## 1. Data Ingestion (Batch Process with Azure Data Factory)

### Step 1.1: Setup Azure Data Lake Storage (ADLS) Gen2

- **Objective**: Store raw and transformed data.
- **Steps**:
    1. In the Azure portal, create an ADLS Gen2 account.
    2. Create a **container** called raw-data for storing raw files.
    3. Create another container called transformed-data for storing cleaned and transformed data.

### Step 1.2: Upload Datasets to ADLS Gen2

- **Objective**: Store raw datasets in the raw-data container.
- **Steps**:
    1. Use Azure Storage Explorer or the Azure Portal to upload the datasets (call_records.csv, customer_usage_data.csv, etc.) to the raw-data container in ADLS Gen2.

### Step 1.3: Set Up Azure Data Factory (ADF) Pipeline

- **Objective**: Perform batch ETL for transforming and loading the data into ADLS Gen2.
- **Steps**:
    1. **Create Azure Data Factory**: Go to the Azure portal and create an ADF instance.
    2. **Create Linked Service to ADLS Gen2**:
        - In ADF, create a **Linked Service** to the ADLS Gen2 account.
    3. **Create a Pipeline**:
        - Define **Copy Activities** in the ADF pipeline to move data from the raw-data container to the transformed-data container after transformation.
    4. **Add Transformation Logic**:
        - Add **Data Flow** in the ADF pipeline to clean and transform the data (e.g., format dates, remove null values).
    5. **Trigger the Pipeline**: Trigger the pipeline to run in batch mode to process the data.

---

## 2. Data Transformation (with Azure Databricks)

### Step 2.1: Set Up Azure Databricks

- **Objective**: Use Databricks for further transformations, analysis, and integration.
- **Steps**:
  1. In Azure Portal, create an Azure Databricks workspace.
  2. Create a **cluster** with appropriate configurations (e.g., Spark runtime).
  3. **Mount ADLS Gen2** in Azure Databricks:
     - Use Databricks to mount the ADLS Gen2 storage so that it can access the data.

python

Copy code

```
configs = {"fs.azure.account.auth.type": "OAuth", ...}  # Azure credentials
dbutils.fs.mount(
  source = "abfss://raw-data@<storage-account-name>.dfs.core.windows.net/",
  mount_point = "/mnt/raw-data",
  extra_configs = configs)
```

### Step 2.2: Load Raw Data into Databricks

- **Objective**: Read the raw data from ADLS Gen2 into Databricks for processing.
- **Steps**:
  1. Use Spark to load datasets like call_records.csv:

python

Copy code

```
call_records_df = spark.read.csv("/mnt/raw-data/call_records.csv", header=True,
inferSchema=True)
```

  2. Perform any necessary transformations (e.g., parsing date formats, dealing with null values).

### Step 2.3: Data Cleansing and Transformation

- **Objective**: Apply transformations to clean the data for further analysis and loading.
- **Steps**:
    1. **Transformation Logic**: Example of cleaning the call records dataset:

python

Copy code

```
from pyspark.sql.functions import to_timestamp, col

call_records_df = call_records_df.withColumn("CallStart", to_timestamp(col("CallStart")))

call_records_df = call_records_df.filter(col("CallDuration") > 0)  # Remove invalid call durations
```

2. Save the cleaned data to the transformed-data container in ADLS Gen2:

python

Copy code

```
call_records_df.write.csv("/mnt/transformed-data/cleaned_call_records", header=True, mode="overwrite")
```

---

**3. Data Aggregation and Modeling (Azure Databricks + Delta Lake)**

**Step 3.1: Create Delta Tables**

- **Objective**: Use Delta Lake to create tables for each dataset and enable faster querying.
- **Steps**:
    1. Convert the cleaned data into **Delta Lake format**:

python

Copy code

```
call_records_df.write.format("delta").save("/mnt/transformed-data/delta/call_records")
```

2. Create Delta tables for the other datasets:

python

Copy code

usage_df.write.format("delta").save("/mnt/transformed-data/delta/usage_data")

billing_df.write.format("delta").save("/mnt/transformed-data/delta/billing_data")

**Step 3.2: Build Aggregations in Databricks**

- **Objective**: Create views or aggregated tables for analysis.
- **Steps**:
    1. Build an aggregated table for **call records**:

python

Copy code

```
call_agg_df = call_records_df.groupBy("Caller").agg(
  sum("CallDuration").alias("TotalCallDuration"),
  count("Caller").alias("TotalCalls")
)
call_agg_df.write.format("delta").save("/mnt/transformed-data/delta/call_aggregates")
```

---

Configuring **Databricks Jobs** allows you to automate the execution of your data processing tasks, such as ETL jobs, machine learning workflows, or scheduled data transformations. Below is a step-by-step guide to configure and schedule Databricks Jobs.

**Steps to Configure Databricks Jobs**

**Step 1: Create a Databricks Cluster**

Before configuring a job, ensure that you have a Databricks cluster running, as jobs will be executed on the cluster.

1. **Log in to Azure Databricks**.
2. **Create a Cluster**:
    - Go to the "Clusters" tab and click on **Create Cluster**.
    - Configure the cluster with the required number of workers and libraries (e.g., Spark versions).
    - **Start the cluster** so it's ready for job execution.

**Step 2: Develop or Upload the Notebook**

Databricks Jobs can run notebooks or JAR/Scala code. The first step is to create or upload the notebook that will be run by the job.

1. **Create or Upload a Notebook**:
    - Go to the **Workspace** tab in Databricks.
    - Click **Create** > **Notebook**.
    - Write your data transformation, ETL logic, or analytics code in Python/Scala/SQL.
    - Save the notebook.

**Step 3: Create a Job**

Now that you have a notebook and a running cluster, you can configure the job.

1. **Navigate to Jobs in Databricks**:
    - In the left navigation pane, click on **Jobs**.
    - Click on **Create Job**.
2. **Configure Job Settings**:
    - **Name**: Give your job a meaningful name, such as Daily_ETL_Pipeline.
    - **Task Type**: Choose **Notebook** or **JAR** depending on what you want to execute.
3. **Set the Task**:
    - **Task Name**: Name your task (e.g., Process_Call_Records).
    - **Notebook Path**: Select the path to the notebook you created earlier by clicking **Browse** and navigating to the appropriate notebook in your workspace.
    - **Cluster**: Select the cluster that will run this job. You can either use an existing cluster or create a new one specifically for this job.
4. **Parameters (Optional)**:
    - If your notebook accepts parameters (e.g., date range), you can pass these as parameters during job creation.
    - Click on **Add Parameter** to specify any notebook parameters like start date, end date, or data paths.
5. **Configure Retry Policies**:

    - Set how many times the job should retry if it fails. You can specify a **retry policy** (e.g., retry 3 times after failure).

**Step 4: Configure Job Scheduling**

You can schedule your Databricks job to run at regular intervals, such as daily or weekly, using a cron-like scheduler.

1. **Enable Job Scheduling**:
    - In the "Schedule" section, click on **Add schedule**.
2. **Choose the Frequency**:
    - Select how frequently you want the job to run:
        - **Hourly**: Run the job every X hours.
        - **Daily**: Run the job every X days at a specific time.
        - **Weekly**: Specify the days of the week and the time for the job.
        - **Cron Expression**: For custom schedules, you can define cron expressions (e.g., 0 0 * * * for midnight daily).
3. **Time Zone**:
    - Select the time zone in which the schedule will run (e.g., UTC).

**Step 5: Set Job Notifications (Optional)**

You can configure notifications to receive alerts when the job succeeds, fails, or is retried.

1. **Notification Settings**:
    - In the "Notifications" section, you can configure email alerts.
    - Specify who should receive alerts on job status (e.g., completion, failure).
    - This is useful for monitoring long-running or critical jobs.

**Step 6: Review and Save the Job**

After configuring the task, schedule, and notifications, review the job configuration.

1. **Save the Job**:
    - Click **Create** to save the job configuration.
2. **View the Job Dashboard**:

    - Once created, the job appears on the **Jobs Dashboard** where you can monitor its execution, review logs, and see history.

**Step 7: Monitor Job Execution**

After creating the job, it can be monitored directly in the Databricks Jobs Dashboard:

1. **View Logs**:
    - Click on the job in the Jobs dashboard to view its run history.
    - You can check logs, including errors, warnings, and detailed execution statistics.
2. **Job Reruns**:
    - If a job fails, you can manually rerun it from the dashboard.
    - Alternatively, the job will automatically rerun if you set up retry logic in step 3.

---

**Advanced Configuration: Job Dependencies**

You can configure jobs to run in sequence or based on dependencies:

- **Multiple Tasks**: If your workflow has multiple steps (e.g., load data, process it, and save results), you can add **multiple tasks** to a job. Each task can depend on the previous task's success or failure.
- **Chained Jobs**: You can chain multiple notebooks or scripts together by adding dependent tasks in a single job configuration.

**8. Data Visualization and Reporting (Power BI/Fabric)**

**Step 8.1: Set Up Power BI to Connect to ADLS or Azure Databricks**

- **Objective**: Use Power BI to visualize the data and create dashboards.
- **Steps**:
    1. **Connect to Azure Databricks**: In Power BI, select the **Azure Databricks** connector and connect using your workspace URL.
    2. **Load Delta Tables**: Import Delta tables created in Databricks into Power BI.
    3. **Create Visualizations**: Design visualizations such as:
        - Call Duration Analysis (Total Calls, Call Duration over time).

        - Customer Usage (Data usage trends).
        - Billing Reports (Monthly billing summary).

**Step 8.2: Develop Real-Time Reports with Power BI/Fabric**

- **Objective**: Develop reports and dashboards for real-time analysis.
- **Steps**:
    1. Use Power BI to design dashboards for each key area (e.g., billing, usage, network performance).
    2. Schedule automatic data refresh to keep reports updated in real time.

---

### 9. Monitor, Schedule, and Automate the Pipeline (ADF + Databricks Jobs)

**Step 5.1: Schedule the ADF Pipeline**

- **Objective**: Set up scheduled runs for the ETL process.
- **Steps**:
    1. In Azure Data Factory, schedule the pipeline to run on a daily or weekly basis for batch processing.

**Step 5.2: Schedule Databricks Jobs**

- **Objective**: Automate data processing in Databricks.
- **Steps**:
    1. Use Databricks Jobs to schedule the Delta Lake aggregations and data transformations.

---

### 10. Conclusion

In this solution, the process follows an end-to-end data engineering pipeline:

1. **Ingestion**: Data is ingested into ADLS Gen2.
2. **Transformation**: Data is cleaned and transformed using Azure Data Factory and Azure Databricks.
3. **Storage**: Cleaned and transformed data is stored in Delta Lake for faster querying.
4. **Analysis**: Power BI is used for data visualization and reporting, connecting directly to Azure Databricks.

5. **Automation**: The entire pipeline is automated using ADF and Databricks Jobs for batch processing.

This ensures that telecom data from various sources like call records, billing, and network metrics is properly processed and available for reporting and analysis.

Incorporating **Apache Airflow** into the above scenario allows you to orchestrate and manage complex workflows, such as ETL pipelines, across multiple services (Azure ADLS, Azure Databricks, etc.). Airflow is a powerful workflow management tool that lets you define and schedule tasks in a directed acyclic graph (DAG). Here's a step-by-step guide on how to use Airflow to automate the ETL process outlined above, leveraging Azure components.

**Steps to Integrate Apache Airflow**

---

**Step 1: Set Up Apache Airflow**

1. **Install Airflow**:
    - Install Airflow locally or on a cloud VM (e.g., an Azure VM).
    - You can install Airflow using the following command (ensure you're using a virtual environment):

bash

Copy code

```
pip install apache-airflow
```

2. **Configure Airflow**:
    - Configure Airflow with the necessary settings in the airflow.cfg file, including setting up storage and the database backend (e.g., PostgreSQL or SQLite for development).
3. **Create a DAGs Directory**:
    - Create a directory called dags/ to store your workflows.
    - Airflow will automatically detect and load DAGs from this folder.

---

**Step 2: Define a DAG for the Workflow**

In this scenario, we’ll define an **Airflow DAG** to orchestrate the following steps:

1. Extract data from **Azure Data Lake Storage (ADLS Gen2)**.

2. Perform ETL in **Azure Databricks**.
3. Store the results back to **ADLS Gen2**.
4. Trigger data refresh in **Power BI**.

**Step 2.1: Create the Airflow DAG**

Here's an example of the Airflow DAG that will automate this workflow:

```python
from airflow import DAG

from airflow.providers.microsoft.azure.hooks.wasb import WasbHook

from airflow.providers.microsoft.azure.operators.databricks import
DatabricksSubmitRunOperator

from airflow.utils.dates import days_ago

from airflow.operators.python import PythonOperator

from datetime import timedelta


# Define default args for the DAG

default_args = {

    'owner': 'airflow',

    'depends_on_past': False,

    'start_date': days_ago(1),

    'email_on_failure': False,

    'email_on_retry': False,

    'retries': 2,

    'retry_delay': timedelta(minutes=5),

}


# Define the DAG

dag = DAG(

    'telecom_etl_workflow',
    default_args=default_args,
    description='A simple ETL workflow for telecom data',
    schedule_interval=timedelta(days=1),  # Run daily
)

# Step 1: Define Python function to check data availability in ADLS Gen2
def check_data_availability(**kwargs):
    wasb_hook = WasbHook(wasb_conn_id="azure_adls_gen2_connection")
    container = "raw-data"
    exists = wasb_hook.check_for_blob(container, "call_records.csv")
    if not exists:
        raise FileNotFoundError(f"call_records.csv not found in {container}.")

# Step 2: Define Python task to check for data availability in ADLS Gen2
check_data = PythonOperator(
    task_id='check_data_availability',
    python_callable=check_data_availability,
    dag=dag,
)

# Step 3: Submit a Databricks Job to process the data (ETL)
databricks_task = DatabricksSubmitRunOperator(
    task_id='databricks_etl_job',
    databricks_conn_id='databricks_default',
    new_cluster={
        'spark_version': '7.3.x-scala2.12',
        'num_workers': 2,
        'node_type_id': 'Standard_DS3_v2'
    },
    spark_jar_task={
        'main_class_name': 'com.example.telecom.ETLProcess',
        'parameters': ['call_records', 'cleaned_call_records']
    },
    dag=dag,
)

# Step 4: Task to trigger data refresh in Power BI
def trigger_powerbi_refresh(**kwargs):
    # Logic to refresh Power BI datasets via REST API
    print("Triggering Power BI data refresh")

powerbi_refresh = PythonOperator(
    task_id='trigger_powerbi_refresh',
    python_callable=trigger_powerbi_refresh,
    dag=dag,
)

# Define task dependencies
check_data >> databricks_task >> powerbi_refresh
```

---

**Step 3: Set Up Airflow Components**

**Step 3.1: Create Airflow Connections**

- **Azure Data Lake Storage (ADLS) Connection**:
    - In the Airflow UI, go to **Admin** > **Connections** and create a new connection:
        - **Connection ID**: azure_adls_gen2_connection

        - **Connection Type**: Azure Blob Storage
        - **Extra**: Provide your ADLS Gen2 credentials (account_name, account_key).
- **Databricks Connection**:
    - In the Airflow UI, create another connection for Databricks:
        - **Connection ID**: databricks_default
        - **Connection Type**: Databricks
        - **Host**: Provide your Azure Databricks workspace URL (e.g., https://<databricks-instance>.azuredatabricks.net).
        - **Token**: Use a Databricks access token for authentication.

**Step 3.2: Add Databricks Job and Notebook**

- **Databricks Notebook**: The DatabricksSubmitRunOperator triggers a Databricks job. You'll need to create a Databricks job or notebook that processes the data from ADLS and runs the ETL process.
- **Databricks Job Parameters**:
    - The DatabricksSubmitRunOperator can submit a job defined in Databricks or directly run a notebook or JAR.
    - Example for running a notebook:

python

```
databricks_task = DatabricksSubmitRunOperator(
  task_id='run_databricks_notebook',
  databricks_conn_id='databricks_default',
  notebook_task={
    'notebook_path': '/Workspace/telecom_etl',
    'base_parameters': {'input': 'call_records', 'output': 'cleaned_call_records'}
  },
  new_cluster={
    'spark_version': '7.3.x-scala2.12',
    'num_workers': 2,
  },
  dag=dag
)
```

---

**Step 4: Automating the Pipeline**

**Step 4.1: Scheduling**

- The schedule_interval argument in the DAG defines how often the ETL process runs. You can set it to @daily, @weekly, or use a custom cron schedule.

**Step 4.2: Monitoring the Workflow**

- Airflow provides a rich UI to monitor your DAGs, tasks, and their statuses. You can check:
    - Whether tasks are completed successfully.
    - Logs for each task (e.g., check_data_availability, databricks_etl_job).
    - Failures, retries, and detailed logs for debugging.

---

**Step 5: Triggering Data Refresh in Power BI**

You can automate the Power BI data refresh as the final step in your workflow. Here's how:

1. **Power BI REST API**: Use Power BI's REST API to refresh the datasets in your workspace.
    - Example API call:

python

Copy code

```
import requests


def trigger_powerbi_refresh():
    url = 'https://api.powerbi.com/v1.0/myorg/groups/{group_id}/datasets/{dataset_id}/refreshes'
    headers = {'Authorization': 'Bearer YOUR_ACCESS_TOKEN'}
```

```
response = requests.post(url, headers=headers)
if response.status_code == 202:
  print("Power BI refresh triggered successfully.")
else:
  print(f"Error triggering Power BI refresh: {response.content}")
```

2. **Airflow Task**: Add a PythonOperator to run this function as the final task in the DAG.

---

**Step 6: Running and Monitoring the DAG**

- Once the DAG is defined, it will appear in the Airflow UI under the **DAGs** tab.
- You can manually trigger the DAG, or let it run on the defined schedule.
- Use the **Graph View** in Airflow to visualize the workflow and monitor task statuses.

---

**Conclusion**

By integrating Airflow into this scenario:

- **Orchestration**: Airflow orchestrates the entire ETL pipeline, from data extraction in ADLS to transformation in Databricks, and finally refreshing reports in Power BI.
- **Automation**: The workflow is fully automated, and Airflow schedules and retries tasks based on the defined configuration.
- **Monitoring**: Airflow provides an easy-to-use interface to monitor task execution, logs, and retries, ensuring that complex workflows run smoothly.