# CDW Documentation

## Azure ML Pipeline Test

# Azure ML Pipeline Test

## Purpose

Test deployment of Azure ML Pipeline and look at outputs and download trained model.

## Key Things Learned

1. When you create train.py and prep.py or any other environment files, they should be stored under ./src in your notebook directory.
2. You need to understand a bit of what you are trying to have it do as it can't think or make suppositions in a normal way. Garbage in, garbage out.

## Final Code

train.py

```
import pandas as pd
import argparse
import os
from sklearn.linear_model import LogisticRegression
import joblib

def main():
    parser = argparse.ArgumentParser()
    parser.add_argument("--training_data", type=str)
    parser.add_argument("--model_output", type=str)
    args = parser.parse_args()

    df = pd.read_csv(os.path.join(args.training_data, "prepped.csv"))
    X = df[["feature1", "feature2", "feature_sum"]]
    y = df["label"]

    model = LogisticRegression()
    model.fit(X, y)

    os.makedirs(args.model_output, exist_ok=True)
    joblib.dump(model, os.path.join(args.model_output, "model.joblib"))

if __name__ == "__main__":
    main()
print("Model output path:", args.model_output)
print("Directory contents after writing:")
print(os.listdir(args.model_output))
print("Writing model to:", args.model_output)
```

```
print("Files in output dir:", os.listdir(args.model_output))
```

Download

Download

```
NOTE:  The print statements on the end were for troubleshooting and
shouldn't be there for production runs.
```

prep.py

```
import pandas as pd
import argparse
import os

def main():
    parser = argparse.ArgumentParser()
    parser.add_argument("--input_data", type=str)
    parser.add_argument("--output_data", type=str)
    args = parser.parse_args()

    df = pd.read_csv(args.input_data)
    df["feature_sum"] = df["feature1"] + df["feature2"]
    os.makedirs(args.output_data, exist_ok=True)
    df.to_csv(os.path.join(args.output_data, "prepped.csv"), index=False)

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

deployment_script.py

```
# Step 1: Install SDK
!pip install --quiet --upgrade azure-ai-ml

# Step 2: Imports and MLClient setup
from azure.ai.ml import MLClient, Input, Output, dsl
from azure.identity import DefaultAzureCredential
from azure.ai.ml.entities import Environment, CommandComponent, Data
from azure.ai.ml.constants import AssetTypes
import pandas as pd
import os
from uuid import uuid4

# Step 3: Connect to workspace
ml_client = MLClient(
    DefaultAzureCredential(),
    subscription_id="baa29726-b3e6-4910-bb9b-b585c655322c",
    resource_group_name="don-test-rg-SCUS",
    workspace_name="don-ml-workspace-fixed"
)
```

```
# Step 4: Create sample data and register it
df = pd.DataFrame({
    "feature1": [1, 2, 3, 4, 5],
    "feature2": [10, 20, 30, 40, 50],
    "label": [0, 1, 0, 1, 0]
})
df.to_csv("data.csv", index=False)

data_asset = Data(
    path="data.csv",
    type=AssetTypes.URI_FILE,
    description="Sample training data",
    name="sample-csv-data"
)
ml_client.data.create_or_update(data_asset)

# Step 5: Create Python scripts
os.makedirs("src", exist_ok=True)

# Leave train.py and prep.py creation to previous steps or user updates

# Step 6: Define Environment
env = Environment(
    name="basic-env",
    image="mcr.microsoft.com/azureml/openmpi4.1.0-ubuntu20.04:latest",
    conda_file={
        "name": "basic",
        "dependencies": [
            "python=3.8",
            "pandas",
            "scikit-learn",
            {
                "pip": [
                    "joblib"
                ]
            }
        ]
    }
)
ml_client.environments.create_or_update(env)

# Step 7: Create components from source
prep_component = CommandComponent(
    name="prep_data",
    description="Prep data component",
    inputs={"input_data": Input(type=AssetTypes.URI_FILE)},
    outputs={"output_data": Output(type=AssetTypes.URI_FOLDER)},
    code="./src",
    command="python prep.py --input_data ${{inputs.input_data}} --
output_data ${{outputs.output_data}}",
    environment=env,
```

```
    compute="cpu-cluster"
)
ml_client.components.create_or_update(prep_component)

# Force new train component to avoid cache issues
train_component = CommandComponent(
    name=f"train_model_{uuid4().hex[:8]}",  # unique name to bust cache
    description="Train model component",
    inputs={"training_data": Input(type=AssetTypes.URI_FOLDER)},
    outputs={"model_output": Output(type=AssetTypes.URI_FOLDER)},
    code="./src",
    command="python train.py --training_data ${{inputs.training_data}} --
model_output ${{outputs.model_output}}",
    environment=env,
    compute="cpu-cluster"
)
ml_client.components.create_or_update(train_component)

# Step 8: Define pipeline function
@dsl.pipeline(default_compute="cpu-cluster")
def ml_pipeline(input_data):
    prep_step = prep_component(input_data=input_data)
    train_step =
train_component(training_data=prep_step.outputs.output_data)
    return {"model_output": train_step.outputs.model_output}

# Step 9: Submit pipeline with explicit output registration
pipeline_job = ml_pipeline(
    input_data=Input(type=AssetTypes.URI_FILE, path="azureml:sample-csv-
data:4")
)

# Force Azure ML to track output
pipeline_job.outputs["model_output"] = Output(
    type=AssetTypes.URI_FOLDER,
    mode="rw_mount"
)

pipeline_job = ml_client.jobs.create_or_update(pipeline_job)

# Step 10: Stream logs
ml_client.jobs.stream(pipeline_job.name)
```

Download

Download

```
NOTE:  This is ran from the Notebook, not from a python script.  At least
not without changes.
```

# Explanation of Final Code

## □ Step 1: Install SDK

This ensures the latest version of the `azure-ai-ml` SDK is installed. It's essential for interacting with the Azure ML workspace, defining and submitting jobs, registering assets like datasets, environments, and components, and managing outputs. This step guarantees compatibility with the latest features and syntax of the Azure ML v2 SDK.

---

## □ Step 2: Imports and MLClient Setup

This step loads all necessary modules from the Azure ML SDK (`MLClient`, `Input`, `Output`, `dsl`, `CommandComponent`, etc.) as well as supporting packages like `pandas`, `uuid`, and `os`. It also establishes a connection to your Azure ML workspace by authenticating using `DefaultAzureCredential` and creating an `MLClient` instance with your subscription ID, resource group, and workspace name. This authenticated client (`ml_client`) will be used throughout the script to register assets and submit pipeline jobs.

---

## □ Step 3: Connect to Workspace

Here, the `MLClient` is initialized using the identity of the executing user or notebook (via `DefaultAzureCredential`). This is a required step to securely interact with the Azure ML control plane and asset registry. Without it, you cannot register data, submit jobs, or fetch pipeline results.

---

## □ Step 4: Create and Register Sample Data

This step uses `pandas` to create a small, structured CSV dataset with three columns: `feature1`, `feature2`, and `label`. The dataset is saved locally as `data.csv` and registered in Azure ML as a named `Data` asset of type `URI_FILE`. Registering it makes it accessible in a reproducible way across pipeline runs and compute targets. The registration is required because pipeline inputs in Azure ML must be trackable, versioned assets.

---

## □ Step 5: Create Python Scripts

This step creates a folder called `src` and writes two Python scripts into it: `prep.py` and `train.py`.

- `prep.py` reads the raw CSV, adds a derived column `feature_sum`, and writes a new prepped dataset to a specified output folder. This simulates feature engineering or transformation logic.
- `train.py` loads the preprocessed data, fits a `LogisticRegression` model from `scikit-`

`learn`, and writes the trained model to the output directory as `model.joblib`.

These scripts are necessary because Azure ML pipelines use self-contained, executable components. All logic must reside in standalone scripts for them to be used inside components.

---

## Step 6: Define Environment

This step defines a custom environment (`basic-env`) with a base Docker image and a set of conda dependencies. The environment includes `python`, `pandas`, `scikit-learn`, and `joblib`. This ensures the scripts run in a reproducible environment with the exact dependencies they need.

The environment is registered in Azure ML and then reused in both the prep and training components. This decouples dependency management from the code logic and avoids issues from differing environments between local and remote runs.

---

## Step 7: Create Components

Two reusable components are defined using `CommandComponent`:

- `prep_component` wraps `prep.py` and defines its input (`input_data`) and output (`output_data`) as URI-based file/folder assets.
- `train_component` wraps `train.py` and defines its input (`training_data`) and output (`model_output`) the same way.

Both components reference the `basic-env` and specify the same compute target (`cpu-cluster`). By wrapping your logic in these components, you modularize and isolate each stage of the ML workflow for reuse, testing, and composition into pipelines.

Importantly, the `train_component` name is suffixed with a UUID to force Azure to register a **new version**, avoiding potential issues from using a stale or cached version.

---

## Step 8: Define Pipeline Function

This defines the actual **DSL pipeline** using the `@dsl.pipeline` decorator. It chains `prep_component` and `train_component` together such that:

- The raw `input_data` is passed to `prep_component`
- The output of `prep_component` is passed as input to `train_component`
- The pipeline **returns `model_output`** to make it accessible post-run

This encapsulates the ML process in a declarative, reusable pipeline structure. The `default_compute` is set to `cpu-cluster`, which is used unless overridden.

## Step 9: Submit Pipeline Job

This constructs a pipeline job from the `ml_pipeline(…)` function by passing it the registered dataset (`azureml:sample-csv-data:4`). This is the correct way to reference versioned data assets in pipelines.

To ensure the `model_output` is tracked and accessible, it is explicitly declared as a pipeline output using:

```
pythonCopyEditpipeline_job.outputs["model_output"] = Output(...)
```

This tells Azure ML to persist and expose this output after the run, regardless of whether it's returned by the pipeline function or not. Without this, even correctly written files would not show up in the portal or SDK.

The job is then submitted using `ml_client.jobs.create_or_update(…)`.

---

## Step 10: Stream Logs

This line attaches to the running pipeline job and streams logs back to the notebook. It helps monitor progress in real-time and identify any failures as they happen.

---

## Bonus: Post-Run Output Download (Outside Numbered Steps)

After the job completes, `ml_client.jobs.download(…)` is used to fetch the `model_output` folder locally. This allows you to verify that the pipeline succeeded and inspect the actual contents of the trained model artifact (`model.joblib`).

---

## Why This Pipeline Matters

This example demonstrates how to build a **modular, reproducible, and trackable ML pipeline** in Azure ML using best practices:

- Everything (code, data, environment, output) is registered and versioned
- Each step is independent, testable, and reusable
- Logs and artifacts are persisted and inspectable via both SDK and UI
- The pipeline can now be scheduled, automated, and extended

# Uses for the output model

## 1. Deploy the Model for Real-Time Inference

### Purpose:

Allow other applications (e.g., web apps, mobile apps, services) to query the model in real time via an API.

### Implementation:

- Deploy the model using **Azure ML Online Endpoints**
- Wrap it in a scoring script (`score.py`) with a defined input/output schema
- Use Azure's **managed REST API** for secure, scalable access

### Example Use Cases:

- Predict customer churn during a support call
- Make fraud detection decisions as a transaction is processed
- Recommend next-best-actions in a CRM interface

---

## 2. Use the Model for Batch Scoring

#### Purpose:

Process large datasets periodically to generate predictions at scale.

#### Implementation:

- Use **Azure ML batch endpoints**, or submit a batch scoring pipeline job
- Read input from blob storage or a database
- Write predictions back to storage for analysis or ingestion into other systems

#### Example Use Cases:

- Score all users nightly to update risk profiles
- Predict part failures across all equipment in a factory
- Run loan approval predictions across pending applications

---

## 3. Evaluate and Explain the Model

**Purpose:**

Ensure the model is fair, explainable, and performant — especially critical in regulated environments.

**Tools:**

- **Responsible AI Dashboard** for fairness, explanation, counterfactuals
- **SHAP or LIME** for feature importance
- **Model metrics dashboards** for precision, recall, ROC, etc.

**Example Use Cases:**

- Validate that your loan approval model isn't biased against a demographic group
- Provide per-prediction feature attributions for compliance
- Tune decision thresholds based on business objectives

---

## 4. Embed the Model in a Business Workflow

**Purpose:**

Integrate predictions into real-time or batch operational systems to drive action.

**Integration Options:**

- Azure Functions or Logic Apps (real-time triggers)
- Azure Data Factory or Synapse pipelines (batch workflows)
- Event Grid / Event Hub for prediction-driven messaging

**Example Use Cases:**

- Auto-assign support tickets based on urgency prediction
- Escalate flagged transactions to fraud review team
- Enqueue predicted high-risk patients into care follow-up workflow

---

## 5. Monitor and Manage the Model in Production

**Purpose:**

Ensure the model performs well over time as real-world data changes.

**Actions:**

- Monitor prediction drift and data quality with **Azure ML Data Monitor**
- Set up retraining pipelines if performance drops
- Use **MLflow** or Azure model registry to version models and manage lifecycles

**Example Use Cases:**

- Detect concept drift in customer behavior post-promotion
- Auto-retrain recommendation model every 2 weeks
- Compare performance of two deployed model versions (A/B testing)

---

## 6. Retrain or Fine-Tune the Model

**Purpose:**

Keep the model up-to-date with fresh data, domain changes, or new features.

**Strategies:**

- Use a scheduled pipeline to retrain with new labeled data
- Add new features or tune hyperparameters
- Replace the model with an upgraded architecture (e.g., switching from logistic regression to XGBoost)

---

## Real-World Examples by Industry

| **Industry** | **Use of `model.joblib`** |
|---|---|
| Finance | Credit risk scoring, fraud detection |
| Retail | Product recommendation, churn prediction |
| Healthcare | Diagnosis support, patient readmission risk |
| Manufacturing | Predictive maintenance, quality defect scoring |
| Logistics | Delivery delay prediction, route optimization |
| Cybersecurity | Threat classification, anomaly detection |

# Reusability

# ✅ Reusable As-Is If:

You are solving **the same kind of problem** (e.g., binary classification using logistic regression) and the following stay consistent:

- **Input data structure**: New datasets have the same column names:
  - `feature1`, `feature2`, `label`
- **Preprocessing logic**: You still just sum `feature1 + feature2` to create `feature_sum`
- **Model type**: You're still using a `LogisticRegression` model from scikit-learn
- **Output format**: You expect the model to be saved as `model.joblib`

## In this case:

✅ You only need to change the **CSV file** and re-register it as a new version of `sample-csv-data`, then update the pipeline call with the new version:

```
pythonCopyEditinput_data=Input(type=AssetTypes.URI_FILE,
path="azureml:sample-csv-data:5")
```

---

## ❌ Requires Changes If:

Your pipeline needs to be adapted for a different data structure or task. Here’s when you'd need to modify the scripts:

### 🔧 If your data columns change:

- You'll need to update:
  - `prep.py` to transform new columns appropriately
  - `train.py` to use the correct feature and label columns
  - Possibly retrain on different targets (multi-class, regression, etc.)

### 🔧 If your model type changes:

- If you switch from `LogisticRegression` to `XGBoost`, `RandomForest`, or a neural network:
  - Update `train.py` to import and instantiate the new model
  - Possibly adjust hyperparameters and training logic

### 🔧 If your pipeline steps change:

- Want to add validation?

- Want to split data into train/test?
- Want to evaluate model metrics?
    - You'll need new component scripts and return more outputs (e.g., `metrics.json`)

### **If your deployment format changes:**

- If your consumers expect ONNX or TensorFlow SavedModel instead of `joblib`, you'll need to:
    - Serialize the model differently
    - Possibly update the pipeline to convert formats

---

## To Make it Highly Reusable:

You can make the pipeline truly production-grade and reusable by:

| Feature | How to Do It |
|---|---|
| Parametrize column names | Add `--feature_cols` and `--label_col` arguments |
| Generalize preprocessing | Add preprocessing config file or flags |
| Model selector | Add `--model_type` argument (`logistic`, `xgb`, etc.) |
| Versioned output naming | Return `model_output` with model name + timestamp |
| Dynamic data input | Register new data via CLI, UI, or pipeline parameter |

---

## Summary

| Scenario | Reusable? | What to Change |
|---|---|---|
| Same data structure and model type | | Just update the input dataset version |
| Same structure, different model | | Modify `train.py` only |
| Different data columns or prediction target | | Modify `prep.py` and `train.py` |
| More complex workflow (e.g., evaluation, deployment) | | Add steps and new component scripts |

AI Knowledge