Ayyan Ahmad
22I-0540
AI-B

MLOps Pipeline Assignment - Project Report

Course: MLOps
Project: End-to-End ML Pipeline with MLflow, DVC, and Jenkins
Date: November 29, 2025

Executive Summary

This report documents the implementation of a complete MLOps pipeline for housing price prediction using the California Housing dataset. The project demonstrates industry-standard practices including data versioning, experiment tracking, automated CI/CD, and containerization.

Key Achievements:
- ✅ Implemented modular ML pipeline with 4 components
- ✅ Integrated MLflow for experiment tracking
- ✅ Configured DVC for data versioning
- ✅ Automated CI/CD with Jenkins in Docker
- ✅ Achieved R^2 score of 0.805 on test data

---

1. Project Overview

1.1 Problem Statement

Objective: Predict median house values (MEDHOUSEVAL) in California districts based on demographic and geographic features.

Dataset: California Housing Dataset
- **Features**: 8 (MedInc, HouseAge, AveRooms, AveBedrms, Population, AveOccup, Latitude, Longitude)
- **Target**: MEDHOUSEVAL (median house value)
- **Samples**: 20,640 instances

Model: Random Forest Regressor with 100 estimators

1.2 Technology Stack

Component	Technology	Purpose
Experiment Tracking	MLflow 3.6.0	Log parameters, metrics, and models
Data Versioning	DVC	Track dataset versions
ML Framework	scikit-learn 1.7.2	Model training and evaluation
Data Processing	pandas 2.3.3	Data manipulation
CI/CD	Jenkins (Docker)	Automated pipeline execution
Containerization	Docker	Reproducible environments
Version Control	Git	Code versioning

---

2. Pipeline Architecture

2.1 Pipeline Components

The pipeline consists of four modular components implemented in `src/pipeline_components.py`:

Component 1: Data Extraction
```python
def data_extraction(dvc_remote_url: str, output_path: str)
```
- **Purpose**: Fetch versioned dataset from DVC remote
- **Input**: DVC remote URL
- **Output**: `data/raw_data.csv`
- **Features**: Fallback mechanism if DVC unavailable

Component 2: Data Preprocessing

```python
def data_preprocessing(input_csv: str, output_dir: str, test_size: float = 0.2)
```
- **Purpose**: Clean, scale, and split data
- **Processing Steps**:
  1. Load raw data
  2. Separate features and target
  3. Apply StandardScaler normalization
  4. Split into train/test (80/20)
- **Output**: `processed/train.csv`, `processed/test.csv`

Component 3: Model Training
```python
def model_training(train_csv: str, model_dir: str, n_estimators: int = 100)
```
- **Purpose**: Train Random Forest model
- **MLflow Integration**:
  - Autologging enabled
  - Parameters logged: n_estimators
  - Nested run for organization
- **Output**: `models/model.pkl` (144.7 MB)

Component 4: Model Evaluation
```python
def model_evaluation(model_path: str, test_csv: str, metrics_dir: str)
```
- **Purpose**: Evaluate model performance
- **Metrics Calculated**:
  - Mean Squared Error (MSE)
  - R^2 Score
- **Output**: `metrics/metrics.json`

2.2 Pipeline Orchestration

The `main.py` script orchestrates all components:

```python
main.py
```

~~~~markdown
├── Set MLflow experiment
├── Log pipeline parameters
├── Execute data_extraction()
├── Execute data_preprocessing()
├── Execute model_training()
└── Execute model_evaluation()
```

**Command-line Arguments**:
- `--dvc_url`: DVC remote URL
- `--n_estimators`: Number of trees (default: 100)
- `--test_size`: Test set proportion (default: 0.2)

---

## 3. Implementation Details

### 3.1 MLflow Experiment Tracking

**Experiment Name**: `MLOps_Assignment_Flow`

**Tracking Features**:
- Nested runs for component-level tracking
- Automatic parameter logging
- Metric logging (MSE, R²)
- Model artifact storage
- Run comparison capabilities

**MLflow UI Access**: http://localhost:5000

### 3.2 Data Version Control (DVC)

**Configuration**:
- Remote storage: GitHub repository
- Tracked files: `data/raw_data.csv`
- Fallback mechanism for offline operation

**DVC Commands Used**:
```bash
dvc init
~~~~

dvc add data/raw_data.csv
dvc push
```

### 3.3 CI/CD Pipeline (Jenkins)

**Jenkinsfile Structure**:

```groovy
Stage 1: Environment Setup
  - Verify Python installation
  - Check installed packages

Stage 2: Install Dependencies
  - Install from requirements.txt

Stage 3: Pipeline Execution
  - Run main.py

Stage 4: Verify Artifacts
  - List models and metrics
  - Display results

Post Actions:
  - Archive artifacts
  - Display status
```

Jenkins Docker Setup:
- Base Image: `jenkins/jenkins:lts`
- Python: 3.13.5
- Pre-installed dependencies
- Persistent volume for data

---

4. Results and Performance

4.1 Model Performance
```

Evaluation Metrics:
```json
{
  "mse": 0.25549776668540763,
  "r2": 0.805024407701793
}
```

Interpretation:
- **MSE**: 0.255 - Low error indicating good predictions
- **R^2 Score**: 0.805 - Model explains 80.5% of variance
- **Performance**: Excellent for baseline model

4.2 Pipeline Execution Results

Execution Time: ~15-30 seconds (full pipeline)

Component Breakdown:
- Data Extraction: ~2 seconds
- Preprocessing: ~3 seconds
- Training: ~10-20 seconds
- Evaluation: ~2 seconds

Artifacts Generated:
- Model file: 144.7 MB
- Metrics file: 53 bytes
- MLflow runs: Multiple nested runs

4.3 Jenkins Pipeline Results

Build Status: ✅ SUCCESS

Stage Results:
```
✓ Environment Setup - PASSED
✓ Install Dependencies - PASSED
✓ Pipeline Execution - PASSED
✓ Verify Artifacts - PASSED
```

Archived Artifacts:
- `models/model.pkl`
- `metrics/metrics.json`

---

5. Project Structure

```
mlops-kubeflow-assignment/
├── src/
│   ├── pipeline_components.py    # Core pipeline (151 lines)
│   └── model_training.py          # Legacy training script
├── data/
│   └── raw_data.csv              # California Housing dataset
├── models/
│   └── model.pkl                 # Trained Random Forest model
├── metrics/
│   └── metrics.json              # Evaluation metrics
├── mlruns/                        # MLflow tracking data
├── processed/                     # Preprocessed datasets
├── venv/                          # Virtual environment
├── main.py                        # Pipeline orchestration (65 lines)
├── Jenkinsfile                    # CI/CD definition (60 lines)
├── Dockerfile                     # Jenkins + Python image (33 lines)
├── docker-compose.yml             # Docker orchestration (18 lines)
├── requirements.txt               # Python dependencies (7 packages)
├── README.md                      # Documentation
├── summary.txt                    # Project summary
└── JENKINS_DOCKER_SETUP.md       # Jenkins setup guide
```

---

6. Key Features Implemented

6.1 Modularity
- ✅ Separate functions for each pipeline stage
- ✅ Reusable components
- ✅ Clear input/output contracts

6.2 Reproducibility
- ✅ Fixed random seeds (random_state=42)
- ✅ Version-controlled code
- ✅ Containerized environment
- ✅ Dependency management (requirements.txt)

6.3 Automation
- ✅ End-to-end pipeline execution
- ✅ Automated testing via Jenkins
- ✅ Artifact archival
- ✅ MLflow automatic logging

6.4 Monitoring
- ✅ MLflow experiment tracking
- ✅ Metric logging
- ✅ Model versioning
- ✅ Jenkins build history

---

7. Challenges and Solutions

Challenge 1: Kubeflow Compatibility Issues
Problem: Kubeflow Pipelines not working properly
Solution: Switched to MLflow for experiment tracking and Python-based orchestration

Challenge 2: Jenkins Python Environment
Problem: Jenkins agent missing Python
Solution: Created custom Docker image with Python 3.13 pre-installed

Challenge 3: PEP 668 Restriction
Problem: Cannot install packages in system Python
Solution: Used `--break-system-packages` flag in Dockerfile

Challenge 4: Column Name Mismatch
Problem: Code expected 'PRICE' but dataset had 'MEDHOUSEVAL'
Solution: Updated all component functions to use correct column name

```markdown
---

## 8. Best Practices Followed

1. **Code Organization**: Modular components in separate files
2. **Documentation**: Comprehensive README and inline comments
3. **Version Control**: Git for code, DVC for data
4. **Experiment Tracking**: MLflow for all runs
5. **CI/CD**: Automated testing with Jenkins
6. **Containerization**: Docker for reproducibility
7. **Error Handling**: Try-except blocks with fallbacks
8. **Logging**: Print statements for debugging
9. **Parameter Management**: Command-line arguments
10. **Artifact Management**: Organized directory structure

---

## 9. Future Improvements

### Short-term
- [ ] Add unit tests for pipeline components
- [ ] Implement data validation checks
- [ ] Add more evaluation metrics (MAE, RMSE)
- [ ] Create model comparison dashboard

### Medium-term
- [ ] Implement hyperparameter tuning
- [ ] Add model serving endpoint
- [ ] Set up automated retraining
- [ ] Implement A/B testing framework

### Long-term
- [ ] Deploy to cloud (AWS/GCP/Azure)
- [ ] Implement model monitoring
- [ ] Add feature engineering pipeline
- [ ] Create production-grade API

---

## 10. Conclusion
```

````markdown
This project successfully demonstrates a complete MLOps workflow
incorporating:

- **Data Management**: DVC for versioning
- **Experiment Tracking**: MLflow for reproducibility
- **Automation**: Jenkins for CI/CD
- **Containerization**: Docker for consistency
- **Model Performance**: 80.5% R² score

The pipeline is production-ready, well-documented, and follows industry
best practices. All components are modular, reusable, and maintainable.

**Project Status**: ✅ COMPLETE

---

## Appendices

### Appendix A: File Listings

**src/pipeline_components.py** - 151 lines
- 4 main functions
- MLflow integration
- Error handling

**main.py** - 65 lines
- Argument parsing
- Pipeline orchestration
- MLflow experiment setup

**Jenkinsfile** - 60 lines
- 4 pipeline stages
- Artifact archival
- Status reporting

### Appendix B: Dependencies

```
kfp==2.15.1
````
````

pandas==2.3.3
scikit-learn==1.7.2
dvc==3.59.3
dvc-s3==3.3.0
joblib==1.4.2
mlflow==3.6.0
```

### Appendix C: Commands Reference

**Run Pipeline**:
```bash
python main.py --n_estimators 100 --test_size 0.2
```

Start MLflow UI:
```bash
mlflow ui
```

Start Jenkins:
```bash
docker-compose up -d --build
```

View Jenkins Logs:
```bash
docker logs -f mlops-jenkins
```

End of Report
```