



CI/CD Pipeline Report

1. System Design Overview

1.1 Architecture Overview

Our project follows a microservices architecture consisting of multiple backend services, an admin panel, and a frontend client. Here's a breakdown:

- **Frontend:** React application (served via Docker on EC2)
- **Backend Services:** Node.js-based microservices including:
 - user-service
 - auth-service
 - problem-service
 - lesson-service
 - match-making-service
- **Admin Panel:** React

- **Deployment Targets:**
 - Backend services are deployed to **AWS Lambda** using **Docker images from ECR**
 - Frontend is deployed to an **EC2 instance** via **Docker**
-

2. CI/CD Pipeline Workflow Implementation

2.1 Overview

We use **GitHub Actions** as our CI/CD platform and integrate with both **Docker Hub** and **AWS** (ECR + Lambda + EC2). The pipeline is defined in **.github/workflows/main.yml**.

2.2 Continuous Integration (CI) Workflow

Trigger:

- On push to **main** or **fix/user-service-tests**

Step-by-Step:

Job: test

- Runs on **ubuntu-latest**
- Matrix strategy runs Jest-based tests across all services:
 1. **frontend**, **admin**, and all backend services
- Steps:
 1. Checkout repository
 2. Setup Node.js (version 20)
 3. Install dependencies using **npm install** (per service)
 4. Run tests using **npm test**

Notes: **continue-on-error: true** ensures that the pipeline continues even if some services fail to build/test

2.3 Continuous Deployment (CD) Workflow

Job: **build-and-push**

- Builds Docker images for backend services
- Pushes images to:
 - Docker Hub
 - AWS ECR

Steps:

1. Checkout code
2. Log in to Docker Hub using **docker/login-action**
3. Build Docker image using service-specific Dockerfiles
4. Push image to Docker Hub
5. Log in to AWS via **aws-actions/configure-aws-credentials**
6. Log in to AWS ECR and push image

Job: **build-and-push-frontend**

- Builds and pushes the frontend image to Docker Hub only
 - Uses similar steps as above
-

2.4 Deployment Jobs

Job: **deploy-backend**

- Triggered after **build-and-push**
- Uses matrix strategy to deploy each backend service to AWS Lambda

Steps:

1. Log in to AWS
2. Use AWS CLI to update the Lambda function with the new ECR image

*Lambda function names are prefixed with **codeify-** and slashes replaced with dashes.*

Job: **deploy-frontend**

- Triggered after **build-and-push-frontend**
- Uses SSH (**appleboy/ssh-action**) to access an EC2 instance and deploy the new frontend image

Steps:

1. SSH into EC2
 2. Stop and remove all running containers
 3. Remove all Docker images
 4. Pull the latest frontend image
 5. Run a new container on port 5173
-

3. Dockerfile Configuration

Each microservice contains a **Dockerfile** with the following common structure:

```
FROM node:16
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
EXPOSE PORT_NUMBER
CMD ["npm", "run"]
```

This ensures that the service:

- Uses Node.js v16
 - Installs dependencies
 - Exposes a default port (e.g., 8085)
 - Runs using a defined script (can be changed to **npm start** or **node index.js** as needed)
-

4. Key Notes & Considerations

- Secrets for Docker and AWS are securely stored in GitHub repository secrets.
- We use a matrix strategy to parallelize testing and deployment across services.
- Frontend deployment is manual via EC2 and Docker, while backend is fully serverless (Lambda).
- CI runs for every push to selected branches ensuring code quality.
- CD uses GitHub Actions for fully automated and reproducible deployments.

5. File Structure Summary

```
.github/  
├── workflows/  
│   └── main.yml  
frontend/  
admin/  
backend/  
├── user-service/  
├── auth-service/  
├── problem-service/  
├── lesson-service/  
└── match-making-service/
```

Each service folder contains:

- **Dockerfile**
- **package.json**
- Test files (***.test.js**) for Jest

6. Conclusion

This CI/CD setup streamlines our development pipeline using GitHub Actions, Docker, and AWS. It ensures that:

- Code changes are continuously tested
- Images are built and stored in registries
- Deployments are automated and reproducible
- Backend is serverless; frontend is hosted on EC2

We have achieved a complete CI/CD workflow for our microservices architecture with best practices and automation in mind.