DevOps Internship - Task 1: Automate Code Deployment Using CI/CD Pipeline

Intern Name: Soumen Das
Organization: Elevate Labs / Ministry of MSME, Government of India
Date: November 13, 2025
Task: Automate Code Deployment Using CI/CD Pipeline (GitHub Actions)

Executive Summary

This document provides a complete solution for **Task 1** of the DevOps internship program. The task involves setting up a robust CI/CD pipeline using GitHub Actions to automate the build, test, and deployment of a Node.js web application. The pipeline integrates continuous integration (automated testing) with continuous deployment (Docker containerization and DockerHub push), eliminating manual deployment processes and reducing human error.

1. Objective and Deliverables

Objective

Set up a CI/CD pipeline to automatically build and deploy a web application whenever code is pushed to the main branch.

Deliverables

- GitHub repository with `.github/workflows/main.yml` CI/CD workflow file
- Automated testing pipeline
- Docker containerization setup
- Automated Docker image build and push to DockerHub
- Triggered on every push to the main branch

Tools Required

- GitHub (Repository hosting)
- GitHub Actions (CI/CD automation)
- Node.js (Application runtime)
- Docker (Containerization)
- DockerHub (Container registry)

2. Project Structure Setup

Step 1: Repository Directory Structure

Create the following structure in your repository:

```
nodejs-demo-app/
├── .github/
│   └── workflows/
│       └── main.yml
├── src/
│   ├── index.js
│   ├── app.js
│   └── server.js
├── tests/
│   └── app.test.js
├── Dockerfile
├── .dockerignore
├── package.json
├── package-lock.json
├── .gitignore
└── README.md
```

Step 2: Initialize Node.js Project

If starting fresh, initialize the project:

```
npm init -y
npm install express
npm install --save-dev jest supertest
```

Step 3: Create Basic Express Application

File: `src/app.js`

```
const express = require('express');
const app = express();

app.use(express.json());

app.get('/', (req, res) =&gt; {
  res.status(200).json({
    message: 'Welcome to Node.js Demo App',
    status: 'running'
  });
});

app.get('/health', (req, res) =&gt; {
  res.status(200).json({
    health: 'OK',
    timestamp: new Date().toISOString()
  });
});

app.post('/api/data', (req, res) =&gt; {
  const { name } = req.body;
  if (!name) {
    return res.status(400).json({ error: 'Name is required' });
  }
  res.status(201).json({
    message: `Hello, ${name}!`,
    received: req.body
  });
});

module.exports = app;
```

File: `src/index.js`

```
const app = require('./app');

const PORT = process.env.PORT || 3000;

app.listen(PORT, () =&gt; {
  console.log(`Server is running on port ${PORT}`);
});
```

Step 4: Create Test Suite

File: `tests/app.test.js`

```
const request = require('supertest');
const app = require('../src/app');

describe('Express App Tests', () =&gt; {

  test('GET / should return welcome message', async () =&gt; {
    const response = await request(app).get('/');
    expect(response.status).toBe(200);
    expect(response.body.status).toBe('running');
  });

  test('GET /health should return OK status', async () =&gt; {
    const response = await request(app).get('/health');
    expect(response.status).toBe(200);
    expect(response.body.health).toBe('OK');
  });

  test('POST /api/data should accept name parameter', async () =&gt; {
    const response = await request(app)
      .post('/api/data')
      .send({ name: 'Soumen' });
    expect(response.status).toBe(201);
    expect(response.body.message).toContain('Soumen');
  });

  test('POST /api/data should return 400 without name', async () =&gt; {
    const response = await request(app)
      .post('/api/data')
      .send({});
    expect(response.status).toBe(400);
  });
});
```

Step 5: Update package.json

File: `package.json`

```
{
  "name": "nodejs-demo-app",
  "version": "1.0.0",
  "description": "Node.js demo app for CI/CD pipeline",
  "main": "src/index.js",
  "scripts": {
    "start": "node src/index.js",
    "test": "jest",
    "build": "echo 'Build completed successfully'"
  },
  "keywords": ["nodejs", "cicd", "devops"],
  "author": "Soumen Das",
  "license": "MIT",
  "dependencies": {
    "express": "^4.18.2"
  },
  "devDependencies": {
    "jest": "^29.7.0",
    "supertest": "^6.3.3"
  }
}
```

3. Dockerfile Configuration

Step 1: Multi-Stage Dockerfile

File: `Dockerfile`

```
# Stage 1: Build Stage
FROM node:18-alpine AS builder

WORKDIR /app

COPY package*.json ./

RUN npm ci

COPY . .

RUN npm test

RUN npm run build

# Stage 2: Production Stage
FROM node:18-alpine

WORKDIR /app

ENV NODE_ENV=production
ENV PORT=3000

COPY package*.json ./

RUN npm ci --only=production

COPY --from=builder /app/src ./src

EXPOSE 3000

HEALTHCHECK --interval=30s --timeout=10s --start-period=5s --retries=3 \
  CMD node -e "require('http').get('http://localhost:3000/health', (r) =&gt; {if (r.status(

CMD ["npm", "start"]
```

Step 2: Docker Ignore File

File: `.dockerignore`

```
node_modules
npm-debug.log
.git
.gitignore
```

```
.env
.env.local
tests
.github
```

4. GitHub Actions Workflow Configuration

Step 1: Create Workflow Directory

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

Step 2: Create Main Workflow File

File: `.github/workflows/main.yml`

```
name: Build and Deploy Node.js App

on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]

env:
  REGISTRY: docker.io
  IMAGE_NAME: nodejs-demo-app

jobs:

  # CI Job: Test and Build
  test-and-build:
    runs-on: ubuntu-latest

    strategy:
      matrix:
        node-version: [18.x]

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

      - name: Setup Node.js ${{ matrix.node-version }}
        uses: actions/setup-node@v3
        with:
          node-version: ${{ matrix.node-version }}
          cache: 'npm'

      - name: Install Dependencies
        run: npm ci

      - name: Run Linting
        run: npm run lint --if-present || true

      - name: Run Tests
        run: npm test

      - name: Build Application
        run: npm run build

  # CD Job: Build Docker Image and Push
  build-and-push:
    needs: test-and-build
    runs-on: ubuntu-latest

    if: github.event_name == 'push' &amp;&amp; github.ref == 'refs/heads/main'

    permissions:
      contents: read
      packages: write

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

      - name: Set up QEMU
        uses: docker/setup-qemu-action@v2

      - name: Set up Docker Buildx
        uses: docker/setup-buildx-action@v2

      - name: Log in to Docker Hub
        uses: docker/login-action@v2
        with:
          username: ${{ secrets.DOCKER_USERNAME }}
          password: ${{ secrets.DOCKER_PASSWORD }}

      - name: Extract Metadata
        id: meta
        uses: docker/metadata-action@v4
        with:
          images: ${{ env.REGISTRY }}/${{ secrets.DOCKER_USERNAME }}/${{ env.IMAGE_NAME }}
          tags: |
            type=ref,event=branch
            type=sha,prefix={{branch}}-
            type=semver,pattern={{version}}
            type=semver,pattern={{major}}.{{minor}}
            type=raw,value=latest,enable={{is_default_branch}}

      - name: Build and Push to Docker Hub
        uses: docker/build-push-action@v4
        with:
          context: .
          push: true
          tags: ${{ steps.meta.outputs.tags }}
          labels: ${{ steps.meta.outputs.labels }}
          cache-from: type=gha
          cache-to: type=gha,mode=max
```

5. GitHub Secrets Configuration

Step 1: Set Up DockerHub Credentials

Navigate to your GitHub repository and set up the following secrets:

1. **DOCKER_USERNAME**: Your DockerHub username
2. **DOCKER_PASSWORD**: Your DockerHub Personal Access Token (PAT)

Steps to Create Secrets:

1. Go to your repository on GitHub
2. Click **Settings → Secrets and variables → Actions**
3. Click **New repository secret**
4. Add the following:
   - Name: `DOCKER_USERNAME`, Value: `your_dockerhub_username`
   - Name: `DOCKER_PASSWORD`, Value: `your_dockerhub_pat_token`

Creating DockerHub PAT:

1. Go to DockerHub Account Settings
2. Click **Security → New Access Token**
3. Give it a descriptive name
4. Select appropriate permissions (Read & Write)
5. Copy and save the token

6. Pushing Code and Triggering Pipeline

Step 1: Commit and Push

```
git add .
git commit -m "Initial commit: Set up CI/CD pipeline"
git push origin main
```

Step 2: Monitor Pipeline Execution

1. Go to your GitHub repository
2. Click **Actions** tab
3. View the running workflow
4. Click on the workflow run to see detailed logs
5. Check each job status (test, build-and-push)

Step 3: Verify Docker Image

After successful pipeline:

1. Go to DockerHub
2. Navigate to your repository
3. Verify the newly pushed image with the correct tag

7. Pipeline Workflow Stages

Stage 1: Code Checkout

- Action: `actions/checkout@v3`
- Purpose: Clone the repository code into the GitHub runner
- Output: Repository files available in the runner environment

Stage 2: Environment Setup

- Action: `actions/setup-node@v3`
- Purpose: Install specified Node.js version
- Output: Node.js runtime ready for execution

Stage 3: Dependency Installation

- Command: `npm ci`
- Purpose: Install project dependencies in CI environment
- Output: `node_modules` directory populated

Stage 4: Testing

- Command: `npm test`
- Purpose: Run automated test suite
- Outcome: Pass/Fail determines pipeline continuation

Stage 5: Application Build

- Command: `npm run build`
- Purpose: Compile and prepare application
- Output: Build artifacts ready for containerization

Stage 6: Docker Build and Push

- Action: `docker/build-push-action@v4`
- Purpose: Build Docker image and push to registry
- Output: Image published on DockerHub with version tags

8. Troubleshooting Guide

Issue 1: GitHub Actions Workflow Not Triggering

Solution:

- Verify workflow file is in `.github/workflows/` directory
- Check YAML syntax (use YAML validator)
- Ensure branch is 'main' (not 'master')
- Verify `on: push` trigger is configured

Issue 2: Docker Login Failed

Solution:

- Verify `DOCKER_USERNAME` and `DOCKER_PASSWORD` secrets are set
- Ensure secrets are added to the correct repository
- Check PAT token has necessary permissions
- Verify no typos in secret names in workflow file

Issue 3: Tests Failing in CI

Solution:

- Run `npm test` locally to verify tests pass
- Check test configuration in `package.json`
- Verify all dependencies are listed in `package.json`
- Check for environment variables needed by tests

Issue 4: Docker Image Build Failing

Solution:

- Verify `Dockerfile` syntax is correct
- Check `.dockerignore` is not excluding necessary files
- Ensure all dependencies are in `package.json`
- Verify working directory paths in Dockerfile

9. Best Practices

Security

- Never commit secrets to repository
- Use GitHub Secrets for sensitive data
- Use minimal Docker base images
- Implement HEALTHCHECK in Dockerfile

Performance

- Use npm cache action to speed up builds
- Implement Docker layer caching
- Use alpine-based images for smaller sizes
- Parallelize jobs where possible

Maintainability

- Use meaningful commit messages
- Keep workflow files organized
- Document pipeline steps
- Version Docker images appropriately

Testing

- Write comprehensive test cases
- Run tests before deployment
- Implement code coverage reporting
- Test different Node.js versions

10. Next Steps and Enhancements

Recommended Enhancements

1. Add code coverage reporting
2. Implement security scanning (Snyk, Trivy)
3. Add staging environment deployment
4. Implement database migrations in pipeline
5. Add performance benchmarking
6. Implement automated rollback on failure
7. Add Slack/email notifications
8. Implement blue-green deployment strategy

Advanced Features

- Multi-environment deployments (dev, staging, production)
- Automated versioning and tagging
- Container registry cleanup policies
- Kubernetes deployment integration
- Infrastructure as Code (IaC) integration

Conclusion

This CI/CD pipeline automates the entire development workflow from code push to production deployment. By implementing this setup, Soumen Das has established:

✓ **Continuous Integration**: Automated testing on every code change
✓ **Continuous Deployment**: Automated Docker image build and push
✓ **Reduced Manual Errors**: Elimination of manual deployment steps
✓ **Scalability**: Foundation for multi-environment deployments
✓ **Code Quality**: Automated testing ensures only quality code is deployed

The pipeline is production-ready and can be extended with additional stages and environments as project requirements grow.