install
```

- `COPY` : Copies the `package.json` and `package-lock.json` files into the container.
- `RUN npm install` : Installs dependencies before copying the app source code. This allows Docker to cache the layer and avoid reinstalling dependencies if source files change but `package.json` doesn't.

```
# 4. Copy remaining app source code
COPY . .
```

- Copies the rest of the application code into the container.

```
# 5. Expose the port your app listens on
EXPOSE 3000
```

- Tells Docker that the container will listen on port 3000 (used by the Node.js app).

```
# 6. Start the app using npm
CMD ["npm", "start"]
```

- `CMD` : Specifies the default command to run when the container starts.

Step - 2 - Test Build Locally

After the Dockerfile is created, you can build and run your Docker image on your machine to ensure it's working.

Build the Docker Image

```
docker build -t nodejs-ci-cd-demo .
```

- `docker build`: Builds a Docker image from the current directory (indicated by `.`).
- `t nodejs-ci-cd-demo`: Tags the image with the name `nodejs-ci-cd-demo`, which makes it easier to refer to later.

Run the Docker Container

```
docker run -p 3000:3000 nodejs-ci-cd-demo
```

- `docker run`: Runs a container based on the image you built.
- `p 3000:3000`: Maps port `3000` of the container to port `3000` on your local machine, allowing access via browser or curl.
- `nodejs-ci-cd-demo`: The name/tag of the Docker image you just built.

Access the Application

- Visit http://localhost:3000 in your browser.
- You should see the homepage response from your Node.js app (e.g., "Welcome to the CI/CD Node.js App!")

III. Set Up AWS ECR & AWS CLI for Docker Image Deployment

Step - 1 - Log Into AWS Console and Create ECR Repository

1. Go to https://console.aws.amazon.com/ and sign in with your **AWS root or IAM account**.
2. In the search bar, type **ECR** and open **Elastic Container Registry**.
3. Click on **"Create repository"**.
4. Fill in:
    - **Repository name:** `nodejs-ci-cd-repo`
    - Visibility: **Private**
5. Leave other defaults as is and click **Create repository**.

Step - 2 - Install AWS CLI Based on Your Operating System

Windows:

- Download the AWS CLI installer: https://aws.amazon.com/cli/
- Run the installer, follow the instructions.
- After installation, verify by running in Command Prompt:

```
aws --version
```

MacOS:

Use **Homebrew** (recommended):

```
brew install awscli
```

Then verify:

```
aws --version
```

Linux:

```
curl "https://awscli.amazonaws.com/awscli-exe-linux-x86_64.zip" -o "awscliv2.zip"
unzip awscliv2.zip
sudo ./aws/install
```

Then verify:

```
aws --version
```

Step - 3 - What If IAM Users Section Is Empty?

If you go to **IAM > Users** and see "No users found", follow these steps:

Step 1: Log in with Root User

- Use your **root account** credentials (email + password from AWS sign-up).
- Navigate to the **IAM** dashboard.

Step 2: Create IAM User

1. Go to **IAM > Users > Add User**
2. Name the user: `ci-cd-user` (or any name)

3. Check **Programmatic Access**
4. On the next screen, choose:
    - **Attach existing policies directly**
    - Select `AmazonEC2ContainerRegistryFullAccess`
    - Optionally add `AmazonECS_FullAccess` if you plan to deploy to ECS
5. Click **Next** > **Create User**

Step 3: Save Access Credentials

- Download or securely copy:
    - **Access Key ID**
    - **Secret Access Key**

These will be used to authenticate via CLI.

Step - 4 - Configure AWS CLI with Your IAM Credentials

Run this command:

```
aws configure
```

Then input:

- **Access Key ID** → (from IAM step)
- **Secret Access Key** → (from IAM step)
- **Default region** → `ap-south-1` (or your region)
- **Output format** → `json`

Step - 5 - Authenticate Docker to AWS ECR

You need to log Docker in to push images:

```
aws ecr get-login-password --region ap-south-1 | docker login --username AWS --password-stdin <your-account-id>.dkr.ecr.ap-south-1.amazonaws.com
```

- Replace <your-account-id> with your actual AWS account ID. You can find this on the top-right corner of the AWS console.

If successful, Docker will be authenticated with ECR and you can now push images.

IV. Update GitHub Secrets

To allow GitHub Actions to securely access your AWS resources, you'll need to add **secrets**—sensitive variables that your workflow will reference without hardcoding them into your codebase.

1. **Open Your GitHub Repository**
    - Navigate to your GitHub repo (e.g., `nodejs-ci-cd-demo`)
2. **Go to Settings → Secrets and Variables → Actions**
    - This section allows you to define secrets that GitHub Actions can use securely.
3. **Click "New Repository Secret"** for each of the following secrets:

Name	Value	Description
`AWS_ACCESS_KEY_ID`	Your AWS IAM access key	Used by GitHub to authenticate with AWS
`AWS_SECRET_ACCESS_KEY`	Your AWS IAM secret key	Paired with access key to authorize CLI commands
`AWS_REGION`	`ap-south-1` (or your region)	AWS region where ECR repo exists
`ECR_REPOSITORY`	`nodejs-ci-cd-repo`	Name of your ECR repo
`AWS_ACCOUNT_ID`	`123456789012` (your 12-digit AWS Account ID)	Needed to construct the ECR registry URL

These secrets are encrypted and can only be used by GitHub Actions workflows. Never hardcode them into files.

V. GitHub Actions Workflow for Docker + ECR

We will now automate the **Build → Login → Push** steps using GitHub Actions.

Step - 1 - Create Workflow File

Create a new file in your project at:

```
.github/workflows/cd.yml
```

Add the following content:

```
name: Build and Push to ECR

on:
  push:
    branches: [ main ]

jobs:
  deploy:
    name: Build and Push Docker Image to AWS ECR
    runs-on: ubuntu-latest

    steps:
      - name: Checkout Repository
        uses: actions/checkout@v3

      - name: Configure AWS Credentials
        uses: aws-actions/configure-aws-credentials@v4
        with:
          aws-access-key-id: ${{ secrets.AWS_ACCESS_KEY_ID }}
          aws-secret-access-key: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
          aws-region: ${{ secrets.AWS_REGION }}

      - name: Login to Amazon ECR
        run: |
          aws ecr get-login-password --region ${{ secrets.AWS_REGION }} | docker login --username AWS --password-stdin ${{ secrets.AWS_ACCOUNT_ID }}.dkr.ecr.${{ secrets.AWS_REGION }}.amazonaws.com

      - name: Build, Tag, and Push Image
        run: |
          IMAGE_URI=${{ secrets.AWS_ACCOUNT_ID }}.dkr.ecr.${{ secrets.AWS_REGION }}.amazonaws.com/${{ secrets.ECR_REPOSITORY }}
          docker build -t $IMAGE_URI .
          docker push $IMAGE_URI
```

Step - 2 - Commit and Push the Workflow

Open a terminal in your project directory and run:

```
git add .
git commit -m "Add Docker and CD workflow"
git push origin main
```

This triggers the GitHub Actions workflow defined above.

VI. Observe CI/CD Workflow

1. Go to the **"Actions" tab** in your GitHub repository.
2. You'll see a workflow titled **"Build and Push to ECR"**.
3. Click it to monitor the execution of each step:
    - Code checkout
    - AWS credentials configuration
    - Docker login to ECR
    - Docker build & push

Final Verification:

- Open **AWS Console > ECR > Your Repo (** `nodejs-ci-cd-repo` **)**
- Confirm that a new **Docker image** has been uploaded under the "Images" tab.

Outcomes

- The Node.js application was successfully containerized using Docker.
- The Docker image was built and securely pushed to a private AWS ECR repository.
- A GitHub Actions workflow was configured to automate the Docker build and push process.
- CI/CD integration was achieved, enabling continuous delivery of the Node.js app to AWS ECR.
- GitHub Secrets were used to securely manage AWS credentials and configuration.

Session - 3 - Continuous Deployment and App Management

Objectives

- Deploy the Docker image, built and stored in AWS ECR, to AWS ECS Fargate for seamless, serverless container orchestration.
- Automate the process of building the Docker image, pushing it to ECR, and deploying it to ECS via GitHub Actions.
- Configure the app's environment variables, add health check routes to ensure reliability, and enable logging for better monitoring.
- Monitor the deployed app, track its health, and ensure it is running optimally with ECS and CloudWatch.

I. Introduction to Deployment

What is Deployment in CI/CD?

Deployment in the context of CI/CD is the process of taking a built Docker image, pushing it to a runtime platform (like ECS), and making it accessible to end users. This is the final step in the pipeline, where the application is made publicly available for use.

Achieved using ECS Fargate (Serverless Containers):

Why ECS + Fargate?

- **Fully Managed Container Orchestration:**

  AWS ECS (Elastic Container Service) is a fully managed service that allows easy deployment and management of Docker containers.

- **No EC2 Management:**

  With Fargate, there's no need to manage EC2 instances directly. Fargate abstracts the underlying infrastructure, making it easier to focus on your containers and applications.

- **Integrated with ECR:**

ECS integrates seamlessly with AWS Elastic Container Registry (ECR), enabling smooth workflows for pulling Docker images and deploying them to ECS.

II. Set up AWS ECS Cluster with Fargate

Step - 1 - Create ECS Cluster

1. **Go to AWS Console → ECS → Clusters → Create Cluster**
    - Select the **Networking only (Fargate)** option for serverless containers.
    - **Cluster Name:** `ci-cd-cluster`
    - For the VPC, create a new one with the default settings.
    - Click **Create** to finish setting up the ECS cluster.

Step - 2 - Create Task Definition

1. **Go to ECS → Task Definitions → Create New Task Definition**
    - **Launch Type:** Fargate
    - **Task Role:** Select `ecsTaskExecutionRole` (create one if not available).
    - **Task Resources:**
        - Memory: 512 MiB
        - CPU: 0.25 vCPU
    - **Add Container:**
        - **Container Name:** `nodejs-app`
        - **Image URI:** `<AWS_ACCOUNT_ID>.dkr.ecr.<region>.amazonaws.com/nodejs-ci-cd-repo:latest`
        - **Port Mappings:** 3000 (the port your Node.js app will listen to)
        - **Log Configuration:** Use `awslogs` to log in AWS CloudWatch (specify the region, log group, and stream prefix).

Step - 3 - Create ECS Service

1. **Go to ECS → Clusters → ci-cd-cluster → Services → Create**
    - **Launch Type:** Fargate

- **Task Definition:** `ci-cd-task`
- **Service Name:** `ci-cd-service`
- **Number of Tasks:** 1
- **VPC/Subnets:** Use the ones created earlier.
- **Auto-assign Public IP:** Enable this to ensure the task is publicly accessible.
- **Load Balancer:** This can be optional, but for production environments, using an application load balancer (ALB) is recommended.

Step - 4 - Add Security Group Rule

1. **Security Group Settings:**
    - Allow inbound traffic on **Port 3000** to ensure the Node.js app is accessible from the outside.

III. Automate Deployment with GitHub Actions

Step - 1 - Update GitHub Secrets

- Go to GitHub Repository → **Settings → Secrets and Variables → Actions** → **New Repository Secret**.
- Add the following secrets:
    - **AWS_ACCESS_KEY_ID**: Your AWS Access Key
    - **AWS_SECRET_ACCESS_KEY**: Your AWS Secret Key
    - **AWS_REGION**: e.g., `ap-south-1`
    - **ECR_REPOSITORY**: `nodejs-ci-cd-repo`
    - **AWS_ACCOUNT_ID**: Your 12-digit AWS account number
    - **ECS_CLUSTER_NAME**: `ci-cd-cluster`
    - **ECS_SERVICE_NAME**: `ci-cd-service`
    - **ECS_TASK_DEFINITION**: `ci-cd-task`

Step - 2 - Add deploy.yml in .github/workflows/

Create GitHub Actions Workflow File

Create a new file at `.github/workflows/deploy.yml` with the following content:

```
name: Deploy to ECS

on:
  push:
    branches: [ main ]

jobs:
  deploy:
    name: Deploy to ECS Fargate
    runs-on: ubuntu-latest

    steps:
      - name: Checkout Repo
        uses: actions/checkout@v3

      - name: Configure AWS Credentials
        uses: aws-actions/configure-aws-credentials@v4
        with:
          aws-access-key-id: ${{ secrets.AWS_ACCESS_KEY_ID }}
          aws-secret-access-key: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
          aws-region: ${{ secrets.AWS_REGION }}

      - name: Login to Amazon ECR
        run: |
          aws ecr get-login-password --region ${{ secrets.AWS_REGION }} |
          docker login --username AWS --password-stdin ${{ secrets.AWS_ACCOUNT_ID }}.dkr.ecr.${{ secrets.AWS_REGION }}.amazonaws.com

      - name: Build and Push Docker Image
        run: |
          IMAGE_URI=${{ secrets.AWS_ACCOUNT_ID }}.dkr.ecr.${{ secrets.AWS_REGION }}.amazonaws.com/${{ secrets.ECR_REPOSITORY }}
          docker build -t $IMAGE_URI .
          docker push $IMAGE_URI

      - name: Deploy to ECS
```

```
        run: |
          aws ecs update-service \
            --cluster ${{ secrets.ECS_CLUSTER_NAME }} \
            --service ${{ secrets.ECS_SERVICE_NAME }} \
            --force-new-deployment
```

Step - 3 - Commit and Push

- **Run the following commands to push your changes**:

```
git add .
git commit -m "Add Docker and ECS deployment workflow"
git push origin main
```

IV. Observe Deployment

1. **Go to AWS ECS → Services → Events tab**

   Observe the ECS service deploying your app by updating the task and pulling the latest Docker image.

2. **Check Task Restart:**

   After pushing changes, ECS will automatically restart the task using the updated Docker image.

3. **Access the App:**

   You can access your app by using the **Public IP** assigned to the task (if no load balancer is used) or through an Application Load Balancer if set up.

V. Application Management & Monitoring

Step - 1 - Health Check

Add `/health` Route to Node.js:

```
app.get('/health', (req, res) => {
  res.status(200).send('App is healthy');
});
```

Configure Health Check in ECS Task Definition:

- Set up ECS to check the `/health` route as part of the health check configuration in the task definition.

Step - 2 - Logs

CloudWatch Logs:

- Logs from the container are automatically sent to **AWS CloudWatch Logs**. You can view these logs under **ECS → Task → Logs**.

Step - 3 - Monitoring

Create Alarms in CloudWatch:

- **Task Crash:** Set up an alarm for task restarts or crashes.
- **CPU/Memory Usage:** Set up alarms to monitor resource usage to ensure optimal performance.
- **Deploy Failure:** Monitor for deployment failures and setup alarms for troubleshooting.

Outcome

- The Dockerized Node.js app was successfully deployed to **AWS ECS Fargate**.
- **GitHub Actions** was configured to automate the full pipeline (build, push, deploy).
- The app is now being **monitored and managed** through AWS tools (CloudWatch and ECS).