**TASK 3**

STEPS TO CREATE A CICD PIPELINE:

1. Create a new project.
2. Clone the project folder using command git clone.
3. Move to that specified folder and open the same in code editor.
4. Now create the required files in the code editor (VS Code)
5. Raw\_data.py:

import pandas as pd

data = [

{"order\_id": 1, "customer": "Alice", "region": "East", "revenue": 120000, "cost": 80000},

{"order\_id": 2, "customer": "Bob", "region": "West", "revenue": 60000, "cost": 40000},

{"order\_id": 3, "customer": "Charlie", "region": None, "revenue": None, "cost": 20000},

{"order\_id": 4, "customer": "David", "region": "South", "revenue": 40000, "cost": 25000},

{"order\_id": 2, "customer": "Bob", "region": "West", "revenue": 60000, "cost": 40000},

]

df = pd.DataFrame(data)

df.to\_csv("raw\_sales\_data.csv", index=False)

print("raw\_sales\_data.csv created")

This code is executed with the help of Python’s built-in **csv** library. It uses the sample data given in the list data. Upon execution, it opens a new file named raw\_sales\_data.csv and writes the sales records into it.

1. Sales\_data.py

import pandas as pd

raw\_file = "raw\_sales\_data.csv"

processed\_file = "processed\_sales\_data.csv"

df = pd.read\_csv(raw\_file)

df = df.drop\_duplicates(subset=["order\_id"])

df["region"].fillna("Unknown", inplace=True)

df["revenue"].fillna(0, inplace=True)

df["profit\_margin"] = ((df["revenue"] - df["cost"]) /

df["revenue"].replace(0, 1))

df["customer\_segment"] = df["revenue"].apply(

lambda r: "Platinum" if r > 100000 else "Gold" if r > 50000 else "Standard"

)

df.to\_csv(processed\_file, index=False)

print("Processed sales data saved to", processed\_file)

This script cleans sales\_data.csv by removing duplicates, filling missing values, calculating profit margin, segmenting customers (Platinum/Gold/Standard), and saving the results into processed\_sales\_data.csv.

1. Azure\_pipelines.yml:

trigger:

- main

pool:

vmImage: 'ubuntu-latest'

variables:

pythonVersion: '3.10'

steps:

# Step 1: Setup Python

- task: UsePythonVersion@0

inputs:

versionSpec: '$(pythonVersion)'

# Step 2: Install dependencies

- script: |

python -m pip install --upgrade pip

pip install -r requirements.txt

displayName: 'Install dependencies'

# Step 3: Run Data Pipeline (generate files)

- script: |

python raw\_data.py

python sales\_data.py

displayName: 'Run Data Processing'

# Step 4: Upload to Azure Blob Storage

- task: AzureCLI@2

inputs:

azureSubscription: 'YourServiceConnectionName' # Create a service connection in DevOps

scriptType: 'bash'

scriptLocation: 'inlineScript'

inlineScript: |

echo "Uploading raw\_sales\_data.csv to Blob..."

az storage blob upload \

--account-name $(AZURE\_STORAGE\_ACCOUNT\_NAME) \

--account-key $(AZURE\_STORAGE\_ACCOUNT\_KEY) \

--container-name $(AZURE\_CONTAINER\_NAME) \

--file raw\_sales\_data.csv \

--name raw\_sales\_data.csv \

--overwrite true

echo "Uploading processed\_sales\_data.csv to Blob..."

az storage blob upload \

--account-name $(AZURE\_STORAGE\_ACCOUNT\_NAME) \

--account-key $(AZURE\_STORAGE\_ACCOUNT\_KEY) \

--container-name $(AZURE\_CONTAINER\_NAME) \

--file processed\_sales\_data.csv \

--name processed\_sales\_data.csv \

--overwrite true

# Step 5: Publish files as artifacts

- publish: raw\_sales\_data.csv

artifact: raw-data

- publish: processed\_sales\_data.csv

artifact: processed-data

**trigger: - main**

* This tells Azure DevOps to **run the pipeline automatically** every time someone pushes commits to the **main** branch.
* If you commit to any other branch, this pipeline won’t start (unless you run it manually).

**pool: vmImage: ubuntu-latest**

* The pipeline will run on a **Microsoft-hosted Ubuntu Linux VM**.
* Think of it as a fresh, temporary machine that Azure gives you just for this run.

**steps:**

This is the list of actions the VM will perform, in order.

**Step 1 — Set up Python**Installs and activates Python 3.10 on the build agent so python is available for next steps.

**Step 2 — Install dependencies**  
Upgrades pip and installs packages from requirements.txt, preparing the environment for your scripts.

**Step 3 — Run data processing**

* Executes raw\_data.py to generate raw\_sales\_data.csv.
* Executes sales\_data.py to clean/enrich data and produce processed\_sales\_data.csv.

**Step 4 — Upload to Azure Blob Storage**  
Uses AzureCLI@2 (authenticated via your Service Connection) to upload both CSVs to your specified storage account and container, overwriting existing blobs if present.

**Step 5 — Publish artifacts**  
Publishes raw\_sales\_data.csv and processed\_sales\_data.csv as pipeline artifacts, so you can download and audit them from the pipeline run.

**BONUS :**

1. Why is storing cleaned data in Azure Blob Storage important for real-time pipelines?
   * **Centralized storage**: Blob Storage acts as a single source of truth, accessible by multiple downstream services (Spark, Synapse, Databricks, Power BI).
   * **Scalability**: It can handle large volumes of structured/unstructured data at scale.
   * **Durability & reliability**: Data stored in Blob is geo-redundant and highly available.
   * **Integration**: Real-time pipelines (Azure Data Factory, Event Grid, Databricks streaming) can pick up cleaned data directly from Blob Storage to drive analytics, dashboards, or ML models.
2. What’s the difference between pipeline artifacts and Blob Storage uploads?
   * **Pipeline Artifacts**: Temporary outputs generated during a pipeline run. They are mainly for debugging, testing, or passing files between stages in the same pipeline. They don’t persist after retention policies expire.
   * **Blob Storage**: A permanent storage service where data can be kept long-term. It is production-ready, supports large-scale datasets, and can be accessed across multiple tools, pipelines, and even external apps.
3. How would you handle failures in file uploads in a production setup?
   * **Retry policies**: Automatically retry upload a few times with exponential backoff (to handle network glitches).
   * **Error handling**: Wrap Azure CLI/SDK upload calls in try–except (Python) or check exit codes in YAML.
   * **Alerts/Monitoring**: Use Azure Monitor or Log Analytics to track failures and send alerts (email/Teams).
   * **Transactional flow**: Upload to a staging container first → validate → then move to final container.