Machine Learning / Applied AI Engineer Case study

Production-Minded ML Pipeline & Inference Service (ML Engineer / Applied AI)

Objective

You've joined a team that needs a **minimal but production-minded ML pipeline** up and running quickly. Your task is to:

1. Build a **small, reproducible training pipeline** for a simple classification problem.
2. Expose the trained model via a **minimal inference API**.
3. Demonstrate **applied ML thinking** (model choice, metrics, trade-offs) and **MLOps fundamentals** (reproducibility, artifacts, validation, basic monitoring plan).

Use a dataset that's available offline.

Timebox: ~60 minutes total (Proposed timeline for case study creation)

- Focus on **Core tasks** first.
- **Stretch tasks** are optional if time allows.

Tech Constraints

Constraint	Details
Language	Python 3.8+
Allowed Libraries	scikit-learn, FastAPI or Flask, pydantic, joblib/pickle, numpy, pandas (optional), MLflow (optional)
External Services	None required; everything runs locally on CPU
Notebooks	Not allowed; deliver runnable scripts and a short README

Problem

- **Dataset**: `Iris` dataset from `sklearn.datasets.load_iris`
- **Task**: Train a **multiclass classifier** to predict iris species from four numeric features

---

Core Tasks (Required)

1. Training Pipeline (`train.py`)

Implement a **reproducible training script** that:

Data & Split

- Loads dataset: `sklearn.datasets.load_iris`
- Performs a **stratified train/test split** with:
  - Configurable random seed
  - Configurable test size

Preprocessing & Model

- Preprocessing:
  - Standardize numeric features with `StandardScaler`
- Model:
  - Use either `LogisticRegression` or `RandomForestClassifier`
- Build a **single scikit-learn** `Pipeline` that includes both:
  - Preprocessing
  - Model

Metrics & Evaluation

- Compute at least:
  - **Accuracy**
  - **Macro F1 score** on the test set
- Additionally, save a simple **confusion matrix** (per-class performance) as JSON (e.g., `artifacts/confusion_matrix.json`).

Artifacts

Write the following to an `artifacts/` directory:

- `artifacts/model.joblib`
  - The fitted sklearn `Pipeline`
- `artifacts/metrics.json`
  - At minimum: accuracy, macro F1
- `artifacts/params.json`
  - All training parameters (CLI args used)
- `artifacts/confusion_matrix.json`
  - Confusion matrix (e.g., scikit-learn output converted to JSON-serializable format)

Configuration & Reproducibility

The script should accept CLI arguments:

- `--seed` (int, for all relevant random_state usage)
- `--test-size` (float, e.g., 0.2)
- `--model` `{logreg, rf}`
- At least one model hyperparameter (for ML thinking):
  - For `logreg`: e.g., `--C`

- For `rf`: e.g., `--n-estimators`
- Use the seed to ensure **deterministic runs** given the same configuration.

Behavior:

- Print key metrics (accuracy and macro F1) to stdout.
- Clearly log/print which model and hyperparameters were used.

Applied ML Note (in README)

In your README, add a **short "ML Design Notes" section** (3–6 sentences) that explains:

- Why you chose the default model and hyperparameters.
- Why you chose the metrics you report.
- Any trade-offs you'd consider in a real production setting (e.g., interpretability vs. performance, training speed vs. complexity).

---

2. Inference API (`serve.py`)

Implement an API that uses the **saved pipeline** for inference.

Framework

- Use **FastAPI** (preferred) or **Flask**.

Startup

- On startup, **load the saved pipeline** from `artifacts/model.joblib`.
- Derive a simple `model_version` from:
  - A timestamp in the model filename, or
  - A short hash (e.g., SHA256 of the model file truncated).

Endpoints

1. **Health Check**
   - `GET /health`
   - Returns:

```
{ "status": "ok" }
```

2. **Prediction**
   - `POST /predict`
   - Request JSON:

```
{
  "sepal_length": float,
  "sepal_width": float,
  "petal_length": float,
  "petal_width": float
}
```

   - Use **pydantic** (or similar) for:
     - Type validation
     - Presence of all required fields
   - Response JSON:

```
{
  "predicted_label": "string species name",
  "class_probabilities": { "class_name": float, ... },
  "model_version": "string",
  "latency_ms": float
}
```

- Measure and include **latency_ms** around the prediction call.

Error Handling

- Invalid inputs should return a clear **4xx** error with a useful message.
- Empty or malformed JSON should not crash the service.

---

3. Project Hygiene & Run Instructions

Provide:

- `requirements.txt` with pinned versions or reasonable minimums.
- `README.md` that includes:
  - How to set up and run training (example command).
  - Where artifacts are written and what each file represents.
  - How to start the API.
  - Example `curl` request.

Example commands:

```
python train.py --seed 42 --test-size 0.2 --model logreg --C 1.0

uvicorn serve:app --host 0.0.0.0 --port 8000

curl -X POST http://localhost:8000/predict \
  -H "Content-Type: application/json" \
  -d
'{"sepal_length":5.1,"sepal_width":3.5,"petal_length":1.4,"petal_width"
:0.2}'
```

Nice-to-have (not required):

- Simple commands (e.g., `Makefile`) to simplify common runs.

Stretch Tasks (Optional, if time permits)

Pick one or two; depth is better than breadth.

1. Experiment Tracking (MLflow or Similar)

- Log parameters and metrics to a local MLflow run **in addition to** JSON artifacts.
- Show how you'd use experiment tracking in a more complex setting.

2. Dockerization

- Add a simple `Dockerfile`:
  - Build and run the API.
- Document build/run commands in README.

3. Basic Tests (pytest)

Add a small `tests/` folder with, for example:

- A test for:
  - Input schema validation (e.g., invalid field types or missing fields).
- A test that:
  - Loads the saved pipeline and runs prediction on a sample input without errors.

4. Simple CI (GitHub Actions)

- GitHub Actions workflow (`.github/workflows/ci.yml`) that:
  - Installs dependencies.
  - Runs linting (optional) and tests.

5. Observability & Monitoring Plan

- **Code-level**:
  - Log each request with a request ID and latency.
  - Optionally aggregate basic rate/latency stats and print them periodically.
- **Monitoring Plan (in README)**:
  - A short note (5–10 sentences) on:
    - How you'd monitor **data drift**.
    - How you'd monitor **model performance** post-deployment.
    - What signals or thresholds might trigger retraining or rollback.

Acceptance Criteria

Training

- Running the training command produces:
  - `artifacts/model.joblib`
  - `artifacts/metrics.json` (accuracy, macro F1)
  - `artifacts/params.json`
  - `artifacts/confusion_matrix.json`
- Metrics are **reasonable for Iris** (e.g., >0.9 accuracy).
- Re-running with the same seed and parameters yields identical metrics.

API

- Loads the saved model and responds correctly on:
  - `GET /health`
  - `POST /predict` with:
    - Valid `predicted_label`
    - `class_probabilities` for each class
    - `model_version`

    - `latency_ms`
- Input validation errors return informative **4xx** responses.

Applied ML & MLOps

- Short, clear **ML design notes** in README (model, metrics, trade-offs).
- Evidence of **reproducibility** and **artifact management**.
- Clean, minimal **logging** (at least model choice, metrics, and basic server logs).

Code Quality

- Clear structure and readable code.
- No notebooks; all runnable via scripts and documented commands.
- Project is easy to set up and run following the README.

Stretch (Bonus)

- Any of the Stretch tasks implemented well.

Suggested Directory Structure

```
README.md
requirements.txt
train.py
serve.py
artifacts/                 # created by the code
tests/                     # (optional)
Dockerfile                 # (optional)
.github/workflows/ci.yml   # (optional)
Makefile                   # (optional)
```

Deliverables

Please submit either:

- A **Git repository URL** (GitHub, GitLab, Bitbucket, etc.), **or**
- A **single ZIP archive** containing the project folder.