MLOps Workshop: Complete Pipeline Implementation

Professional Knowledge Transfer Session

Table of Contents

Introduction & Overview

What is MLOps?

MLOps (Machine Learning Operations) is a practice that aims to deploy and maintain machine learning models in production reliably and efficiently.

Key Principles:

- **Automation**: Automated pipelines for training, testing, and deployment
- **Monitoring**: Continuous monitoring of model performance and data quality
- **Versioning**: Track changes in data, code, and models
- **Collaboration**: Bridge the gap between data science and operations teams
- **Reproducibility**: Ensure consistent results across environments

Why MLOps Matters:

- **Business Impact**: 85% of ML projects fail to reach production
- **Technical Debt**: Manual processes create maintenance overhead
- **Compliance**: Regulatory requirements for model explainability
- **Scalability**: Handle multiple models across different environments

Workshop Objectives

By the end of this session, you will understand: - Complete MLOps pipeline architecture - 8 essential MLOps tools and their integration - Practical implementation strategies - Production deployment considerations

---

MLOps Architecture

Overall System Architecture

```
┌─────────────────────────────────────────────────────────────────┐
│                      MLOps Complete Pipeline                    │
├─────────────────────────────────────────────────────────────────┤
│                                                                 │
│  ┌─────────────┐    ┌─────────────┐    ┌─────────────┐         │
│  │    Data     │────│  Training   │────│   Serving   │         │
│  │   Layer     │    │    Layer    │    │    Layer    │         │
│  └─────────────┘    └─────────────┘    └─────────────┘         │
│         │                   │                   │              │
│         └───────────────────┼───────────────────┘              │
│                             │                                  │
│  ┌─────────────────────────────────────────────────────────┐   │
│  │               Monitoring & Observability               │    │
│  └─────────────────────────────────────────────────────────┘   │
│                                                                 │
└─────────────────────────────────────────────────────────────────┘
```

Technology Stack

Layer	Technology	Purpose
Data	DVC	Data versioning and pipeline management
Training	MLflow	Experiment tracking and model registry
Serving	FastAPI	REST API for model inference
Monitoring	Prometheus + Grafana	Metrics collection and visualization
CI/CD	GitHub Actions	Automated testing and deployment
Infrastructure	Docker	Containerization and orchestration
Quality	Evidently AI	Data drift and model performance monitoring
Testing	Pytest	Automated testing framework

Data Versioning (DVC)

What is DVC?

Data Version Control (DVC) is an open-source tool for data science and machine learning projects that provides: - Git-like versioning for datasets and ML models - Reproducible ML pipelines - Experiment management

Key Features

Advantages ✅

- **Version Control**: Track changes in large datasets
- **Reproducibility**: Recreate exact experiment conditions
- **Pipeline Management**: Define and execute ML workflows
- **Storage Agnostic**: Works with S3, GCS, Azure, local storage
- **Git Integration**: Seamless integration with Git workflows

Disadvantages ❌

- **Learning Curve**: Additional complexity for simple projects
- **Storage Costs**: Requires external storage for large datasets
- **Performance**: Can be slow with very large files

Usage Example

```
# Initialize DVC in your project
dvc init

# Add data to DVC tracking
dvc add data/train.csv

# Create a pipeline stage
dvc stage add -n train \
  -d data/train.csv \
  -d scripts/train.py \
  -o models/model.pkl \
  python scripts/train.py

# Run the pipeline
dvc repro

# Push data to remote storage
dvc push
```

DVC Pipeline Configuration

```yaml
# dvc.yaml
stages:
  prepare:
    cmd: python src/prepare.py
    deps:
    - src/prepare.py
    - data/raw
    outs:
    - data/processed

  train:
    cmd: python src/train.py
    deps:
    - src/train.py
    - data/processed
    outs:
    - models/model.pkl
    metrics:
    - metrics.json
```

DVC in Our Architecture

```
┌─────────────────────────────────────────┐
│              Data Layer                  │
├─────────────────────────────────────────┤
│  Raw Data ──→ DVC ──→ Processed Data     │
│       │        │          │              │
│       │        │          └──→ Training  │
│       │        │                         │
│       │        └──→ Version Control      │
│       │                                  │
│       └──→ Remote Storage (S3/GCS)       │
└─────────────────────────────────────────┘
```

Official Documentation

- **Website**: https://dvc.org/
- **GitHub**: https://github.com/iterative/dvc
- **Documentation**: https://dvc.org/doc

Model Versioning & Experiment Tracking (MLflow)

What is MLflow?

MLflow is an open-source platform for managing the ML lifecycle, including experimentation, reproducibility, deployment, and model registry.

Core Components

1. MLflow Tracking

- Log parameters, metrics, and artifacts
- Compare experiments and runs
- Organize experiments by project

2. MLflow Models

- Standard format for packaging ML models
- Multiple deployment options (REST API, batch, streaming)

3. MLflow Model Registry

- Central model store with versioning
- Stage transitions (staging, production, archived)
- Model lineage and annotations

4. MLflow Projects

- Reusable and reproducible ML code
- Define dependencies and entry points

Advantages ✅

- **Comprehensive Tracking**: End-to-end experiment management
- **Model Registry**: Centralized model versioning and deployment
- **Technology Agnostic**: Works with any ML library
- **UI Interface**: Rich web interface for experiment comparison
- **REST API**: Programmatic access to all features

Disadvantages ❌

- **Complexity**: Can be overkill for simple projects
- **Resource Usage**: Requires dedicated infrastructure
- **Learning Curve**: Multiple concepts to understand

Usage Example

```
import mlflow
import mlflow.sklearn
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score

# Start MLflow experiment
mlflow.set_experiment("iris_classification")

with mlflow.start_run():
    # Log parameters
    n_estimators = 100
    max_depth = 5
    mlflow.log_param("n_estimators", n_estimators)
    mlflow.log_param("max_depth", max_depth)

    # Train model
    model = RandomForestClassifier(
        n_estimators=n_estimators,
        max_depth=max_depth
    )
    model.fit(X_train, y_train)

    # Log metrics
    predictions = model.predict(X_test)
    accuracy = accuracy_score(y_test, predictions)
    mlflow.log_metric("accuracy", accuracy)

    # Log model
    mlflow.sklearn.log_model(
        model,
        "model",
        registered_model_name="iris_classifier"
    )
```

MLflow UI Features

The MLflow UI provides: - **Experiment Comparison**: Side-by-side parameter and metric comparison - **Run Details**: Detailed view of individual experiments - **Model Registry**: Model version management interface - **Artifact Browser**: View and download logged artifacts

MLflow in Our Architecture

```
┌─────────────────────────────────────────┐
│              Training Layer             │
├─────────────────────────────────────────┤
│  Training Script ──→ MLflow Tracking    │
│          │                │             │
│          │                ├──→ Metrics  │
│          │                ├──→ Parameters │
│          │                └──→ Artifacts │
│          │                              │
│          └──→ MLflow Registry ──→ Models │
│                           │             │
│                           └──→ Serving  │
└─────────────────────────────────────────┘
```

Official Documentation

- **Website**: https://mlflow.org/
- **GitHub**: https://github.com/mlflow/mlflow
- **Documentation**: https://mlflow.org/docs/latest/index.html

---

Model Serving (FastAPI)

What is FastAPI?

FastAPI is a modern, fast web framework for building APIs with Python 3.7+ based on standard Python type hints.

Key Features

Advantages ✅

- **High Performance**: One of the fastest Python frameworks
- **Automatic Documentation**: Interactive API docs (Swagger UI)
- **Type Safety**: Built-in validation with Python type hints
- **Async Support**: Native async/await support
- **Easy Testing**: Built-in testing support

- **Standards-based**: OpenAPI and JSON Schema compliance

Disadvantages ❌

- **Relatively New**: Smaller ecosystem compared to Flask/Django
- **Learning Curve**: Type hints and async concepts
- **Dependency Management**: Can become complex with many dependencies

Usage Example

```
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel
import pickle
import numpy as np

app = FastAPI(
    title="ML Model API",
    description="API for serving ML models",
    version="1.0.0"
)

# Load model
with open("models/iris_model.pkl", "rb") as f:
    model = pickle.load(f)

class PredictionRequest(BaseModel):
    sepal_length: float
    sepal_width: float
    petal_length: float
    petal_width: float

class PredictionResponse(BaseModel):
    prediction: str
    confidence: float
    probabilities: dict

@app.post("/predict/iris", response_model=PredictionResponse)
async def predict_iris(request: PredictionRequest):
    try:
        # Prepare features
        features = np.array([[
            request.sepal_length,
            request.sepal_width,
            request.petal_length,
            request.petal_width
        ]])

        # Make prediction
        prediction = model.predict(features)[0]
        probabilities = model.predict_proba(features)[0]

        class_names = ['setosa', 'versicolor', 'virginica']

        return PredictionResponse(
            prediction=class_names[prediction],
            confidence=float(max(probabilities)),
            probabilities={
                class_names[i]: float(prob)
                for i, prob in enumerate(probabilities)
            }
        )
    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))
```

```
@app.get("/health")
async def health_check():
    return {"status": "healthy"}
```

FastAPI Features

Automatic Documentation

- **Swagger UI**: Interactive API documentation at `/docs`
- **ReDoc**: Alternative documentation at `/redoc`
- **OpenAPI Schema**: Machine-readable API specification

Request/Response Models

```
from pydantic import BaseModel, validator

class ModelInput(BaseModel):
    feature1: float
    feature2: float

    @validator('feature1')
    def validate_feature1(cls, v):
        if v < 0:
            raise ValueError('feature1 must be positive')
        return v
```

FastAPI in Our Architecture

```
┌──────────────────────────────────────────┐
│             Serving Layer                │
├──────────────────────────────────────────┤
│  Client Request ──→ FastAPI Router       │
│         │                │               │
│         │                ├──→ Validation │
│         │                ├──→ Model Load │
│         │                ├──→ Prediction │
│         │                └──→ Response   │
│         │                                │
│         └──→ Monitoring ──→ Metrics      │
└──────────────────────────────────────────┘
```

Official Documentation

- **Website**: https://fastapi.tiangolo.com/
- **GitHub**: https://github.com/tiangolo/fastapi
- **Documentation**: https://fastapi.tiangolo.com/

Monitoring & Observability (Prometheus + Grafana)

What is Prometheus?

Prometheus is an open-source monitoring and alerting toolkit designed for reliability and scalability.

What is Grafana?

Grafana is an open-source analytics and interactive visualization web application.

Prometheus Features

Advantages ✅

- **Time Series Database**: Efficient storage and querying
- **Pull-based Model**: Scrapes metrics from targets
- **PromQL**: Powerful query language
- **Service Discovery**: Automatic target discovery
- **Alerting**: Built-in alerting capabilities

Disadvantages ❌

- **Local Storage**: Not suitable for long-term storage
- **Complexity**: Can be complex to set up initially
- **Resource Usage**: Memory intensive for large deployments

Grafana Features

Advantages ✅

- **Rich Visualizations**: Multiple chart types and dashboards
- **Data Source Agnostic**: Supports many data sources
- **Alerting**: Advanced alerting capabilities
- **User Management**: Role-based access control
- **Plugins**: Extensive plugin ecosystem

Disadvantages ❌

- **Performance**: Can be slow with large datasets
- **Complexity**: Dashboard creation can be complex
- **Resource Usage**: Memory and CPU intensive

Usage Example

Prometheus Configuration

```
# prometheus.yml
global:
  scrape_interval: 15s

scrape_configs:
  - job_name: 'mlops-api'
    static_configs:
      - targets: ['api:8000']
    metrics_path: '/metrics'
    scrape_interval: 5s

  - job_name: 'prometheus'
    static_configs:
      - targets: ['localhost:9090']
```

Python Metrics Collection

```
from prometheus_client import Counter, Histogram, Gauge
import time

# Define metrics
PREDICTION_COUNTER = Counter(
    'ml_predictions_total',
    'Total number of predictions',
    ['model_name', 'model_version']
)

PREDICTION_LATENCY = Histogram(
    'ml_prediction_duration_seconds',
    'Time spent on predictions',
    ['model_name']
)

MODEL_ACCURACY = Gauge(
    'ml_model_accuracy',
    'Current model accuracy',
    ['model_name', 'model_version']
)

@app.post("/predict")
async def predict(request: PredictionRequest):
    start_time = time.time()

    # Make prediction
    result = model.predict(request.features)

    # Record metrics
    PREDICTION_COUNTER.labels(
        model_name='iris',
        model_version='v1.0'
    ).inc()

    PREDICTION_LATENCY.labels(
        model_name='iris'
    ).observe(time.time() - start_time)

    return result
```

Grafana Dashboard Configuration

```
{
  "dashboard": {
    "title": "MLOps Monitoring",
    "panels": [
      {
        "title": "Prediction Rate",
        "type": "graph",
        "targets": [
          {
            "expr": "rate(ml_predictions_total[5m])",
            "legendFormat": "{{model_name}}"
          }
        ]
      },
      {
        "title": "Prediction Latency",
        "type": "graph",
        "targets": [
          {
            "expr": "histogram_quantile(0.95, ml_prediction_duration_seconds_bucket)",
            "legendFormat": "95th percentile"
          }
        ]
      }
    ]
  }
}
```

Key Metrics for ML Systems

Business Metrics

- **Prediction Volume**: Number of predictions per time unit
- **Response Time**: API response latency
- **Error Rate**: Failed predictions percentage
- **User Engagement**: API usage patterns

Model Performance Metrics

- **Accuracy**: Model accuracy over time
- **Data Drift**: Input data distribution changes
- **Model Drift**: Prediction distribution changes

- **Feature Importance**: Changes in feature contributions

Infrastructure Metrics

- **CPU/Memory Usage**: Resource utilization
- **Disk I/O**: Model loading performance
- **Network**: API traffic patterns
- **Container Health**: Service availability

Monitoring Architecture

```
┌─────────────────────────────────────────┐
│          Monitoring Layer               │
├─────────────────────────────────────────┤
│  Application ──→ Prometheus ──→ Grafana │
│      │              │            │      │
│      ├──→ Metrics   │            │      │
│      ├──→ Logs      │            │      │
│      └──→ Traces    │            │      │
│                     │            │      │
│                     ├──→ Alerts  │      │
│                     └──→ Storage │      │
│                                  │      │
│                          ┌───────┘      │
│                          │              │
│                          └──→ Dashboards│
└─────────────────────────────────────────┘
```

Official Documentation

- **Prometheus**: https://prometheus.io/docs/
- **Grafana**: https://grafana.com/docs/grafana/latest/

CI/CD Pipelines (GitHub Actions)

What is GitHub Actions?

GitHub Actions is a CI/CD platform that allows you to automate your build, test, and deployment pipeline directly from your GitHub repository.

Key Concepts

Workflows

- YAML files that define automated processes
- Triggered by events (push, pull request, schedule)
- Composed of one or more jobs

Jobs

- Set of steps that execute on the same runner
- Can run in parallel or sequentially
- Each job runs in a fresh virtual environment

Actions

- Reusable units of code
- Can be custom or from GitHub Marketplace
- Input/output parameters for flexibility

Advantages ✅

- **Native Integration**: Built into GitHub ecosystem
- **Extensive Marketplace**: Thousands of pre-built actions
- **Matrix Builds**: Test across multiple environments
- **Secrets Management**: Secure environment variables
- **Free Tier**: Generous free usage for public repos

Disadvantages ❌

- **GitHub Dependency**: Locked to GitHub platform
- **Learning Curve**: YAML syntax and concepts
- **Cost**: Can be expensive for private repos with heavy usage
- **Limited Runners**: Resource constraints on hosted runners

Usage Example

Basic ML Pipeline

```yaml
# .github/workflows/ml-pipeline.yml
name: ML Pipeline

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

jobs:
  test:
    runs-on: ubuntu-latest

    strategy:
      matrix:
        python-version: [3.8, 3.9]

    steps:
    - uses: actions/checkout@v3

    - name: Set up Python ${{ matrix.python-version }}
      uses: actions/setup-python@v3
      with:
        python-version: ${{ matrix.python-version }}

    - name: Install dependencies
      run: |
        python -m pip install --upgrade pip
        pip install -r requirements.txt

    - name: Run tests
      run: |
        pytest tests/ -v --cov=src

    - name: Run data quality checks
      run: |
        python scripts/data_validation.py

    - name: Train models
      run: |
        python scripts/train_all_models.py

    - name: Model validation
      run: |
        python scripts/model_validation.py

  deploy:
    needs: test
    runs-on: ubuntu-latest
    if: github.ref == 'refs/heads/main'

    steps:
    - uses: actions/checkout@v3
```

```
    - name: Build Docker image
      run: |
        docker build -t mlops-api .

    - name: Deploy to staging
      run: |
        docker-compose -f docker-compose.staging.yml up -d

    - name: Run integration tests
      run: |
        python tests/integration_tests.py

    - name: Deploy to production
      if: success()
      run: |
        echo "Deploying to production"
        # Production deployment commands
```

Advanced ML Workflow

```
name: Advanced ML Pipeline

on:
  push:
    branches: [ main ]
    paths: ['models/**', 'data/**', 'api/**']

env:
  MLFLOW_TRACKING_URI: ${{ secrets.MLFLOW_TRACKING_URI }}
  AWS_ACCESS_KEY_ID: ${{ secrets.AWS_ACCESS_KEY_ID }}
  AWS_SECRET_ACCESS_KEY: ${{ secrets.AWS_SECRET_ACCESS_KEY }}

jobs:
  data-validation:
    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v3
      with:
        lfs: true

    - name: DVC Setup
      uses: iterative/setup-dvc@v1

    - name: Data validation
      run: |
        dvc pull
        python scripts/validate_data.py

    - name: Upload validation report
      uses: actions/upload-artifact@v3
      with:
        name: data-validation-report
        path: reports/data_validation.html

  model-training:
    needs: data-validation
    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v3

    - name: Train models
      run: |
        python scripts/train_all_models.py

    - name: Model evaluation
      run: |
        python scripts/evaluate_models.py

    - name: Register models
      run: |
        python scripts/register_models.py
```

```
  security-scan:
    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v3

    - name: Run security scan
      uses: pypa/gh-action-pip-audit@v1.0.0
      with:
        inputs: requirements.txt

    - name: Container security scan
      uses: aquasecurity/trivy-action@master
      with:
        image-ref: 'mlops-api:latest'
        format: 'sarif'
        output: 'trivy-results.sarif'

  performance-testing:
    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v3

    - name: Start services
      run: |
        docker-compose up -d

    - name: Load testing
      run: |
        pip install locust
        locust -f tests/load_test.py --headless -u 50 -r 5 -t 300s --host http://localhost:8000

    - name: Cleanup
      run: |
        docker-compose down
```

GitHub Actions in Our Architecture

```
┌─────────────────────────────────────────┐
│              CI/CD Layer                │
├─────────────────────────────────────────┤
│  Code Push ──→ GitHub Actions           │
│       │             │                  │
│       │             ├──→ Test Stage     │
│       │             ├──→ Build Stage    │
│       │             ├──→ Deploy Stage   │
│       │             └──→ Monitor Stage  │
│       │                                │
│       └──→ Notifications ──→ Slack/Email│
└─────────────────────────────────────────┘
```

Best Practices

Security

- Use secrets for sensitive data
- Limit workflow permissions
- Pin action versions
- Review third-party actions

Performance

- Use caching for dependencies
- Minimize workflow runtime
- Use self-hosted runners for heavy workloads
- Parallelize independent jobs

Reliability

- Add retry mechanisms
- Use status checks
- Implement proper error handling
- Monitor workflow performance

Official Documentation

- **Website**: https://github.com/features/actions
- **Documentation**: https://docs.github.com/en/actions

Containerization (Docker)

What is Docker?

Docker is a platform that enables developers to package applications and their dependencies into lightweight, portable containers.

Key Concepts

Containers

- Lightweight, standalone packages
- Include application code, runtime, libraries, and dependencies
- Isolated from the host system

Images

- Read-only templates for creating containers
- Built using Dockerfiles
- Can be stored in registries (Docker Hub, ECR, GCR)

Dockerfile

- Text file with instructions to build images
- Defines the application environment step by step

Advantages ✅

- **Consistency**: Same environment across dev/staging/prod
- **Portability**: Runs anywhere Docker is installed

- **Scalability**: Easy horizontal scaling
- **Resource Efficiency**: Lower overhead than VMs
- **Isolation**: Process and filesystem isolation

Disadvantages ❌

- **Complexity**: Additional abstraction layer
- **Storage Overhead**: Images can be large
- **Security Concerns**: Shared kernel with host
- **Learning Curve**: New concepts and commands

Usage Example

Dockerfile for ML API

```
# Dockerfile
FROM python:3.9-slim

# Set working directory
WORKDIR /app

# Install system dependencies
RUN apt-get update && apt-get install -y \
    gcc \
    && rm -rf /var/lib/apt/lists/*

# Copy requirements first for better caching
COPY requirements.txt .

# Install Python dependencies
RUN pip install --no-cache-dir -r requirements.txt

# Copy application code
COPY . .

# Create non-root user
RUN useradd --create-home --shell /bin/bash mlops
USER mlops

# Expose port
EXPOSE 8000

# Health check
HEALTHCHECK --interval=30s --timeout=10s --start-period=5s --retries=3 \
    CMD curl -f http://localhost:8000/health || exit 1

# Start application
CMD ["uvicorn", "api.main:app", "--host", "0.0.0.0", "--port", "8000"]
```

Multi-stage Build

```
# Multi-stage Dockerfile for optimization
FROM python:3.9 as builder

WORKDIR /app

# Install build dependencies
RUN pip install --no-cache-dir poetry

# Copy dependency files
COPY pyproject.toml poetry.lock ./

# Install dependencies
RUN poetry config virtualenvs.create false \
    && poetry install --no-dev --no-interaction --no-ansi

# Production stage
FROM python:3.9-slim

WORKDIR /app

# Copy installed packages from builder
COPY --from=builder /usr/local/lib/python3.9/site-packages /usr/local/lib/python3.9/site-packages
COPY --from=builder /usr/local/bin /usr/local/bin

# Copy application
COPY . .

# Run as non-root
RUN useradd --create-home mlops
USER mlops

EXPOSE 8000

CMD ["uvicorn", "api.main:app", "--host", "0.0.0.0", "--port", "8000"]
```

Docker Compose for MLOps Stack

```
# docker-compose.yml
version: '3.8'

services:
  api:
    build: .
    ports:
      - "8000:8000"
    environment:
      - MLFLOW_TRACKING_URI=http://mlflow:5000
      - PROMETHEUS_MULTIPROC_DIR=/tmp
    depends_on:
      - mlflow
      - prometheus
    volumes:
      - ./models:/app/models
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:8000/health"]
      interval: 30s
      timeout: 10s
      retries: 3

  mlflow:
    image: mlflow/mlflow:latest
    ports:
      - "5000:5000"
    environment:
      - BACKEND_STORE_URI=sqlite:///mlflow.db
      - DEFAULT_ARTIFACT_ROOT=/mlflow/artifacts
    volumes:
      - mlflow_data:/mlflow
    command: >
      mlflow server
      --backend-store-uri sqlite:///mlflow.db
      --default-artifact-root /mlflow/artifacts
      --host 0.0.0.0
      --port 5000

  prometheus:
    image: prom/prometheus:latest
    ports:
      - "9090:9090"
    volumes:
      - ./monitoring/prometheus.yml:/etc/prometheus/prometheus.yml
      - prometheus_data:/prometheus
    command:
      - '--config.file=/etc/prometheus/prometheus.yml'
      - '--storage.tsdb.path=/prometheus'
      - '--web.console.libraries=/usr/share/prometheus/console_libraries'
      - '--web.console.templates=/usr/share/prometheus/consoles'

  grafana:
    image: grafana/grafana:latest
```

```
    ports:
      - "3000:3000"
    environment:
      - GF_SECURITY_ADMIN_PASSWORD=admin123
    volumes:
      - grafana_data:/var/lib/grafana
      - ./monitoring/grafana/datasources:/etc/grafana/provisioning/datasources
      - ./monitoring/grafana/dashboards:/etc/grafana/provisioning/dashboards

  jupyter:
    image: jupyter/datascience-notebook:latest
    ports:
      - "8888:8888"
    environment:
      - JUPYTER_ENABLE_LAB=yes
    volumes:
      - ./notebooks:/home/jovyan/work
    command: start-notebook.sh --NotebookApp.token=''

volumes:
  mlflow_data:
  prometheus_data:
  grafana_data:
```

Docker in Our Architecture

```
┌─────────────────────────────────────────┐
│          Container Layer                │
├─────────────────────────────────────────┤
│                                         │
│  ┌─────────┐  ┌─────────┐  ┌─────────┐  │
│  │   API   │  │ MLflow  │  │Prometheus│  │
│  │Container│  │Container│  │Container │  │
│  └─────────┘  └─────────┘  └─────────┘  │
│       │            │            │       │
│       └────────────┼────────────┘       │
│                    │                    │
│  ┌─────────┐  ┌─────────┐  ┌─────────┐  │
│  │ Grafana │  │ Jupyter │  │  Redis  │  │
│  │Container│  │Container│  │Container│  │
│  └─────────┘  └─────────┘  └─────────┘  │
└─────────────────────────────────────────┘
```

Container Best Practices

Security

- Use official base images

- Run as non-root user
- Scan images for vulnerabilities
- Keep images updated
- Use multi-stage builds

Performance

- Minimize layers
- Use .dockerignore
- Leverage build cache
- Optimize image size
- Use specific tags

Reliability

- Implement health checks
- Set resource limits
- Use restart policies
- Monitor container metrics
- Log to stdout/stderr

Official Documentation

- **Website**: https://www.docker.com/
- **Documentation**: https://docs.docker.com/

Data Drift Detection (Evidently AI)

What is Evidently AI?

Evidently AI is an open-source Python library for evaluating, testing, and monitoring ML model quality throughout the model lifecycle.

Key Capabilities

Data Drift Detection

- Statistical tests for feature distribution changes
- Visual drift analysis and reporting
- Automatic drift detection algorithms

Model Performance Monitoring

- Classification and regression metrics
- Performance degradation detection
- Target drift analysis

Data Quality Assessment

- Missing values analysis
- Data integrity checks
- Feature correlation analysis

Advantages ✅

- **Comprehensive Analysis**: Multiple drift detection methods
- **Visual Reports**: Rich HTML reports and dashboards
- **Real-time Monitoring**: Integration with monitoring systems
- **No Target Required**: Works without ground truth labels
- **Easy Integration**: Simple Python API

Disadvantages ❌

- **Resource Intensive**: Can be slow with large datasets
- **Learning Curve**: Understanding statistical concepts
- **Memory Usage**: High memory requirements for large datasets
- **Limited Customization**: Fixed report templates

Usage Example

Basic Drift Detection

```
import pandas as pd
from evidently import ColumnMapping
from evidently.report import Report
from evidently.metric_suite import MetricSuite
from evidently.metrics import DataDriftMetric, DatasetDriftMetric

# Load reference and current data
reference_data = pd.read_csv('data/reference.csv')
current_data = pd.read_csv('data/current.csv')

# Define column mapping
column_mapping = ColumnMapping(
    target='target',
    prediction='prediction',
    numerical_features=['feature1', 'feature2', 'feature3'],
    categorical_features=['category1', 'category2']
)

# Create drift report
report = Report(metrics=[
    DatasetDriftMetric(),
    DataDriftMetric()
])

# Generate report
report.run(
    reference_data=reference_data,
    current_data=current_data,
    column_mapping=column_mapping
)

# Save report
report.save_html('reports/data_drift_report.html')

# Get drift results
drift_results = report.as_dict()
dataset_drift = drift_results['metrics'][0]['result']['dataset_drift']
print(f"Dataset drift detected: {dataset_drift}")
```

Model Performance Monitoring

```
from evidently.metrics import ClassificationPerformanceMetric
from evidently.metric_suite import MetricSuite

# Create performance report
report = Report(metrics=[
    ClassificationPerformanceMetric(),
    DataDriftMetric(),
    DatasetDriftMetric()
])

# Run analysis
report.run(
    reference_data=reference_data,
    current_data=current_data,
    column_mapping=column_mapping
)

# Extract metrics
results = report.as_dict()
accuracy = results['metrics'][0]['result']['accuracy']
precision = results['metrics'][0]['result']['precision']
recall = results['metrics'][0]['result']['recall']

print(f"Current accuracy: {accuracy}")
print(f"Current precision: {precision}")
print(f"Current recall: {recall}")
```

Real-time Monitoring Integration

```
from evidently.monitors import MonitoringService
from evidently.options import DataDriftOptions
import json

class DriftMonitor:
    def __init__(self, reference_data, column_mapping):
        self.reference_data = reference_data
        self.column_mapping = column_mapping

    def check_drift(self, current_batch):
        """Check for drift in current batch"""
        report = Report(metrics=[
            DatasetDriftMetric(
                options=DataDriftOptions(
                    drift_threshold=0.3
                )
            )
        ])

        report.run(
            reference_data=self.reference_data,
            current_data=current_batch,
            column_mapping=self.column_mapping
        )

        results = report.as_dict()
        drift_detected = results['metrics'][0]['result']['dataset_drift']

        if drift_detected:
            self._send_alert(results)

        return drift_detected

    def _send_alert(self, results):
        """Send drift alert"""
        alert = {
            'timestamp': pd.Timestamp.now().isoformat(),
            'drift_detected': True,
            'drift_score': results['metrics'][0]['result']['drift_score'],
            'affected_features': [
                feature for feature, details in
                results['metrics'][0]['result']['drift_by_columns'].items()
                if details['drift_detected']
            ]
        }

        # Send to monitoring system
        print(f"DRIFT ALERT: {json.dumps(alert, indent=2)}")

# Usage
monitor = DriftMonitor(reference_data, column_mapping)

# Monitor incoming batches
```

```
for batch in data_stream:
    drift_detected = monitor.check_drift(batch)
    if drift_detected:
        print("Drift detected! Consider retraining model.")
```

Drift Detection Methods

Statistical Tests

- **Kolmogorov-Smirnov**: Distribution comparison for numerical features
- **Chi-squared**: Independence test for categorical features
- **Jensen-Shannon**: Divergence-based drift detection
- **Population Stability Index**: Stability measurement

Visualization

- **Distribution Plots**: Feature distribution comparison
- **Drift Score Heatmap**: Feature-wise drift visualization
- **Correlation Analysis**: Feature relationship changes
- **Time Series Plots**: Drift over time

Evidently in Our Architecture

```
┌──────────────────────────────────────────┐
│          Monitoring Layer                │
├──────────────────────────────────────────┤
│  Incoming Data ──→ Evidently AI          │
│         │                │               │
│         │                ├──→ Drift Check│
│         │                ├──→ Reports    │
│         │                └──→ Alerts     │
│         │                                │
│         └──→ Model Retraining ←──────────│
│                     │                    │
│                     └──→ MLflow          │
└──────────────────────────────────────────┘
```

Integration with MLOps Pipeline

```
# Automated drift detection in MLOps pipeline
class MLOpsPipeline:
    def __init__(self):
        self.drift_monitor = DriftMonitor()
        self.model_trainer = ModelTrainer()
        self.model_deployer = ModelDeployer()

    def process_batch(self, data_batch):
        # Check for drift
        drift_detected = self.drift_monitor.check_drift(data_batch)

        if drift_detected:
            print("Drift detected. Initiating retraining...")

            # Retrain model
            new_model = self.model_trainer.retrain(data_batch)

            # Validate new model
            if self._validate_model(new_model):
                # Deploy new model
                self.model_deployer.deploy(new_model)
                print("New model deployed successfully.")
            else:
                print("New model validation failed. Keeping current model.")

        return drift_detected
```

Official Documentation

- **Website**: https://evidentlyai.com/
- **GitHub**: https://github.com/evidentlyai/evidently
- **Documentation**: https://docs.evidentlyai.com/

Automated Testing

What is Automated Testing in MLOps?

Automated Testing in MLOps involves systematic testing of data, models, and infrastructure to ensure reliability and performance of ML systems.

Types of ML Testing

1. Data Testing

- **Schema Validation**: Ensure data structure consistency
- **Data Quality**: Check for missing values, outliers, duplicates
- **Data Drift**: Monitor distribution changes over time
- **Feature Engineering**: Validate feature transformations

2. Model Testing

- **Unit Tests**: Test individual model components
- **Integration Tests**: Test end-to-end model pipeline
- **Performance Tests**: Validate model accuracy and speed
- **Regression Tests**: Ensure model improvements don't break existing functionality

3. Infrastructure Testing

- **API Testing**: Test model serving endpoints
- **Load Testing**: Validate system performance under load
- **Security Testing**: Check for vulnerabilities
- **Deployment Testing**: Validate deployment processes

Advantages ✅

- **Early Bug Detection**: Catch issues before production
- **Regression Prevention**: Ensure changes don't break existing functionality
- **Quality Assurance**: Maintain high standards across deployments
- **Faster Development**: Automated feedback enables rapid iteration
- **Documentation**: Tests serve as living documentation

Disadvantages ❌

- **Initial Setup Cost**: Time investment to create comprehensive tests
- **Maintenance Overhead**: Tests need to be updated with code changes
- **False Positives**: Flaky tests can slow development

- **Coverage Complexity**: Difficult to test all edge cases in ML

Testing Framework Structure

```
tests/
├── unit/
│   ├── test_data_processing.py
│   ├── test_feature_engineering.py
│   └── test_model_components.py
├── integration/
│   ├── test_model_pipeline.py
│   ├── test_api_endpoints.py
│   └── test_monitoring.py
├── performance/
│   ├── test_model_latency.py
│   ├── test_memory_usage.py
│   └── test_load_testing.py
└── fixtures/
    ├── sample_data.py
    └── mock_models.py
```

Usage Examples

Data Testing

```
import pytest
import pandas as pd
import numpy as np
from src.data_validation import DataValidator

class TestDataValidation:

    @pytest.fixture
    def sample_data(self):
        return pd.DataFrame({
            'feature1': [1, 2, 3, 4, 5],
            'feature2': [0.1, 0.2, 0.3, 0.4, 0.5],
            'target': [0, 1, 0, 1, 0]
        })

    def test_data_schema(self, sample_data):
        """Test data schema validation"""
        validator = DataValidator()

        # Expected schema
        expected_columns = ['feature1', 'feature2', 'target']
        expected_types = {
            'feature1': 'int64',
            'feature2': 'float64',
            'target': 'int64'
        }

        # Validate schema
        assert list(sample_data.columns) == expected_columns
        for col, dtype in expected_types.items():
            assert sample_data[col].dtype == dtype

    def test_data_quality(self, sample_data):
        """Test data quality checks"""
        # No missing values
        assert not sample_data.isnull().any().any()

        # Value ranges
        assert sample_data['feature1'].between(1, 10).all()
        assert sample_data['feature2'].between(0, 1).all()
        assert sample_data['target'].isin([0, 1]).all()

    def test_data_drift(self):
        """Test data drift detection"""
        # Reference data
        reference = np.random.normal(0, 1, 1000)

        # Current data (no drift)
        current_no_drift = np.random.normal(0, 1, 1000)

        # Current data (with drift)
        current_with_drift = np.random.normal(2, 1, 1000)
```

```
from scipy.stats import ks_2samp

# No drift case
_, p_value_no_drift = ks_2samp(reference, current_no_drift)
assert p_value_no_drift > 0.05  # No significant difference

# Drift case
_, p_value_drift = ks_2samp(reference, current_with_drift)
assert p_value_drift < 0.05  # Significant difference
```

Model Testing

```
import pytest
import pickle
import numpy as np
from sklearn.metrics import accuracy_score
from src.models import IrisClassifier

class TestIrisModel:

    @pytest.fixture
    def trained_model(self):
        """Load trained model for testing"""
        with open('models/iris/artifacts/iris_model.pkl', 'rb') as f:
            return pickle.load(f)

    @pytest.fixture
    def test_data(self):
        """Sample test data"""
        return {
            'X': np.array([[5.1, 3.5, 1.4, 0.2]]),  # Setosa
            'y': np.array([0])
        }

    def test_model_prediction(self, trained_model, test_data):
        """Test model prediction functionality"""
        prediction = trained_model.predict(test_data['X'])

        # Check prediction format
        assert isinstance(prediction, np.ndarray)
        assert len(prediction) == len(test_data['X'])
        assert prediction[0] in [0, 1, 2]  # Valid class labels

    def test_model_accuracy(self, trained_model):
        """Test model accuracy threshold"""
        # Load test dataset
        from sklearn.datasets import load_iris
        from sklearn.model_selection import train_test_split

        iris = load_iris()
        X_train, X_test, y_train, y_test = train_test_split(
            iris.data, iris.target, test_size=0.2, random_state=42
        )

        # Test accuracy
        predictions = trained_model.predict(X_test)
        accuracy = accuracy_score(y_test, predictions)

        assert accuracy >= 0.9  # Minimum 90% accuracy

    def test_model_latency(self, trained_model, test_data):
        """Test prediction latency"""
        import time

        start_time = time.time()
```

```
        prediction = trained_model.predict(test_data['X'])
        end_time = time.time()

        latency = end_time - start_time
        assert latency < 0.1  # Less than 100ms

    def test_model_robustness(self, trained_model):
        """Test model robustness to edge cases"""
        # Edge cases
        edge_cases = [
            [0, 0, 0, 0],          # All zeros
            [10, 10, 10, 10],      # High values
            [0.1, 0.1, 0.1, 0.1]   # Very small values
        ]

        for case in edge_cases:
            prediction = trained_model.predict([case])
            assert prediction[0] in [0, 1, 2]
```

API Testing

```
import pytest
from fastapi.testclient import TestClient
from api.main import app

class TestMLAPI:

    @pytest.fixture
    def client(self):
        return TestClient(app)

    def test_health_endpoint(self, client):
        """Test health check endpoint"""
        response = client.get("/health")
        assert response.status_code == 200
        assert response.json()["status"] == "healthy"

    def test_iris_prediction(self, client):
        """Test iris prediction endpoint"""
        test_data = {
            "sepal_length": 5.1,
            "sepal_width": 3.5,
            "petal_length": 1.4,
            "petal_width": 0.2
        }

        response = client.post("/predict/iris", json=test_data)

        assert response.status_code == 200
        result = response.json()

        # Check response structure
        assert "prediction" in result
        assert "confidence" in result
        assert "probabilities" in result

        # Check data types
        assert isinstance(result["prediction"], str)
        assert isinstance(result["confidence"], float)
        assert isinstance(result["probabilities"], dict)

        # Check value ranges
        assert 0 <= result["confidence"] <= 1
        assert result["prediction"] in ["setosa", "versicolor", "virginica"]

    def test_invalid_input(self, client):
        """Test API with invalid input"""
        invalid_data = {
            "sepal_length": "invalid",
            "sepal_width": 3.5,
            "petal_length": 1.4,
            "petal_width": 0.2
        }
```

```
        response = client.post("/predict/iris", json=invalid_data)
        assert response.status_code == 422  # Validation error

    def test_api_performance(self, client):
        """Test API response time"""
        import time

        test_data = {
            "sepal_length": 5.1,
            "sepal_width": 3.5,
            "petal_length": 1.4,
            "petal_width": 0.2
        }

        start_time = time.time()
        response = client.post("/predict/iris", json=test_data)
        end_time = time.time()

        assert response.status_code == 200
        assert (end_time - start_time) < 1.0  # Less than 1 second
```

Load Testing

```
# tests/performance/test_load.py
import concurrent.futures
import time
import requests
import statistics

class TestLoadPerformance:

    def test_concurrent_requests(self):
        """Test API under concurrent load"""
        base_url = "http://localhost:8000"
        test_data = {
            "sepal_length": 5.1,
            "sepal_width": 3.5,
            "petal_length": 1.4,
            "petal_width": 0.2
        }

        def make_request():
            start_time = time.time()
            response = requests.post(
                f"{base_url}/predict/iris",
                json=test_data
            )
            end_time = time.time()
            return {
                'status_code': response.status_code,
                'response_time': end_time - start_time
            }

        # Simulate 50 concurrent users
        with concurrent.futures.ThreadPoolExecutor(max_workers=50) as executor:
            futures = [executor.submit(make_request) for _ in range(100)]
            results = [future.result() for future in futures]

        # Analyze results
        success_count = sum(1 for r in results if r['status_code'] == 200)
        response_times = [r['response_time'] for r in results]

        # Assertions
        assert success_count >= 95  # 95% success rate
        assert statistics.mean(response_times) < 0.5  # Average < 500ms
        assert max(response_times) < 2.0  # Max < 2 seconds
```

Test Configuration

pytest.ini

```
[tool:pytest]
testpaths = tests
python_files = test_*.py
python_classes = Test*
python_functions = test_*
addopts =
    -v
    --tb=short
    --cov=src
    --cov-report=html
    --cov-report=term-missing
    --cov-fail-under=80
markers =
    unit: Unit tests
    integration: Integration tests
    performance: Performance tests
    slow: Slow running tests
```

conftest.py

```
# tests/conftest.py
import pytest
import pandas as pd
import numpy as np
from unittest.mock import Mock

@pytest.fixture(scope="session")
def sample_iris_data():
    """Sample iris dataset for testing"""
    np.random.seed(42)
    return pd.DataFrame({
        'sepal_length': np.random.uniform(4, 8, 100),
        'sepal_width': np.random.uniform(2, 5, 100),
        'petal_length': np.random.uniform(1, 7, 100),
        'petal_width': np.random.uniform(0, 3, 100),
        'target': np.random.choice([0, 1, 2], 100)
    })

@pytest.fixture
def mock_model():
    """Mock model for testing"""
    model = Mock()
    model.predict.return_value = np.array([0])
    model.predict_proba.return_value = np.array([[0.8, 0.1, 0.1]])
    return model

@pytest.fixture(scope="session", autouse=True)
def setup_test_environment():
    """Setup test environment"""
    import os
    os.environ['TESTING'] = 'true'
    yield
    del os.environ['TESTING']
```

Testing in Our Architecture

```
┌─────────────────────────────────────────────────────────┐
│                 Testing Layer                           │
├─────────────────────────────────────────────────────────┤
│                                                         │
│   ┌─────────────┐  ┌─────────────┐  ┌─────────────┐     │
│   │    Unit     │  │Integration  │  │Performance│   │
│   │   Tests     │  │   Tests     │  │   Tests     │   │
│   └─────────────┘  └─────────────┘  └─────────────┘     │
│                                                         │
│          │                │                │            │
│          └────────────────┼────────────────┘            │
│                           │                             │
│          ┌────────────────▼────────────────┐            │
│          │      Continuous Testing         │            │
│          │     (GitHub Actions)            │            │
│          └─────────────────────────────────┘            │
└─────────────────────────────────────────────────────────┘
```

Official Documentation

- **Pytest**: https://docs.pytest.org/
- **FastAPI Testing**: https://fastapi.tiangolo.com/tutorial/testing/
- **MLOps Testing**: https://ml-ops.org/content/testing

Integration & Architecture

Complete MLOps Architecture

```
┌─────────────────────────────────────────────────────────────────┐
│                      MLOps Complete Stack                       │
├─────────────────────────────────────────────────────────────────┤
│                                                                 │
│  ┌───────────────────┐    ┌───────────────────┐                │
│  │   Development     │    │    Production     │                │
│  │   Environment     │    │   Environment     │                │
│  │                   │    │                   │                │
│  │  ┌─────────────┐  │    │  ┌─────────────┐  │                │
│  │  │    Data     │  │    │  │    Data     │  │                │
│  │  │ Versioning  │◄─┼────┼──┤ Monitoring  │  │                │
│  │  │   (DVC)     │  │    │  │(Evidently)  │  │                │
│  │  └─────────────┘  │    │  └─────────────┘  │                │
│  │         │         │    │         │         │                 │
│  │  ┌─────────────┐  │    │  ┌─────────────┐  │                │
│  │  │   Model     │  │    │  │   Model     │  │                │
│  │  │  Training   │◄─┼────┼──┤  Serving    │  │                │
│  │  │  (MLflow)   │  │    │  │ (FastAPI)   │  │                │
│  │  └─────────────┘  │    │  └─────────────┘  │                │
│  │         │         │    │         │         │                 │
│  │  ┌─────────────┐  │    │  ┌─────────────┐  │                │
│  │  │  Testing    │  │    │  │ Monitoring  │  │                │
│  │  │  (Pytest)   │  │    │  │(Prometheus/ │  │                │
│  │  │             │  │    │  │  Grafana)   │  │                │
│  │  └─────────────┘  │    │  └─────────────┘  │                │
│  └───────────────────┘    └───────────────────┘                │
│            │                          │                         │
│            └──────────────┬───────────┘                         │
│                           │                                     │
│                 ┌─────────────────┐                             │
│                 │     CI/CD       │                             │
│                 │ (GitHub Actions)│                             │
│                 │                 │                             │
│                 │  ┌─────────────┐│                             │
│                 │  │Containerization│                           │
│                 │  │  (Docker)   │ │                            │
│                 │  └─────────────┘│                             │
│                 └─────────────────┘                             │
│                                                                 │
└─────────────────────────────────────────────────────────────────┘
```

Data Flow Architecture

```
Input Data ──┐
             │
             ▼
    ┌─────────────────┐
    │  Data Pipeline  │
    │                 │
    │ ┌─────────────┐ │     ┌─────────────┐
    │ │     DVC     │◄┼─────┤ Data Store  │
    │ │ Versioning  │ │     │    (S3)     │
    │ └─────────────┘ │     └─────────────┘
    └─────────────────┘
             │
             ▼
    ┌─────────────────┐
    │ Model Training  │
    │                 │
    │ ┌─────────────┐ │     ┌─────────────┐
    │ │   MLflow    │◄┼─────┤Model Registry│
    │ │  Tracking   │ │     │  (MLflow)   │
    │ └─────────────┘ │     └─────────────┘
    └─────────────────┘
             │
             ▼
    ┌─────────────────┐
    │ Model Serving   │
    │                 │
    │ ┌─────────────┐ │     ┌─────────────┐
    │ │   FastAPI   │◄┼─────┤   Docker    │
    │ │    Server   │ │     │ Container   │
    │ └─────────────┘ │     └─────────────┘
    └─────────────────┘
             │
             ▼
    ┌─────────────────┐
    │   Monitoring    │
    │                 │
    │ ┌─────────────┐ │     ┌─────────────┐
    │ │ Prometheus  │◄┼─────┤   Grafana   │
    │ │   Metrics   │ │     │ Dashboard   │
    │ └─────────────┘ │     └─────────────┘
    └─────────────────┘
             │
             ▼
    ┌─────────────────┐
    │ Quality Gates   │
    │                 │
    │ ┌─────────────┐ │
    │ │ Evidently   │ │
    │ │   Drift     │ │
    │ └─────────────┘ │
    └─────────────────┘
```

Service Communication

```
┌─────────────────────────────────────────────────────────────────────┐
│                         Service Mesh                                │
├─────────────────────────────────────────────────────────────────────┤
│                                                                     │
│  ┌─────────────┐    ┌─────────────┐    ┌─────────────┐             │
│  │   FastAPI   │    │   MLflow    │    │ Prometheus  │             │
│  │   :8000     │◄──►│   :5000     │◄──►│   :9090     │             │
│  └─────────────┘    └─────────────┘    └─────────────┘             │
│         │                   │                   │                  │
│         │                   │                   │                  │
│  ┌─────────────┐    ┌─────────────┐    ┌─────────────┐             │
│  │   Grafana   │    │   Jupyter   │    │    Redis    │             │
│  │   :3000     │◄──►│   :8888     │◄──►│   :6379     │             │
│  └─────────────┘    └─────────────┘    └─────────────┘             │
│                                                                     │
│  ┌───────────────────────────────────────────────────────────┐     │
│  │              Load Balancer / API Gateway                  │     │
│  │                          :80                              │     │
│  └───────────────────────────────────────────────────────────┘     │
│                                                                     │
└─────────────────────────────────────────────────────────────────────┘
```

Deployment Pipeline

```
┌─────────────────────────────────────────────────────────────┐
│                    Deployment Pipeline                      │
├─────────────────────────────────────────────────────────────┤
│                                                             │
│  Developer Push                                             │
│         │                                                   │
│         ▼                                                   │
│  ┌─────────────┐                                            │
│  │   GitHub    │                                            │
│  │ Repository  │                                            │
│  └─────────────┘                                            │
│         │                                                   │
│         ▼                                                  │
│  ┌─────────────┐    ┌─────────────┐    ┌─────────────┐     │
│  │   Trigger   │───▶│    Test     │───▶│    Build    │     │
│  │    CI/CD    │    │   Suite     │    │   Docker    │     │
│  └─────────────┘    └─────────────┘    └─────────────┘     │
│                            │                  │            │
│                            ▼                  │            │
│                     ┌─────────────┐           │            │
│                     │  Security   │           │            │
│                     │    Scan     │           │            │
│                     └─────────────┘           │            │
│                            │                  │            │
│                            ▼                  ▼            │
│                     ┌─────────────┐    ┌─────────────┐     │
│                     │  Quality    │───▶│   Deploy    │     │
│                     │   Gates     │    │    Stage    │     │
│                     └─────────────┘    └─────────────┘     │
│                                               │            │
│                                               ▼            │
│                                        ┌─────────────┐     │
│                                        │   Deploy    │     │
│                                        │ Production  │     │
│                                        └─────────────┘     │
│                                                             │
└─────────────────────────────────────────────────────────────┘
```

Key Integration Patterns

1. Event-Driven Architecture

```
# Event-driven model retraining
class ModelRetrainingOrchestrator:
    def __init__(self):
        self.drift_monitor = Evidently()
        self.model_trainer = MLflowTrainer()
        self.deployer = DockerDeployer()

    async def handle_drift_event(self, event):
        if event.drift_score > 0.3:
            # Trigger retraining
            new_model = await self.model_trainer.retrain()

            # Deploy if validation passes
            if self.validate_model(new_model):
                await self.deployer.deploy(new_model)
```

2. Circuit Breaker Pattern

```
from circuit_breaker import CircuitBreaker

class ModelService:
    def __init__(self):
        self.primary_model = load_model("v2.0")
        self.fallback_model = load_model("v1.0")
        self.circuit_breaker = CircuitBreaker(
            failure_threshold=5,
            timeout=60
        )

    @circuit_breaker.protected
    def predict(self, data):
        try:
            return self.primary_model.predict(data)
        except Exception:
            return self.fallback_model.predict(data)
```

3. Blue-Green Deployment

```yaml
# docker-compose.blue-green.yml
version: '3.8'

services:
  api-blue:
    image: mlops-api:v1.0
    environment:
      - MODEL_VERSION=v1.0
    ports:
      - "8001:8000"

  api-green:
    image: mlops-api:v2.0
    environment:
      - MODEL_VERSION=v2.0
    ports:
      - "8002:8000"

  load-balancer:
    image: nginx:alpine
    ports:
      - "80:80"
    depends_on:
      - api-blue
      - api-green
```

Monitoring Integration

```python
# Comprehensive monitoring setup
class MLOpsMonitoring:
    def __init__(self):
        self.prometheus = PrometheusClient()
        self.grafana = GrafanaClient()
        self.evidently = EvidentialAI()

    def setup_monitoring(self):
        # Model performance metrics
        self.setup_model_metrics()

        # Infrastructure metrics
        self.setup_infrastructure_metrics()

        # Business metrics
        self.setup_business_metrics()

        # Alerting rules
        self.setup_alerts()

    def setup_model_metrics(self):
        """Setup ML-specific monitoring"""
        metrics = [
            'model_prediction_accuracy',
            'model_latency_p95',
            'data_drift_score',
            'feature_importance_changes'
        ]

        for metric in metrics:
            self.prometheus.create_gauge(metric)

    def setup_alerts(self):
        """Setup alerting rules"""
        alerts = [
            {
                'name': 'ModelAccuracyDrop',
                'condition': 'model_accuracy < 0.85',
                'action': 'trigger_retraining'
            },
            {
                'name': 'HighLatency',
                'condition': 'model_latency_p95 > 500ms',
                'action': 'scale_service'
            },
            {
                'name': 'DataDrift',
                'condition': 'data_drift_score > 0.3',
                'action': 'alert_data_team'
            }
        ]
```

```
        for alert in alerts:
            self.prometheus.create_alert(alert)
```

Best Practices & Next Steps

MLOps Best Practices

1. Data Management

- **Version Everything**: Data, code, models, and configurations
- **Data Quality**: Implement comprehensive validation
- **Schema Evolution**: Handle data schema changes gracefully
- **Data Lineage**: Track data provenance and transformations

2. Model Development

- **Experiment Tracking**: Log all experiments with MLflow
- **Model Validation**: Cross-validation and holdout testing
- **Feature Stores**: Centralized feature management
- **Model Interpretability**: Ensure models are explainable

3. Deployment

- **Gradual Rollouts**: Use canary or blue-green deployments
- **Health Checks**: Implement comprehensive health monitoring
- **Rollback Strategy**: Quick rollback mechanisms
- **Load Testing**: Validate performance under load

4. Monitoring

- **Multi-layer Monitoring**: Infrastructure, application, and business metrics
- **Alerting**: Proactive alerting with clear escalation paths
- **Drift Detection**: Continuous monitoring for data and model drift
- **SLA Monitoring**: Track service level agreements

5. Security

- **Model Security**: Protect against adversarial attacks
- **Data Privacy**: Implement privacy-preserving techniques
- **Access Control**: Role-based access to systems
- **Audit Logs**: Comprehensive logging for compliance

Implementation Roadmap

Phase 1: Foundation (Weeks 1-4)

- [ ] Set up version control (Git + DVC)
- [ ] Implement basic CI/CD pipeline
- [ ] Create initial model training pipeline
- [ ] Set up MLflow for experiment tracking

Phase 2: Serving & Monitoring (Weeks 5-8)

- [ ] Deploy models with FastAPI
- [ ] Implement Prometheus monitoring
- [ ] Create Grafana dashboards
- [ ] Set up automated testing

Phase 3: Advanced Features (Weeks 9-12)

- [ ] Implement data drift detection
- [ ] Add model retraining automation
- [ ] Enhance security measures
- [ ] Optimize performance

Phase 4: Production Hardening (Weeks 13-16)

- [ ] Load testing and optimization
- [ ] Disaster recovery planning
- [ ] Documentation and training
- [ ] Compliance and governance

Common Pitfalls to Avoid

Technical Pitfalls

- **Data Leakage**: Ensure proper train/test splits
- **Model Overfitting**: Use proper validation techniques
- **Infrastructure Coupling**: Avoid tight coupling between components
- **Technical Debt**: Regular refactoring and cleanup

Organizational Pitfalls

- **Siloed Teams**: Foster collaboration between teams
- **Unclear Ownership**: Define clear roles and responsibilities
- **Inadequate Testing**: Comprehensive testing strategy
- **Poor Documentation**: Maintain up-to-date documentation

Next Steps

Immediate Actions

1. **Start Small**: Begin with one model and expand
2. **Focus on Automation**: Automate repetitive tasks
3. **Implement Monitoring**: Start monitoring from day one
4. **Build Team Skills**: Invest in team training

Advanced Topics to Explore

- **Feature Stores**: Centralized feature management
- **Model Mesh**: Multi-model serving architecture
- **MLOps Platforms**: Kubeflow, Seldon, or MLflow
- **Edge Deployment**: Mobile and IoT model deployment

Resources for Continued Learning

Books - "Designing Machine Learning Systems" by Chip Huyen - "Building Machine Learning Pipelines" by Hannes Hapke - "MLOps: Continuous Delivery for Machine Learning" by Gil Hoffer

Courses - MLOps Specialization (Coursera) - Made With ML Course - Full Stack Deep Learning

Communities - MLOps Community Slack - Reddit r/MachineLearning - MLOps Conference talks

Tools to Explore - **Kubeflow**: Kubernetes-native ML workflows - **Seldon Core**: Enterprise model deployment - **Feast**: Feature store solution - **Great Expectations**: Data validation framework

---

Workshop Summary

What We've Covered

✅ **Complete MLOps Pipeline Architecture** - End-to-end understanding of MLOps components - Integration patterns and best practices - Real-world implementation examples

✅ **8 Essential MLOps Tools** - DVC for data versioning - MLflow for experiment tracking - FastAPI for model serving - Prometheus & Grafana for monitoring - GitHub Actions for CI/CD - Docker for containerization - Evidently AI for drift detection - Pytest for automated testing

✅ **Production-Ready Practices** - Security considerations - Performance optimization - Scalability patterns - Monitoring strategies

Key Takeaways

1. **MLOps is a Journey**: Start simple and evolve
2. **Automation is Key**: Automate everything possible
3. **Monitoring is Critical**: Monitor models, data, and infrastructure
4. **Culture Matters**: Foster collaboration between teams
5. **Continuous Learning**: MLOps tools and practices evolve rapidly

Workshop Resources

All code, configurations, and documentation are available in our demo repository: - **GitHub Repository**: [MLOps Demo Project] - **Documentation**: Complete setup guides and tutorials - **Examples**: Working code for all components - **Tests**: Comprehensive test suites

---

Thank you for attending the MLOps Workshop!

Questions and Discussion
