# Milestone 2: Containerization & CI/CD Pipeline

MLOps Course - Module 3

### Aligned Learning Objectives

- **CG2.LO1:** Containerize ML environments and configure registries.
- **CG2.LO3:** Integrate CI/CD for testing and rollout.

---

### Assignment Overview

You will build a complete containerized ML service deployment pipeline that emphasizes security, reproducibility, and automation. This milestone integrates Docker containerization best practices with automated CI/CD workflows to create a production-ready deployment system.

This assignment combines three critical MLOps competencies:

1. **Multi-stage Docker image** with optimized layers and security considerations
2. **Automated CI/CD pipeline** that builds, tests, and publishes to a container registry
3. **Operations runbook** documenting the complete workflow

By completing this milestone, you'll demonstrate your ability to create reproducible, automated deployment pipelines-a fundamental skill for modern ML engineering roles.

---

### Why This Matters

Containerization and CI/CD automation are foundational to modern MLOps practice. In production environments, manual deployment processes are error-prone, time-consuming, and don't scale. This milestone teaches you to build automated pipelines that ensure every deployment is tested, versioned, and reproducible.

These skills directly translate to industry practice where you'll need to:

- Deploy models reliably across development, staging, and production environments
- Maintain security and compliance through automated scanning and testing
- Enable rapid iteration while maintaining system stability

- Document operational procedures for team collaboration

---

## Deliverables

### Part 1: Multi-Stage Docker Image (4 points)

Create an optimized, production-ready container image:

1. **`Dockerfile`** - Multi-stage build with:
   - Builder stage for dependency installation
   - Minimal runtime stage with only necessary components
   - Clear layer optimization and caching strategy
2. **`app/`** directory containing:
   - Minimal inference script (e.g., `app.py`)
   - Pinned dependencies (`requirements.txt` or `pyproject.toml`)
   - Any necessary model artifacts or configuration
3. **`docker-compose.yaml`** (optional) - For local testing and development
4. **Registry verification** - Screenshot or link proving successful image push with proper tagging (e.g., `registry.example.com/username/ml-service:v1.0.0`)

### Part 2: CI/CD Pipeline (4 points)

Implement automated build, test, and deployment:

5. **`.github/workflows/build.yml`** - GitHub Actions workflow including:
   - **Test job:** Run pytest on your code
   - **Build job:** Build Docker image
   - **Authenticate:** Login to container registry
   - **Publish:** Push image with semantic version tags
6. **`README.md`** - Project documentation with:
   - CI/CD status badge showing build status
   - Clear instructions for pulling and running the image
   - Quick start guide for local development
7. **`tests/test_app.py`** - Unit tests for your ML service:
   - Test inference endpoint functionality
   - Validate input/output formats
   - Check error handling

### Part 3: Operations Runbook (2 points)

Document your deployment system:

8. **`RUNBOOK.md`** - Comprehensive operations guide covering:
    - **Dependency pinning strategy:** How you ensure reproducibility
    - **Image optimization:** Size before/after optimization with techniques used
    - **Security considerations:** Vulnerability scanning, minimal attack surface
    - **CI/CD workflow:** Step-by-step explanation of the automation pipeline
    - **Versioning strategy:** How you use semantic versioning (`vX.Y.Z`)
    - **Troubleshooting:** Common issues and solutions

---

## Requirements & Constraints

**Required:**

- Python-based ML service only
- Docker for containerization
- GitHub Actions for CI/CD (no other CI systems)
- Push to the assigned course container registry
- Use per-student namespace/tag in registry
- Semantic versioning for all image tags

**Prohibited:**

- No external CI/CD systems (CircleCI, Jenkins, etc.)
- No hardcoded credentials in repository
- No large binary files committed to Git

---

## Rubric (10 points)

### Docker Image Quality (4 points)

| Criteria | Points | Description |
|---|---|---|
| **Multi-stage Build** | 2 | Correct multi-stage Dockerfile with clean layer separation and optimization |
| **Runtime Environment** | 1 | Minimal, deterministic runtime with pinned dependencies |
| **Registry Integration** | 1 | Successful push to registry with proper semantic versioning |

### CI/CD Pipeline (4 points)

| Criteria | Points | Description |
|---|---|---|
| **Pipeline Functionality** | 2 | Complete GitHub Actions workflow executing build-test-deploy successfully |
| **Test Integration** | 1 | Tests execute correctly and failures cause CI to fail |
| **Authentication & Versioning** | 1 | Proper registry authentication and semantic version tagging |

**Documentation & Operations (2 points)**

| Criteria | Points | Description |
|---|---|---|
| **Runbook Quality** | 1 | Clear, technically accurate operations documentation |
| **Project Organization** | 0.5 | Clean repository structure with logical file organization |
| **README & Instructions** | 0.5 | Professional README with CI badge and clear setup instructions |

**Total: 10 points**

---

## What Success Looks Like

A high-quality submission demonstrates:

- **Reproducible Docker image** with clear layer optimization and minimal size
- **Automated pipeline** triggered by version tags that builds, tests, and deploys without manual intervention
- **Comprehensive documentation** enabling team members to understand and replicate your workflow
- **Security-conscious practices** including pinned dependencies and minimal attack surface
- **Professional presentation** with clear organization and attention to detail

---

## Getting Started

Follow these steps to complete this milestone:

1. **Start with your Milestone 1 service** or create a minimal ML inference script
2. **Write a basic Dockerfile** that runs your application successfully
3. **Convert to multi-stage build** separating builder and runtime stages
4. **Optimize layers** by ordering commands from least to most frequently changing
5. **Create a simple test** that validates your inference endpoint
6. **Set up GitHub Actions** workflow with build and test jobs
7. **Configure registry authentication** using GitHub Secrets
8. **Push your first image** with a semantic version tag (e.g., `v0.1.0`)
9. **Write your RUNBOOK.md** documenting the complete workflow

---

## Tips for Success

### Docker Optimization

- **Layer caching:** Order Dockerfile commands from least to most frequently changing
- **Multi-stage builds:** Use builder stage for compilation, minimal runtime stage for execution
- **Base image selection:** Consider Alpine for size, Debian-slim for compatibility
- **Common pitfall:** Installing unnecessary development dependencies in runtime image

### CI/CD Best Practices

- **Test first:** Ensure tests pass locally before pushing
- **Secrets management:** Use GitHub Secrets for registry credentials
- **Conditional execution:** Only push images on successful tests
- **Common pitfall:** Forgetting to authenticate before pushing to registry

### Documentation

- **Be specific:** Include exact commands, not just descriptions
- **Assume nothing:** Write for someone unfamiliar with your setup
- **Update regularly:** Keep documentation in sync with code changes
- **Common pitfall:** Outdated documentation that doesn't match current implementation

### Example Multi-Stage Dockerfile Structure

```
# Builder stage
FROM python:3.11-slim as builder
WORKDIR /build
COPY requirements.txt .
RUN pip install --user --no-cache-dir -r requirements.txt

# Runtime stage
FROM python:3.11-slim
WORKDIR /app
```

```
COPY --from=builder /root/.local /root/.local
COPY app/ .
ENV PATH=/root/.local/bin:$PATH
CMD ["python", "app.py"]
```

---

### Challenge Extensions (Optional)

Push your containerization skills further:

1. **Base Image Comparison:**
   - Build versions using Alpine, Debian-slim, and Ubuntu
   - Compare image sizes, build times, and compatibility
   - Document trade-offs and recommendations
2. **Security Scanning:**
   - Integrate Trivy or Snyk vulnerability scanning (free tier)
   - Add security scan step to CI/CD pipeline
   - Document and remediate any critical vulnerabilities
3. **Multi-Architecture Builds:**
   - Build images for both amd64 and arm64 architectures
   - Use Docker buildx for cross-platform builds
   - Test on different CPU architectures
4. **Advanced CI/CD:**
   - Implement separate staging and production workflows
   - Add automated rollback on deployment failure
   - Create deployment notifications (Slack, email)

---

### Submission Requirements

- Push all files to your course repository under `module3/milestone2/`
- Ensure your CI/CD pipeline runs successfully on push
- Tag your final submission as `m2-submission` in Git
- Verify your image is accessible in the course registry
- **Deadline:** [To be announced in course schedule]

---

### Common Issues & Solutions

#### Issue: Image push fails with authentication error

**Solution:** Verify GitHub Secrets are configured correctly and workflow has registry credentials

**Issue: Tests pass locally but fail in CI**

**Solution:** Check for environment-specific dependencies or hardcoded paths

**Issue: Image size is too large (>1GB)**

**Solution:** Review installed dependencies, use multi-stage builds, consider Alpine base

**Issue: Build takes too long**

**Solution:** Optimize layer caching, use `.dockerignore`, parallelize independent steps

---

## Resources

- Docker Multi-Stage Builds
- GitHub Actions Documentation
- Container Registry Authentication
- Semantic Versioning Specification
- Docker Best Practices