CI/CD Pipeline Documentation

Online Learning Portal - Microservices Demo

1. Project Overview

I built a complete CI/CD pipeline for a microservices-based online learning portal. The system automatically tests, builds, and deploys 5 independent services whenever code is pushed to GitHub.**What it does:**

- Automatically runs tests when I push code
- Builds Docker containers for each service
- Deploys to AWS cloud infrastructure
- All services can communicate with each other
- Has a unified API interface (Swagger UI)

2. Architecture Diagram

```
GitHub Repository
     │ (push code)
     ▼
Jenkins Server (on EC2)
     │ (webhook trigger)
     ├─→ Run Tests (pytest)
     ├─→ Build Docker Image
     ├─→ Push to ECR (image storage)
     └─→ Deploy to ECS (run containers)


AWS Cloud
     ├─→ ECS Fargate (6 running services)
     ├─→ DynamoDB (5 database tables)]
     ├─→ Cloud Map (service discovery)
     └─→ Load Balancer (public access)
```

3. Microservices

I created 5 independent services that work together:

1. **User Service** - handles registration and login
2. **Course Service** - manages course catalog
3. **Enrollment Service** - students enroll in courses
4. **Payment Service** - processes payments
5. **Notification Service** - sends notification

Plus a **Swagger UI** service that acts as an API Gateway to access all services from one place.Each service:

- Has its own code repository branch
- Has its own database table
- Can be deployed independently
- Runs in its own Docker container

4. Technology Stack

Why I chose these tools:

Python + FastAPI

- Easy to learn and write APIs quickly
- Automatic API documentation (Swagger)
- Good for microservices

Docker

- Packages everything the app needs
- Works the same everywhere (my laptop, AWS cloud)
- Easy to deploy and scale

Jenkins

- Free and popular CI/CD tool
- Automatically triggers on code push
- Supports pipeline-as-code (Jenkinsfile)

AWS Services:

- **ECR** - stores my Docker images (like Docker Hub but private)
- **ECS Fargate** - runs containers without managing servers
- **DynamoDB** - NoSQL database, easy to use, scales automatically
- **Cloud Map** - lets services find each other by name
- **EC2** - runs my Jenkins server

pytest

- Simple testing framework for Python
- Integrated into my pipeline

5. AWS Services Explained

Amazon ECS (Elastic Container Service)

What it does: Runs my Docker containers in the cloud without me managing servers.**Why I used it:**

- No need to set up/maintain servers
- Automatically handles container lifecycle
- Fargate = serverless (just run containers)

My setup:

- 6 services running (5 microservices + API gateway)
- Each gets 0.25 vCPU and 512MB memory
- All containers listen on port 8080

Amazon ECR (Elastic Container Registry)

What it does: Stores my Docker images privately.**Why I used it:**

- Integrates directly with ECS
- More secure than public Docker Hub
- Fast image pulls from within AWS

My setup:

- 6 repositories (one per service)
- Jenkins pushes new images on every build
- Uses `latest` tag (simple for demo)

Amazon DynamoDB

What it does: NoSQL database for storing application data.**Why I used it:**

- No need to manage database servers
- Simple key-value + JSON storage
- Good for microservices (each service gets own table)
- Automatically scales

My setup:

- 5 tables (one per service)
- Each table has a primary key (like `user_id`, `course_id`)
- Some have indexes for fast lookups (like search by email)

AWS Cloud Map

What it does: Service discovery - services can find each other by name.**Why I used it:**

- Services have dynamic IPs in cloud
- Cloud Map gives them DNS names like `user-service.local`
- Services can call each other without hardcoding Ips

My setup:

- Created namespace: `learning-portal.local`
- Each service gets a DNS name
- Example: payment service calls `http://notification-service.learning-portal.local:8080`

Amazon EC2

What it does: Virtual machine in the cloud.**Why I used it:**

- Needed somewhere to run Jenkins
- Cheaper than managed Jenkins options

My setup:

- Medium-sized instance (2 CPUs, 4GB RAM)
- Installed Jenkins, Docker, AWS CLI
- Opens port 8080 for Jenkins web interface

6. CI/CD Pipeline Flow

Step-by-Step Process:

1. **Developer pushes code to GitHub**
   - I work on a service branch (like `user-service`)
   - Push changes to GitHub

2. **GitHub webhook triggers Jenkins**
   - Automatic notification to Jenkins
   - Jenkins pulls the latest code

3. **Jenkins runs tests**
   - Installs Python dependencies
   - Runs pytest unit tests
   - Pipeline stops if tests fail

4. **Jenkins builds Docker image**
   - Uses Dockerfile in service folder
   - Tags image with service name
   - Cross-platform build (works on any CPU)

5. **Jenkins pushes to ECR**
   - Authenticates to AWS
   - Pushes image to private registry
   - Overwrites `latest` tag

6. **Jenkins deploys to ECS**
   - Creates/updates task definition
   - Forces ECS to pull new image
   - ECS starts new container, stops old one
   - Zero-downtime deployment

Pipeline Stages (in Jenkinsfile):

```
stages {
    stage('Run Tests') {
        // Install dependencies and run pytest
    }
    stage('Build Docker Image') {
        // Build container image
    }
    stage('Push to ECR') {
        // Upload to AWS registry
    }
    stage('Deploy to ECS') {
        // Update running service
    }
}
```

7. Security & Access

IAM Roles (AWS Permissions)

ecsTaskExecutionRole

- Allows ECS to pull images from ECR
- Allows ECS to write logs to CloudWatch
- Infrastructure-level permissions

ecsTaskRole

- Allows my app code to access DynamoDB
- Application-level permissions
- Each container runs with this role

EC2 Instance Role

- Allows Jenkins to access AWS services
- Can push to ECR, update ECS, read DynamoDB

Security Groups (Firewall Rules)

Jenkins Server:

- Port 8080: Open to internet (for web UI and GitHub webhooks)
- Port 22: SSH access (for management)

ECS Services:

- Port 8080: Open within VPC (services talk to each other)
- Swagger UI: Open to internet (public API gateway)

8. Testing Strategy

I wrote 18 unit tests total across all services:**What tests check:**

- Health endpoints work
- Input validation works (reject bad data)
- Database queries work
- Error handling (404, 401, 422 errors)

Example test:

```
def test_register_user_missing_fields():
    response = client.post("/users/register", json={})
    assert response.status_code == 422  # Validation error
```

Why testing matters:

- Catches bugs before deployment
- Gives confidence code works
- Required for professional CI/CD

9. Challenges & Solutions

Challenge 1: Architecture Mismatch

Problem: Jenkins server is ARM64, but I was building for AMD64. Builds took 10+ minutes.
Solution: Use Docker Buildx for cross-platform builds. Now builds finish in 2-3 minutes.

Challenge 2: Service Communication

Problem: Services have dynamic IPs in ECS. How do they find each other?
Solution: AWS Cloud Map gives each service a DNS name. Services use names instead of IPs.

Challenge 3: Python Package Installation

Problem: Ubuntu 24.04 blocks pip installations (PEP 668).
Solution: Use `--break-system-packages` flag in Jenkins. Safe for CI/CD environments.

Challenge 4: DynamoDB Access from Tests

Problem: Tests failed because Jenkins couldn't access DynamoDB.
Solution: Added DynamoDB permissions to EC2 instance IAM role.

10. Tool Justification

Why Jenkins?

- Popular industry standard
- Free and open source
- Good documentation and community support
- Pipeline-as-code (Jenkinsfile)

Why Docker?

- Industry standard for containerization
- Consistent environments
- Easy to deploy anywhere

Why AWS ECS instead of Kubernetes?

- Simpler to learn and set up
- Fargate = no server management
- Good enough for this demo
- Cheaper for small projects

Why FastAPI?

- Modern Python framework
- Automatic API docs (Swagger)
- Fast performance
- Easy to learn

Why DynamoDB instead of PostgreSQL?

- Serverless (no database server to manage)
- Each microservice gets own table
- Scales automatically
- Good fit for NoSQL data

11. Results

Pipeline Success Rate: 100% (all builds passing)
Deployment Time: 3-5 minutes per service
Tests Passing: 18/18
Services Running: 6/6 active in ECS
Automation Level: Fully automated from git push to deployment

12. Conclusion

This project demonstrates a production-ready CI/CD pipeline using industry-standard tools.
I successfully implemented:

- Microservices architecture with 5 independent services
- Automated testing integrated into CI/CD
- Containerization with Docker
- Cloud deployment on AWS
- Infrastructure as code (Jenkinsfiles)
- Service discovery and communication
- Database integration

What I learned:

- How CI/CD pipelines work in real companies
- Docker and containerization
- AWS cloud services
- Microservices communication
- Infrastructure automation

GitHub: https://github.com/HAR5HA-7663/demo-for-CI_CD-integration