Technology Perception

ISSN(Online): 3082-4451

Volume- 1| Issue- 02| 2025

Research Article **Received:** 05-08-2025 | **Accepted:** 25-08-2025 | **Published:** 19-09-2025

A Practical Guide to Building End-to-End Azure Machine Learning Training Pipelines with Python SDK v2

Jayakanth Pujuri

Senior Member, IEEE, ORCID iD: 0009-0006-3471-0977

Abstract: This paper delivers a practical, step-by-step guide to constructing, automating, and managing machine learning training pipelines using the Azure Machine Learning (Azure ML) Python SDK v2. It systematically navigates the essential stages, commencing with Azure ML workspace connection and the programmatic setup of requisite compute infrastructure. The guide then details the creation of versioned foundational MLOps assets, including Data Assets for traceable data handling, custom Conda-based Environments for execution reproducibility, and modular Components for discrete pipeline tasks such as data preparation and model training. Emphasis is placed on integrating MLflow for comprehensive experiment tracking and model registration. The methodology culminates in the assembly and execution of an end-to-end training pipeline, exemplified by an NYC Taxi fare prediction model, illustrating the orchestration of these elements into a cohesive workflow. This tutorial aims to empower developers and MLOps practitioners with the skills to develop modular, scalable, and reproducible ML solutions in the Azure cloud environment.

Keywords: Python, SDK v2, Machine Learning

INTRODUCTION

The Increasing Complexity and scale of machine learning endeavors demand rigorous, automated, and reproducible MLOps practices. A cornerstone of effective MLOps is the systematic automation of training pipelines, which significantly accelerates the path from model conception to reliable production deployment. Azure Machine Learning (Azure ML) is a comprehensive cloud platform for managing the complete ML lifecycle (Microsoft, 2024). Specifically, the Azure ML Python SDK v2 provides a sophisticated, code-first interface for granular programmatic control over all facets of ML experimentation and pipeline orchestration (Microsoft, 2024). This empowers practitioners with enhanced automation and a deeper understanding of operational mechanics.

This paper delivers a practical guide for constructing and automating ML training pipelines using the Azure ML Python SDK v2. It addresses core MLOps principles: ensuring reproducibility through versioned assets and defined software environments; promoting automation of intricate workflows with modular, reusable components; and enabling robust management of ML artifacts, including MLflow integration for experiment tracking and model registration. The tutorial progresses from the programmatic setup of foundational infrastructure, including compute resources, to exploring core concepts like data asset management, custom component creation, and environment specification. By completing this guide, readers will have assembled an end-to-end training pipeline for a sample NYC Taxi fare prediction model, culminating in a versioned, MLflow-tracked model registered in Azure ML, and thereby gaining tangible skills in building automated, production-oriented ML solutions.

This guide is principally designed for developers, and also for experienced data scientists, ML engineers, and MLOps specialists, who possess proficient Python programming skills and seek to build production-oriented solutions. Access to an Azure subscription is necessary; notably, the entirety of this tutorial can be executed utilizing the Azure free tier. However, practitioners are strongly advised to exercise diligent cost management, particularly concerning the provisioning and utilization of compute resources, to prevent inadvertent expenditures.

PREREQUISITES AND ENVIRONMENT SETUP

This tutorial is accompanied by a GitHub repository containing all Python scripts, component YAML files, and the necessary data for building the example pipeline (available at [Link to Repository Placeholder - e.g., https://github.com/yourusername/azureml_sdk_v2_tutorial]).

Before proceeding with the programmatic steps outlined in this paper, users must ensure their local development environment and Azure account are correctly configured. Key prerequisites include an active Azure subscription, an Azure Machine Learning (Azure ML) Workspace, the config.json file downloaded from this workspace, Azure Command-Line Interface (CLI) authentication, and a Python virtual environment into which the azure-

***Corresponding Author:** Jayakanth Pujuri

DOI- https://doi.org/10.5281/zenodo.17162321

ai-ml (Microsoft, 2024) and azure-identity packages are installed.

For comprehensive, step-by-step instructions to fulfill these requirements, please consult the README.md file within the aforementioned GitHub repository. The subsequent sections assume these foundational setup tasks have been completed.

Establishing a Connection to the Azure ML Workspace

Programmatic interaction with Azure ML services begins with establishing an authenticated connection to the workspace. Throughout this tutorial, this connection is managed using a dedicated utility script, ml_client_connection_utils.py, located in the pipeline_scripts directory of the companion repository. This script encapsulates the logic for instantiating the azure.ai.ml.MLClient, which serves as the primary programmatic interface to the Azure ML workspace.

The core function within this utility, get_ml_client(), is designed for robustness and ease of use. It primarily attempts to establish a connection by loading workspace details from a config.json file. The script intelligently searches several common project directory locations for this file, such as the project root or a dedicated .azureml subfolder. This method, internally using MLClient.from_config(credential=DefaultAzureCredential(), path=...), uses the DefaultAzureCredential from the azure-identity library, which supports multiple authentication flows (e.g., Azure CLI login, environment variables, managed identity) to suit various execution contexts. This config.json-based approach is the recommended pathway for this tutorial.

As a fallback, if config.json is not found or if explicitly bypassed, the get_ml_client() function can utilize manually provided subscription ID, resource group name, and workspace name parameters for connection.

In the operational scripts that follow, such as 01_setup_azure_environment.py, the MLClient is typically initialized by importing and calling this utility function:

```
# pipeline_scripts/01_setup_azure_environment.py (Example Usage)
from ml_client_connection_utils import get_ml_client

# Attempt to connect to the Azure ML workspace
ml_client = get_ml_client()

if not ml_client:
    # Further error handling or script exit would occur here
    print("MLClient could not be initialized. Exiting script.")
    exit(1)

print(f"Successfully connected to Azure ML workspace: {ml_client.workspace_name}")
# Azure ML operations using ml_client can now proceed
```

Fig. 1. Excerpt of 01_setup_azure_environment.py.

The complete, well-commented source code for the ml_client_connection_utils.py utility, detailing its search logic and error handling, is available in the project's GitHub repository for review. This utility ensures a consistent and simplified connection mechanism across all subsequent tutorial scripts.

Provisioning Essential Compute Resources

With the MLClient established, the next step is to ensure the necessary Azure ML compute infrastructure is available for executing pipeline jobs. This tutorial primarily utilizes a managed Azure ML Compute Cluster for running training and data processing tasks. Optionally, an Azure ML Compute Instance can be configured for interactive development, though it is not essential for the main pipeline execution.

The script pipeline_scripts/01_setup_azure_environment.py automates the provisioning of the required compute cluster, named cc-nyc-taxi-cpu. This script performs the following actions:

- Establishes a connection to the Azure ML workspace using the get_ml_client() utility discussed previously.

Publisher: SARC Publisher

- Checks for the existence of the cc-nyc-taxi-cpu compute cluster.
- If the cluster exists, it verifies if its configuration (VM size, min/max instances) matches the desired settings for the tutorial. If not, it attempts to update the cluster.
- If the cluster does not exist, it creates a new AmlCompute cluster with a specified VM size (e.g., Standard_DS11_v2), configured to scale down to zero instances when idle to optimize costs, and a defined maximum number of instances.

Key considerations when provisioning the compute cluster include selecting an appropriate VM size available in the user's Azure region and ensuring sufficient vCPU quota for the chosen VM family. The script 01_setup_azure_environment.py encapsulates the SDK v2 logic for this:

```
# pipeline_scripts/01_setup_azure_environment.py (Snippet for AmlCompute)
from azure.ai.ml.entities import AmlCompute

cluster_name = "cc-nyc-taxi-cpu"
desired_vm_size = "Standard_DS11_v2"
# ... (other parameters like min/max instances)

try:
   compute_cluster = ml_client.compute.get(cluster_name)
   print(f"Found existing compute cluster '{cluster_name}'.")
   # ... (logic for checking and updating if necessary) ...
except ResourceNotFoundError:
   print(f"Compute cluster '{cluster_name}' not found. Creating new...")
   compute_cluster_config = AmlCompute(
      name=cluster_name,
      type="amlcompute",
      size=desired_vm_size,
      min_instances=0, # Essential for cost-saving
      max_instances=1, # Tutorial default
      idle_time_before_scale_down=120,
      # ...
   )
   ml_client.compute.begin_create_or_update(compute_cluster_config).result()
   print(f"Compute cluster '{cluster_name}' created.")
# ... (error handling) ...
```

Fig. 2. Excerpt of 01_setup_azure_environment.py.

The script also includes commented-out sections for creating a ComputeInstance (ci-nyc-taxi-dev), which users can enable if an interactive development environment within Azure ML Studio is desired. The full 01_setup_azure_environment.py script is available in the project repository.

With the workspace connection active and the compute cluster provisioned, the environment is now prepared for defining and managing the core assets of the machine learning pipeline, which are detailed in Section III.

FOUNDATIONAL ELEMENTS OF AZURE ML PIPELINES

With the Azure ML workspace connection established and essential compute infrastructure provisioned as detailed in Section II, the focus now shifts to creating the core, versioned assets that form the building blocks of an automated machine learning pipeline.

This section details the programmatic creation of data assets for traceable data handling, custom environments for reproducible execution, and modular components for individual pipeline steps, culminating with the integration of MLflow for robust experiment tracking and model governance. These elements are created using specific Python scripts that use the Azure ML SDK v2.

Data Management: Assets and Versioning

Effective MLOps practices mandate robust data management, ensuring that data used for training and evaluation is traceable, versioned, and easily

Publisher: SARC Publisher

accessible within the ML workflow. Azure ML Data Assets fulfill this role by providing versioned pointers to data. This data can be uploaded from a local source to an Azure ML Datastore—a secure connection to Azure storage services like Azure Blob Storage, which typically serves as the default datastore for an Azure ML workspace—or it can reference data already in the cloud.

For our NYC Taxi fare prediction example, this tutorial first creates a URI_FILE Data Asset by uploading the raw CSV data, and subsequently, an MLTable Data Asset is created to provide a structured, typed interface to this data for pipeline consumption.

The initial step involves uploading the local yellowTaxiData.csv file (located in the raw_data_to_upload directory of the companion repository) to the workspace's default datastore and registering it as a versioned URI_FILE asset. This is accomplished by the pipeline_scripts/02_upload_and_register_csv_asset.py script. The script defines a Data object, specifying its name, the desired version (e.g., "1" for the initial run), a description, the local path to the data file, and the type as AssetTypes.URI_FILE. The Azure ML SDK then handles the upload and registration when ml_client.data.create_or_update() is called.

```
# pipeline_scripts/02_upload_and_register_csv_asset.py (Key Snippet)
from azure.ai.ml.entities import Data
from azure.ai.ml.constants import AssetTypes

# ... (ml_client and local_data_path are defined) ...

data_asset_name = "nyc-taxi-raw-yellow-csv"
data_asset_version = "1"

my_raw_data_asset = Data(
    name=data_asset_name,
    version=data_asset_version,
    description=f"Raw NYC Yellow Taxi data ({local_csv_file_name}) uploaded...",
    path=local_data_path, # SDK handles upload from this local path
    type=AssetTypes.URI_FILE
)
registered_data_asset = ml_client.data.create_or_update(my_raw_data_asset)
                        # The registered_data_asset.path will now point to the cloud URI
```

Fig. 3. Excerpt of 02_upload_and_register_csv_asset.py.

This process results in the nyc-taxi-raw-yellow-csv:1 data asset being available in the workspace, with its path attribute reflecting its new location in Azure cloud storage.

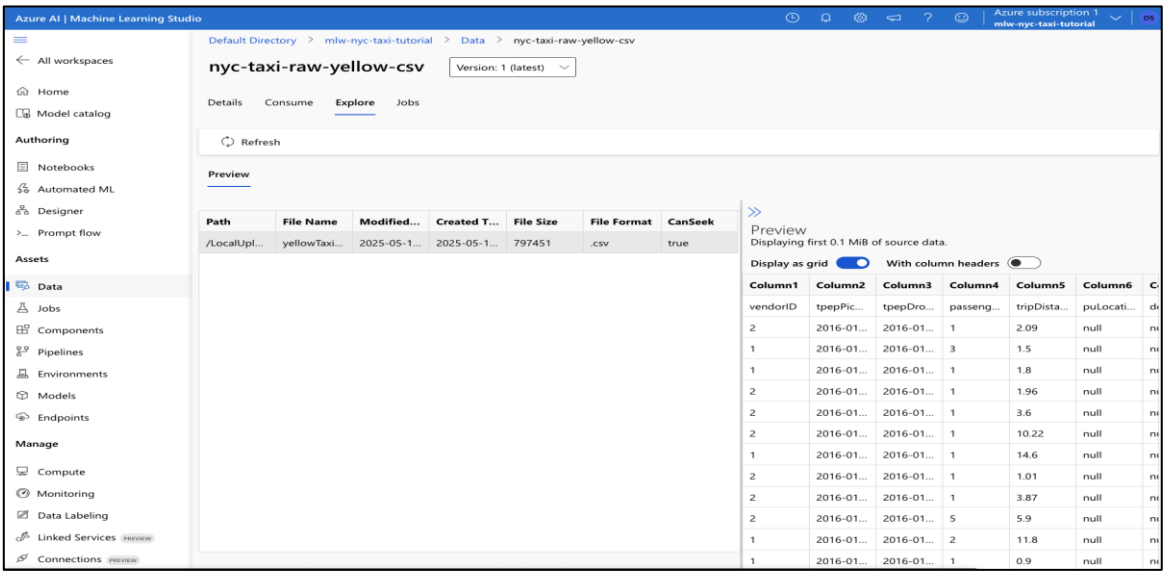

Fig. 4. The nyc-taxi-raw-yellow-csv, uploaded and registered to Azure's default data store.

Publisher: SARC Publisher

Once the raw CSV data is in the cloud and registered as a URI_FILE asset, an MLTable asset is created to provide a structured, versioned pointer to this tabular data. The MLTable itself, in this tutorial, does not perform transformations; instead, it acts as a schema-aware reference, with CSV parsing details deferred to the data preparation component for explicit control.

The script pipeline_scripts/03_create_mltable_asset.py orchestrates this. It first retrieves the cloud URI of the previously created nyc-taxi-raw-yellow-csv:1 asset. Then, it dynamically generates the content for an MLTable definition file. This definition simply contains a paths directive pointing to the cloud URI of the CSV file.

```
# pipeline_scripts/03_create_mltable_asset.py (Key Snippet for MLTable definition)
# ... (fully_qualified_csv_uri obtained from the URI_FILE asset) ...

mltable_definition_content = {
   "type": "mltable",
   "paths": [{"file": fully_qualified_csv_uri}]
}
# This content is written to a local temporary file named 'MLTable'
# in a temporary directory (e.g., '../temp_mltable_def_csv_pointer/MLTable').
```

Fig. 5. Excerpt of 03_create_mltable_asset.py.

This local directory containing the MLTable file is then used as the path when creating a new Data asset of type AssetTypes.MLTABLE.

```
# pipeline_scripts/03_create_mltable_asset.py (Key Snippet for MLTable registration)
# ... (local_mltable_def_dir is the path to the directory holding the 'MLTable' file) ...

mltable_asset_name = "nyc-taxi-mltable-yellow-csv"
mltable_asset_version = "1"

my_mltable_asset = Data(
   name=mltable_asset_name,
   version=mltable_asset_version,
   description=f"MLTable (simple pointer) referencing ...",
   path=local_mltable_def_dir, # Path to the folder containing the MLTable definition file
   type=AssetTypes.MLTABLE
)

registered_mltable_asset = ml_client.data.create_or_update(my_mltable_asset)
```

Fig. 6. Excerpt of 03_create_mltable_asset.py.

Upon successful execution, the nyc-taxi-mltable-yellow-csv:1 asset is registered.

Publisher: SARC Publisher

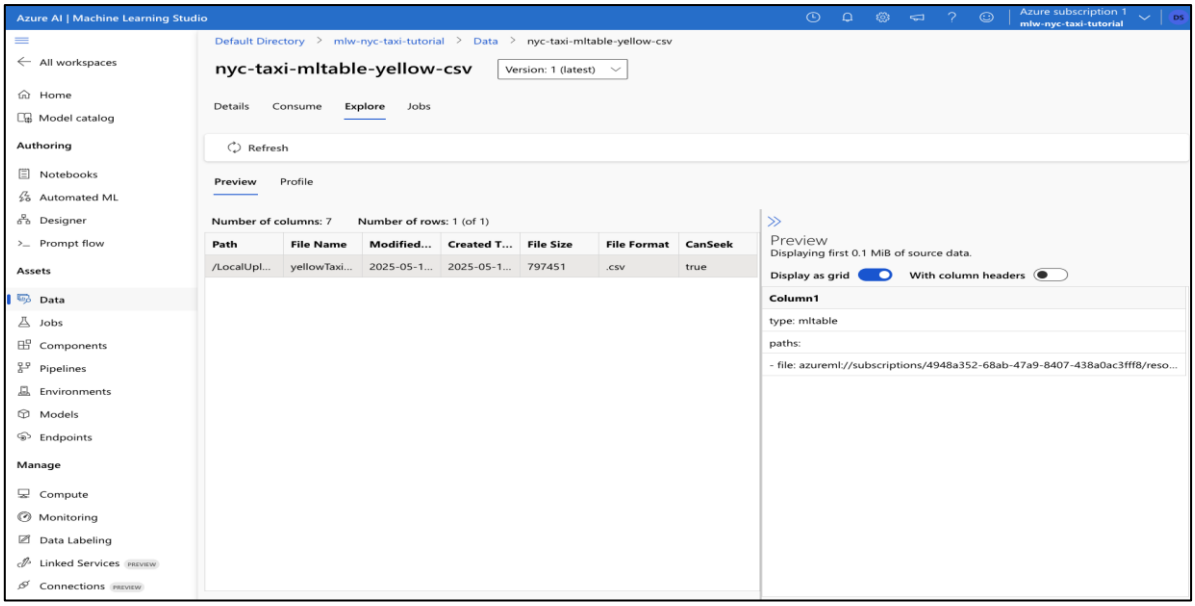

Fig. 7. The nyc-taxi-mltable-yellow-csv, uploaded and registered to Azure's default data store.

This MLTable asset will serve as the typed input for the data preparation step in the main pipeline, ensuring that the pipeline consumes a specific, versioned representation of the tabular data.

Custom Components for Modularity and Reusability

Azure ML Components are self-contained, reusable units of code that perform specific tasks within a pipeline, such as data preparation, model training, or evaluation. They encapsulate the execution logic, define necessary inputs and outputs, and specify the runtime environment. This component-based architecture is fundamental to building modular, scalable, and maintainable ML pipelines, aligning with MLOps best practices by promoting the separation of concerns and facilitating the independent development and testing of pipeline steps.

Defining Reproducible Runtime Environments: To ensure that component code executes consistently and reliably across different development and execution contexts, Azure ML utilizes Environments. An Environment defines the software runtime, including the operating system (via a base Docker image), Python packages, system libraries, and environment variables.

For this tutorial, a custom Conda environment is defined in environments/nyc_taxi_env.yml to specify all dependencies required by the pipeline components, such as scikit-learn, pandas, mlflow, and azureml-mlflow.

The environments/nyc_taxi_env.yml file lists Python and package versions to ensure reproducibility:

 # ~/azureml_sdk_v2_tutorial/environments/nyc_taxi_env.yml (Key Snippet)
name: nyc-taxi-tutorial-conda-env
channels:
- conda-forge
- defaults
dependencies:
- python=3.8.13
- pip=22.3.1
- scikit-learn=1.0.2
- pandas=1.5.3
- numpy=1.23.5
- mlflow==2.3.0
- **pip**:
  - azureml-mlflow==1.50.0
  - mltable==1.5.0

Fig. 8. Excerpt of nyc_taxi_env.yml.

Publisher: SARC Publisher

It is important to note that the name field within the Conda YAML (nyc-taxi-tutorial-conda-env) is for Conda's internal use and is distinct from the name given to the Azure ML Environment asset when it is registered.

This Conda definition is then registered as a versioned Azure ML Environment asset using the script pipeline_scripts/04_create_environment_asset.py.

This script instantiates an Environment object, providing a name for the asset (e.g., nyc-taxi-component-env), a version string (e.g., "1"), a description, the path to the local Conda YAML file, and a reference to a base Docker image (e.g., mcr.microsoft.com/azureml/openmpi4.1.0-ubuntu20.04:latest) upon which Azure ML will build the specified Conda environment.

```
# pipeline_scripts/04_create_environment_asset.py (Key Snippet)
from azure.ai.ml.entities import Environment

# ... (ml_client and conda_env_file_path_local are defined) ...

environment_name = "nyc-taxi-component-env"
environment_version = "1"

custom_environment = Environment(
    name=environment_name,
    version=environment_version,
    description="Custom Conda environment for NYC Taxi pipeline components...",
    conda_file=conda_env_file_path_local,
    image="[mcr.microsoft.com/azureml/openmpi4.1.0-
ubuntu20.04:latest](https://mcr.microsoft.com/azureml/openmpi4.1.0-ubuntu20.04:latest)"
)

registered_environment = ml_client.environments.create_or_update(custom_environment)
# The actual Docker image build occurs when this environment is first used by a job.
```

Fig. 9. Excerpt of 04_create_environment_asset.py.

Registering the environment (e.g., as nyc-taxi-component-env:1) makes it available within the Azure ML workspace.

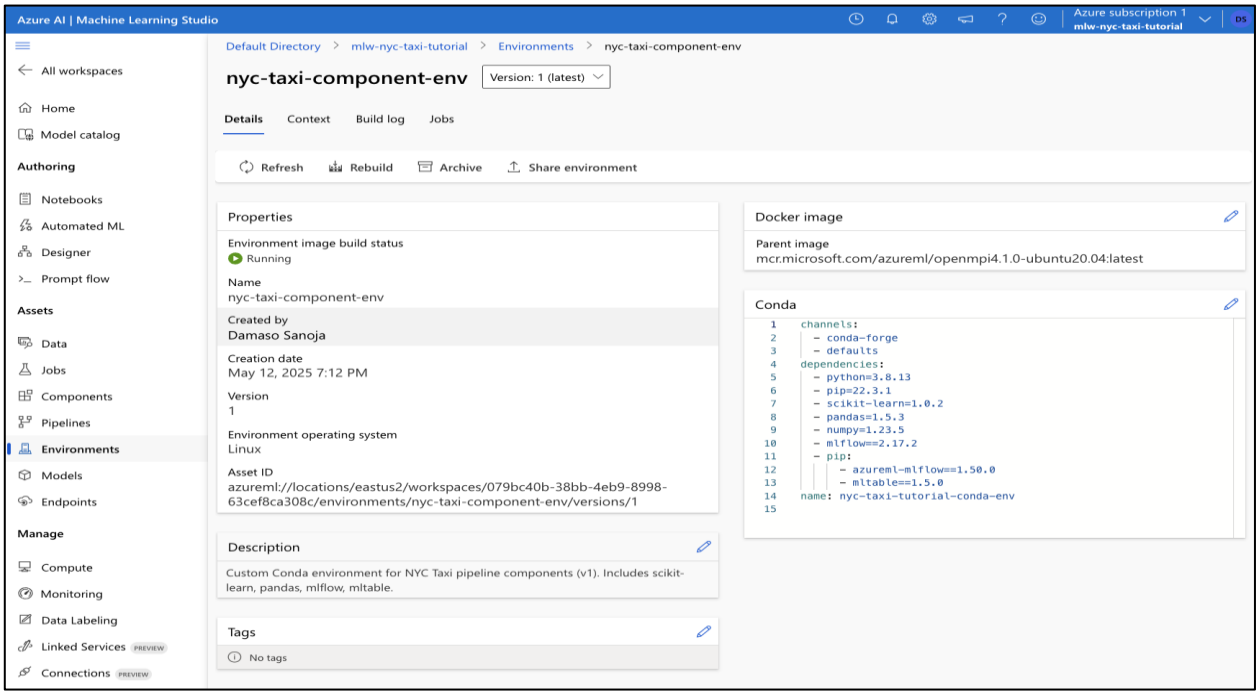

Fig. 10. The nyc-taxi-component-env, registered as a Conda environment in Azure Studio.

Publisher: SARC Publisher

Subsequent pipeline components will reference this registered environment to ensure that their code executes within this exact, reproducible software configuration. This practice mitigates issues related to dependency conflicts or "works on my machine" problems, which are critical concerns in production ML systems.

Creating a Custom Component: With a reproducible environment defined and registered, the next step is to create custom components that encapsulate specific stages of the ML workflow.

For this tutorial, a data preparation component is developed first. This involves two main parts: the Python script containing the data processing logic, and a YAML component definition file that specifies the component's interface, inputs, outputs, the script to execute, and the environment it runs in.

The core logic for data preparation is implemented in the Python script components/prep_data/src/prep_data.py. This script is designed to be executed from the command line and accepts arguments for the raw data input path, the output path for prepared data, a test split ratio, and a random state for reproducibility. Internally, it performs several operations:

- Loads the raw CSV data (passed as a URI_FILE by the pipeline) into a pandas DataFrame (The Pandas Development Team, 2020).
- Applies basic filtering (e.g., removing records with zero passenger count or fare amount).
- Converts and validates datetime columns, handling potential NaT values.
- Engineers new temporal features such as hour of day, day of week, and month from the pickup datetime.
- Selects relevant features for model training and the target variable (fareAmount).
- Handles potential infinite values and missing values in the target variable.
- Ensures all feature columns are numeric, coercing types and imputing remaining NaNs in feature columns using the median.
- Splits the processed data into training and testing sets using sklearn.model_selection.train_test_split. (F. Pedregosa *et al.*, 2011)
- Saves the resulting training and testing DataFrames as Parquet files (train_data.parquet, test_data.parquet) into designated subfolders within the component's output path.

The interface for this data preparation component is defined in components/prep_data/prep_data_component.yml. This YAML file specifies metadata such as the component's name (prep_nyc_taxi_data_custom), version ("1"), display_name, and description. Crucially, it defines the expected inputs (e.g., raw_data_input_path of type uri_file, test_split_ratio of type number) and outputs (e.g., prepared_data_path of type uri_folder). The code key points to the src/ directory containing prep_data.py, and the environment key references the previously registered nyc-taxi-component-env:1. The command section details how to execute prep_data.py, mapping the component's inputs and outputs to the script's command-line arguments using Azure ML's expression syntax.

A key snippet from prep_data_component.yml illustrates the input, output, and command definition:

```
# components/prep_data/prep_data_component.yml (Key Snippet)
$schema: https://azuremlschemas.azureedge.net/latest/commandComponent.schema.json
name: prep_nyc_taxi_data_custom
version: 1
type: command
inputs:
  raw_data_input_path:
    type: uri_file
  test_split_ratio:
    type: number
    default: 0.2
# ... (other inputs)
outputs:
  prepared_data_path:
```


Publisher: SARC Publisher

```
  type: uri_folder
code: ./src/ # Points to the directory containing prep_data.py
command: >-
  python prep_data.py
  --raw_data_input_path ${{inputs.raw_data_input_path}}
  --prepared_data_path ${{outputs.prepared_data_path}}
  --test_split_ratio ${{inputs.test_split_ratio}}
  # ... (other arguments)
environment: azureml:nyc-taxi-component-env:1
```

Fig. 11. Excerpt of prep_data_component.yml.

To make this component available for use in pipelines, it is registered in the Azure ML workspace using the script pipeline_scripts/05_register_prep_component.py. This script uses the Azure ML SDK's load_component() function to parse the YAML definition file. The loaded component object is then passed to ml_client.components.create_or_update() for registration.

```
# pipeline_scripts/05_register_prep_component.py (Key Snippet)
from azure.ai.ml import load_component

# ... (ml_client and component_yaml_path_local are defined) ...

# Load component definition from YAML
data_prep_component = load_component(source=component_yaml_path_local)

# Register the component
registered_component = ml_client.components.create_or_update(data_prep_component)

print(f"Successfully registered component: {registered_component.name} version {registered_component.version}")
```

Fig. 12. Excerpt of 05_register_prep_component.py.

Once registered (e.g., as prep_nyc_taxi_data_custom:1), this data preparation component can be readily incorporated as a versioned, reusable step in various machine learning pipelines.

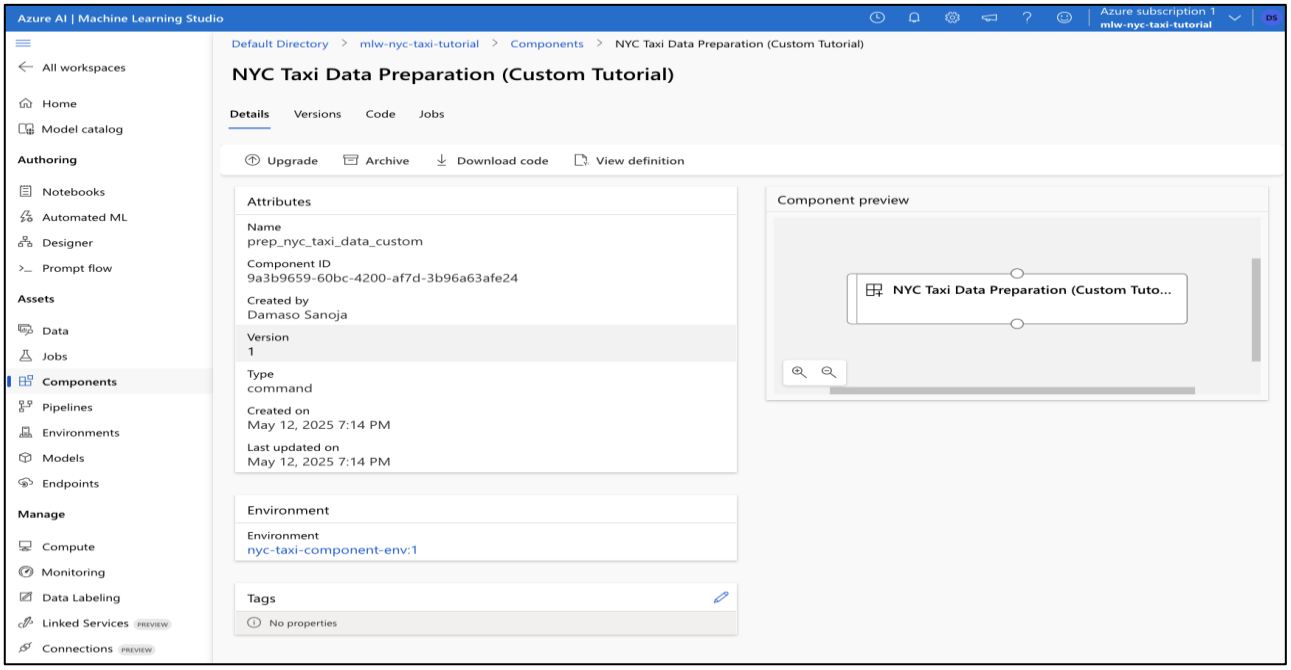

Fig. 13. The prep_nyc_taxi_data_custom Component preview in Azure Studio.

Publisher: SARC Publisher

This modular approach simplifies pipeline construction and maintenance, as the component's logic can be updated and versioned independently.

Experiment Tracking and Model Governance with MLflow

Comprehensive experiment tracking and model management are critical for iterative development and production-grade MLOps. Azure Machine Learning seamlessly integrates with MLflow, an open-source platform for managing the end-to-end machine learning lifecycle (Zaharia, M. *et al.*, 2018). This integration allows for automatic tracking of runs, parameters, metrics, artifacts, and models within the Azure ML workspace, providing a centralized location for governance and reproducibility.

The model training component for the NYC Taxi fare prediction task, defined by components/train_model/train_model_component.yml and its corresponding script components/train_model/src/train_model.py, uses this MLflow integration extensively. The train_model.py script is responsible for:

- Loading the prepared training data (output from the data preparation component).
- Training a regression model (e.g., sklearn.linear_model.Ridge in this tutorial).
- Within an mlflow.start_run() context, logging key aspects of the training process:

Parameters: Hyperparameters like alpha and other configuration details are logged using mlflow.log_param().

Metrics: Evaluation metrics calculated on the training data, such as Root Mean Squared Error (RMSE) and R-squared (R2) score, are logged using mlflow.log_metric().

Model Artifacts: The trained scikit-learn model is logged using mlflow.sklearn.log_model(). This function also facilitates including a model signature (inferred from sample input and output data using mlflow.models.signature.infer_signature) and an input_example. Crucially, by specifying the registered_model_name argument within mlflow.sklearn.log_model(), the model is not only saved as an artifact of the MLflow run but is also automatically registered or versioned within the Azure ML Model Registry.

Other Artifacts: Additional files, such as a text file containing the names of features used for training (feature_names.txt), can be logged using mlflow.log_artifact().

Key snippets from components/train_model/src/train_model.py illustrate these MLflow logging operations:

```
# components/train_model/src/train_model.py (Key MLflow Snippets)
import mlflow
import mlflow.sklearn
from mlflow.models.signature import infer_signature

# ... (inside main function and after loading data as X_train, y_train) ...

with mlflow.start_run() as run:
    mlflow.log_param("alpha_hyperparameter", args.alpha)
    # ... (other parameters) ...

    model = Ridge(alpha=args.alpha, random_state=args.random_state)
    model.fit(X_train, y_train)

    predictions_train = model.predict(X_train)
    rmse_train = mean_squared_error(y_train, predictions_train, squared=False)
    r2_train = r2_score(y_train, predictions_train)
    mlflow.log_metric("training_rmse", rmse_train)
    mlflow.log_metric("training_r2_score", r2_train)

    input_example_df = X_train.head()
    example_predictions = model.predict(input_example_df)
    signature = infer_signature(input_example_df, example_predictions)
```


Publisher: SARC Publisher

```
  mlflow.sklearn.log_model(
      sk_model=model,
      artifact_path="nyc_taxi_fare_model_files",
      registered_model_name=args.registered_model_name,
      signature=signature,
      input_example=input_example_df
  )

  # Example of logging an additional artifact
  # with open("feature_names.txt", 'w') as f: # features are written to this file
  #     # ... write feature names ...
  # mlflow.log_artifact("feature_names.txt", artifact_path="training_run_details")
```

Fig. 14. Excerpt of train_model.py.

The components/train_model/train_model_component.yml file defines the interface for this training component. It specifies inputs such as the path to prepared data (prepared_data_input_path), hyperparameters like alpha, and the registered_model_name. It also declares an output (model_output_mlflow of type uri_folder), which can be used for auxiliary files like a training summary, distinct from the MLflow-managed model artifacts. Similar to the data preparation component, it references the nyc-taxi-component-env:1 environment and defines the command to execute train_model.py with the necessary argument bindings.

Finally, the script pipeline_scripts/06_register_train_component.py registers this training component. It follows the same pattern as the data preparation component registration: it loads the component definition from train_model_component.yml using load_component() and then uses ml_client.components.create_or_update() to register it in the Azure ML workspace (e.g., as train_nyc_taxi_model_custom:1).

```
# pipeline_scripts/06_register_train_component.py (Key Snippet)
from azure.ai.ml import load_component

# ... (ml_client and component_yaml_path_local for train_model_component.yml are defined) ...

model_train_component = load_component(source=component_yaml_path_local)
registered_component = ml_client.components.create_or_update(model_train_component)

print(f"Successfully registered component: {registered_component.name} version {registered_component.version}")
```

Fig. 15. Excerpt of 06_register_train_component.py.

By registering this training component, which internally handles all MLflow logging and model registration, a modular and traceable approach to model training and governance is established.

Publisher: SARC Publisher

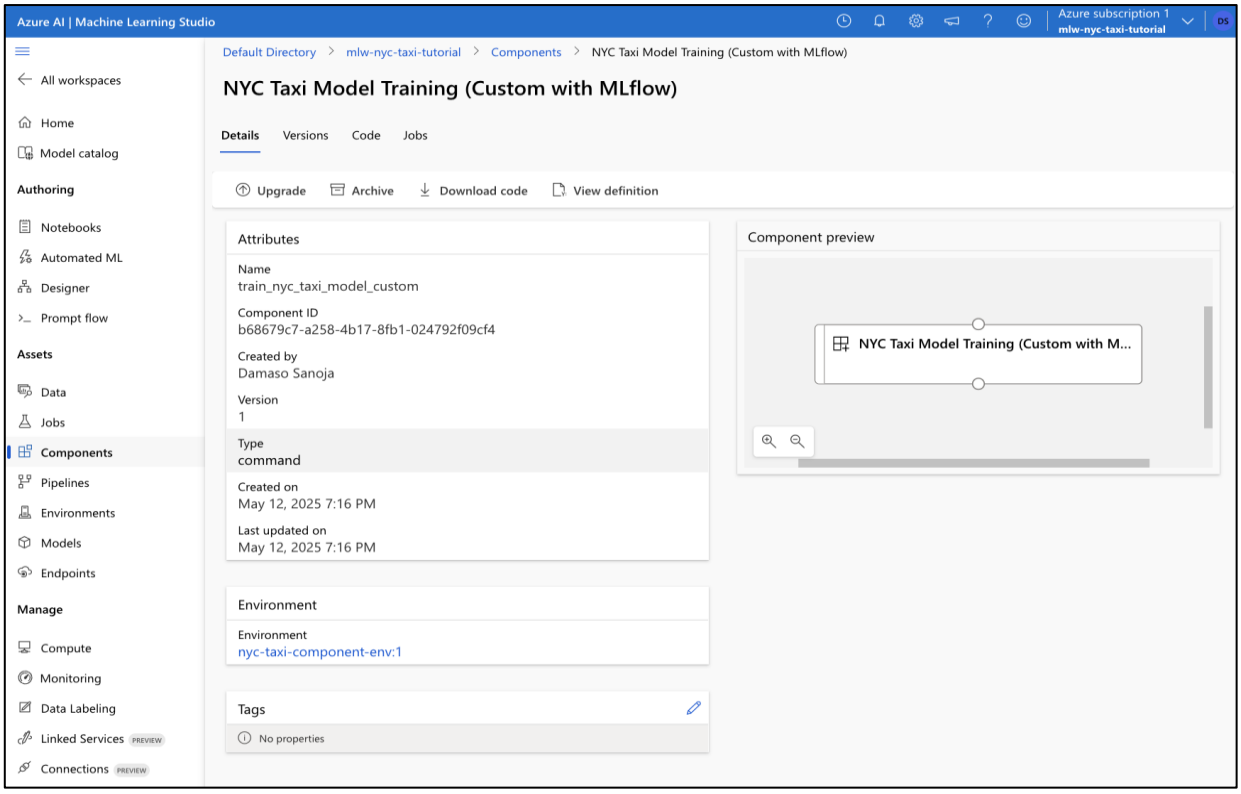

Fig. 16. The train_nyc_taxi_model_custom Component in Azure Studio.

The resulting MLflow runs and registered models in Azure ML provide a comprehensive audit trail and facilitate model deployment and lifecycle management, which will be demonstrated when the full pipeline is assembled in Section IV.

ASSEMBLING AND EXECUTING THE NYC TAXI FARE PREDICTION PIPELINE

This section demonstrates the culmination of the concepts and assets developed in Section III, detailing the construction, execution, and monitoring of an automated training pipeline for the NYC Taxi fare prediction example.

The primary script guiding this process is pipeline_scripts/07_run_nyc_taxi_pipeline.py. This walkthrough will show how the versioned Data Assets, custom Environment, and modular Components are orchestrated into a cohesive and reproducible MLOps workflow using the Azure ML Python SDK v2.

Pipeline Definition: Orchestrating Components

The core of the automated workflow is an Azure ML pipeline defined programmatically using a Python function and the @pipeline decorator from the azure.ai.ml.dsl module. This approach provides a clear and maintainable way to define the sequence of operations, data dependencies, and parameterization of the entire training process.

The pipeline_scripts/07_run_nyc_taxi_pipeline.py script begins by establishing a connection to the Azure ML workspace and verifying the essential cc-nyc-taxi-cpu compute target. It then proceeds to load the specific, versioned components that were registered in Section III:

Loading Registered Components: The prep_nyc_taxi_data_custom:1 (data preparation) and train_nyc_taxi_model_custom:1 (model training) components are retrieved from the workspace using ml_client.components.get(). This ensures that the pipeline consistently utilizes version "1" of these building blocks, which is the version created by the tutorial's preceding scripts.

Publisher: SARC Publisher

```
# pipeline_scripts/07_run_nyc_taxi_pipeline.py (Snippet: Loading components)
prep_component_name = "prep_nyc_taxi_data_custom"
prep_component_version = "1"
train_component_name = "train_nyc_taxi_model_custom"
train_component_version = "1"
prep_data_component_func = ml_client.components.get(name=prep_component_name, version=prep_component_version)
train_model_component_func = ml_client.components.get(name=train_component_name, version=train_component_version)
```

Fig. 17. Excerpt of 07_run_nyc_taxi_pipeline.py.

Defining the Pipeline Function: A Python function, nyc_taxi_pipeline_tutorial_run in this case, is decorated with @pipeline. This decorator registers the function as a pipeline definition with Azure ML, allowing it to be instantiated and run as a job. The function defines:

Pipeline-Level Inputs: These are parameters that can be passed to the pipeline at runtime. For this example, pipeline_input_data (of type Input from azure.ai.ml) is defined to accept the raw input data, pipeline_alpha_for_training allows specifying the Ridge regression hyperparameter, and pipeline_model_registration_name dictates the name under which the trained model will be registered.

Component Instantiation as Jobs: Inside the pipeline function, jobs are created by calling the loaded component functions (e.g., prep_data_component_func(...)). Inputs are passed to these component jobs, including pipeline-level inputs or outputs from preceding steps. For instance, parameters like test_split_ratio for the data preparation job are explicitly set within the pipeline, overriding any defaults from the component's YAML if needed.

Data Flow and Dependency Management: The critical connection between components is established by directing the output of one component job to the input of a subsequent one. The prepared_data_path output from the prep_data_job is passed as the prepared_data_input_path to the train_model_job. Azure ML automatically manages the data transfer or mounting based on these defined dependencies.

Pipeline Outputs: The pipeline can also declare its own outputs, often sourced from the outputs of its constituent component jobs, such as the path to the prepared data and the training summary location.

A conceptual representation of the pipeline definition within 07_run_nyc_taxi_pipeline.py:

```
# pipeline_scripts/07_run_nyc_taxi_pipeline.py (Conceptual Pipeline Definition)
from azure.ai.ml.dsl import pipeline
from azure.ai.ml import Input

@pipeline(
    name="nyc_taxi_training_pipeline_v1_final",
    display_name="NYC Taxi Fare Prediction Training Pipeline (v1 - Tutorial Run)"
    # ... other pipeline metadata
)
def nyc_taxi_pipeline_tutorial_run(
    pipeline_input_data: Input,
    pipeline_alpha_for_training: float = 1.0,
    pipeline_model_registration_name: str = "nyc-taxi-fare-predictor"
):
    prep_data_job = prep_data_component_func( # Loaded component
        raw_data_input_path=pipeline_input_data,
        test_split_ratio=0.2
        # ... other inputs
    )

                train_model_job = train_model_component_func( # Loaded component
```


Publisher: SARC Publisher

```
        prepared_data_input_path=prep_data_job.outputs.prepared_data_path, # Wiring output to input
        alpha=pipeline_alpha_for_training,
        registered_model_name=pipeline_model_registration_name
        # ... other inputs
    )
    return { # Defining pipeline outputs
        "pipeline_prepared_data_output": prep_data_job.outputs.prepared_data_path,
        "pipeline_model_training_summary_output": train_model_job.outputs.model_output_mlflow
    }
```

Fig. 18. Excerpt of 07_run_nyc_taxi_pipeline.py.

Pipeline Instantiation and Execution

Once the pipeline structure is defined via the decorated Python function, it must be instantiated as a PipelineJob object and configured for execution:

Instantiating the Pipeline: The pipeline function (nyc_taxi_pipeline_tutorial_run) is called. For this tutorial's initial run, it is invoked by passing an Input object that references the nyc-taxi-raw-yellow-csv:1 data asset (created in Section III.A) as the pipeline_input_data. Specific values for other pipeline parameters like pipeline_alpha_for_training and pipeline_model_registration_name are also provided at this stage.

```
# pipeline_scripts/07_run_nyc_taxi_pipeline.py (Snippet: Instantiating the pipeline)
input_data_asset_name = "nyc-taxi-raw-yellow-csv"
input_data_asset_version = "1"
# ... (code to get input_data_for_pipeline object) ...
pipeline_job = nyc_taxi_pipeline_tutorial_run(
    pipeline_input_data=Input(type="uri_file",
path=f"azureml:{input_data_for_pipeline.name}:{input_data_for_pipeline.version}"),
    pipeline_alpha_for_training=0.5,
    pipeline_model_registration_name="nyc-taxi-fare-predictor-tutorial-v1"
)
```

Fig. 19. Excerpt of 07_run_nyc_taxi_pipeline.py.

Setting Default Compute Target: A crucial step for ensuring the pipeline jobs run on the intended infrastructure is to assign a default compute target to the pipeline job instance. This is done programmatically after the pipeline job is instantiated by setting its settings.default_compute attribute. For this tutorial, the cc-nyc-taxi-cpu compute cluster (provisioned in Section II.B) is set as the default.

```
# pipeline_scripts/07_run_nyc_taxi_pipeline.py (Snippet: Setting default compute)
pipeline_job.settings.default_compute = "cc-nyc-taxi-cpu"
```

Submitting the Pipeline Job: The configured pipeline_job is then submitted to Azure ML using ml_client.jobs.create_or_update(). An experiment_name (e.g., nyc_taxi_tutorial_pipeline_v1_runs) is provided to group and organize related pipeline runs within the Azure ML Studio, facilitating easier tracking and comparison.

Publisher: SARC Publisher

```
# pipeline_scripts/07_run_nyc_taxi_pipeline.py (Snippet: Submitting the job)
returned_pipeline_job = ml_client.jobs.create_or_update(
    pipeline_job,
    experiment_name="nyc_taxi_tutorial_pipeline_v1_runs"
)
print(f"Pipeline job submitted successfully.")
print(f"  Job Name: {returned_pipeline_job.name}")
print(f"  View in Azure ML Studio: {returned_pipeline_job.studio_url}")
```

Monitoring and Reviewing Pipeline Results

Upon successful submission, the 07_run_nyc_taxi_pipeline.py script outputs a URL to the pipeline run in Azure ML Studio. This web interface is central to monitoring and understanding the pipeline's execution:

Visualize the Pipeline Graph: The Studio presents a directed acyclic graph (DAG) of the pipeline, visually representing the components, their execution order, and data dependencies.

Monitor Run Status: Users can track the real-time status of the overall pipeline and each individual component job (e.g., Preparing, Running, Completed, Failed).

Inspect Component Details: Each component job in the graph can be selected to view its specific inputs, outputs, parameters, and, importantly, its execution logs. These logs include standard output (std_log.txt), which contains messages printed by the component scripts (e.g., prep_data.py and train_model.py), crucial for debugging and verification.

Review MLflow Tracking Data: For the model training component (train_nyc_taxi_model_custom:1), the "Metrics" tab will display metrics logged via MLflow (e.g., training_rmse, training_r2_score). The "Outputs + logs" tab will contain artifacts logged by MLflow, including the model files themselves (typically under a path like nyc_taxi_fare_model_files) and any additional artifacts like feature_names.txt. The model registered via MLflow will also be versioned and available in the workspace's central "Models" registry.

Access Pipeline Outputs: If the pipeline definition included a return statement to declare pipeline-level outputs (as shown in the conceptual snippet), these outputs can be found and downloaded from the "Outputs + logs" tab of the parent pipeline run.

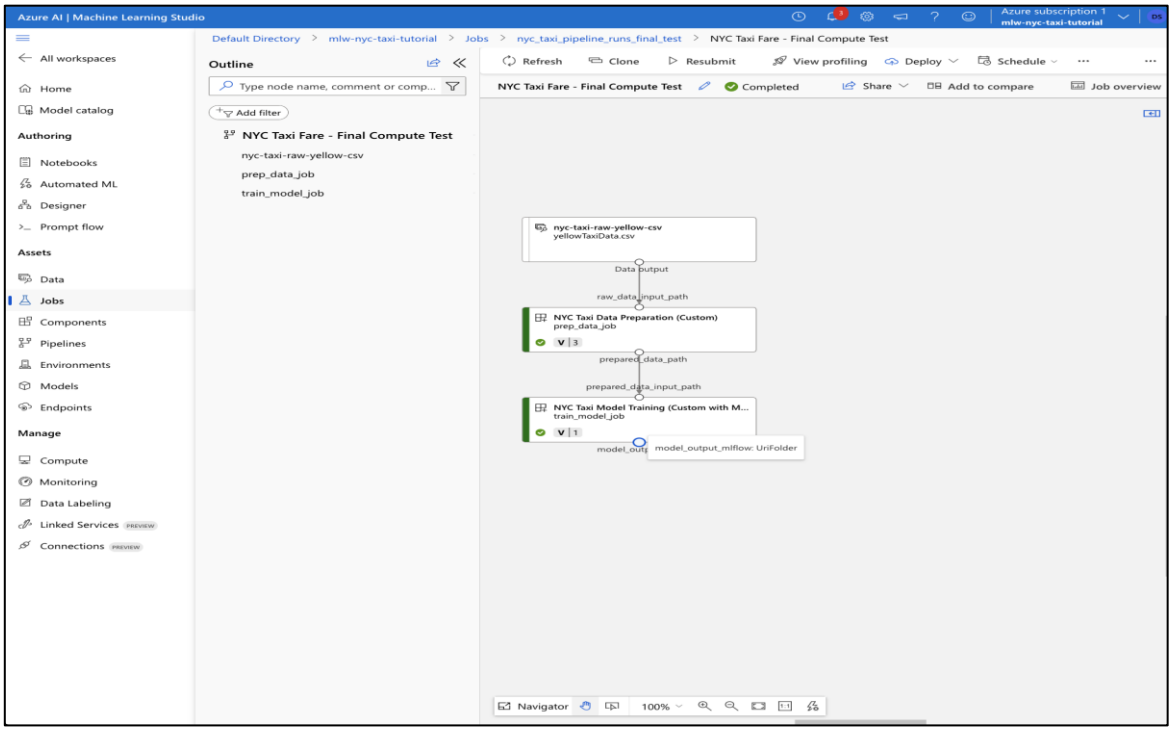

Fig. 20. Completed NYC Taxi Fare pipeline in Azure Studio.

Publisher: SARC Publisher

Considerations for Iteration and Version Management

The scripts and pipeline definition detailed in this tutorial are intentionally structured for a "clean run," consistently utilizing version "1" of all created Data Assets, the custom Environment, and both custom Components. This approach is designed to ensure predictability and reproducibility for readers undertaking the tutorial for the first time.

However, real-world machine learning development is an inherently iterative process. Should a user modify the underlying logic or interface of a component (e.g., change feature engineering steps in prep_data.py, alter its YAML definition, or update the model architecture in train_model.py), it is a fundamental MLOps best practice to register this modified entity as a *new version* in their Azure ML workspace (e.g., prep_nyc_taxi_data_custom:2). This preserves the integrity and lineage of all asset versions, allowing for rollback and comparison. Similarly, significant changes to data sources or the software environment would warrant the creation of new versions for those respective assets.

To assist users in managing multiple asset versions during such experimentation, the companion GitHub repository includes a utility script, pipeline_scripts/check_asset_versions.py. This script connects to the workspace and provides a listing of all available versions for specified data assets, components, and environments. While not part of the core pipeline execution flow, this utility can be a valuable tool for tracking the assets created during iterative development.

To execute the pipeline with these newer, user-modified asset versions, the 07_run_nyc_taxi_pipeline.py script would require corresponding adjustments. Specifically, the sections where components are loaded (e.g., ml_client.components.get(name="prep_nyc_taxi_data_custom", version="2")) and where input data assets are referenced (e.g., Input(path="azureml:my_data_asset:new_version")) would need to be updated to point to the new, desired version numbers. This disciplined approach to versioning all dependencies is key to maintaining robust, traceable, and adaptable machine learning workflows as projects evolve.

CONCLUSION AND OUTLOOK

This paper has furnished a practical, step-by-step guide to building and automating end-to-end machine learning training pipelines using the Azure Machine Learning Python SDK v2. Through a sequence of programmatic operations, readers have learned to establish a connection to an Azure ML workspace and provision essential compute infrastructure (Section II). Subsequently, the tutorial detailed the creation and registration of core MLOps assets: versioned Data Assets (URI_FILE and MLTable) for traceable data inputs, a custom Conda-based Environment for reproducible execution contexts, and modular, reusable Components for distinct pipeline tasks such as data preparation and model training, including the integration of MLflow for comprehensive experiment tracking and model registration (Section III). The assembly of these foundational elements into a complete, executable training pipeline for an NYC Taxi fare prediction example was then demonstrated, covering pipeline definition, submission, and result monitoring (Section IV).

By completing this tutorial, practitioners gain not only a functional Azure ML training pipeline but also a suite of versioned assets—including a registered MLflow model—and, more importantly, the applied knowledge to use Azure ML for robust, automated, and reproducible MLOps workflows. The presented approach highlights key benefits: automation of the entire training lifecycle, reproducibility stemming from versioned assets and defined environments, modularity through self-contained components, and enhanced governance via MLflow's tracking capabilities.

While this guide provides a strong foundational example, several considerations arise when extending these concepts to more complex, production-grade scenarios. Scaling pipelines may involve utilizing larger compute clusters, distributing component workloads, or optimizing data access patterns. Effective component design—emphasizing clear interfaces, independent testability, and minimal dependencies—along with diligent environment management, such as using lean, optimized base Docker images and regularly auditing package versions, are crucial for long-term maintainability. Practitioners should also be mindful of potential challenges, such as managing intricate data lineage across multiple pipelines or integrating with enterprise security and networking policies, which often require bespoke solutions. The discussion on iteration and versioning (Section IV.D) provides initial guidance for managing the evolution of these ML systems.

Publisher: SARC Publisher

Future enhancements to the demonstrated pipeline could include the addition of a dedicated model evaluation component, the integration of automated model deployment to an Azure ML Managed Endpoint for real-time or batch inference, and the implementation of automated retraining triggers based on data drift detection or model performance degradation monitoring. Incorporating advanced Responsible AI tools for fairness assessment and model explainability would further enrich the pipeline's production readiness.

Ultimately, this tutorial equips developers and MLOps engineers with the foundational skills and understanding necessary to confidently develop, deploy, and manage sophisticated and reliable machine learning & training pipelines on the Azure platform, fostering a more mature and efficient approach to the machine learning lifecycle.

ACKNOWLEDGMENT

The author would like to acknowledge the use of Google's Gemini model for AI-assisted editing to improve the manuscript's grammar and clarity.

REFERENCES

1. Microsoft, "What is Azure Machine Learning? - Azure Machine Learning." *Microsoft Docs*, (2024).
2. Microsoft, "What is Azure Machine Learning CLI and Python SDK v2? - Azure Machine Learning," *Microsoft Docs*, (2024).
3. Microsoft, "azure-ai-ml – PyPI." *Python Package Index*, (2024).
4. The Pandas Development Team, "pandas-dev/pandas: Pandas." *Zenodo*, (2020).
5. Pedregosa, F. *et al.*, "Scikit-learn: Machine Learning in Python," Journal of Machine Learning Research, vol. 12, pp. 2825-2830, (2011).
6. Zaharia, M., Chen, A., Davidson, A., Ghodsi, A., Hong, S. A., Konwinski, A., ... & Zumar, C. "Accelerating the machine learning lifecycle with MLflow." *IEEE Data Eng. Bull*. 41.4 (2018): 39-45.

Source of support: Nil; **Conflict of interest:** Nil.

Cite this article as:
Pujuri, J. "A Practical Guide to Building End-to-End Azure Machine Learning Training Pipelines with Python SDK v2". *Technology Perception* 1.2 (2025): pp 1-17.

Publisher: SARC Publisher