A1: The "Manual" MLOps Challenge

DA5402: A1: The "Manual" MLOps Challenge

Duration: 10 Days

Objective: To build, deploy, and maintain a production-ready AI system using "primitive" manual methods. By the end, you should understand the pain points of manual ML management and why automation is a necessity, not a luxury.

---

1. Problem Statement

You are the first ML Engineer at a startup that lacks any infrastructure. Your task is to build a **Predictive Maintenance System**. You must manage the entire lifecycle—data engineering, model versioning, deployment, and "production" monitoring—using only standard programming tools (Python, Git, CSV files, and basic Web Frameworks like Flask/FastAPI).

Constraint: You are strictly forbidden from using MLOps-specific tools (e.g., DVC, MLflow, Airflow, or Kubernetes). You must create your own "manual" versions of these capabilities.

---

2. Suggested Datasets

Choose one of the following for your project:

- **Predictive Maintenance (Binary Classification):** UCI Machine Learning Repository - AI4I 2020 Predictive Maintenance Dataset.
    - *Challenge:* The data represents sensor readings. You must handle "drift" by splitting the data chronologically. The data represents 10,000 data points. You must simulate "Time-Series" drift by training on the first 7,000 points and "monitoring" the last 3,000 as if they were arriving in production.

---

3. Project Requirements & Deliverables

Phase A: Data & Configuration Management (Days 1–3)

- **Manual Data Versioning:** Create a directory structure that acts as a manual "Data Store." Store your raw data, and as you clean it, save a new version (e.g., `v1_raw.csv`, `v2_cleaned.csv`). Keep a text-based `manifest.txt` file that logs exactly which script produced which version.
- **Hard-Coded Configs:** Move all hyperparameters and file paths out of your code and into a `config.json` or `config.yaml` file. Your code must read from this file.

Phase B: The "Model Registry" (Days 4–5)

- **Training & Artifacts:** Train your model. After training, save the model object (Pickle/Joblib) and a `metadata.json` file containing the training date, the specific dataset version used, the Git commit hash of the code, and the final accuracy/loss.
- **Reproducibility Check:** A peer should be able to run your training script and get the exact same results based solely on your `config.json` and versioned CSVs.

Phase C: Manual Deployment (Days 6–8)

- **API Wrapper:** Wrap your model in a Flask or FastAPI endpoint.
- **Deployment Log:** Create a local `deployment_log.csv`. Every time you "deploy" (run the server), log which model version is currently live.
- **Testing:** Write three "smoke tests" in a separate Python script that send requests to your local API and verify the output format.

Phase D: Post-Deployment & Maintenance (Days 9–10)

- **Simulation of Drift:** Create a "Day 2" dataset (a subset of your original data with modified values or a later timeframe). Run this through your API.
- **Manual Monitoring:** Create a script that periodically reads your API logs and calculates the "Production Error Rate" relative to your training error.
- **Retraining Trigger:** If the error exceeds a threshold, manually re-run your training script, update your `config.json`, and restart your API with the new model.

---

4. Sample Directory Structure

To pass the "Reproducibility Test," your project must follow this rigorous structure. This ensures that a peer can find the exact data version that corresponds to a specific model.

```
/manual_mlops_project
│
├── /data
│   ├── /raw                # Immutable raw CSV from UCI
│   ├── /processed          # Versioned cleaned data (e.g., v1_train.csv, v2_train.csv)
│   └── /production         # Simulated "new" data for monitoring
│
├── /models                 # Saved .pkl or .joblib files
│   └── model_metadata.log # Manual text log of model versions & metrics
│
├── /src
│   ├── data_prep.py        # Script for cleaning and feature engineering
│   ├── train.py            # Model training script
│   ├── inference.py        # API code (FastAPI/Flask)
│   └── monitor.py          # Script to calculate drift/accuracy on prod data
│
├── config.yaml             # The "Single Source of Truth" for parameters
├── requirements.txt        # Environment dependencies
└── deployment_log.csv      # Manual log of which model is currently live
```

5. Template/Sample Configuration File (config.yaml)

Your Python scripts should **never** contain hard-coded paths or hyperparameters. Use this template to control your pipeline:

```yaml
# Project Metadata
project_name: "Predictive_Maintenance_v1"
author: "Graduate_Student_Name"

# Data Paths
data:
  raw_path: "data/raw/ai4i2020.csv"
  processed_dir: "data/processed/"
  current_version: "v1"

# Hyperparameters
model_params:
  algorithm: "RandomForest"
  n_estimators: 100
  max_depth: 10
  random_state: 42

# Deployment Settings
deployment:
  model_path: "models/model_v1.pkl"
  port: 5000
  threshold: 0.75  # Classification threshold for failure
```

6. Final Submission

Submit a ZIP file or Git repository containing:

1. **The Codebase:** Organized into /data, /models, /scripts, and /api.
2. **The Documentation:** A 2-page report answering:
    - What was the most difficult part of managing data versions manually?
    - How did you ensure that the model in production was the same one you evaluated during training?
    - How did you ensure that train.py always used the correct version of the data specified in config.yaml?
    - Based on this experience, what are the top three features you would want in an automated MLOps tool?
    - Describe the "breakdown" you experienced when trying to track which model version was currently serving requests in your API.
    - How would an automated **Model Registry** have made Phase B easier?

Grading Rubric: Manual MLOps Assignment 1

Category	Weight	Criteria
Manual Versioning & Data Engineering	**30%**	**Data Lineage: Is there a clear, manual trail from raw data to processed versions? Does the `manifest.txt` or log accurately link data versions to specific processing scripts?** **Feature Engineering: Proper handling of categorical variables and scaling for the predictive maintenance dataset.**
Reproducibility & Configuration	**25%**	**Config Isolation: Are *all* paths, versions, and hyperparameters stored in `config.yaml`?** **Zero-Hardcoding: Does the training script fail if the config file is removed? Can the instructor change the `n_estimators` in the config and see the model update without touching the code?**
Deployment & Inference	**20%**	**API Functionality: Does the FastAPI/Flask endpoint return predictions successfully?** **Deployment Logging: Is the `deployment_log.csv` maintained? Does it show a history of which model version was active at what time?**

Monitoring & Drift Simulation	**15%**	**Manual Analysis: Did the student successfully compare production sensor data against the training baseline?** **Logic: Is the manual retraining trigger logical (e.g., a script that flags if precision/recall drops below the threshold set in `config.yaml`)?**
AI Ethics & Documentation	**10%**	**Code of Conduct Adherence: Is there an "AI Disclosure Appendix"?** **Critical Thinking: Does the post-mortem report demonstrate a deep understanding of the friction points in the manual process? (e.g., identifying why manual versioning is prone to human error).**

Scoring Guide

- **Exemplary (90-100%): The project is a "clockwork" manual system. Every artifact is meticulously logged. The student provides insightful reflections on the limitations of manual MLOps.**
- **Proficient (75-89%): All components work. Configuration is largely isolated, though 1-2 hardcoded paths might remain. AI disclosure is present.**
- **Developing (60-74%): The API works, but the versioning is messy or inconsistent. The connection between data versions and model artifacts is difficult to trace.**
- **Insufficient (<60%): The project uses an automated MLOps tool (DVC/MLflow) despite the prohibition, or there is evidence of AI-generated logic without attribution.**
