

Exercise: Model Promotion Pipeline (Staging -> Production)

Our focus:

- model lifecycle and environments
- automated quality gates
- model registry promotion
- deployment strategies (staging vs production)
- rollback readiness

Concept map (core ideas)

Model lifecycle

- *Train -> Evaluate -> Register -> Deploy to staging -> Promote -> Deploy to production*

Quality gate

A rule that must pass before promotion. Examples:

- accuracy ≥ 0.90
- latency ≤ 50 ms
- schema compatible with current API contract
- smoke tests pass

Model registry as the source of truth

Instead of "whatever is in the repo", production should serve:

- a specific model *name + version* from MLflow
- with a recorded metric and lineage (data version, code commit)

Exercise scenario

Your team trains a new model. Every time a candidate model is produced, the pipeline should:

1. Register it in MLflow as a new version.
2. Deploy that version to *staging* automatically.
3. Run automated tests (quality gates) against staging.
4. If gates pass, promote the model in MLflow to *Production* stage.
5. Deploy production API to serve the latest *Production*-staged model.

If gates fail:

- the model stays in *Staging*
- production is NOT affected

Code main components

- training/evaluation (ml/)
 - serving (backend/)
 - UI (frontend/)
 - workflows (.github/workflows/)
-

Setup

A) Secrets in GitHub

Create GitHub Actions secrets:

- `DAGSHUB_MLFLOW_TRACKING_URI` (example: `https://dagshub.com/<org>/<repo>.mlflow`)
- `DAGSHUB_TOKEN` (personal access token)
- `DOCKERHUB_USERNAME`
- `DOCKERHUB_TOKEN` (or password)

We can run deployments locally with docker-compose and still learn the pipeline logic.

B) MLflow model name

Pick one model registry name:

- `churn-model`

Provided minimal code

These snippets are intentionally small to aid you in writing, not write the full code for you.

1) ML training + registration (ml/train_and_register.py)

This script:

- trains a tiny model
- logs metrics and artifacts to MLflow
- registers a new model version in the MLflow Model Registry

```
import os
import json
import time
import mlflow
import mlflow.sklearn
from sklearn.datasets import make_classification
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

MODEL_NAME = os.getenv("MODEL_NAME", "churn-model")
```

```
def main():
    tracking_uri = os.environ["MLFLOW_TRACKING_URI"]
    token = os.environ["MLFLOW_TRACKING_TOKEN"]
    mlflow.set_tracking_uri(tracking_uri)
    os.environ["MLFLOW_TRACKING_USERNAME"] = token
    os.environ["MLFLOW_TRACKING_PASSWORD"] = token

    X, y = make_classification(n_samples=2000, n_features=10,
                              random_state=42)
    X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                         test_size=0.2, random_state=42)

    model = LogisticRegression(max_iter=200)
    model.fit(X_train, y_train)
    preds = model.predict(X_test)
    acc = accuracy_score(y_test, preds)

    run_name = f"candidate-{int(time.time())}"
    with mlflow.start_run(run_name=run_name) as run:
        mlflow.log_metric("accuracy", float(acc))
        mlflow.log_param("model_type", "logreg")
        mlflow.log_param("data_version", os.getenv("DATA_VERSION",
                                                    "dvc:unknown"))
        mlflow.sklearn.log_model(model, artifact_path="model")

        # Register model
        model_uri = f"runs:{run.info.run_id}/model"
        mv = mlflow.register_model(model_uri=model_uri, name=MODEL_NAME)

        out = {"run_id": run.info.run_id, "accuracy": float(acc),
               "model_version": mv.version}
        print(json.dumps(out))

if __name__ == "__main__":
    main()
```

2) Gate decision (ml/evaluate.py)

This script:

- fetches metrics for a candidate run (or uses provided)
- applies a simple threshold gate

```
import os
import json
import sys

ACCURACY_THRESHOLD = float(os.getenv("ACCURACY_THRESHOLD", "0.90"))

def main():
    # Accept JSON from stdin or argv for simplicity
```

```

if not sys.stdin.isatty():
    payload = sys.stdin.read().strip()
else:
    payload = sys.argv[1]

data = json.loads(payload)
acc = float(data["accuracy"])

passed = acc >= ACCURACY_THRESHOLD
result = {
    "passed": passed,
    "accuracy": acc,
    "threshold": ACCURACY_THRESHOLD,
    "model_version": data["model_version"],
    "run_id": data["run_id"],
}
print(json.dumps(result))

if not passed:
    sys.exit(2)

if __name__ == "__main__":
    main()

```

3) Backend API serving latest model from MLflow stage (backend/model_loader.py)

This loads:

- latest model in a given registry stage: "Staging" or "Production"

```

import os
import mlflow
import mlflow.pyfunc

MODEL_NAME = os.getenv("MODEL_NAME", "churn-model")
MODEL_STAGE = os.getenv("MODEL_STAGE", "Staging")

def load_model():
    tracking_uri = os.environ["MLFLOW_TRACKING_URI"]
    token = os.environ["MLFLOW_TRACKING_TOKEN"]
    mlflow.set_tracking_uri(tracking_uri)
    os.environ["MLFLOW_TRACKING_USERNAME"] = token
    os.environ["MLFLOW_TRACKING_PASSWORD"] = token

    # MLflow supports stage-based URI like:
    # models:/<name>/<stage>
    uri = f"models:{MODEL_NAME}/{MODEL_STAGE}"
    return mlflow.pyfunc.load_model(uri)

```

4) Backend API (backend/app.py)

A minimal predict endpoint + health endpoint.

```
import os
from flask import Flask, request, jsonify
from model_loader import load_model

app = Flask(__name__)
model = load_model()

@app.get("/health")
def health():
    return {"status": "ok", "stage": os.getenv("MODEL_STAGE", "Staging")}

@app.post("/predict")
def predict():
    payload = request.get_json(force=True)
    # Expect "features": [[...10 floats...], [...]]
    X = payload["features"]
    preds = model.predict(X)
    return jsonify({"predictions": preds.tolist()})

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=8000)
```

5) Simple frontend (frontend/index.html)

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>Model Stage Demo</title>
  </head>
  <body>
    <h1>Model Promotion Demo</h1>
    <p>Calls /health and /predict on the backend.</p>

    <button id="health">Check Health</button>
    <pre id="healthOut"></pre>

    <button id="predict">Send Predict</button>
    <pre id="predOut"></pre>

    <script>
      const backend = "http://localhost:8000";

      document.getElementById("health").onclick = async () => {
        const r = await fetch(`${backend}/health`);
        document.getElementById("healthOut").textContent =
JSON.stringify(await r.json(), null, 2);
      };
    </script>
  </body>
</html>
```

```
document.getElementById("predict").onclick = async () => {
  // 10 features expected
  const payload = { features:
[[0.1,0.2,0.0,0.4,0.5,0.0,0.2,0.1,0.3,0.9]] };
  const r = await fetch(`${backend}/predict`, {
    method: "POST",
    headers: { "Content-Type": "application/json" },
    body: JSON.stringify(payload),
  });
  document.getElementById("predOut").textContent =
JSON.stringify(await r.json(), null, 2);
};
</script>
</body>
</html>
```

Dockerfiles (minimal)

backend/Dockerfile

```
FROM python:3.11-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY . .
EXPOSE 8000
CMD ["python", "app.py"]
```

backend/requirements.txt

```
flask==3.0.3
mlflow==2.14.3
scikit-learn==1.5.1
pandas==2.2.2
```

ml/Dockerfile

```
FROM python:3.11-slim
WORKDIR /ml
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY . .
CMD ["python", "train_and_register.py"]
```

ml/requirements.txt

```
mlflow==2.14.3
scikit-learn==1.5.1
```

Local deployment (docker-compose)

deploy/docker-compose.staging.yml

Staging serves the model at MLflow stage "Staging".

```
services:
  backend-staging:
    build: ../backend
    ports:
      - "8000:8000"
    environment:
      MODEL_NAME: churn-model
      MODEL_STAGE: Staging
      MLFLOW_TRACKING_URI: ${MLFLOW_TRACKING_URI}
      MLFLOW_TRACKING_TOKEN: ${MLFLOW_TRACKING_TOKEN}
```

deploy/docker-compose.production.yml

Production serves the model at MLflow stage "Production".

```
services:
  backend-production:
    build: ../backend
    ports:
      - "8001:8000"
    environment:
      MODEL_NAME: churn-model
      MODEL_STAGE: Production
      MLFLOW_TRACKING_URI: ${MLFLOW_TRACKING_URI}
      MLFLOW_TRACKING_TOKEN: ${MLFLOW_TRACKING_TOKEN}
```

The pipeline: two workflows

You will implement the pipeline in two workflows:

1. Train/Register -> Deploy to Staging -> Run gates
2. If gates pass -> Promote -> Deploy to Production

This separation mirrors real teams:

- A model candidate can be tested many times in staging.
- Production promotion is deliberate and auditable.

Workflow 1: Candidate -> Staging + Gates

Create `.github/workflows/train_register_deploy_staging.yml`

```
name: Candidate to Staging

on:
  workflow_dispatch:
  push:
    branches: ["main"]
    paths:
      - "ml/_."
      - ".github/workflows/train_register_deploy_staging.yml"

jobs:
  train_register:
    runs-on: ubuntu-latest
    outputs:
      result_json: ${ steps.train.outputs.result_json }
    steps:
      - uses: actions/checkout@v4

      - name: Set up Python
        uses: actions/setup-python@v5
        with:
          python-version: "3.11"

      - name: Install ML deps
        run: |
          pip install -r ml/requirements.txt

      - name: Train + register in MLflow
        id: train
        env:
          MLFLOW_TRACKING_URI: ${ secrets.DAGSHUB_MLFLOW_TRACKING_URI }
          MLFLOW_TRACKING_TOKEN: ${ secrets.DAGSHUB_TOKEN }
          MODEL_NAME: churn-model
          DATA_VERSION: "dvc:v1" # in a real repo: derive from
                                `dvc.lock` or commit hash
        run: |
          python ml/train_and_register.py | tee train_out.json
          echo "result_json=$(cat train_out.json)" >> "$GITHUB_OUTPUT"

      - name: Evaluate gate
        env:
          ACCURACY_THRESHOLD: "0.90"
        run: |
          echo '${ steps.train.outputs.result_json }' | python
```

```

ml/evaluate.py

deploy_staging:
  needs: train_register
  runs-on: ubuntu-latest
  steps:
    - uses: actions/checkout@v4

    - name: Build + push backend image (staging)
      env:
        DOCKERHUB_USERNAME: ${ secrets.DOCKERHUB_USERNAME }
        DOCKERHUB_TOKEN: ${ secrets.DOCKERHUB_TOKEN }
      run: |
        echo "$DOCKERHUB_TOKEN" | docker login -u "$DOCKERHUB_USERNAME"
--password-stdin
        docker build -t "$DOCKERHUB_USERNAME/churn-backend:staging"
backend
        docker push "$DOCKERHUB_USERNAME/churn-backend:staging"

    - name: Deploy to staging (local-compose example)
      # For class: this is illustrative.
      # In real deployment: SSH into staging host and run compose
pulling the image.
      run: |
        echo "Deploy step placeholder: run docker-compose on staging
host."

staging_smoke_test:
  needs: deploy_staging
  runs-on: ubuntu-latest
  steps:
    - name: Smoke test placeholder
      run: |
        echo "In real life, curl staging /health and /predict here."

```

What to notice:

- `train_register` produces a JSON payload with `accuracy`, `run_id`, `model_version`.
- `evaluate.py` is the quality gate. It fails the job if accuracy is too low.
- Deployment happens only if training + gate passed.
- This workflow DOES NOT promote to production. It only proves "safe candidate" in staging.

Workflow 2: Promote -> Production Deploy

Create `.github/workflows/promote_deploy_production.yml`

This one is typically manual (or requires approvals). For class: use `workflow_dispatch`.

```

name: Promote to Production

```

```

on:
  workflow_dispatch:
    inputs:
      model_version:
        description: "Model version to promote"
        required: true

jobs:
  promote:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - name: Set up Python
        uses: actions/setup-python@v5
        with:
          python-version: "3.11"

      - name: Install MLflow
        run: |
          pip install mlflow==2.14.3

      - name: Promote model version in MLflow registry
        env:
          MLFLOW_TRACKING_URI: ${ secrets.DAGSHUB_MLFLOW_TRACKING_URI }
          MLFLOW_TRACKING_TOKEN: ${ secrets.DAGSHUB_TOKEN }
          MODEL_NAME: churn-model
          MODEL_VERSION: ${ inputs.model_version }
        run: |
          python - << 'PY'
          import os, mlflow
          from mlflow.tracking import MlflowClient

          mlflow.set_tracking_uri(os.environ["MLFLOW_TRACKING_URI"])
          token = os.environ["MLFLOW_TRACKING_TOKEN"]
          os.environ["MLFLOW_TRACKING_USERNAME"] = token
          os.environ["MLFLOW_TRACKING_PASSWORD"] = token

          client = MlflowClient()
          name = os.environ["MODEL_NAME"]
          version = os.environ["MODEL_VERSION"]

          # Transition version to Production (optionally archive existing)
          client.transition_model_version_stage(
              name=name,
              version=version,
              stage="Production",
              archive_existing_versions=True
          )
          print(f"Promoted {name} v{version} to Production")
          PY

  deploy_production:
    needs: promote

```

```

runs-on: ubuntu-latest
steps:
  - uses: actions/checkout@v4

  - name: Build + push backend image (production)
    env:
      DOCKERHUB_USERNAME: ${ secrets.DOCKERHUB_USERNAME }
      DOCKERHUB_TOKEN: ${ secrets.DOCKERHUB_TOKEN }
    run: |
      echo "$DOCKERHUB_TOKEN" | docker login -u "$DOCKERHUB_USERNAME"
      --password-stdin
      docker build -t "$DOCKERHUB_USERNAME/churn-backend:production"
      backend
      docker push "$DOCKERHUB_USERNAME/churn-backend:production"

  - name: Deploy to production (placeholder)
    run: |
      echo "Deploy step placeholder: run docker-compose on production
      host."

```

What to notice:

- Promotion happens in the model registry, not by changing code.
- Production backend always loads `models:/churn-model/Production`.
- This is the key: production pulls the "truth" from registry stage.

Our tasks (what to do)

Task 1: Run Candidate -> Staging workflow

1. Trigger `Candidate to Staging` workflow (workflow_dispatch) OR push a change under `ml/`.
2. Observe logs:
 - candidate run created
 - accuracy logged
 - model version registered
 - gate evaluation passed/failed

Deliverable: write down:

- model version number
- accuracy
- did the gate pass?

Task 2: Explain what "staging" proves

Answer (short):

- What staging tests that offline evaluation does not

Task 3: Promote to production

1. Trigger **Promote to Production**
2. Provide **model_version** from Task 1
3. Observe:
 - MLflow stage transition to Production
 - deployment job

Deliverable: screenshot of the promotion log line.

Task 4: Prove production uses registry stage, not "latest code"

Locally:

- Run staging backend reading "Staging" and prod backend reading "Production"
- Verify **/health** returns correct stage

Example local run:

```
export MLFLOW_TRACKING_URI="..."
export MLFLOW_TRACKING_TOKEN="..."

docker compose -f deploy/docker-compose.staging.yml up --build
# in another terminal:
docker compose -f deploy/docker-compose.production.yml up --build
```

Then:

- **curl http://localhost:8000/health** -> stage should be Staging
- **curl http://localhost:8001/health** -> stage should be Production

Discussion questions

Answer these questions in your submission:

1. Why is it dangerous to deploy "whatever just merged to main" as the model?
2. What does the registry stage give you that a Git tag does not?
3. If staging passes but production fails, what could be the causes?
4. Where should DVC fit in a serious pipeline:
 - training data snapshot
 - evaluation dataset snapshot
 - drift reference dataset
5. What should be added to the gate beyond accuracy?

- latency
 - schema checks
 - fairness constraints
 - adversarial or robustness tests
-

Extensions (optional)

ngrok

use ngrok to add a public-facing endpoint that you can use to automatically run deployments locally