

1. Introduction

1.1 Project Overview

The Malia Project is designed to streamline and automate data extraction, transformation, storage, and analysis for business decision-making. The project integrates various data sources into a centralized system, ensuring data consistency and availability for real-time reporting and advanced analytics.

The objective of this project is to address the current challenges of manual data entry and reconciliation between website orders, ERP system transactions, and shipping company collections in an effort to increase Malia's data accuracy and integrity.

1.2 Purpose

The purpose of this technical documentation is to provide detailed information about the Malia Project's **architecture, data sources, data pipeline, storage, analysis, reporting, and technical setup**. It serves as a comprehensive guide for developers, data engineers, analysts, and stakeholders involved in the project.

1.3 Audience

This documentation is intended for:

- **Developers:** To understand the system architecture, code structure, and deployment process.
- **Data Engineers:** To gain insights into the data pipeline, ETL processes, and data storage.
- **Data Analysts:** To learn about the available data, KPIs, and analysis techniques.
- **Project Stakeholders:** To get an overview of the project's capabilities and benefits.

2. Setting Up the Development Environment

2.1 Prerequisites:

Hardware Requirements

- Minimum 8 GB RAM (16 GB recommended)
- At least 50 GB of free disk space
- Multi-core CPU (Quad-core recommended)

Software Requirements

- Operating System: Windows 10/11, macOS, or Linux
- Python 3.8 or later - Current version: 3.11.7
- Docker version 26.1.1 and Docker Compose v2.27.0-desktop.2
- MySQL
- Power BI Desktop (for Windows)

2.2 Installation

Python and Libraries

1. Install Python:

- Download and install the latest version of Python from the [official website](#).
- Verify the installation:

```
python --version
```

2. **Install Libraries:** Python libraries will later be automatically installed later in the project environment through a `requirements.txt` file.

Docker and Docker Compose

Step by Step Guide on Installing Airflow on Docker Desktop:

1. Create a Project Folder on the Machine

- Open VsCode and create a new project folder. Current Name: `AIRFLOW_DOCKER_1`
- Open the terminal in VsCode and ensure that the directory is changed to the folder's directory
- The followinf is the structure of the `AIRFLOW_DOCKER_1` folder: (instructions on when to create the folders will be given later)

```
AIRFLOW_DOCKER_1/
├── config/
├── dags/
├── logs/
├── plugins/
├── scripts/
└── technical_documentation/
```

2. Install Docker and Docker Compose

- Download and install Docker from the [official website](#).
- Docker Compose is included with Docker Desktop for Windows and macOS.
- For Linux, follow the instructions on the [official Docker Compose website](#).

3. Lunch the Newly Installed Docker Desktop Application

- Note that it may take a couple of minutes to initialize based on the system's performance.

4. Verify Docker Installation in the VsCode Terminal

```
docker --version  
docker-compose --version
```

5. Download the Airflow Configuration File for Docker

- install the .yaml file from the [Aiflow official website](#) and save it as a .yaml file in the project directory. (**not recommended**)
- or execute the following command in the terminal to fetch the .yaml file (**not recommended**)

```
curl -Lfo 'https://airflow.apache.org/docs/apache-airflow/2.9.2/docker-compose.yaml'
```

- or download the modified docker file configured for this exact project using [the following link](#) and save it in the project directory (**recommended**)
- If modified file is downloaded: change the docker image back to default value for it to run properly (it will be modified to custom value at a later stage). From:

```
image: ${AIRFLOW_IMAGE_NAME:-extending_airflow:latest}
```

To:

```
image: ${AIRFLOW_IMAGE_NAME:-apache/airflow:2.0.1}
```

- If the default .yaml file was downloaded, there are the key Modification done to the .yaml file:

- 1. Image Customization:** Change the Docker image to a custom image `extending_airflow:latest` for adding custom dependencies or configurations.

```
image: ${AIRFLOW_IMAGE_NAME:-extending_airflow:latest}
```

- 2. Executor Type:** Change the executor type from CeleryExecutor to LocalExecutor for simplicity in a local development environment.

```
AIRFLOW__CORE__EXECUTOR: LocalExecutor
```

- 3. Loading Examples:** Disable loading of example DAGs by setting `AIRFLOW__CORE__LOAD_EXAMPLES` to `false`

```
AIRFLOW__CORE__LOAD_EXAMPLES: 'false'
```

4. **Volume Configuration:** Added additional volumes for `scripts`, `data`, and `requirements.txt` to support custom scripts, data storage, and requirements management.

```
volumes:
- ${AIRFLOW_PROJ_DIR:-.}/dags:/opt/airflow/dags
- ${AIRFLOW_PROJ_DIR:-.}/logs:/opt/airflow/logs
- ${AIRFLOW_PROJ_DIR:-.}/config:/opt/airflow/config
- ${AIRFLOW_PROJ_DIR:-.}/plugins:/opt/airflow/plugins
- ${AIRFLOW_PROJ_DIR:-.}/scripts:/opt/airflow/scripts
- ${AIRFLOW_PROJ_DIR:-.}/data:/opt/airflow/data
- ${AIRFLOW_PROJ_DIR:-.}/requirements.txt:/requirements.txt
```

5. **Removed Services:** Removed the Redis and Airflow worker services as well as the Flower monitoring tool, which are not required for the LocalExecutor setup.
6. **Dependencies:** Removed dependency on Redis, as it is not needed for the LocalExecutor configuration

```
depends_on:
  &airflow-common-depends-on
postgres:
  condition: service_healthy
```

7. **Add Necessary configuration for Email Notification:** Make sure that the `environment` section of the `docker-compose.yaml` file is as follows (this is for gmail configuration):

```
environment:
&airflow-common-env
AIRFLOW__CORE__EXECUTOR: LocalExecutor
AIRFLOW__DATABASE__SQL_ALCHEMY_CONN:
postgresql+psycopg2://airflow:airflow@postgres/airflow
AIRFLOW__CORE__FERNET_KEY: ''
AIRFLOW__CORE__DAGS_ARE_PAUSED_AT_CREATION: 'true'
AIRFLOW__CORE__LOAD_EXAMPLES: 'false'
AIRFLOW__API__AUTH_BACKENDS:
'airflow.api.auth.backend.basic_auth,airflow.api.auth.backend.session'
# yamllint disable rule:line-length
# smtp email notification config
AIRFLOW__EMAIL__EMAIL_BACKEND: airflow.utils.email.send_email_smtp
AIRFLOW__SMTP__SMTP_HOST: smtp.gmail.com
AIRFLOW__SMTP__STARTTLS: True
AIRFLOW__SMTP__SMTP_SSL: False
```

```
AIRFLOW__SMTP__SMTP_USER: your gmail  
AIRFLOW__SMTP__SMTP_PASSWORD: your password  
AIRFLOW__SMTP__SMTP_PORT: 587  
AIRFLOW__SMTP__SMTP_MAIL_FROM:your gmail  
AIRFLOW__SMTP__SMTP_DEBUG: True
```

6. Create dags, logs, and plugins folders in the project folder:

- The folders can be manually added in VsCode or the following command can be run in the terminal

```
mkdir dags logs plugins
```

7. Initialize the Airflow Database in Docker:

- run rhe following command in the terminal:

```
docker-compose up airflow-init
```

Running and Stopping Airflow past the Installation

- **To start all airflow services:**

```
docker-compose up -d
```

The open the browser and access the `localhost:8080` port of your machine to access the airflow webserver usinf 'airflow' as both the unsername and password

- **To cut the webserver:**

```
docker-compose down
```

Installing Python Dependencies to the Airflow Docker Container

1. Create the requirements.txt File in the Project Folder

```
apache-airflow==2.9.2  
pandas  
numpy  
sqlalchemy  
pymysql  
openpyxl  
apache-airflow-providers-mysql
```

```
mlxtend  
matplotlib  
seaborn  
squarify  
fpdf  
requests
```

2. Extend the Airflow Image to Install all the Dependencies in the `requirements.txt` File:

- Create a docker file in the the project root folder: `Dockerfile` and include this code:

```
FROM apache/airflow:2.9.2  
COPY requirements.txt /requirements.txt  
RUN pip install --upgrade pip  
RUN pip install --no-cache-dir -r /requirements.txt
```

3. Build the Extended Image:

- Run the following command in the terminal:

```
docker build . --tag extending_airflow:latest
```

- Change the Docker image to a custom image `extending_airflow:latest` for adding custom dependencies or configurations.

```
image: ${AIRFLOW_IMAGE_NAME:-extending_airflow:latest}
```

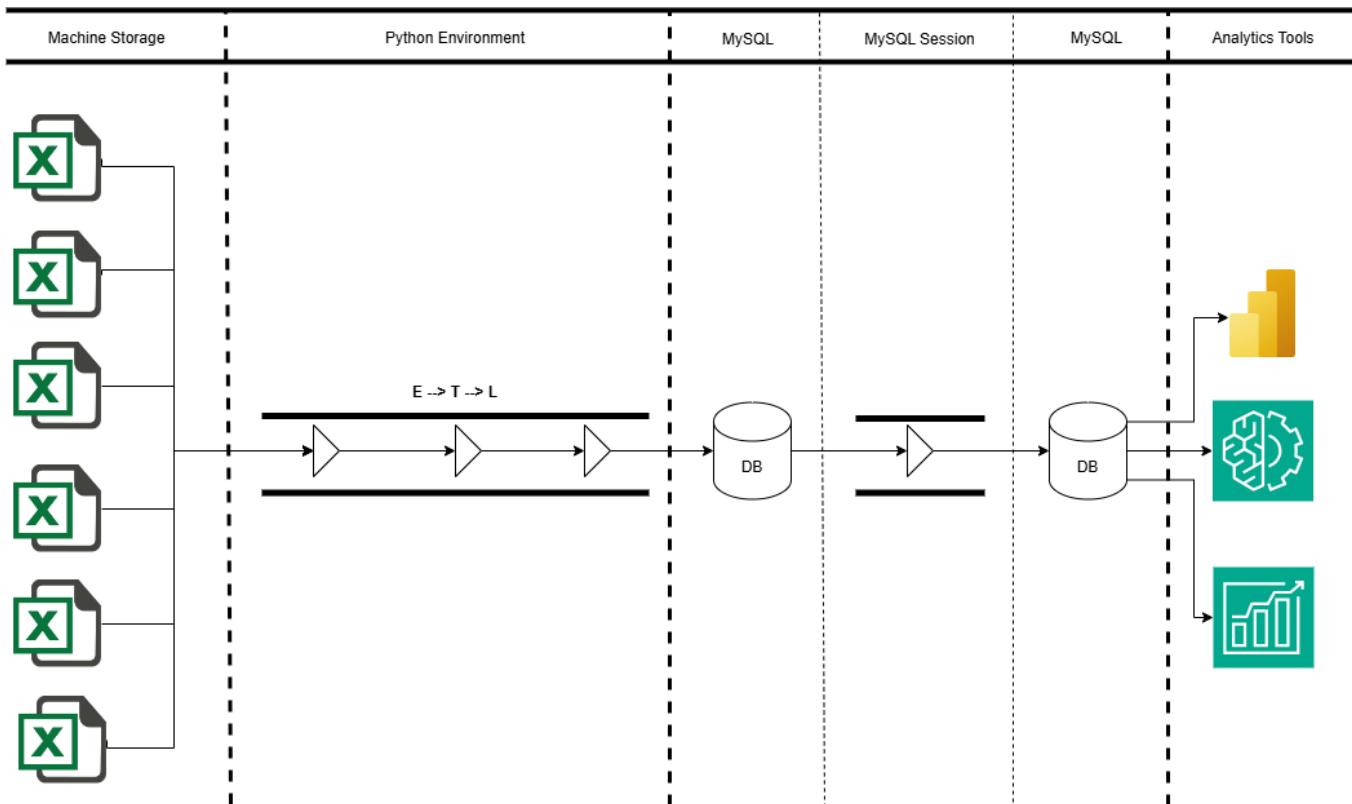
4. Rebuild the Aiflow Webserver and Scheduler Services since the Image Name was Modified:

```
docker-compose up -d --no-deps --build airflow-webserver airflow-scheduler
```

5. Important Note: changing the `requirements.txt` file will require to rebuild the airflow webserver and scheduler containers:

```
docker-compose up -d --no-deps --build airflow-webserver airflow-scheduler
```

3. Data Flow



The data flow in the Malia Project involves several stages, each responsible for handling different aspects of data processing. Here is a detailed description of each stage:

3.1 Data Sources:

1. Website ECOM Data

Description: This sheet contains data related to e-commerce orders processed through the website.

- **Order Number:** Unique identifier for each order.
- **Shipper Name:** Name of the shipping company.
- **Created At:** Date and time when the order was created.
- **Delivered At:** Date and time when the order was delivered.
- **Billing Type:** Type of billing used for the order.
- **Amount:** Total amount for the order.
- **Currency:** Currency in which the amount is specified.
- **Country:** Country to which the order was shipped.
- **AWB:** Air Waybill number for tracking the shipment.

2. Shipped & Collected - Aramex

Description: This sheet contains data about shipments that were shipped and collected by Aramex.

- **ShprNo:** Shipper number.
- **HAWB:** House Air Waybill number.
- **Delivery Date:** Date when the shipment was delivered.
- **CODAmount:** Cash on Delivery amount.
- **CODCurrency:** Currency of the COD amount.
- **O_CODAmount:** Original COD amount.

- **Aramex_Inv_date:** Invoice date from Aramex.
- **TOKENNO:** Token number for tracking.

3. Shipped & Collected - Cosmaline

Description: This sheet contains data about shipments that were shipped and collected by Cosmaline.

- **Driver_Delivery_date:** Date when the driver delivered the shipment.
- **OrderNo:** Order number associated with the shipment.
- **Amount:** Amount associated with the order.
- **Currency:** Currency of the amount.

4. Collected - Credit Card

Description: This sheet contains data about payments collected through credit cards.

- **Value_date:** Date when the payment was processed.
- **Narrative:** Description or details of the transaction.
- **Amount:** Amount collected.
- **Currency:** Currency of the amount.

5. ERP-Oracle Collection

Description: This sheet contains data related to collections recorded in the ERP-Oracle system.

- **RECEIPT_NUMBER:** Unique identifier for each receipt.
- **CURRENCY_CODE:** Currency code of the receipt.
- **EXCHANGE_RATE:** Exchange rate applied to the receipt.
- **RECEIPT_DATE:** Date when the receipt was recorded.
- **CUSTOMER_NUMBER:** Unique identifier for the customer.
- **CUSTOMER_NAME:** Name of the customer.
- **RECEIPT_CLASS:** Classification of the receipt.
- **RECEIPT_AMOUNT:** Amount recorded in the receipt.
- **COMMENTS:** Additional comments or details about the receipt.

6. Oracle Data

Description: This sheet contains detailed order data from the Oracle system.

- **OPERATING_UNIT_NAME:** Name of the operating unit.
- **ORACLE_REFERENCE_ORDER_NUMBER:** Oracle reference order number.
- **ECOM_REFERENCE_ORDER_NUMBER:** E-commerce reference order number.
- **ORDER_TYPE:** Type of the order.
- **ORDER_TYPE_NAME:** Name of the order type.
- **ORDERED_ITEM:** Item that was ordered.
- **ORDERED_QUANTITY:** Quantity of the item ordered.
- **PRICING_QUANTITY_UOM:** Unit of measure for the pricing quantity.
- **UNIT_SELLING_PRICE (after discount without vat):** Selling price per unit after discount and without VAT.

- **UNIT_LIST_PRICE (original price before discount without vat)**: List price per unit before discount and without VAT.
- **ORDERED_DATE**: Date when the order was placed.
- **CUSTOMER_NAME**: Name of the customer.
- **TAX_CODE**: Tax code applicable to the order.

7. Daily Rate

Description: This sheet contains the daily exchange rates.

- **Date**: Date for the exchange rate.
- **Rate**: Exchange rate for the specified date.

3.2 Data Flow Components

1 Data Ingestion

- **Sources**: Data is collected from various sources such as e-commerce platforms, shipping data systems, ERP systems, all of them stored locally as **.txt**, **.csv**, or **.xlsx** files.
- **Process**: Python scripts and Airflow DAGs are used to automate the extraction of data from these sources. The extracted data is saved as separate **.csv** files for initial handling. The data is stored as **.csv** files to avoid memory errors due to handling multiple data sources using the in-build system RAM.
- **Technologies Used**: Python, Apache Airflow.

2 Data Processing

- **Transformation**: The ingested data is loaded into a Python environment where it is cleaned and transformed. This involves standardizing formats, handling missing values, and performing necessary calculations.
- **Output**: The transformed data is saved as **.parquet** files for optimized storage and retrieval.
- **Technologies Used**: Pandas, Apache Airflow.

3 Data Storage

- **Loading**: The **.parquet** files are read and loaded into a MySQL database. SQLAlchemy is used to interact with the database and ensure data integrity.
- **Storage**: The data is stored in a structured format, making it ready for analysis and reporting.
- **Technologies Used**: MySQL, SQLAlchemy, Apache Airflow.

4 Data Reconciliation

- **Purpose**: To ensure that the same transactions present in multiple tables are reconciled, automated SQL scripts are run on the MySQL database.
- **Process**: SQL scripts identify and resolve discrepancies between different data tables to ensure data accuracy.
- **Technologies Used**: SQL, Apache Airflow.

5 Data Cleanup

- **Purpose:** To maintain data integrity and avoid clutter, all temporary tables created during the processing are deleted.
- **Process:** Automated SQL scripts delete temporary tables at the end of each processing cycle.
- **Technologies Used:** SQL, Apache Airflow.

6 Reporting

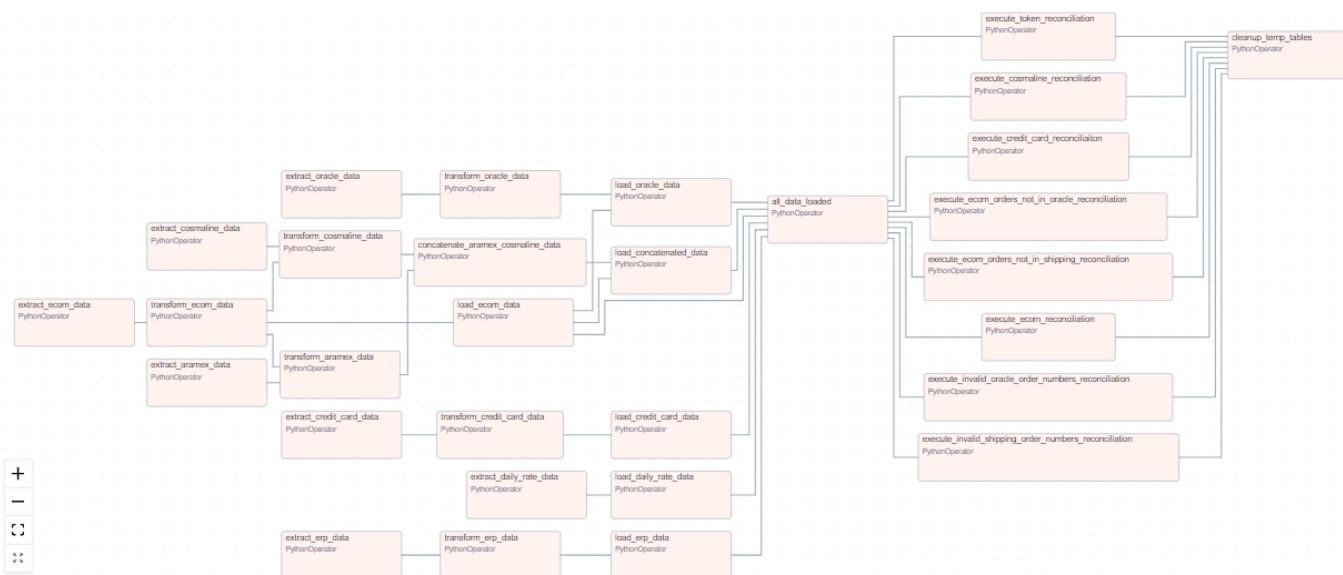
- **Visualization:** Power BI dashboards and reports are generated to provide actionable insights and real-time monitoring of data integrity.
- **Data Source:** The MySQL database serves as the source for the reporting layer.
- **Technologies Used:** Power BI.

7 Analytics

- **Analysis:** Tools and scripts are used to perform data analysis and generate insights.
- **Output:** Results from the analysis can be integrated into the reporting layer for visualization or used for further decision-making.
- **Technologies Used:** Python, Jupyter Notebooks, Machine Learning libraries.

4. Back end System Architecture and Development

4.1 Architechture Diagram



4.2 Data Architecture Overview and Key Componenets (ETL process):

1. Data Extraction:

- The purpose of this step is to extract the data from the different datasources and transfer them to a python environment for transformation.

2. Data Transformation:

- The purpose of this step is to perform transformations operations on the data in the python environment, making it ready for database storage.

3. Data Load:

- The purpose of this step is to load the data from the python environment into the relational database.

4. Data Reconciliation:

- The purpose of this step is to perform reconciliation measures on the database, ensuring Malia's data integrity

5. DAG Implementation:

- The purpose of this step is to implement the ETL + reconciliation pipeline into an automated DAG script that will call the other ETL scripts to run their functions.

4.3 Technical Changes to the Development Environment:

1. Create a `scripts` folder to hold the python and sql scripts to be ran by the system:

```
mkdir scripts
```

2. Create a `data` folder to hold the initial data source .xlsx file:

```
mkdir data
```

3. Create a `tests` folder to hold the unit and integration tests that will insure the reliability of the system

```
mkdir tests
```

4.4 The Extraction Process

To manage the extraction process, create a Python file named `extract.py` in the `scripts` folder to hold the functions related to extracting data from the data sources.

Function to Extract Data from Excel

The following function, `read_data_from_excel`, extracts data from an Excel file into a Pandas DataFrame. This function reads the data from a specified sheet within the Excel file and handles any potential errors that may occur during the process.

- **Purpose:** Reads data from an Excel file into a Pandas DataFrame.
- **Parameters:**
 - `filepath` (str): The path to the Excel file.
 - `sheet_name` (str or int): The sheet name or index to read data from. Defaults to 'Website ECOM Data'.
- **Returns:** `pd.DataFrame`

- **Explanation:** The function handles reading the data and includes error handling to manage potential issues during the read process.

```
import pandas as pd
import numpy as np

def read_data_from_excel(filepath, sheet_name='Website ECOM Data'):
    """
    Reads data from an Excel file into a Pandas DataFrame.

    Parameters:
        filepath (str): The path to the Excel file.
        sheet_name (str or int): The sheet name or index to read data from.
        Defaults to the first sheet.

    Returns:
        pd.DataFrame: The data read from the Excel file.
    """
    try:
        df = pd.read_excel(filepath, sheet_name=sheet_name)
        #print("Data read successfully from Excel.")
        return df
    except Exception as e:
        print(f"An error occurred while reading the data: {e}")
        return None
```

4.5 The Transformation Process

To manage the transformation process, create a Python file named `transform.py` in the scripts folder to hold all the functions that will perform data transformation on the different dataframes. In addition to the `transform.py` file, create a `transform_pipelines.py` files that will hold the transformation pipelines for each data source.

For this process, each data source (dataframe) will have its own transformation pipeline, consisting of multiple functions. Below, we will describe the transformation pipeline for each data source and provide the functions involved.

```
import pandas as pd
import numpy as np
```

4.5.1 Website ECOM Data Transformation

Transformation Pipeline:

The following pipeline is used to transform the Website ECOM Data:

```
def data_preprocessing_pipeline_ECOM_Data(df):
    """
    Process a DataFrame through a series of cleaning and normalization steps.

    Parameters:
        df (pd.DataFrame): The DataFrame to be processed.

    Returns:
        pd.DataFrame: The processed DataFrame.
    """
    # Convert Data types
    df = convert_data_types_ECOM_Data(df)

    # Normalize some columns to lower case
    df = lower_columns(df, ['Shipper Name', 'Billing Type'])

    # Normalize some columns to upper case
    df = uppercase_columns(df, ['Currency', 'Country'])

    # Remove rows with missing values
    df = remove_missing_values(df)

    # Remove duplicate rows, keeping the first occurrence
    df = remove_duplicates(df)

    # Change Column Names
    df = rename_columns_for_website_ecom_data(df)

    return df
```

Function 1: Convert Data Types

This function ensures that the data types of specific columns are consistent and correct. It converts columns like `Created At` and `Delivered At` to datetime, `Amount` to float, and others to object types as needed for further processing and analysis.

```
def convert_data_types_ECOM_Data(df):
    """
    Converts the data types of specified columns in the DataFrame.

    Parameters:
        df (pd.DataFrame): The DataFrame whose columns will have their data types converted.

    Returns:
        pd.DataFrame: The DataFrame with converted data types.
    """
    # Define a dictionary mapping column names to desired data types
    dtype_dict = {
        'Order Number': 'object',
```

```

        'Shipper Name': 'object',
        'Created At': 'datetime64[ns]',
        'Delivered At': 'datetime64[ns]',
        'Billing Type': 'object',
        'Amount': 'float64',
        'Currency': 'object',
        'Country': 'object',
        'AWB': 'object'
    }
df = df.astype(dtype_dict)
return df

```

Function 2: Normalize Columns to Lower Case

This function processes specified columns to ensure consistency in text data by converting them to lowercase and capitalizing the first letter of each word. This is especially useful for columns like **Shipper Name** and **Billing Type** to maintain uniformity.

```

def lower_columns(dataframe, columns):
    """
    Strips, lowers, and capitalizes the first letter of each word in the specified
    columns of a DataFrame.

    Parameters:
        dataframe (pd.DataFrame): The DataFrame to process.
        columns (list): List of column names to normalize.

    Returns:
        pd.DataFrame: The DataFrame with normalized columns.
    """
    for column in columns:
        if (column in dataframe.columns) and (dataframe[column].dtype == object):
            dataframe[column] =
                dataframe[column].str.strip().str.lower().str.capitalize()
        else:
            print(f"Warning: {column} does not exist in the DataFrame or is not of
type object.")
    return dataframe

```

Function 3: Normalize Columns to Upper Case

This function processes specified columns by converting text data to uppercase, which helps in maintaining consistency for categorical data like **Currency** and **Country**.

```

def uppercase_columns(dataframe, columns):
    """
    Strips whitespace and converts the text to upper case for specified columns in
    a DataFrame.

```

```
Parameters:  
    dataframe (pd.DataFrame): The DataFrame to process.  
    columns (list): List of column names to convert to upper case.  
  
Returns:  
    pd.DataFrame: The DataFrame with the specified columns converted to upper case.  
    """  
    for column in columns:  
        if (column in dataframe.columns) and (dataframe[column].dtype == object):  
            dataframe[column] = dataframe[column].str.strip().str.upper()  
        else:  
            print(f"Warning: {column} does not exist in the DataFrame or is not of type object.")  
    return dataframe
```

Function 4: Remove Missing Values

This function cleans the DataFrame by removing any rows that contain missing values. It helps ensure the integrity and completeness of the data before further analysis.

```
def remove_missing_values(df):  
    """  
    Removes rows with any missing values from the DataFrame without printing any output.  
  
    Parameters:  
        df (pd.DataFrame): The DataFrame from which to remove missing values.  
  
    Returns:  
        pd.DataFrame: A new DataFrame with missing values removed.  
    """  
    return df.dropna()
```

Function 5: Remove Duplicate Rows

This function eliminates duplicate rows from the DataFrame, keeping only the first occurrence. This step is crucial for ensuring that each record is unique and avoiding redundancy in the data.

```
def remove_duplicates(df):  
    """  
    Removes duplicate rows from the DataFrame, keeping the first occurrence of each duplicate.  
  
    Parameters:  
        df (pd.DataFrame): The DataFrame from which to remove duplicates.
```

```

Returns:
pd.DataFrame: A new DataFrame with duplicates removed, keeping the first
occurrence.
"""
return df.drop_duplicates()

```

Function 6: Rename Columns

This function standardizes the column names by renaming them according to a predefined mapping. It ensures that column names are consistent, making the DataFrame easier to work with and understand.

```

def rename_columns_for_website_ecom_data(df):
    """
    Renames columns of a specific DataFrame based on a predefined dictionary
    mapping of old names to new names.
    This function is tailored for the 'Website_ECOM_Data' DataFrame.

    Parameters:
    df (pd.DataFrame): The DataFrame whose columns will be renamed.

    Returns:
    pd.DataFrame: The DataFrame with renamed columns.
    """
    # Encapsulated dictionary specific to the Website_ECOM_Data schema
    column_rename_dict = {
        'Order Number': 'order_number',
        'Shipper Name': 'shipper_name',
        'Created At': 'created_at',
        'Delivered At': 'delivered_at',
        'Billing Type': 'billing_type',
        'Amount': 'amount',
        'Currency': 'currency',
        'Country': 'country',
        'AWB': 'AWB'
    }
    df = df.rename(columns=column_rename_dict)
    return df

```

4.5.2 Shipping Aramex Data Transformation

Transformation Pipeline:

The following pipeline is used to transform the Shipping Aramex Data:

```

def data_preprocessing_pipeline_Aramex_Data(df, ECOM_df):
    """
    Process a DataFrame through a series of cleaning and normalization steps.
    """

```

```

Parameters:
    df (pd.DataFrame): The DataFrame to be processed.

Returns:
    pd.DataFrame: The processed DataFrame.
"""

# Convert Data types
df = convert_data_types_Aramex_Data(df)

# Rename the HAWB columns
df = rename_columns_for_website_Aramex_data(df)

# Normalize some columns to upper case
df = uppercase_columns(df, ['CODCurrency'])

# Update CODCurrency
df = update_cod_currency_Aramex_data(df)

# Cleaning the TOKEN column to have the same token per day
df = clean_tokens_by_date_Aramex_Data(df, 'Aramex_Inv_date', 'TOKENNO')

# Adding the order_number column from the ECOM data
df = add_order_number_Aramex_Data(df, ECOM_df)

# Converting data types:
df = convert_data_types_aramecosmoline(df)

return df

```

Function 1: Convert Data Types

This function ensures that the data types of specific columns are consistent and correct. It converts columns like `Delivery_Date` and `Aramex_Inv_date` to datetime, `CODAmount` and `O_CODAmount` to float, and others to object types as needed for further processing and analysis.

```

def convert_data_types_Aramex_Data(df):
    """
    Converts the data types of specified columns in the DataFrame.

    Parameters:
        df (pd.DataFrame): The DataFrame whose columns will have their data types converted.

    Returns:
        pd.DataFrame: The DataFrame with converted data types.
"""

    # Define a dictionary mapping column names to desired data types
    dtype_dict = {
        'ShprNo': 'object',
        'HAWB': 'object',
        'Delivery_Date': 'datetime64[ns]',

```

```

        'CODAmount': 'float64',
        'O_CODAmount': 'float64',
        'Aramex_Inv_date': 'datetime64[ns]',
        'TOKENNO': 'object'
    }

# Convert data types according to the dictionary
for column, dtype in dtype_dict.items():
    if column in df.columns:
        try:
            df[column] = df[column].astype(dtype)
        except ValueError:
            print(f"Conversion error: Could not convert {column} to {dtype}")
    else:
        print(f"Warning: {column} not in DataFrame")

return df

```

Function 2: Rename Columns

This function standardizes the column names by renaming them according to a predefined mapping. It ensures that column names are consistent, making the DataFrame easier to work with and understand.

```

def rename_columns_for_website_Aramex_data(df):
    """
    Renames columns of a specific DataFrame based on a predefined dictionary
    mapping of old names to new names.
    This function is tailored for the 'Website_ECOM_Data' DataFrame.

    Parameters:
        df (pd.DataFrame): The DataFrame whose columns will be renamed.

    Returns:
        pd.DataFrame: The DataFrame with renamed columns.
    """
    # Encapsulated dictionary specific to the Website_ECOM_Data schema
    column_rename_dict = {
        'HAWB': 'AWB',
    }

    return df.rename(columns=column_rename_dict, inplace=False)

```

Function 3: Normalize Columns to Upper Case

This function processes specified columns by converting text data to uppercase, which helps in maintaining consistency for categorical data like CODCurrency.

```

def uppercase_columns(dataframe, columns):
    """
    Strips whitespace and converts the text to upper case for specified columns in
    a DataFrame.

    Parameters:
        dataframe (pd.DataFrame): The DataFrame to process.
        columns (list): List of column names to convert to upper case.

    Returns:
        pd.DataFrame: The DataFrame with the specified columns converted to upper
        case.
    """
    for column in columns:
        if column in dataframe.columns:
            dataframe[column] = dataframe[column].str.strip().str.upper()
        else:
            print(f"Warning: {column} does not exist in the DataFrame.")
    return dataframe

```

Function 4: Upper CODCurrency

This function updates the `CODCurrency` column based on the `CODAmount` column. If `CODAmount` is greater than 0, it sets `CODCurrency` to 'LBP'.

```

def update_cod_currency_Aramex_data(df):
    """
    Updates the CODCurrency column based on the CODAmount column.
    If CODAmount is greater than 0, sets CODCurrency to 'LBP'.

    Parameters:
        df (pd.DataFrame): The DataFrame containing CODAmount and CODCurrency
        columns.

    Returns:
        pd.DataFrame: The DataFrame with updated CODCurrency column.
    """
    df.loc[df['CODAmount'] > 0, 'CODCurrency'] = 'LBP'
    return df

```

Function 5: Clean Tokens by Date

This function ensures that each unique date has the same token, replacing discrepancies with the mode token for that date.

```

def clean_tokens_by_date_Aramex_Data(df, date_column, token_column):
    """

```

Ensures each unique date has the same token, replacing discrepancies with the mode token for that date.

Parameters:

`df (pd.DataFrame)`: The DataFrame containing the data.
`date_column (str)`: The name of the column containing the dates.
`token_column (str)`: The name of the column containing the tokens.

Returns:

`pd.DataFrame`: The DataFrame with tokens cleaned.

"""

`def replace_with_mode(group):`

`mode_token = group[token_column].mode()[0]` # Calculate the mode token for the group
`group[token_column] = mode_token` # Replace all tokens in the group with the mode token
`return group`

Group by the date column and apply the replacement function to each group

`df = df.groupby(date_column).apply(replace_with_mode).reset_index(drop=True)`

`return df`

Function 6: Add Order Number

This function adds the `order_number` column from Website ECOM Data to the Shipping Aramex Data based on matching `AWB` values.

```
def add_order_number_Aramex_Data(df, ECOM_df):
    """
    Adds the order_number column from Website_ECOM_Data to
    ShippedandCollected_Aramex
    based on matching AWB values.

    Parameters:
        ShippedandCollected_Aramex (pd.DataFrame): DataFrame containing shipment
        and collection data.
        Website_ECOM_Data (pd.DataFrame): DataFrame containing e-commerce data
        with order numbers.

    Returns:
        pd.DataFrame: Updated DataFrame with the order_number column added.
    """
    # Ensure AWB columns in both DataFrames have the same type
    df['AWB'] = df['AWB'].astype(str)
    ECOM_df['AWB'] = ECOM_df['AWB'].astype(str)

    # Merge DataFrames on the 'AWB' column, only adding the 'order_number' column
    # to ShippedandCollected_Aramex
    updated_df = pd.merge(
        df,
        ECOM_df[['AWB', 'order_number']],
        on='AWB',
        how='left')
```

```

        on='AWB',
        how='left'
    )
    return updated_df

```

Function 7: Convert Data Types for Aramex Cosmaline

This function converts the data types of specified columns in the Aramex DataFrame to match the database schema.

```

def convert_data_types_aramex_cosmaline(df):
    """
    Converts the data types of specified columns in the Aramex DataFrame to match
    the database schema.

    Parameters:
        df (pd.DataFrame): The DataFrame whose columns will have their data types
        converted.

    Returns:
        pd.DataFrame: The DataFrame with converted data types.
    """
    dtype_dict = {
        'AWB': 'str', # Will be converted to varchar(100) in the database
        'ShprNo': 'object', # Will be converted to varchar(255) in the database
        'Delivery_Date': 'datetime64[ns]', # Will be converted to datetime in the
        database
        'CODAmount': 'float32', # Will be converted to float in the database
        'CODCurrency': 'object', # Will be converted to varchar(100) in the
        database
        'O_CODAmount': 'float32', # Will be converted to float in the database
        'Aramex_Inv_date': 'datetime64[ns]', # Will be converted to datetime in
        the database
        'TOKENNO': 'object', # Will be converted to varchar(255) in the database
        'order_number': 'object' # Will be converted to varchar(255) in the
        database
    }

    for column, dtype in dtype_dict.items():
        if column in df.columns:
            try:
                df[column] = df[column].astype(dtype)
            except ValueError:
                print(f"Conversion error: Could not convert {column} to {dtype}")
        else:
            print(f"Warning: {column} not in DataFrame")

    return df

```

4.5.3 Shipping Cosmaline Data Transformation

Transformation Pipeline:

The following pipeline is used to transform the Shipping Cosmaline Data:

```
def data_preprocessing_pipeline_Cosmaline_Data(df, ECOM_df):
    """
    Process a DataFrame through a series of cleaning and normalization steps.

    Parameters:
        df (pd.DataFrame): The DataFrame to be processed.

    Returns:
        pd.DataFrame: The processed DataFrame.
    """
    # Convert Data types
    df = convert_data_types_Cosmaline_Data(df)

    # Normalize some columns to upper case
    df = uppercase_columns(df, ['Currency'])

    # Renaming the columns:
    df = rename_columns_for_Cosmaline_data(df)

    # Adding CODAmount:
    df = add_cod_amount_column_Cosmaline_data(df)

    # Adding new columns:
    df = add_new_columns_Cosmaline_Data(df)

    # Adding the AWB from ECOM data:
    df = add_AWB_Cosmaline_Data(df, ECOM_df)

    # Converting data types:
    df = convert_data_types_aramelex_cosmaline(df)

    return df
```

Function 1: Convert Data Types:

This function ensures that the data types of specific columns are consistent and correct. It converts columns like `Driver_Delivery_date` to datetime, `Amount` to float, and others to object types as needed for further processing and analysis.

```
def convert_data_types_Cosmaline_Data(df):
    """
    Converts the data types of specified columns in the DataFrame.

    Parameters:
        df (pd.DataFrame): The DataFrame whose columns will have their data types converted.
    """
    # Convert specific columns to the correct data type
    df['Driver_Delivery_date'] = pd.to_datetime(df['Driver_Delivery_date'])
    df['Amount'] = df['Amount'].astype(float)
    # ... (convert other columns as needed)

    return df
```

```

Returns:
    pd.DataFrame: The DataFrame with converted data types.

"""

# Define a dictionary mapping column names to desired data types
dtype_dict = {
    'Currency': 'object',
    'Driver_Delivery_date': 'datetime64[ns]',
    'Amount': 'float64',
    'OrderNo': 'object'
}

# Convert data types according to the dictionary
for column, dtype in dtype_dict.items():
    if column in df.columns:
        try:
            df[column] = df[column].astype(dtype)
        except ValueError:
            print(f"Conversion error: Could not convert {column} to {dtype}")
    else:
        print(f"Warning: {column} not in DataFrame")

return df

```

Function 2: Normalize Columns to Upper Case:

This function processes specified columns by converting text data to uppercase, which helps in maintaining consistency for categorical data like `Currency`.

```

def uppercase_columns(dataframe, columns):
    """
    Strips whitespace and converts the text to upper case for specified columns in
    a DataFrame.

    Parameters:
        dataframe (pd.DataFrame): The DataFrame to process.
        columns (list): List of column names to convert to upper case.

    Returns:
        pd.DataFrame: The DataFrame with the specified columns converted to upper
        case.

    """

    for column in columns:
        if column in dataframe.columns:
            dataframe[column] = dataframe[column].str.strip().str.upper()
        else:
            print(f"Warning: {column} does not exist in the DataFrame.")

    return dataframe

```

Function 3: Rename Columns:

This function standardizes the column names by renaming them according to a predefined mapping. It ensures that column names are consistent, making the DataFrame easier to work with and understand.

```
def rename_columns_for_Cosmaline_data(df):
    """
    Renames columns of a specific DataFrame based on a predefined dictionary
    mapping of old names to new names.
    This function is tailored for the 'Website_ECOM_Data' DataFrame.

    Parameters:
        df (pd.DataFrame): The DataFrame whose columns will be renamed.

    Returns:
        pd.DataFrame: The DataFrame with renamed columns.
    """
    # Encapsulated dictionary specific to the Website_ECOM_Data schema
    column_rename_dict = {
        'Driver_Delivery_date': 'Delivery_Date',
        'OrderNo': 'order_number',
        'Amount': 'O_CODAmount',
        'Currency': 'CODCurrency'
    }

    return df.rename(columns=column_rename_dict, inplace=False)
```

Function 4: Add COD Amount Column:

This function adds a new column `CODAmount` to the DataFrame based on the `CODCurrency` column. If `CODCurrency` is `USD`, `CODAmount` is set to 0. If 'CODCurrency' is `LBP`, `CODAmount` is set to the value in `O_CODAmount`.

```
def add_cod_amount_column_Cosmaline_data(df):
    """
    Adds a new column 'CODAmount' to the DataFrame based on the 'CODCurrency'
    column.
    If 'CODCurrency' is 'USD', 'CODAmount' is set to 0.
    If 'CODCurrency' is 'LBP', 'CODAmount' is set to the value in 'O_CODAmount'.

    Parameters:
        df (pd.DataFrame): The DataFrame to process.

    Returns:
        pd.DataFrame: The DataFrame with the new 'CODAmount' column added.
    """
    df['CODAmount'] = df.apply(
        lambda row: 0 if row['CODCurrency'] == 'USD' else row['O_CODAmount'],
        axis=1
    )
    return df
```

Function 5: Add New Columns:

This function adds three new columns to the DataFrame: `ShprNo` as blank, `Aramex_Inv_date` as blank, and `TOKENNO` as `Shipped With Cosmaline`.

```
def add_new_columns_Cosmaline_Data(df):
    """
    Adds three new columns to the DataFrame:
    - 'ShprNo' as blank
    - 'Aramex_Inv_date' as blank
    - 'TOKENNO' as 'Shipped With Cosmaline'

    Parameters:
        df (pd.DataFrame): The DataFrame to process.

    Returns:
        pd.DataFrame: The DataFrame with the new columns added.
    """
    df['ShprNo'] = pd.NA
    df['Aramex_Inv_date'] = pd.NA
    df['TOKENNO'] = 'Shipped With Cosmaline'
    return df
```

Function 6: Add AWB from ECOM Data:

This function adds the `AWB` column from the ECOM Data to the Cosmaline Data based on matching `order_number` values.

```
def add_AWB_Cosmaline_Data(df, ECOM_df):
    """
    Merges df2 into df1 on the 'order_number' column, including only the 'AWB' column from df2.

    Parameters:
        df1 (pd.DataFrame): The DataFrame to be merged into.
        df2 (pd.DataFrame): The DataFrame containing 'order_number' and 'AWB' columns.

    Returns:
        pd.DataFrame: The resulting DataFrame after the merge.
    """
    updated_df = pd.merge(
        df,
        ECOM_df[['order_number', 'AWB']],
        on='order_number',
        how='left'
    )
    return updated_df
```

Function 7: Convert Data Types for Aramex Cosmaline:

This function converts the data types of specified columns in the Cosmaline DataFrame to match the database schema.

```
def convert_data_types_aramec_cosmaline(df):
    """
    Converts the data types of specified columns in the Aramec DataFrame to match
    the database schema.

    Parameters:
        df (pd.DataFrame): The DataFrame whose columns will have their data types
                            converted.

    Returns:
        pd.DataFrame: The DataFrame with converted data types.
    """
    dtype_dict = {
        'AWB': 'str', # Will be converted to varchar(100) in the database
        'ShprNo': 'object', # Will be converted to varchar(255) in the database
        'Delivery_Date': 'datetime64[ns]', # Will be converted to datetime in the
                                         database
        'CODAmount': 'float32', # Will be converted to float in the database
        'CODCurrency': 'object', # Will be converted to varchar(100) in the
                               database
        'O_CODAmount': 'float32', # Will be converted to float in the database
        'Aramex_Inv_date': 'datetime64[ns]', # Will be converted to datetime in
                                         the database
        'TOKENNO': 'object', # Will be converted to varchar(255) in the database
        'order_number': 'object' # Will be converted to varchar(255) in the
                               database
    }

    for column, dtype in dtype_dict.items():
        if column in df.columns:
            try:
                df[column] = df[column].astype(dtype)
            except ValueError:
                print(f"Conversion error: Could not convert {column} to {dtype}")
        else:
            print(f"Warning: {column} not in DataFrame")

    return df
```

4.5.4 Concatinating the Aramex and Cosmaline shipping data

This function concatenates the Aramex and Cosmaline shipping data transformed above into one `shippedandcollected_aramec_cosmaline` dataframe

```
def concatenate_Aramex_Cosmaline(df1, df2):
    """
    Concatenates two DataFrames vertically and resets the index.

    Parameters:
        df1 (pd.DataFrame): The first DataFrame.
        df2 (pd.DataFrame): The second DataFrame.

    Returns:
        pd.DataFrame: The concatenated DataFrame with reset index.
    """
    concatenated_df = pd.concat([df1, df2], ignore_index=True)
    return concatenated_df
```

4.5.5 Concatenated Aramex and Cosmaline Shipping Data Transformation

Transformation Pipeline: The following pipeline is used to transform the concatenated Aramex and Cosmaline shipping data:

```
def data_preprocessing_pipeline_Cosmaline_Aramex_Data(df):
    """
    Process a DataFrame through a series of cleaning and normalization steps.

    Parameters:
        df (pd.DataFrame): The DataFrame to be processed.

    Returns:
        pd.DataFrame: The processed DataFrame.
    """
    # Remove Duplicates
    df = remove_duplicates(df)

    return df
```

Function: Remove Duplicate Rows:

This function eliminates duplicate rows from the DataFrame, keeping only the first occurrence. This step is crucial for ensuring that each record is unique and avoiding redundancy in the data.

```
def remove_duplicates(df):
    """
    Removes duplicate rows from the DataFrame, keeping the first occurrence of
    each duplicate.

    Parameters:
        df (pd.DataFrame): The DataFrame from which to remove duplicates.

    Returns:
    
```

```
pd.DataFrame: A new DataFrame with duplicates removed, keeping the first occurrence.  
"""  
return df.drop_duplicates()
```

4.5.6 Credit Card Transaction Data Transformation

Transformation Pipeline:

The following pipeline is used to transform the Credit Card transaction data:

```
def data_preprocessing_pipeline_CreditCard_Data(df):  
    """  
    Process a DataFrame through a series of cleaning and normalization steps.  
  
    Parameters:  
        df (pd.DataFrame): The DataFrame to be processed.  
  
    Returns:  
        pd.DataFrame: The processed DataFrame.  
    """  
    # Upper case the Narrative column and Currency  
    df = uppercase_columns(df, ['Narrative', 'Currency'])  
  
    # Converting the Data types:  
    df = convert_data_types_Credit_Card(df)  
  
    # Lowercasing Column Names:  
    df = rename_columns_to_lowercase(df)  
  
    return df
```

Function 1: Normalize Columns to Upper Case:

This function processes specified columns by converting text data to uppercase, which helps in maintaining consistency for categorical data like **Narrative** and **Currency**.

```
def uppercase_columns(dataframe, columns):  
    """  
    Strips whitespace and converts the text to upper case for specified columns in a DataFrame.  
  
    Parameters:  
        dataframe (pd.DataFrame): The DataFrame to process.  
        columns (list): List of column names to convert to upper case.  
  
    Returns:  
        pd.DataFrame: The DataFrame with the specified columns converted to upper
```

```
case.  
    """  
  
    for column in columns:  
        if column in dataframe.columns:  
            dataframe[column] = dataframe[column].str.strip().str.upper()  
        else:  
            print(f"Warning: {column} does not exist in the DataFrame.")  
    return dataframe
```

Function 2: Convert Data Types:

This function ensures that the data types of specific columns are consistent and correct. It converts columns like `Value_date` to datetime, `Amount` to float, and others to string types as needed for further processing and analysis.

```
def convert_data_types_Credit_Card(df):  
    """  
    Converts the data types of specified columns in the Credit Card DataFrame.  
  
    Parameters:  
        df (pd.DataFrame): The DataFrame whose columns will have their data types converted.  
  
    Returns:  
        pd.DataFrame: The DataFrame with converted data types.  
    """  
  
    # Define a dictionary mapping column names to desired data types  
    dtype_dict = {  
        'Value_date': 'datetime64[ns]',  
        'Narrative': 'str',  
        'Amount': 'float32',  
        'Currency': 'str'  
    }  
  
    # Convert data types according to the dictionary  
    for column, dtype in dtype_dict.items():  
        if column in df.columns:  
            try:  
                df[column] = df[column].astype(dtype)  
            except ValueError:  
                print(f"Conversion error: Could not convert {column} to {dtype}")  
        else:  
            print(f"Warning: {column} not in DataFrame")  
  
    return df
```

Function 3: Rename Columns to Lowercase:

This function standardizes the column names by converting them to lowercase. It ensures that column names are consistent, making the DataFrame easier to work with and understand.

```
def rename_columns_to_lowercase(df):
    """
    Renames columns of the DataFrame to lowercase in place.

    Parameters:
        df (pd.DataFrame): The DataFrame whose columns will be renamed.

    Returns:
        pd.DataFrame: The DataFrame with renamed columns.
    """
    return df.rename(columns=str.lower, inplace=False)
```

4.5.7 ERP Data Transformation

Transformation Pipeline:

The following pipeline is used to transform the ERP data:

```
def data_preprocessing_pipeline_ERP_Data(df):
    """
    Process a DataFrame through a series of cleaning and normalization steps.

    Parameters:
        df (pd.DataFrame): The DataFrame to be processed.

    Returns:
        pd.DataFrame: The processed DataFrame.
    """

    # Converting the Customer Number to object
    df = convert_data_types_ERP_Data(df)

    # Converting some columns to upper case:
    df = uppercase_columns(df, ['CURRENCY_CODE'])

    # Converting some columns to lower case:
    df = lower_columns(df, ['CUSTOMER_NAME', 'RECEIPT_CLASS'])

    # Extracting the Token from the Comments column:
    df = extract_token_ERP_Data(df, 'COMMENTS')

    return df
```

Function 1: Convert Data Types:

This function ensures that the data types of specific columns are consistent and correct. It converts columns like `CUSTOMER_NUMBER` to object types as needed for further processing and analysis.

```
def convert_data_types_ERP_Data(df):
    """
    Converts the data types of specified columns in the DataFrame.

    Parameters:
        df (pd.DataFrame): The DataFrame whose columns will have their data types converted.

    Returns:
        pd.DataFrame: The DataFrame with converted data types.

    """
    # Define a dictionary mapping column names to desired data types
    dtype_dict = {
        'CUSTOMER_NUMBER': 'object'
    }

    # Convert data types according to the dictionary
    for column, dtype in dtype_dict.items():
        if column in df.columns:
            try:
                df[column] = df[column].astype(dtype)
            except ValueError:
                print(f"Conversion error: Could not convert {column} to {dtype}")
        else:
            print(f"Warning: {column} not in DataFrame")

    return df
```

Function 2: Normalize Columns to Upper Case:

This function processes specified columns by converting text data to uppercase, which helps in maintaining consistency for categorical data like `CURRENCY_CODE`.

```
def uppercase_columns(dataframe, columns):
    """
    Strips whitespace and converts the text to upper case for specified columns in a DataFrame.

    Parameters:
        dataframe (pd.DataFrame): The DataFrame to process.
        columns (list): List of column names to convert to upper case.

    Returns:
        pd.DataFrame: The DataFrame with the specified columns converted to upper case.

    """
    for column in columns:
```

```

if column in dataframe.columns:
    dataframe[column] = dataframe[column].str.strip().str.upper()
else:
    print(f"Warning: {column} does not exist in the DataFrame.")
return dataframe

```

Function 3: Normalize Columns to Lower Case:

This function processes specified columns to ensure consistency in text data by converting them to lowercase and capitalizing the first letter of each word. This is useful for columns like `CUSTOMER_NAME` and `RECEIPT_CLASS` to maintain uniformity.

```

def lower_columns(dataframe, columns):
    """
    Strips, lowers, and capitalizes the first letter of each word in the specified
    columns of a DataFrame.

    Parameters:
        dataframe (pd.DataFrame): The DataFrame to process.
        columns (list): List of column names to normalize.

    Returns:
        pd.DataFrame: The DataFrame with normalized columns.
    """
    for column in columns:
        if column in dataframe.columns:
            dataframe[column] =
                dataframe[column].str.strip().str.lower().str.capitalize()
        else:
            print(f"Warning: {column} does not exist in the DataFrame.")
    return dataframe

```

Function 4: Extract Token from Comments:

This function extracts the token number from the specified column in the DataFrame and updates the column with the extracted token. If the token is not found, it keeps the original value.

```

def extract_token_ERP_Data(df, column_name):
    """
    Extracts the token number from the specified column in the DataFrame and
    updates the column with the extracted token.

    If the token is not found, it keeps the original value.

    Parameters:
        df (pd.DataFrame): The DataFrame containing the column.
        column_name (str): The name of the column from which to extract the token.

    Returns:
    """

```

```

pd.DataFrame: The updated DataFrame with the extracted tokens in the specified
column.
"""
# Define the regex pattern
pattern = r'Token (\w+)'

# Extract the token using str.extract
tokens = df[column_name].str.extract(pattern, expand=False)

# Update the column with the extracted tokens only where the pattern is
matched
df[column_name] = tokens.fillna(df[column_name])

return df

```

4.5.7 Oracle Data Transformation

Transformation Pipeline:

The following pipeline is used to transform the Oracle data:

```

def data_preprocessing_pipeline_Oracle_Data(df):
    """
    Process a DataFrame through a series of cleaning and normalization steps.

    Parameters:
        df (pd.DataFrame): The DataFrame to be processed.

    Returns:
        pd.DataFrame: The processed DataFrame.
    """

    # Removing the hours from the date column:
    df = remove_time_from_datetime_Oracle_Data(df, 'ORDERED_DATE')

    # Renaming the Columns:
    df = rename_columns_for_Oracle_data(df)

    # Converting the ECOM reference order number to object
    df = convert_data_types_Oracle_Data(df)

    return df

```

Function 1: Remove Time from Datetime:

This function removes the time portion from a datetime column in a DataFrame while preserving the datatype as datetime.

```

def remove_time_from_datetime_Oracle_Data(df, column_name):
    """
    Removes the time portion from a datetime column in a DataFrame while
    preserving the datatype as datetime.

    Parameters:
    df (pd.DataFrame): The DataFrame containing the column.
    column_name (str): The name of the column to process.

    Returns:
    pd.DataFrame: The updated DataFrame with the time portion removed from the
    specified column.

    """
    # Convert the column to datetime format if it's not already
    df[column_name] = pd.to_datetime(df[column_name])

    # Remove the time portion and preserve the datatype
    df[column_name] = df[column_name].dt.date

    # Convert back to datetime to ensure it remains a datetime object
    df[column_name] = pd.to_datetime(df[column_name])

    return df

```

Function 2: Rename Columns:

This function standardizes the column names by renaming them according to a predefined mapping. It ensures that column names are consistent, making the DataFrame easier to work with and understand.

```

def rename_columns_for_Oracle_data(df):
    """
    Renames columns of a specific DataFrame based on a predefined dictionary
    mapping of old names to new names.
    This function is tailored for the 'Website_ECOM_Data' DataFrame.

    Parameters:
    df (pd.DataFrame): The DataFrame whose columns will be renamed.

    Returns:
    pd.DataFrame: The DataFrame with renamed columns.

    """
    # Encapsulated dictionary specific to the Website_ECOM_Data schema
    column_rename_dict = {
        'OPERATING_UNIT_NAME': 'operating_unit_name',
        'ORACLE_REFERENCE_ORDER_NUMBER': 'oracle_reference_order_number',
        'ECOM_REFERENCE_ORDER_NUMBER': 'ecom_reference_order_number',
        'ORDER_TYPE': 'order_type',
        'ORDER_TYPE_NAME': 'order_type_name',
        'ORDERED_ITEM': 'ordered_item',
        'ORDERED_QUANTITY': 'ordered_quantity',
    }

```

```

        'PRICING_QUANTITY_UOM': 'pricing_quantity_uom',
        'UNIT_SELLING_PRICE (after discount without vat)': 'unit_selling_price',
        'UNIT_LIST_PRICE( original price before discount without vat)': 
        'unit_list_price',
        'ORDERED_DATE': 'ordered_date',
        'CUSTOMER_NAME': 'customer_name',
        'TAX_CODE': 'tax_code'
    }

    return df.rename(columns=column_rename_dict, inplace=False)

```

Function 3: Convert Data Types:

This function ensures that the data types of specific columns are consistent and correct. It converts columns like `ecom_reference_order_number` to object types, `ordered_date` to datetime, and others to their appropriate types as needed for further processing and analysis.

```

def convert_data_types_Oracle_Data(df):
    """
    Converts the data types of specified columns in the Oracle Data DataFrame.

    Parameters:
        df (pd.DataFrame): The DataFrame whose columns will have their data types converted.

    Returns:
        pd.DataFrame: The DataFrame with converted data types.
    """

    dtype_dict = {
        'operating_unit_name': 'object',
        'oracle_reference_order_number': 'int32',
        'ecom_reference_order_number': 'object',
        'order_type': 'object',
        'order_type_name': 'object',
        'ordered_item': 'object',
        'ordered_quantity': 'int32',
        'pricing_quantity_uom': 'object',
        'unit_selling_price': 'float32',
        'unit_list_price': 'float32',
        'ordered_date': 'datetime64[ns]',
        'customer_name': 'object',
        'tax_code': 'object'
    }

    for column, dtype in dtype_dict.items():
        if column in df.columns:
            try:
                df[column] = df[column].astype(dtype)
            except ValueError:
                print(f"Conversion error: Could not convert {column} to {dtype}")
        else:
            pass

```

```
    print(f"Warning: {column} not in DataFrame")  
  
    return df
```

4.5.8 Daily Rate Data Transformation

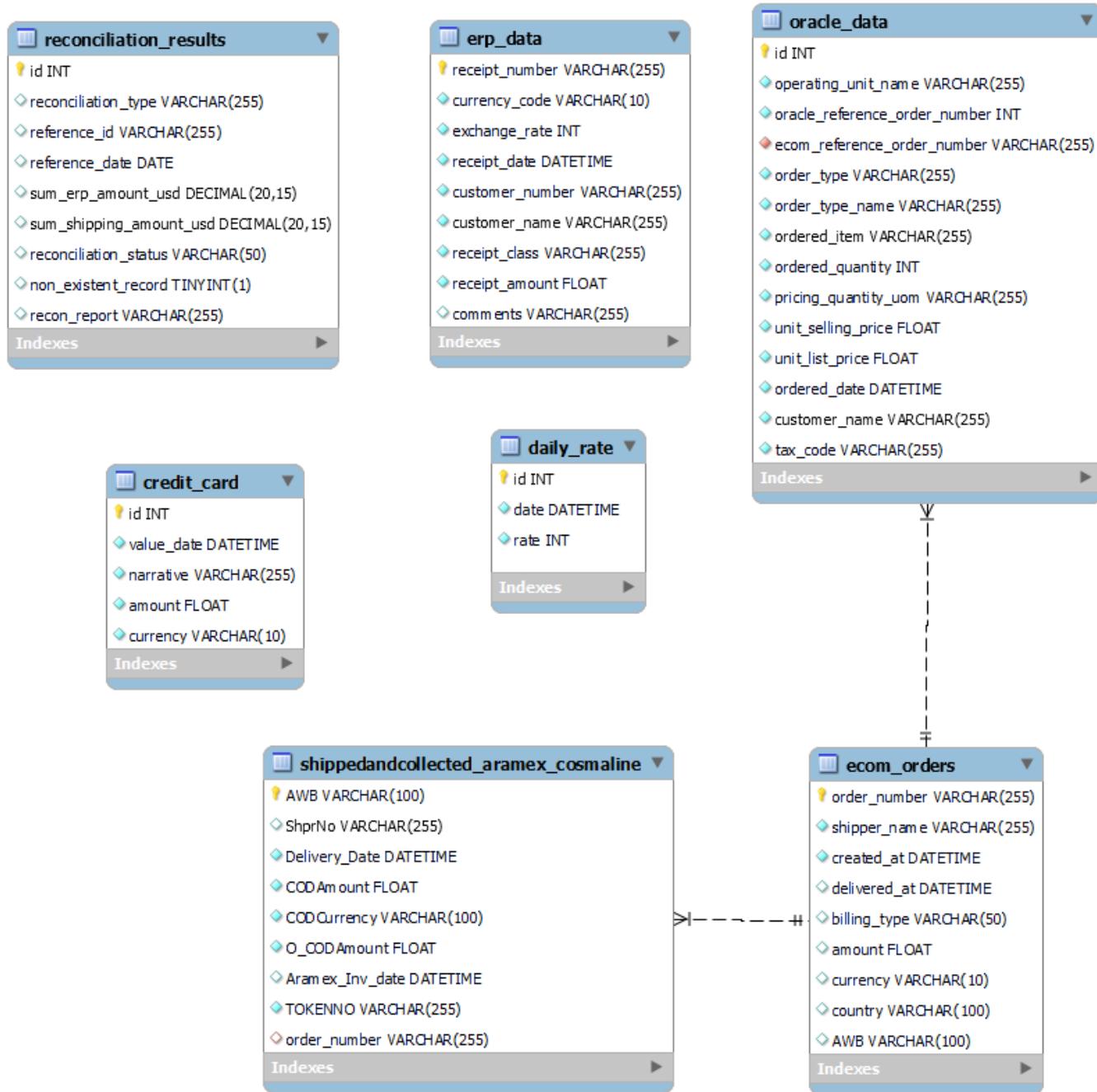
The Daily Rate data does not require any preprocessing. This dataset is already in a format suitable for further analysis and processing.

4.6 The Load Process

To manage the transformation process, create a Python file named `load.py` in the `scripts` folder to hold all the functions that will load the data from the python environment to the relational database

4.6.1 Setting up the Database in MySQL

Database Schema



The database schema consists of the following tables:

- ecom_orders
- shippedandcollected_aramex_cosmaline
- erp_data
- credit_card
- oracle_data
- reconciliation_results
- daily_rate

ecom_order Table

The **ecom_orders** table stores order details from the e-commerce platform. Each order is uniquely identified by **order_number**.

```

CREATE TABLE `ecom_orders` (
  `order_number` varchar(255) NOT NULL,
  `shipper_name` varchar(255) NOT NULL,
  `created_at` datetime NOT NULL,
  `delivered_at` datetime DEFAULT NULL,
  `billing_type` varchar(50) DEFAULT NULL,
  `amount` float DEFAULT NULL,
  `currency` varchar(10) DEFAULT NULL,
  `country` varchar(100) DEFAULT NULL,
  `AWB` varchar(100) DEFAULT NULL,
  `db_entry_time` timestamp NULL DEFAULT CURRENT_TIMESTAMP,
  `db_update_time` timestamp NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
  PRIMARY KEY (`order_number`)
);

```

shippedandcollected_aramex_cosmaline Table

The `shippedandcollected_aramex_cosmaline` table stores shipment details collected by Aramex and Cosmaline, with a foreign key reference to `ecom_orders`.

```

CREATE TABLE `shippedandcollected_aramex_cosmaline` (
  `AWB` varchar(100) NOT NULL,
  `ShprNo` varchar(255) DEFAULT NULL,
  `Delivery_Date` datetime NOT NULL,
  `CODAmount` float NOT NULL,
  `CODCurrency` varchar(100) NOT NULL,
  `O_CODAmount` float NOT NULL,
  `Aramex_Inv_date` datetime DEFAULT NULL,
  `TOKENNO` varchar(255) NOT NULL,
  `order_number` varchar(255) DEFAULT NULL,
  `db_entry_time` timestamp NULL DEFAULT CURRENT_TIMESTAMP,
  `db_update_time` timestamp NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
  PRIMARY KEY (`AWB`),
  KEY `order_number` (`order_number`),
  CONSTRAINT `shippedandcollected_aramex_cosmaline_ibfk_1` FOREIGN KEY (`order_number`) REFERENCES `ecom_orders` (`order_number`)
);

```

erp_data Table

The `erp_data` table stores financial receipt data from the ERP system, with each receipt uniquely identified by `receipt_number`.

```

CREATE TABLE `erp_data` (
  `receipt_number` varchar(255) NOT NULL,

```

```

`currency_code` varchar(10) NOT NULL,
`exchange_rate` int NOT NULL,
`receipt_date` datetime NOT NULL,
`customer_number` varchar(255) NOT NULL,
`customer_name` varchar(255) NOT NULL,
`receipt_class` varchar(255) NOT NULL,
`receipt_amount` float NOT NULL,
`comments` varchar(255) DEFAULT NULL,
`db_entry_time` timestamp NULL DEFAULT CURRENT_TIMESTAMP,
`db_update_time` timestamp NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,
PRIMARY KEY (`receipt_number`)
);

```

credit_card table

The `credit_card` table stores credit card transaction data, with each transaction uniquely identified by `id`.

```

CREATE TABLE `credit_card` (
`id` int NOT NULL AUTO_INCREMENT,
`value_date` datetime NOT NULL,
`narrative` varchar(255) NOT NULL,
`amount` float NOT NULL,
`currency` varchar(10) NOT NULL,
`db_entry_time` timestamp NULL DEFAULT CURRENT_TIMESTAMP,
`db_update_time` timestamp NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,
PRIMARY KEY (`id`)
);

```

oracle_data Table

The `oracle_data` table stores order data from the Oracle system, with a foreign key reference to `ecom_orders`.

```

CREATE TABLE `oracle_data` (
`id` int NOT NULL AUTO_INCREMENT,
`operating_unit_name` varchar(255) NOT NULL,
`oracle_reference_order_number` int NOT NULL,
`ecom_reference_order_number` varchar(255) NOT NULL,
`order_type` varchar(255) NOT NULL,
`order_type_name` varchar(255) NOT NULL,
`ordered_item` varchar(255) NOT NULL,
`ordered_quantity` int NOT NULL,
`pricing_quantity_uom` varchar(255) NOT NULL,
`unit_selling_price` float NOT NULL,
`unit_list_price` float NOT NULL,
`ordered_date` datetime NOT NULL,

```

```

`customer_name` varchar(255) NOT NULL,
`tax_code` varchar(255) NOT NULL,
`db_entry_time` timestamp NULL DEFAULT CURRENT_TIMESTAMP,
`db_update_time` timestamp NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,
PRIMARY KEY (`id`),
KEY `ecom_reference_order_number` (`ecom_reference_order_number`),
CONSTRAINT `oracle_data_ibfk_1` FOREIGN KEY (`ecom_reference_order_number`)
REFERENCES `ecom_orders` (`order_number`)
);

```

reconciliation_results Table

The `reconciliation_results` table stores the results of reconciliation processes, including details such as reconciliation type, reference ID, and amounts.

```

CREATE TABLE `reconciliation_results` (
`id` int NOT NULL AUTO_INCREMENT,
`reconciliation_type` varchar(255) DEFAULT NULL,
`reference_id` varchar(255) DEFAULT NULL,
`reference_date` date DEFAULT NULL,
`sum_erp_amount_usd` decimal(20,15) DEFAULT NULL,
`sum_shipping_amount_usd` decimal(20,15) DEFAULT NULL,
`reconciliation_status` varchar(50) DEFAULT NULL,
`non_existent_record` tinyint(1) DEFAULT NULL,
`recon_report` varchar(255) DEFAULT NULL,
`db_entry_time` timestamp NULL DEFAULT CURRENT_TIMESTAMP,
`db_update_time` timestamp NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,
PRIMARY KEY (`id`)
);

```

daily_rate Table

The `daily_rate` table stores daily exchange rates, with each rate uniquely identified by `id`.

```

CREATE TABLE `daily_rate` (
`id` int NOT NULL AUTO_INCREMENT,
`date` datetime NOT NULL,
`rate` int NOT NULL,
`db_entry_time` timestamp NULL DEFAULT CURRENT_TIMESTAMP,
`db_update_time` timestamp NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,
PRIMARY KEY (`id`)
);

```

4.6.2 Connecting to the Database

The following function connects to the MySQL database using SQLAlchemy. This function creates an SQLAlchemy engine with hardcoded credentials. Note: Be cautious with sensitive information in production environments and consider using environment variables or a secure secrets management solution.

```
import pandas as pd
from sqlalchemy import create_engine
import numpy as np
from sqlalchemy import inspect
```

```
def create_db_engine():
    """
    Creates a SQLAlchemy engine with hardcoded credentials.

    Returns:
        engine: An SQLAlchemy Engine instance.

    """
    user = 'yourUserName'
    password = 'yourPassword' # Be cautious with sensitive info in production environments
    host = 'host.docker.internal'
    database = 'CapstoneTest'
    url = f'mysql+pymysql://{user}:{password}@{host}/{database}'
    engine = create_engine(url, echo=True) # Set echo=False in production for less verbosity
    return engine
```

Explanation:

- User: The username for the database connection.
- Password: The password for the database connection.
- Host: The database server's hostname. host.docker.internal is used for Docker-based environments.
- Database: The name of the database to connect to.
- URL: The connection URL formatted for SQLAlchemy and MySQL.
- Engine: The SQLAlchemy engine instance created with the specified URL. The echo parameter is set to True for verbose logging during development; set it to False in production.

4.6.2 Loading the data into the Database

To ensure that if the automation is run multiple times on the same data, there are no duplicate values in the permanent tables. The data is first loaded into temporary tables, and then these tables are compared to the permanent tables using two functions depending on whether the dataframe has a primary key or not. Finally, a function cleans up the temporary tables.

Function to Upload DataFrame to Temporary SQL Table

This function uploads a DataFrame to a specified temporary SQL table using SQLAlchemy engine.

```

def upload_dataframe_to_temp_sql(df, temp_table_name, engine, index=False):
    """
    Uploads a DataFrame to a specified temporary SQL table using SQLAlchemy
    engine.

    Parameters:
        df (pd.DataFrame): The DataFrame to upload.
        temp_table_name (str): The name of the temporary SQL table to which the
        data should be uploaded.
        engine: The SQLAlchemy engine object connected to the database.
        index (bool): Whether to write the DataFrame's index as a column in the
        table.

    Returns:
        None
    """
    try:
        df.to_sql(name=temp_table_name, con=engine, if_exists='replace',
        index=index)
        print("Data uploaded successfully to the temporary table.")
    except Exception as e:
        print(f"An error occurred while uploading data: {e}")

```

Function to Merge Data from Temporary to Main Table with Primary Key:

This function merges data from a temporary table to the main table, handling duplicates using the primary key.

```

def merge_data_from_temp_to_main_with_pk(temp_table_name, main_table_name,
engine):
    """
    Merges data from a temporary table to the main table.

    Parameters:
        temp_table_name (str): The name of the temporary SQL table.
        main_table_name (str): The name of the main SQL table.
        engine: The SQLAlchemy engine object connected to the database.

    Returns:
        None
    """
    # Inspect the main table to get the column names
    inspector = inspect(engine)
    # Get columns excluding 'id', 'db_entry_time', and 'db_update_time'
    columns = [column['name'] for column in inspector.get_columns(main_table_name)]
    if column['name'] not in ['id', 'db_entry_time', 'db_update_time']]

    # Construct the ON DUPLICATE KEY UPDATE part of the SQL statement
    on_duplicate_key_update = ", ".join([f"{col} = VALUES({col})" for col in
columns])

```

```

# Construct the full merge SQL statement
merge_sql = f"""
INSERT INTO {main_table_name} (', '.join(columns))
SELECT ', '.join(columns) FROM {temp_table_name}
ON DUPLICATE KEY UPDATE
{on_duplicate_key_update}
"""

# Execute the merge SQL statement
with engine.connect() as connection:
    connection.execute(merge_sql)
print(f"Data from {temp_table_name} merged into {main_table_name} successfully.")

```

Function to Merge Data from Temporary to Main Table without Primary Key:

This function merges data from a temporary table to the main table, ensuring no duplicates for tables without a primary key.

```

def merge_data_from_temp_to_main_without_pk(temp_table_name, main_table_name,
engine, columns):
    """
    Merges data from a temporary table to the main table, ensuring no duplicates
    for tables without a primary key.

    Parameters:
        temp_table_name (str): The name of the temporary SQL table.
        main_table_name (str): The name of the main SQL table.
        engine: The SQLAlchemy engine object connected to the database.
        columns (list): List of columns to compare for duplicates.

    Returns:
        None
    """

    # Exclude 'db_entry_time' and 'db_update_time' from the columns list
    columns = [col for col in columns if col not in ['db_entry_time',
'db_update_time']]

    unique_columns_select = " AND ".join([f"{main_table_name}.{col} = "
{temp_table_name}.{col}" for col in columns])
    merge_sql = f"""
INSERT INTO {main_table_name} (', '.join(columns))
SELECT ', '.join([f'{temp_table_name}.{col}' for col in columns])
FROM {temp_table_name}
WHERE NOT EXISTS (
    SELECT 1
    FROM {main_table_name}
    WHERE {unique_columns_select}
)
"""

    print(f"Merge SQL:\n{merge_sql}")

```

```
with engine.connect() as connection:
    result = connection.execute(merge_sql)
    print(f"Rows affected: {result.rowcount}")
    print(f"Data from {temp_table_name} merged into {main_table_name}
successfully.")
```

Function to Cleanup Temporary Tables:

This function drops the temporary tables used during the ETL process.

```
def cleanup_temp_tables(engine):
    """
    Drops the temporary tables used during the ETL process.

    Parameters:
        engine: The SQLAlchemy engine object connected to the database.

    Returns:
        None
    """
    temp_tables = [
        'temp_ecom_orders',
        'temp_aramex_data',
        'temp_cosmaline_data',
        'temp_oracle_data',
        'temp_credit_card_data',
        'temp_erp_data',
        'temp_daily_rate',
        'temp_credit_card',
        'temp_ecom_orders',
        'temp_shippedandcollected_aramex_cosmaline',
        'erp_data_tokens',
        'aramex_cod_sum',
        'erp_receipt_sum',
        'erp_cosmaline',
        'aramex_cosmaline_sum',
        'erp_cosmaline_sum',
        'erp_cybersource_sum',
        'credit_card_sum'
    ]

    with engine.connect() as connection:
        for table in temp_tables:
            connection.execute(f"DROP TABLE IF EXISTS {table}")
    print("Temporary tables dropped successfully.")
```

4.7 The Reconciliation Process

To manage the reconciliation process efficiently, create a Python file named `reconciliation.py` in the `scripts` folder. This file will contain the functions required to run the reconciliation scripts. Additionally, place all the SQL scripts related to the reconciliation process in the `scripts` folder for better organization and management.

4.7.1 Function to Execute Scripts

The `reconciliation.py` file will include functions to execute the SQL reconciliation scripts. Below is the function that reads the SQL scripts and executes them:

```
import sqlalchemy
from sqlalchemy.orm import sessionmaker
import sys
sys.path.append('/opt/airflow/scripts') # Ensure the scripts directory is in the
system path

from load import create_db_engine

from sqlalchemy.orm import sessionmaker
import sqlalchemy as sa
```

```
def execute_reconciliation_script(script_filename):
    """
    Executes a reconciliation SQL script.

    Parameters:
        script_filename (str): The filename of the SQL script to execute.

    Returns:
        None
    """
    # Create a database engine using the function from load.py
    engine = create_db_engine()

    # Create a session
    Session = sessionmaker(bind=engine)
    session = Session()

    try:
        # Read the SQL script
        script_path = f'/opt/airflow/scripts/{script_filename}'
        with open(script_path, 'r') as file:
            sql_script = file.read()

        # Split the script into individual statements
        statements = sql_script.split(';')

        # Execute each statement individually
        for statement in statements:
            if statement.strip():
```

```

        session.execute(sa.text(statement))

    session.commit()
    print(f"Script {script_filename} executed successfully.")

except Exception as e:
    session.rollback()
    print(f"An error occurred: {e}")

finally:
    session.close()

```

Explanation:

- execute_reconciliation_script: This function reads a specified SQL script from the scripts folder and executes it. It uses SQLAlchemy to handle the database connection and session management.
- Reading the SQL Script: The script is read from the specified path.
- Splitting the Script: The script is split into individual SQL statements.
- Executing Statements: Each statement is executed individually.
- Session Management: The session is committed if all statements execute successfully. In case of an error, the session is rolled back.

4.7.2 Reconciliation Scripts

ECOM Orders Reconciliation

This script reconciles the e-commerce orders by comparing the order amounts in the `ecom_orders` table with the corresponding amounts in the `shippedandcollected_aramex_cosmaline` table. It inserts the reconciliation results into the `reconciliation_results` table.

```

-- Step 1: Create necessary temporary tables for ECOM Orders Reconciliation

-- Note: Assuming 'ecom_orders' and 'shippedandcollected_aramex_cosmaline' are
existing tables and do not need to be created temporarily.

-- Step 2: Insert into reconciliation_results table with duplicate check
INSERT INTO reconciliation_results (reconciliation_type, reference_id,
sum_erp_amount_usd, sum_shipping_amount_usd, reconciliation_status,
non_existent_record, recon_report)
SELECT
    'ECOM Orders Reconciliation' AS reconciliation_type,
    e.order_number AS reference_id,
    e.amount AS sum_erp_amount_usd,
    SUM(
        CASE
            WHEN a.CODCurrency = 'USD' THEN a.O_CODAmount
            WHEN a.CODCurrency = 'LBP' THEN a.CODAmount / COALESCE((SELECT rate
FROM daily_rate WHERE date = a.Delivery_Date), 89500)
            ELSE 0
        END
    )

```

```

) AS sum_cod_amount_usd,
CASE
    WHEN ABS(e.amount - SUM(
        CASE
            WHEN a.CODCurrency = 'USD' THEN a.O_CODAmount
            WHEN a.CODCurrency = 'LBP' THEN a.CODAmount / COALESCE((SELECT
rate FROM daily_rate WHERE date = a.Delivery_Date), 89500)
            ELSE 0
        END
    )) <= 1 THEN 'Match'
    ELSE 'Mismatch'
END AS reconciliation_status,
CASE
    WHEN COUNT(a.order_number) = 0 THEN 1
    ELSE 0
END AS non_existent_record,
CASE
    WHEN ABS(e.amount - SUM(
        CASE
            WHEN a.CODCurrency = 'USD' THEN a.O_CODAmount
            WHEN a.CODCurrency = 'LBP' THEN a.CODAmount / COALESCE((SELECT
rate FROM daily_rate WHERE date = a.Delivery_Date), 89500)
            ELSE 0
        END
    )) <= 1 THEN 'Match'
    WHEN COUNT(a.order_number) = 0 THEN 'Mismatch due to non-existent record'
    ELSE 'Mismatch due to different amounts'
END AS recon_report
FROM
    ecom_orders e
LEFT JOIN
    shippedandcollected_aramex_cosmaline a ON e.order_number = a.order_number
WHERE NOT EXISTS (
    SELECT 1 FROM reconciliation_results rr
    WHERE rr.reference_id = e.order_number AND rr.reconciliation_type = 'ECOM
Orders Reconciliation'
)
GROUP BY
    e.order_number;

COMMIT;

```

Tokens Reconciliation

This script reconciles the tokens by comparing the token amounts in the `erp_data` and `shippedandcollected_aramex_cosmaline` tables. It inserts the reconciliation results into the `reconciliation_results` table.

```
-- Step 1: Create necessary temporary tables for Tokens Reconciliation
-- Create erp_data_tokens
```

```

CREATE TEMPORARY TABLE IF NOT EXISTS erp_data_tokens AS
SELECT
    receipt_number,
    CASE
        WHEN currency_code = 'LBP' THEN receipt_amount / exchange_rate
        ELSE receipt_amount
    END AS receipt_amount_usd,
    REGEXP_SUBSTR(comments, '[0-9]{6}[A-Z]{2}|[0-9]{7}[A-Z]{2}') AS token
FROM
    erp_data
WHERE
    comments REGEXP '[0-9]{6}[A-Z]{2}|[0-9]{7}[A-Z]{2}';

-- Create aramex_cod_sum
CREATE TEMPORARY TABLE IF NOT EXISTS aramex_cod_sum AS
SELECT
    TOKENNO AS token,
    SUM(
        CASE
            WHEN CODCurrency = 'LBP' THEN
                CODAmount / COALESCE((SELECT rate FROM daily_rate WHERE date =
aramex.Delivery_Date), 89500)
            ELSE
                0_CODAmount
        END
    ) AS sum_cod_amount_usd
FROM
    shippedandcollected_aramex_cosmaline aramex
GROUP BY
    TOKENNO;

-- Create erp_receipt_sum
CREATE TEMPORARY TABLE IF NOT EXISTS erp_receipt_sum AS
SELECT
    token,
    SUM(receipt_amount_usd) AS sum_receipt_amount_usd
FROM
    erp_data_tokens
GROUP BY
    token;

-- Step 2: Insert into reconciliation_results table with duplicate check
INSERT INTO reconciliation_results (reconciliation_type, reference_id,
sum_erp_amount_usd, sum_shipping_amount_usd, reconciliation_status,
non_existent_record, recon_report)
SELECT
    'Tokens Reconciliation' AS reconciliation_type,
    e.token AS reference_id,
    e.sum_receipt_amount_usd,
    a.sum_cod_amount_usd,
    CASE
        WHEN ABS(e.sum_receipt_amount_usd - a.sum_cod_amount_usd) <= 0.05 *
a.sum_cod_amount_usd THEN 'Match'
        ELSE 'Mismatch'
    END

```

```

        END AS reconciliation_status,
        0 AS non_existent_record,
        CASE
            WHEN ABS(e.sum_receipt_amount_usd - a.sum_cod_amount_usd) <= 0.05 *
a.sum_cod_amount_usd THEN 'Match'
            ELSE 'Mismatch due to different amounts'
        END AS recon_report
    FROM
        erp_receipt_sum e
    LEFT JOIN
        aramex_cod_sum a ON e.token = a.token
    WHERE NOT EXISTS (
        SELECT 1 FROM reconciliation_results rr
        WHERE rr.reference_id = e.token AND rr.reconciliation_type = 'Tokens
Reconciliation'
    );
-- Clean up temporary tables
DROP TABLE IF EXISTS erp_data_tokens, aramex_cod_sum, erp_receipt_sum;

COMMIT;

```

Shipped With Cosmaline Reconciliation

This script reconciles the shipments marked as **Shipped With Cosmaline** by comparing the amounts in the `erp_data` and `shippedandcollected_aramex_cosmaline` tables. It inserts the reconciliation results into the `reconciliation_results` table.

```

-- Step 1: Create necessary temporary tables for Shipped With Cosmaline
Reconciliation

-- Create erp_cosmaline
CREATE TEMPORARY TABLE IF NOT EXISTS erp_cosmaline AS
SELECT
    receipt_date,
    CASE
        WHEN currency_code = 'LBP' THEN receipt_amount / exchange_rate
        ELSE receipt_amount
    END AS receipt_amount_usd
FROM
    erp_data
WHERE
    comments = 'Shipped With Cosmaline';

-- Create aramex_cosmaline_sum
CREATE TEMPORARY TABLE IF NOT EXISTS aramex_cosmaline_sum AS
SELECT
    Delivery_Date,
    SUM(
        CASE
            WHEN

```

Cybersource Reconciliation

This script reconciles the Cybersource credit card transactions by comparing the amounts in the `erp_data` and `credit_card` tables. It inserts the reconciliation results into the `reconciliation_results` table.

```
-- Step 1: Create necessary temporary tables for Cybersource Reconciliation

-- Create erp_cybersource_sum
CREATE TEMPORARY TABLE IF NOT EXISTS erp_cybersource_sum AS
SELECT
    DATE(receipt_date) AS date,
    SUM(
        CASE
            WHEN currency_code = 'LBP' THEN receipt_amount / exchange_rate
            ELSE receipt_amount
        END
    ) AS sum_receipt_amount_usd
FROM
    erp_data
WHERE
    comments = 'Cybersource'
GROUP BY
    DATE(receipt_date);

-- Create credit_card_sum
CREATE TEMPORARY TABLE IF NOT EXISTS credit_card_sum AS
SELECT
    DATE(value_date) AS date,
    SUM(amount) AS sum_credit_card_amount
FROM
    credit_card
GROUP BY
    DATE(value_date);

-- Step 2: Insert into reconciliation_results table with duplicate check
INSERT INTO reconciliation_results (reconciliation_type, reference_date,
sum_erp_amount_usd, sum_shipping_amount_usd, reconciliation_status,
non_existent_record, recon_report)
SELECT
    'Cybersource Reconciliation' AS reconciliation_type,
    e.date AS reference_date,
    e.sum_receipt_amount_usd,
    c.sum_credit_card_amount AS sum_shipping_amount_usd,
    CASE
        WHEN ABS(e.sum_receipt_amount_usd - c.sum_credit_card_amount) <= 0.05 *
c.sum_credit_card_amount THEN 'Match'
        ELSE 'Mismatch'
    END AS reconciliation_status,
    0 AS non_existent_record,
    CASE
```

```

        WHEN ABS(e.sum_receipt_amount_usd - c.sum_credit_card_amount) <= 0.05 *
c.sum_credit_card_amount THEN 'Match'
        ELSE 'Mismatch due to different amounts'
    END AS recon_report
FROM
    erp_cybersource_sum e
LEFT JOIN
    credit_card_sum c ON e.date = c.date
WHERE NOT EXISTS (
    SELECT 1 FROM reconciliation_results rr
    WHERE rr.reference_date = e.date AND rr.reconciliation_type = 'Cybersource
Reconciliation'
);
-- Clean up temporary tables
DROP TABLE IF EXISTS erp_cybersource_sum, credit_card_sum;

COMMIT;

```

ECOM Orders Not in Oracle Reconciliation

This script identifies e-commerce orders that exist in the `ecom_orders` table but are not found in the `oracle_data` table. It inserts the results into the `reconciliation_results` table.

```

-- Step 1: Create necessary temporary tables for ECOM Orders not in Oracle
Reconciliation

-- Note: Assuming 'ecom_orders' and 'oracle_data' are existing tables and do not
need to be created temporarily.

-- Step 2: Insert into reconciliation_results table with duplicate check
INSERT INTO reconciliation_results (reconciliation_type, reference_id,
reconciliation_status, non_existent_record, recon_report)
SELECT
    'ECOM Orders not in Oracle' AS reconciliation_type,
    e.order_number AS reference_id,
    'Mismatch' AS reconciliation_status,
    1 AS non_existent_record,
    'Mismatch due to non-existent record' AS recon_report
FROM
    ecom_orders e
LEFT JOIN
    oracle_data o ON e.order_number = o.ecom_reference_order_number
WHERE
    o.ecom_reference_order_number IS NULL
AND NOT EXISTS (
    SELECT 1 FROM reconciliation_results rr
    WHERE rr.reference_id = e.order_number AND rr.reconciliation_type = 'ECOM
Orders not in Oracle'
);

```

```
COMMIT;
```

ECOM Orders Not in Shipping Reconciliation

This script identifies e-commerce orders that exist in the `ecom_orders` table but are not found in the `shippedandcollected_aramex_cosmaline` table. It inserts the results into the `reconciliation_results` table.

```
-- Step 1: Create necessary temporary tables for ECOM Orders not in Shipping
Reconciliation

-- Note: Assuming 'ecom_orders' and 'shippedandcollected_aramex_cosmaline' are
existing tables and do not need to be created temporarily.

-- Step 2: Insert into reconciliation_results table with duplicate check
INSERT INTO reconciliation_results (reconciliation_type, reference_id,
reconciliation_status, non_existent_record, recon_report)
SELECT
    'ECOM Orders not in Shipping' AS reconciliation_type,
    e.order_number AS reference_id,
    'Mismatch' AS reconciliation_status,
    1 AS non_existent_record,
    'Mismatch due to non-existent record' AS recon_report
FROM
    ecom_orders e
LEFT JOIN
    shippedandcollected_aramex_cosmaline a ON e.order_number = a.order_number
WHERE
    a.order_number IS NULL
AND NOT EXISTS (
    SELECT 1 FROM reconciliation_results rr
    WHERE rr.reference_id = e.order_number AND rr.reconciliation_type = 'ECOM
Orders not in Shipping'
);
COMMIT;
```

Invalid Oracle Order Numbers Reconciliation

This script identifies invalid Oracle order numbers by finding records in the `oracle_data` table that do not have a corresponding order number in the `ecom_orders` table. It inserts the results into the `reconciliation_results` table.

```
-- Step 1: Create necessary temporary tables for Invalid Oracle Order Numbers
Reconciliation

-- Note: Assuming 'oracle_data' and 'ecom_orders' are existing tables and do not
```

need to be created temporarily.

```
-- Step 2: Insert into reconciliation_results table with duplicate check
INSERT INTO reconciliation_results (reconciliation_type, reference_id,
reconciliation_status, non_existent_record, recon_report)
SELECT
    'Invalid Oracle Order Numbers' AS reconciliation_type,
    o.ecom_reference_order_number AS reference_id,
    'Mismatch' AS reconciliation_status,
    1 AS non_existent_record,
    'Mismatch due to non-existent record' AS recon_report
FROM
    oracle_data o
LEFT JOIN
    ecom_orders e ON o.ecom_reference_order_number = e.order_number
WHERE
    e.order_number IS NULL AND o.ecom_reference_order_number IS NOT NULL
AND NOT EXISTS (
    SELECT 1 FROM reconciliation_results rr
    WHERE rr.reference_id = o.ecom_reference_order_number AND
    rr.reconciliation_type = 'Invalid Oracle Order Numbers'
);
COMMIT;
```

Invalid Shipping Order Numbers Reconciliation

This script identifies invalid shipping order numbers by finding records in the `shippedandcollected_aramex_cosmaline` table that do not have a corresponding order number in the `ecom_orders` table. It inserts the results into the `reconciliation_results` table.

```
-- Step 1: Create necessary temporary tables for Invalid Shipping Order Numbers
Reconciliation

-- Note: Assuming 'shippedandcollected_aramex_cosmaline' and 'ecom_orders' are
existing tables and do not need to be created temporarily.

-- Step 2: Insert into reconciliation_results table with duplicate check
INSERT INTO reconciliation_results (reconciliation_type, reference_id,
reconciliation_status, non_existent_record, recon_report)
SELECT
    'Invalid Shipping Order Numbers' AS reconciliation_type,
    a.order_number AS reference_id,
    'Mismatch' AS reconciliation_status,
    1 AS non_existent_record,
    'Mismatch due to non-existent record' AS recon_report
FROM
    shippedandcollected_aramex_cosmaline a
LEFT JOIN
    ecom_orders e ON a.order_number = e.order_number
WHERE
```

```

e.order_number IS NULL AND a.order_number IS NOT NULL
AND NOT EXISTS (
    SELECT 1 FROM reconciliation_results rr
    WHERE rr.reference_id = a.order_number AND rr.reconciliation_type = 'Invalid
Shipping Order Numbers'
);

COMMIT;

```

4.8 The Email Notification Process

To manage the email notification process efficiently, create a Python file named `email_notifications.py` in the `scripts` folder. This file will contain the functions required to send email notifications concerning the status of the automation process to the stakeholders.

Functions to Format and Style the Email Notifications

Success Email Formatting

This function formats the email body for a success notification.

```

def format_success_email_body():
    return f"""
<html>
<body>
    <h2>ETL Automated Process Notification</h2>
    <p>The ETL process has completed successfully.</p>
    <p>All tasks have been executed without any issues.</p>
    <br>
    <p>Best Regards,</p>
    <p>Your ETL Automation System</p>
</body>
</html>
"""

```

Failure Email Formatting

This function formats the email body for a failure notification.

```

def format_failure_email_body(task_instance, task_status):
    return f"""
<html>
<body>
    <h2>ETL Automated Process Notification</h2>
    <p><strong>Task:</strong> {task_instance.task_id}</p>
    <p><strong>Status:</strong> {task_status}</p>
    <p><strong>DAG:</strong> {task_instance.dag_id}</p>
    <p><strong>Execution Date:</strong> {task_instance.execution_date}</p>

```

```

<p><strong>Log URL:</strong> <a href="{task_instance.log_url}">
{task_instance.log_url}</a></p>
<br>
<p>Best Regards,</p>
<p>Your ETL Automation System</p>
</body>
</html>
"""

```

Functions to Send the Email Notifications

Success Email Function

This function sends an email notification when the entire process succeeds using the formatted email body.

```

def success_email(context):
    subject = 'ETL Process Success'
    body = format_success_email_body()
    to_email = ['rar37@mail.aub.edu', 'hmf31@mail.aub.edu']
    send_email(to=to_email, subject=subject, html_content=body)

```

Failure Email Function

This function sends an email notification when a task fails using the formatted email body.

```

def failure_email(context):
    task_instance = context['task_instance']
    task_status = 'Failed'
    subject = f'Airflow Task {task_instance.task_id} {task_status}'
    body = format_failure_email_body(task_instance, task_status)
    to_email = ['rar37@mail.aub.edu', 'hmf31@mail.aub.edu']
    send_email(to=to_email, subject=subject, html_content=body)

```

4.9 Market Basket Analysis

In this section, we document the process of automating the Market Basket Analysis (MBA) using the Apriori and FP-Growth algorithms. The automation was achieved through a structured pipeline involving data extraction, transformation, and model generation. The results were then stored in a specified folder for further analysis.

1 - Folder Structure

To organize the automation scripts, a `models` folder was created within the `scripts` directory. Inside the `models` folder, a `data` subfolder was set up to hold the data to be analyzed by the MBA models. The scripts responsible for different parts of the analysis were also placed within the `models` folder.

- scripts/
 - models/
 - data/ (to hold the data to be analyzed)
 - generate_apriori.py (script to run the Apriori algorithm)
 - generate_fpgrowth.py (script to run the FP-Growth algorithm)
 - transform_pipeline_for_model.py (script containing the transformation pipeline)
 - extract.py (script for data extraction)
 - transform.py (script containing transformation functions)

2 - Data Extraction

The first step in the automation pipeline is data extraction. The data is extracted from the database using a query that joins relevant tables to obtain necessary fields like `product_name`, `product_category`, `ordered_quantity`, and `unit_selling_price`. The extraction function is defined in the `extract.py` script.

This function fetches the data and stores it in a Pandas DataFrame for subsequent processing:

```
def fetch_data_from_db_for_models():
    engine = create_db_engine()
    connection = engine.connect()

    query = """
SELECT
    oracle_data.operating_unit_name,
    oracle_data.ecom_reference_order_number,
    oracle_data.ordered_item,
    oracle_data.ordered_quantity,
    oracle_data.unit_selling_price,
    oracle_data.unit_list_price,
    oracle_data.ordered_date,
    oracle_data.tax_code,
    oracle_data_product_name.product_id,
    oracle_data_product_name.product_name,
    oracle_data_product_name.product_category
FROM
    oracle_data
JOIN
    oracle_data_product_name
ON
    oracle_data.ordered_item = oracle_data_product_name.product_id;
"""

    result = connection.execute(query)
    df = pd.DataFrame(result.fetchall(), columns=result.keys())

    connection.close()
    return df
```

3 - Data Transformation

After extraction, the data undergoes a series of transformations to prepare it for MBA. The transformation pipeline is defined in the `transform_pipeline_for_model.py` script, which calls several functions from the `transform.py` script.

```
def preprocessing_pipeline_for_model(df):
    df = convert_strings_to_lists(df)
    df = remove_product_not_found(df)
    df = apply_preprocess_category_list(df)
    df = convert_lists_to_strings(df)
    df = remove_higher_selling_price(df)
    df = calculate_total_sales(df)
    df = create_relevant_columns_df(df)
    df = remove_shipping_local(df)
    return df
```

The individual transformation functions are responsible for:

- `convert_strings_to_lists(df)`: Converts strings in the `product_category` column into lists.
- `remove_product_not_found(df)`: Removes rows where the product name is 'Product Not Found'.
- `apply_preprocess_category_list(df)`: Cleans the category list by removing special characters.
- `convert_lists_to_strings(df)`: Converts lists back to comma-separated strings.
- `remove_higher_selling_price(df)`: Removes rows where the selling price is higher than the list price.
- `calculate_total_sales(df)`: Calculates total sales with and without discounts.
- `create_relevant_columns_df(df)`: Selects relevant columns for the MBA process.
- `remove_shipping_local(df)`: Removes rows related to local shipping.
- `expand_column_to_rows(df, column_to_expand)`: Expands a column with multiple values into separate rows.

This pipeline ensures the data is clean, relevant, and in the correct format for the market basket analysis.

Individual Functions from the `transform.py` script:

```
def convert_strings_to_lists(df):
    df['product_category'] = df['product_category'].apply(lambda x:
ast.literal_eval(x))
    return df

def remove_product_not_found(df):
    df = df[df['product_name'] != 'Product Not Found']
    return df

def preprocess_category_list(category_list):
    return [re.sub(r'\.\(\)]', '', category) for category in category_list]

def apply_preprocess_category_list(df):
    df['product_category'] =
df['product_category'].apply(preprocess_category_list)
    return df
```

```

def convert_lists_to_strings(df):
    df['product_category'] = df['product_category'].apply(lambda x: ', '.join(x))
    if x else 'Unknown')
    return df

def remove_higher_selling_price(df):
    df = df[df['unit_selling_price'] <= df['unit_list_price']]
    return df

def calculate_total_sales(df):
    df['total_sales_with_discount'] = df['ordered_quantity'] *
    df['unit_selling_price']
    df['total_sales_without_discount'] = df['ordered_quantity'] *
    df['unit_list_price']
    return df

def create_relevant_columns_df(df):
    df = df[['ecom_reference_order_number', 'product_name']]
    return df

def remove_shipping_local(df):
    df = df[df['product_name'] != 'SHIPPING_LOCAL']
    return df

def calculate_discount_percentage(df, unit_list_price_col, unit_selling_price_col,
new_col_name='discount_percentage'):
    df[new_col_name] = (df[unit_list_price_col] - df[unit_selling_price_col]) /
    df[unit_list_price_col] * 100
    return df

def expand_column_to_rows(df, column_to_expand, delimiter=', ',
new_col_name=None):
    if new_col_name is None:
        new_col_name = column_to_expand

def expand_column_to_rows(df, column_to_expand, delimiter=', ',
new_col_name=None):
    if new_col_name is None:
        new_col_name = column_to_expand

    df_expanded = df.assign(**{new_col_name:
df[column_to_expand].str.split(delimiter)}).explode(new_col_name)
    return df_expanded

```

4- Model Generation

The core of the automation involves running the Apriori and FP-Growth algorithms to generate frequent itemsets and association rules.

Apriori Model

The `generate_apriori.py` script contains the logic for running the Apriori algorithm:

```

from mlxtend.frequent_patterns import apriori, association_rules
import pandas as pd

def generate_apriori_results(df):
    # Create a basket matrix (binary matrix)
    basket = (df
              .groupby(['ecom_reference_order_number', 'product_name'])
              ['product_name']
              .count().unstack().reset_index().fillna(0)
              .set_index('ecom_reference_order_number'))

    # Convert quantities to a binary format (purchased or not purchased)
    basket_sets = basket.applymap(lambda x: 1 if x > 0 else 0)

    # Apply the Apriori algorithm
    frequent_itemsets = apriori(basket_sets, min_support=0.01, use_colnames=True)

    # Generate association rules
    rules = association_rules(frequent_itemsets, metric="lift", min_threshold=1)

    # Filter rules based on support, confidence, and lift
    filtered_rules = rules[(rules['support'] >= 0.01) &
                           (rules['confidence'] >= 0.5) &
                           (rules['lift'] > 1)]

    # Convert frozenset to string
    filtered_rules['antecedents'] = filtered_rules['antecedents'].apply(lambda x: ', '.join(list(x)))
    filtered_rules['consequents'] = filtered_rules['consequents'].apply(lambda x: ', '.join(list(x)))

    return filtered_rules

```

This script creates a binary basket matrix from the transaction data, runs the Apriori algorithm, generates association rules, and filters them based on predefined thresholds for support, confidence, and lift.

FP-Growth

The `generate_fp_growth.py` script contains the logic for running the FP-Growth algorithm:

```

from mlxtend.frequent_patterns import fpgrowth, association_rules
import pandas as pd

def generate_fp_growth_results(df):
    # Prepare the data
    basket = (df
              .groupby(['ecom_reference_order_number', 'product_name'])
              ['product_name']
              .count().unstack().reset_index().fillna(0)
              .set_index('ecom_reference_order_number'))

```

```

basket_sets = basket.applymap(lambda x: 1 if x > 0 else 0)

# Apply the FP-Growth algorithm
min_support_threshold = 0.005
frequent_itemsets = fpgrowth(basket_sets, min_support=min_support_threshold,
use_colnames=True)

# Generate association rules
min_confidence_threshold = 0.5
rules = association_rules(frequent_itemsets, metric="confidence",
min_threshold=min_confidence_threshold)

# Filter rules based on lift
filtered_rules = rules[rules['lift'] > 1]

# Convert frozenset to string
filtered_rules['antecedents'] = filtered_rules['antecedents'].apply(lambda x:
', '.join(list(x)))
filtered_rules['consequents'] = filtered_rules['consequents'].apply(lambda x:
', '.join(list(x)))

return filtered_rules

```

4.10 Exploratory Data Analysis (EDA)

This section describes how the EDA was automated, including the data extraction, transformation, and generation of various statistical summaries and visualizations. These outputs provide a comprehensive understanding of the data, which informs subsequent modeling and decision-making processes.

1 - Folder Structure

A dedicated `visuals` folder was created within the `models` directory to store all the generated visualizations and statistical summaries. This ensures that all outputs from the EDA process are organized and easily accessible for further analysis or inclusion in reports.

- `scripts/`
 - `models/`
 - `visuals/` (to hold all the visualizations and statistics)
 - `visualization.py` (script containing visualization generation functions)
 - `transform_pipeline_for_models.py` (script containing the transformation pipeline for EDA)
- `extract.py` (script for data extraction)

2 - Data Extraction

The data used for the EDA was extracted using the same `fetch_data_from_db_for_models()` function as in the Market Basket Analysis section. This function connects to the database, executes a SQL query to fetch the relevant data, and returns the data as a Pandas DataFrame.

```

def fetch_data_from_db_for_models():
    engine = create_db_engine()

```

```

connection = engine.connect()

query = """
SELECT
    oracle_data.operating_unit_name,
    oracle_data.ecom_reference_order_number,
    oracle_data.ordered_item,
    oracle_data.ordered_quantity,
    oracle_data.unit_selling_price,
    oracle_data.unit_list_price,
    oracle_data.ordered_date,
    oracle_data.tax_code,
    oracle_data_product_name.product_id,
    oracle_data_product_name.product_name,
    oracle_data_product_name.product_category
FROM
    oracle_data
JOIN
    oracle_data_product_name
ON
    oracle_data.ordered_item = oracle_data_product_name.product_id;
"""

result = connection.execute(query)
df = pd.DataFrame(result.fetchall(), columns=result.keys())

connection.close()
return df

```

3 - Data Transformation

Once the data is extracted, it undergoes a transformation pipeline specifically designed for the EDA process. This pipeline is defined in the `transform_pipeline_for_models.py` script.

```

def preprocessing_pipeline_for_report(df):
    df = convert_strings_to_lists(df)
    df = remove_product_not_found(df)
    df = apply_preprocess_category_list(df)
    df = convert_lists_to_strings(df)
    df = remove_higher_selling_price(df)
    df = calculate_total_sales(df)
    df = calculate_discount_percentage(df, 'unit_list_price',
    'unit_selling_price')
    return df

```

Individual Functions:

```

def convert_strings_to_lists(df):
    df['product_category'] = df['product_category'].apply(lambda x:

```

```
ast.literal_eval(x))
    return df

def remove_product_not_found(df):
    df = df[df['product_name'] != 'Product Not Found']
    return df

def preprocess_category_list(category_list):
    return [re.sub(r'\.\(\)\\]', '', category) for category in category_list]

def apply_preprocess_category_list(df):
    df['product_category'] =
df['product_category'].apply(preprocess_category_list)
    return df

def convert_lists_to_strings(df):
    df['product_category'] = df['product_category'].apply(lambda x: ', '.join(x)
if x else 'Unknown')
    return df

def remove_higher_selling_price(df):
    df = df[df['unit_selling_price'] <= df['unit_list_price']]
    return df

def calculate_total_sales(df):
    df['total_sales_with_discount'] = df['ordered_quantity'] *
df['unit_selling_price']
    df['total_sales_without_discount'] = df['ordered_quantity'] *
df['unit_list_price']
    return df

def create_relevant_columns_df(df):
    df = df[['ecom_reference_order_number', 'product_name']]
    return df

def remove_shipping_local(df):
    df = df[df['product_name'] != 'SHIPPING_LOCAL']
    return df

def calculate_discount_percentage(df, unit_list_price_col, unit_selling_price_col,
new_col_name='discount_percentage'):
    df[new_col_name] = (df[unit_list_price_col] - df[unit_selling_price_col]) /
df[unit_list_price_col] * 100
    return df

def expand_column_to_rows(df, column_to_expand, delimiter=', ',
new_col_name=None):
    if new_col_name is None:
        new_col_name = column_to_expand

def expand_column_to_rows(df, column_to_expand, delimiter=', ',
new_col_name=None):
    if new_col_name is None:
        new_col_name = column_to_expand
```

```
df_expanded = df.assign(**{new_col_name:  
df[column_to_expand].str.split(delimiter)}).explode(new_col_name)  
return df_expanded
```

4 - Visualization Generation

The EDA process involves generating a wide range of visualizations and statistical summaries to explore different aspects of the data. These visualizations are generated using various functions defined in the `visualization.py` script. The primary function that orchestrates the creation of all visuals is `generate_all_visualizations()`.

```
def generate_all_visualizations(df, df_expanded, output_dir):  
    import os  
  
    # Ensure the output directory exists  
    if not os.path.exists(output_dir):  
        os.makedirs(output_dir)  
  
    # List of visualization functions using df  
    visualization_functions_df = [  
        (save_summary_statistics, df, 'summary_statistics.csv'),  
        (save_categorical_distribution, df, 'operating_unit_name',  
         'categorical_distribution.csv'),  
        (save_unique_categories_info, df, 'product_category',  
         'unique_categories_info.csv'),  
        (plot_top_products, df, 'top_products.png'),  
        (plot_monthly_sales_trend, df, 'monthly_sales_trend.png'),  
        (plot_order_quantity_distribution, df, 'order_quantity_distribution.png'),  
        (plot_monetary_distribution, df, 'monetary_distribution.png'),  
        (plot_recency_distribution, df, 'recency_distribution.png'),  
        (save_order_quantity_stats, df, 'order_quantity_stats.csv'),  
        (plot_items_per_order_distribution, df,  
         'items_per_order_distribution.png'),  
        (plot_price_distribution, df, 'price_distribution.png'),  
        (plot_discount_percentage_distribution, df,  
         'discount_percentage_distribution.png'),  
        (plot_discount_vs_ordered_quantity, df,  
         'discount_vs_ordered_quantity.png'),  
        (plot_average_discounts_comparison_hist, df,  
         'average_discounts_comparison_hist.png'),  
        (save_average_discounts_summary_stats, df,  
         'average_discounts_summary_stats.csv'),  
        (plot_top_distributors_treemap, df, 'top_distributors_treemap.png'),  
        (plot_transaction_diversity, df, 'transaction_diversity.png'),  
        (plot_item_frequencies, df, 'item_frequencies.png'),  
        (save_transaction_diversity_stats, df, 'transaction_diversity_stats.csv'),  
        (save_item_frequencies_stats, df, 'item_frequencies_stats.csv'),  
        (save_niche_products, df, 'niche_products.csv'),  
        (save_niche_purchase_frequency, df, 'niche_purchase_frequency.csv'),  
        (save_combined_niche_products, df, 'combined_niche_products.csv'),
```

```

        (save_combined_niche_products_count, df,
 'combined_niche_products_count.csv'),
    ]

# List of visualization functions using df_expanded
visualization_functions_df_expanded = [
    (plot_discount_percentage_by_category, df_expanded,
 'discount_percentage_by_category.png'),
    (plot_top_categories_unit_selling_prices, df_expanded,
 'top_categories_unit_selling_prices.png'),
    (get_top_categories, df_expanded, 'top_categories.csv'),
    (plot_category_sales, df_expanded, 'category_sales.png'),
]

# Call each visualization function with df
for func, *args in visualization_functions_df:
    file_name = args[-1]
    file_path = os.path.join(output_dir, file_name)
    func(*args[:-1], file_path) # Pass the args without the last element
(file_name) and append file_path

# Call each visualization function with df_expanded
for func, *args in visualization_functions_df_expanded:
    file_name = args[-1]
    file_path = os.path.join(output_dir, file_name)
    func(*args[:-1], file_path) # Pass the args without the last element
(file_name) and append file_path

```

Individual Functions:

```

def save_summary_statistics(df, file_path):
    summary_stats = df[['ordered_quantity', 'unit_selling_price',
'unit_list_price']].describe()
    with open(file_path, 'w') as f:
        f.write("Summary Statistics for Numerical Columns:\n")
        f.write(summary_stats.to_string())
    print(f"Summary statistics saved to {file_path}")

def save_categorical_distribution(df, column_name, file_path):
    categorical_dist = df[column_name].value_counts()
    with open(file_path, 'w') as f:
        f.write(f"Distribution of {column_name}:\n")
        f.write(categorical_dist.to_string())
    print(f"Categorical distribution saved to {file_path}")

def save_unique_categories_info(df, column_name, file_path):
    # Split the comma-separated strings into individual categories and create a
    set of unique categories
    unique_categories = set()
    df[column_name].apply(lambda x: unique_categories.update(x.split(', ')))

    # Count the number of unique categories

```

```
num_unique_categories = len(unique_categories)

# Save the results to a text file
with open(file_path, 'w') as f:
    f.write(f"Number of unique categories: {num_unique_categories}\n")
    f.write(f"Unique categories: {unique_categories}\n")

print(f"Unique categories information saved to {file_path}")

def plot_top_products(df, file_path):
    # Filter out "Product Not Found" and calculate the top 10 best-selling
    products
    top_products = df[df['product_name'] != 'Product Not Found']\
        .groupby('product_name')['ordered_quantity']\
        .sum().sort_values(ascending=False).head(10)

    # Plot the top 10 best-selling products
    plt.figure(figsize=(12, 6))
    top_products.plot(kind='bar')
    plt.title('Top 10 Best-Selling Products (Excluding "Product Not Found")')
    plt.xlabel('Product Name')
    plt.ylabel('Total Quantity Sold')
    plt.xticks(rotation=45, ha='right')
    plt.tight_layout() # Adjust layout to ensure everything fits without
    overlapping
    plt.savefig(file_path)
    plt.close()
    print(f"Top products plot saved to {file_path}")

def plot_monthly_sales_trend(df, file_path):
    # Ensure that the 'ordered_date' column is in datetime format
    if not pd.api.types.is_datetime64_any_dtype(df['ordered_date']):
        df['ordered_date'] = pd.to_datetime(df['ordered_date'])
    # Extract month and year from ordered_date
    df['year_month'] = df['ordered_date'].dt.to_period('M')

    # Aggregate sales by month
    monthly_sales = df.groupby('year_month')['total_sales_with_discount'].sum()

    # Convert the year_month to a string format for better readability
    monthly_sales.index = monthly_sales.index.strftime('%B %Y')

    # Plot the monthly sales trend as a horizontal bar chart
    plt.figure(figsize=(12, 6))
    monthly_sales.plot(kind='barh')
    plt.title('Bar Chart of Monthly Sales')
    plt.xlabel('Total Sales with Discount')
    plt.ylabel('Month')
    plt.grid(True)
    plt.tight_layout() # Adjust layout to ensure everything fits without
    overlapping
    plt.savefig(file_path)
    plt.close()
    print(f"Monthly sales trend plot saved to {file_path}")
```

```
def get_top_categories(df, file_path):
    # Group by product_category and sum the ordered_quantity
    category_quantity = df.groupby('product_category')
    ['ordered_quantity'].sum().reset_index()

    # Sort the categories by total quantity sold and select the top 10
    top_categories = category_quantity.sort_values(by='ordered_quantity',
    ascending=False).head(10)

    # Save the top categories to a CSV file
    top_categories.to_csv(file_path, index=False)
    print(f"Top categories information saved to {file_path}")

def plot_category_sales(df, file_path):
    # Calculate total sales by product category
    category_sales = df.groupby('product_category')
    ['total_sales_with_discount'].sum().sort_values(ascending=False)

    # Plot the revenue contribution by product category
    plt.figure(figsize=(12, 6))
    category_sales.plot(kind='bar')
    plt.title('Revenue Contribution by Product Category')
    plt.xlabel('Product Category')
    plt.ylabel('Total Sales')
    plt.xticks(rotation=45, ha='right')
    plt.tight_layout() # Adjust layout to ensure everything fits without
    overlapping
    plt.savefig(file_path)
    plt.close()
    print(f"Category sales plot saved to {file_path}")

def plot_order_quantity_distribution(df, file_path):
    # Plot distribution of order quantities
    plt.figure(figsize=(12, 6))
    df['ordered_quantity'].plot(kind='hist', bins=30)
    plt.title('Distribution of Order Quantities')
    plt.xlabel('Order Quantity')
    plt.ylabel('Frequency')

    # Set the x-axis ticks with increments of 5
    max_order_quantity = df['ordered_quantity'].max()
    plt.xticks(range(0, max_order_quantity + 1, 5))

    plt.tight_layout()
    plt.savefig(file_path)
    plt.close()
    print(f"Order quantity distribution plot saved to {file_path}")

def plot_monetary_distribution(df, file_path):
    # Calculate Monetary Value
    monetary = df.groupby('ecom_reference_order_number')
    ['total_sales_with_discount'].sum()
    rfm = pd.DataFrame({'monetary': monetary})
```

```
# Plot the Monetary Distribution
plt.figure(figsize=(12, 6))
sns.histplot(rfm, x='monetary', kde=True)
plt.title('Monetary Distribution')
plt.xlabel('Total Spend')
plt.ylabel('Frequency')
plt.tight_layout()
plt.savefig(file_path)
plt.close()
print(f"Monetary distribution plot saved to {file_path}")

def plot_recency_distribution(df, file_path):
    # Calculate Recency
    df['recency'] = (df['ordered_date'].max() - df['ordered_date']).dt.days

    rfm = pd.DataFrame({
        'recency': df.groupby('ecom_reference_order_number')['recency'].min()
    })

    # Plot Recency Distribution
    plt.figure(figsize=(12, 6))
    sns.histplot(rfm, x='recency', kde=True)
    plt.title('Recency Distribution')
    plt.xlabel('Days Since Last Purchase')
    plt.ylabel('Frequency')
    plt.tight_layout()
    plt.savefig(file_path)
    plt.close()
    print(f"Recency distribution plot saved to {file_path}")

def save_order_quantity_stats(df, file_path):
    # Group by ecom_reference_order_number and calculate the sum of
    # ordered_quantity for each order
    order_quantity_stats = df.groupby('ecom_reference_order_number')[['ordered_quantity']].sum()

    # Calculate summary statistics
    summary_stats = order_quantity_stats.describe()

    # Extract the specific statistics and create a DataFrame
    stats = {
        'Statistic': ['Average', 'Minimum', 'Maximum', 'Q1 (25th percentile)', 'Median (50th percentile)', 'Q3 (75th percentile)'],
        'Value': [
            summary_stats['mean'],
            summary_stats['min'],
            summary_stats['max'],
            summary_stats['25%'],
            summary_stats['50%'],
            summary_stats['75%']
        ]
    }
    stats_df = pd.DataFrame(stats)
```

```
# Save the results to a CSV file
stats_df.to_csv(file_path, index=False)
print(f"Order quantity statistics saved to {file_path}")

def plot_items_per_order_distribution(df, file_path):
    # Group by ecom_reference_order_number and calculate the sum of
    ordered_quantity for each order
    order_quantity_stats = df.groupby('ecom_reference_order_number')
    ['ordered_quantity'].sum()

    # Plotting the histogram
    plt.figure(figsize=(12, 6))
    order_quantity_stats.plot(kind='hist', bins=30, alpha=0.7)
    plt.title('Distribution of Items per Order')
    plt.xlabel('Number of Items per Order')
    plt.ylabel('Frequency')
    plt.tight_layout()
    plt.savefig(file_path)
    plt.close()
    print(f"Items per order distribution plot saved to {file_path}")

def plot_price_distribution(df, file_path):
    # Plot distribution of unit selling prices and unit list prices
    plt.figure(figsize=(12, 6))
    df['unit_selling_price'].plot(kind='hist', bins=20, alpha=0.5, label='Selling
Price')
    df['unit_list_price'].plot(kind='hist', bins=20, alpha=0.5, label='List
Price')
    plt.title('Distribution of Selling Prices and List Prices')
    plt.xlabel('Price')
    plt.ylabel('Frequency')
    plt.legend()
    plt.tight_layout()
    plt.savefig(file_path)
    plt.close()
    print(f"Price distribution plot saved to {file_path}")

def plot_discount_percentage_distribution(df, file_path):
    # Calculate discount percentage
    df['discount_percentage'] = (df['unit_list_price'] - df['unit_selling_price'])
    / df['unit_list_price'] * 100

    # Plot the distribution of discount percentages
    plt.figure(figsize=(12, 6))
    df['discount_percentage'].plot(kind='hist', bins=20)
    plt.title('Distribution of Discount Percentages')
    plt.xlabel('Discount Percentage')
    plt.ylabel('Frequency')
    plt.tight_layout()
    plt.savefig(file_path)
    plt.close()
    print(f"Discount percentage distribution plot saved to {file_path}")
```

```
def plot_discount_percentage_by_category(df_expanded, file_path):
    # Calculate the total sales per category
    category_quantity = df_expanded.groupby('product_category')[['ordered_quantity']].sum().reset_index()

    # Sort the categories by total sales
    sorted_categories = category_quantity.sort_values(by='ordered_quantity', ascending=False)[['product_category']]

    # Convert product_category to a categorical type with the sorted order
    df_expanded['product_category'] =
pd.Categorical(df_expanded['product_category'], categories=sorted_categories, ordered=True)

    # Plot the box plot with sorted categories
    plt.figure(figsize=(15, 8)) # Increase figure width
    sns.boxplot(x='product_category', y='discount_percentage', data=df_expanded, width=0.5)
    plt.title('Box Plot of Discount Percentages by Product Category')
    plt.xlabel('Product Category')
    plt.ylabel('Discount Percentage')
    plt.xticks(rotation=90, ha='right') # Rotate labels more and align to the right
    plt.tight_layout() # Adjust layout to prevent clipping
    plt.savefig(file_path)
    plt.close()
    print(f"Discount percentages by category box plot saved to {file_path}")

def plot_discount_vs_ordered_quantity(df, file_path):
    # Calculate the correlation
    correlation = df[['discount_percentage']].corr(df['ordered_quantity'])

    # Scatter plot of discount percentage vs. ordered quantity
    plt.figure(figsize=(12, 6))
    sns.scatterplot(x='discount_percentage', y='ordered_quantity', data=df, alpha=0.6)
    plt.title('Scatter Plot of Discount Percentage vs. Ordered Quantity')
    plt.xlabel('Discount Percentage')
    plt.ylabel('Ordered Quantity')
    plt.grid(True)

    # Add the correlation value to the plot
    plt.text(0.05, max(df['ordered_quantity']) * 0.95, f'Correlation: {correlation:.2f}', fontsize=12, color='red')

    plt.tight_layout()
    plt.savefig(file_path)
    plt.close()
    print(f"Scatter plot of discount percentage vs. ordered quantity saved to {file_path}")

def plot_average_discounts_comparison_hist(df, file_path_hist):
    # Calculate total quantity ordered for each item
    item_popularity = df.groupby('product_name')['ordered_quantity'].sum()
```

```
# Determine the median popularity
median_popularity = item_popularity.median()

# Categorize items into popular and less popular
df['popularity'] = df['product_name'].map(lambda x: 'Popular' if
item_popularity[x] > median_popularity else 'Less Popular')

# Calculate the average discount percentage for each item
average_discounts = df.groupby('product_name')['discount_percentage'].mean()
df['average_discount'] = df['product_name'].map(average_discounts)

# Create a DataFrame for comparison
comparison_df = df[['product_name', 'popularity',
'average_discount']].drop_duplicates()

# Histogram to compare average discounts
plt.figure(figsize=(12, 6))
sns.histplot(data=comparison_df, x='average_discount', hue='popularity',
element='step', stat='density', common_norm=False, bins=20)
plt.title('Comparison of Average Discounts for Popular and Less Popular
Items')
plt.xlabel('Average Discount Percentage')
plt.ylabel('Density')
plt.tight_layout()
plt.savefig(file_path_hist)
plt.close()
print(f"Histogram of average discounts comparison saved to {file_path_hist}")

def save_average_discounts_summary_stats(df, file_path_stats):
    # Calculate total quantity ordered for each item
    item_popularity = df.groupby('product_name')['ordered_quantity'].sum()

    # Determine the median popularity
    median_popularity = item_popularity.median()

    # Categorize items into popular and less popular
    df['popularity'] = df['product_name'].map(lambda x: 'Popular' if
item_popularity[x] > median_popularity else 'Less Popular')

    # Calculate the average discount percentage for each item
    average_discounts = df.groupby('product_name')['discount_percentage'].mean()
    df['average_discount'] = df['product_name'].map(average_discounts)

    # Create a DataFrame for comparison
    comparison_df = df[['product_name', 'popularity',
'average_discount']].drop_duplicates()

    # Print summary statistics for further insight
    summary_stats = comparison_df.groupby('popularity')[['average_discount']].describe()
    with open(file_path_stats, 'w') as f:
        f.write(summary_stats.to_string())
    print(f"Summary statistics saved to {file_path_stats}")
```

```
def plot_top_categories_unit_selling_prices(df_expanded, file_path):
    # Calculate total sales for each category
    df_expanded['total_sales_with_discount'] = df_expanded['ordered_quantity'] *
df_expanded['unit_selling_price']
    category_sales = df_expanded.groupby('product_category')
    ['total_sales_with_discount'].sum()

    # Identify top performing categories
    top_categories =
category_sales.sort_values(ascending=False).head(10).index.tolist()

    # Filter DataFrame to include only top performing categories
    top_categories_df =
df_expanded[df_expanded['product_category'].isin(top_categories)]

    # Plot the box plot for top performing categories with specified order
    plt.figure(figsize=(14, 8))
    sns.boxplot(x='product_category', y='unit_selling_price',
data=top_categories_df, order=top_categories)
    plt.title('Box Plot of Unit Selling Prices by Top Performing Product
Categories')
    plt.xlabel('Product Category')
    plt.ylabel('Unit Selling Price')
    plt.xticks(rotation=45)
    plt.tight_layout()
    plt.savefig(file_path)
    plt.close()
    print(f"Box plot of unit selling prices by top performing product categories
saved to {file_path}")

def plot_top_distributors_treemap(df, file_path):
    # Top distributing companies
    top_distributors = df.groupby('operating_unit_name')
    ['ordered_quantity'].sum().sort_values(ascending=False).head(10)

    # Create a list of colors in shades of blue and reverse it
    colors = plt.cm.Blues(range(0, 256, int(256/len(top_distributors))))
    colors = colors[::-1] # Reverse the colors

    # Create a treemap
    plt.figure(figsize=(12, 6))
    squarify.plot(sizes=top_distributors.values, label=top_distributors.index,
alpha=0.8, color=colors)
    plt.title('Top Distributing Companies by Quantity Shipped')
    plt.axis('off') # Remove axes
    plt.tight_layout()
    plt.savefig(file_path)
    plt.close()
    print(f"Treemap of top distributing companies saved to {file_path}")

def plot_transaction_diversity(df, file_path):
    # Create a basket matrix (binary matrix)
    basket = (df
```

```
.groupby(['ecom_reference_order_number', 'product_name'])
['product_name']
    .count().unstack().reset_index().fillna(0)
    .set_index('ecom_reference_order_number'))
basket_sets = basket.applymap(lambda x: 1 if x > 0 else 0)

# Count the number of unique items per transaction
unique_items_per_transaction = basket_sets.sum(axis=1)

# Plot the distribution of the number of items per transaction
plt.figure(figsize=(10, 6))
plt.hist(unique_items_per_transaction, bins=range(1,
unique_items_per_transaction.max() + 1), edgecolor='k')
plt.title('Distribution of the Number of Items per Transaction')
plt.xlabel('Number of Items per Transaction')
plt.ylabel('Frequency')
plt.tight_layout()
plt.savefig(file_path)
plt.close()
print(f"Distribution of the number of items per transaction plot saved to {file_path}")

def plot_item_frequencies(df, file_path):
    # Create a basket matrix (binary matrix)
    basket = (df
              .groupby(['ecom_reference_order_number', 'product_name'])
['product_name']
              .count().unstack().reset_index().fillna(0)
              .set_index('ecom_reference_order_number'))
    basket_sets = basket.applymap(lambda x: 1 if x > 0 else 0)

    # Count the frequency of each item across all transactions
    item_frequencies = basket_sets.sum(axis=0)

    # Plot the distribution of item frequencies
    plt.figure(figsize=(10, 6))
    plt.hist(item_frequencies, bins=range(1, item_frequencies.max() + 1),
edgecolor='k')
    plt.title('Distribution of Item Frequencies')
    plt.xlabel('Frequency of Items')
    plt.ylabel('Number of Items')
    plt.tight_layout()
    plt.savefig(file_path)
    plt.close()
    print(f"Distribution of item frequencies plot saved to {file_path}")

def save_transaction_diversity_stats(df, file_path):
    # Create a basket matrix (binary matrix)
    basket = (df
              .groupby(['ecom_reference_order_number', 'product_name'])
['product_name']
              .count().unstack().reset_index().fillna(0)
              .set_index('ecom_reference_order_number'))
    basket_sets = basket.applymap(lambda x: 1 if x > 0 else 0)
```

```
# Count the number of unique items per transaction
unique_items_per_transaction = basket_sets.sum(axis=1)

# Calculate basic statistics
stats = unique_items_per_transaction.describe()

# Convert statistics to DataFrame
stats_df = pd.DataFrame(stats, columns=[ 'Value' ]).reset_index()
stats_df.columns = [ 'Statistic', 'Value' ]

# Save the results to a CSV file
stats_df.to_csv(file_path, index=False)
print(f"Transaction diversity statistics saved to {file_path}")

def save_item_frequencies_stats(df, file_path):
    # Create a basket matrix (binary matrix)
    basket = (df
              .groupby(['ecom_reference_order_number', 'product_name'])
              ['product_name']
              .count().unstack().reset_index().fillna(0)
              .set_index('ecom_reference_order_number'))
    basket_sets = basket.applymap(lambda x: 1 if x > 0 else 0)

    # Count the frequency of each item across all transactions
    item_frequencies = basket_sets.sum(axis=0)

    # Calculate basic statistics
    stats = item_frequencies.describe()

    # Convert statistics to DataFrame
    stats_df = pd.DataFrame(stats, columns=[ 'Value' ]).reset_index()
    stats_df.columns = [ 'Statistic', 'Value' ]

    # Save the results to a CSV file
    stats_df.to_csv(file_path, index=False)
    print(f"Item frequency statistics saved to {file_path}")

def save_niche_products(df, file_path):
    # Calculate sales volume for each product
    sales_volume = df.groupby('product_name')
    ['ordered_quantity'].sum().reset_index()

    # Identify products with lower sales volumes
    niche_products = sales_volume[sales_volume['ordered_quantity'] <
sales_volume['ordered_quantity'].quantile(0.25)]

    # Save the niche products to a text file
    with open(file_path, 'w') as f:
        f.write("Potential Niche Products based on Sales Volume:\n")
        f.write(niche_products.to_string(index=False))
    print(f"Niche products information saved to {file_path}")

def save_niche_purchase_frequency(df, file_path):
```

```
# Calculate purchase frequency for each product
purchase_frequency = df.groupby('product_name')
['ecom_reference_order_number'].nunique().reset_index()

# Identify products with lower purchase frequency
niche_purchase_frequency =
purchase_frequency[purchase_frequency['ecom_reference_order_number'] <
purchase_frequency['ecom_reference_order_number'].quantile(0.25)]

# Save the niche purchase frequency products to a text file
with open(file_path, 'w') as f:
    f.write("Potential Niche Products based on Purchase Frequency:\n")
    f.write(niche_purchase_frequency.to_string(index=False))
print(f"Niche purchase frequency information saved to {file_path}")

def save_combined_niche_products(df, file_path):
    # Calculate sales volume for each product
    sales_volume = df.groupby('product_name')
    ['ordered_quantity'].sum().reset_index()
    niche_products = sales_volume[sales_volume['ordered_quantity'] <
sales_volume['ordered_quantity'].quantile(0.25)]

    # Calculate purchase frequency for each product
    purchase_frequency = df.groupby('product_name')
    ['ecom_reference_order_number'].nunique().reset_index()
    niche_purchase_frequency =
    purchase_frequency[purchase_frequency['ecom_reference_order_number'] <
purchase_frequency['ecom_reference_order_number'].quantile(0.25)]

    # Combine niche products based on different criteria
    combined_niche_products = set(niche_products['product_name']).intersection(
        set(niche_purchase_frequency['product_name']))
    )

    # Create a DataFrame for the combined niche products
    combined_niche_products_df = pd.DataFrame({
        'Product Name': list(combined_niche_products)
    })

    # Save the DataFrame to a CSV file
    combined_niche_products_df.to_csv(file_path, index=False)
    print(f"Combined niche products information saved to {file_path}")

def save_combined_niche_products_count(df, file_path):
    # Calculate sales volume for each product
    sales_volume = df.groupby('product_name')
    ['ordered_quantity'].sum().reset_index()
    niche_products = sales_volume[sales_volume['ordered_quantity'] <
sales_volume['ordered_quantity'].quantile(0.25)]

    # Calculate purchase frequency for each product
    purchase_frequency = df.groupby('product_name')
    ['ecom_reference_order_number'].nunique().reset_index()
    niche_purchase_frequency =
```

```

purchase_frequency[purchase_frequency[ 'ecom_reference_order_number' ] <
purchase_frequency[ 'ecom_reference_order_number' ].quantile(0.25)]

# Combine niche products based on different criteria
combined_niche_products = set(niche_products[ 'product_name' ]).intersection(
    set(niche_purchase_frequency[ 'product_name' ]))
)

# Count the number of combined niche products
num_combined_niche_products = len(combined_niche_products)

# Create a DataFrame for the combined niche products
combined_niche_products_df = pd.DataFrame({
    'Product Name': list(combined_niche_products)
})

# Add the count to the DataFrame
combined_niche_products_df[ 'Count' ] = num_combined_niche_products

# Save the DataFrame to a CSV file
combined_niche_products_df.to_csv(file_path, index=False)
print(f"Combined niche products count and information saved to {file_path}")

```

4.11 PDF Generator

To compile the extensive visualizations and statistics generated during the Exploratory Data Analysis (EDA) into a comprehensive report, a PDF report generator was implemented. This section describes the process of creating the PDF report, including the integration of images, tables, and text descriptions that provide context and insights for each visualization.

The `generate_pdf.py` script was created within the `models` folder to handle the PDF report generation. This script pulls data and `visuals` from the `visuals` folder and compiles them into a formatted PDF document.

`generate_pdf.py` script

```

from fpdf import FPDF
import os
import pandas as pd

class PDFReport(FPDF):
    def __init__(self, output_dir, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.output_dir = output_dir

    def header(self):
        if self.page_no() > 1: # Skip header on cover page
            self.image(os.path.join(self.output_dir, 'visuals/logos.png'), x=80,
y=10, w=50)
            self.set_font('Times', 'B', 12)
            self.cell(0, 20, '', 0, 1, 'C')
            self.ln(10)

```

```
def footer(self):
    self.set_y(-15)
    self.set_font('Times', 'I', 8)
    self.cell(0, 10, f'Page {self.page_no()}', 0, 0, 'C')

def cover_page(self):
    self.add_page()
    self.set_font('Times', 'B', 20)
    self.cell(0, 80, '', 0, 1, 'C') # Initial space before the title
    self.cell(0, 10, 'AUB x Malia Capstone E-commerce Performance Analysis',
0, 1, 'C')
        self.cell(0, 60, '', 0, 1, 'C') # Space between the title and "Created
by" statement
    self.set_font('Times', '', 12)
    self.cell(0, 10, 'Created by: Romanos Rizk and Hadil Fares', 0, 1, 'C')

def chapter_title(self, title):
    self.set_font('Times', 'B', 16)
    self.cell(0, 10, title, 0, 1, 'L')
    self.ln(5)

def chapter_body(self, body):
    self.set_font('Times', '', 12)
    self.multi_cell(0, 10, body)
    self.ln()

def add_image(self, image_path, title, description):
    self.chapter_title(title)
    self.chapter_body(description)
    self.image(image_path, w=180)
    self.ln(10)

def add_csv_as_table(self, file_path, title, description):
    self.chapter_title(title)
    self.chapter_body(description)
    df = pd.read_csv(file_path)
    col_widths = [max(self.get_string_width(str(value))) for value in df[col]
+ 4 for col in df.columns]
    col_widths = [max(width, 40) for width in col_widths] # Set a minimum
column width
    self.set_font('Times', 'B', 10)
    for col, width in zip(df.columns, col_widths):
        self.cell(width, 10, col, border=1)
    self.ln()
    self.set_font('Times', '', 10)
    for i in range(len(df)):
        for value, width in zip(df.iloc[i], col_widths):
            self.cell(width, 10, str(value), border=1)
        self.ln()
    self.ln()

def add_csv_as_paragraph(self, file_path, title, description):
    self.chapter_title(title)
```

```
        self.chapter_body(description)
        df = pd.read_csv(file_path)
        products = df['Product Name'].tolist()
        paragraph = ', '.join(products)
        self.chapter_body(paragraph)

def generate_pdf_report(output_dir, pdf_path):
    pdf = PDFReport(output_dir=output_dir)

    # Set the correct directory for images
    image_dir = os.path.join(output_dir, 'visuals')
    # Add Cover Page
    pdf.cover_page()

    # Section 1: Sales Analysis
    pdf.add_page()
    pdf.chapter_title('Section 1: Sales Analysis')
    pdf.add_image(os.path.join(image_dir, 'top_products.png'), 'Top 10 Best-Selling Products',
                 "The 'Top 10 Best-Selling Products' plot shows the products with the highest sales volume. This visualization highlights the top-performing products in terms of quantity sold, providing insights into customer preferences and popular items.")
    pdf.add_image(os.path.join(image_dir, 'monthly_sales_trend.png'), 'Monthly Sales Trend',
                 "The 'Monthly Sales Trend' plot displays the total sales revenue for each month. It helps in understanding the sales performance over time, identifying peak sales periods, and evaluating seasonal trends or patterns.")
    pdf.add_csv_as_table(os.path.join(image_dir, 'top_categories.csv'), 'Top Categories by Sales Volume',
                         "The 'Top Categories by Sales Volume' table ranks product categories by their total sales volume. This information is useful for understanding which categories are most popular and contribute most significantly to overall sales.")
    pdf.add_image(os.path.join(image_dir, 'category_sales.png'), 'Revenue Contribution by Product Category',
                 "The 'Revenue Contribution by Product Category' plot shows the percentage of total revenue generated by each product category. It highlights the most financially significant categories and helps identify which ones drive the most revenue.")
    pdf.add_image(os.path.join(image_dir, 'order_quantity_distribution.png'), 'Distribution of Order Quantities',
                 "The 'Distribution of Order Quantities' plot is a histogram that visualizes the frequency of various order quantities placed by customers. It provides insight into purchasing behavior by showing how many items are typically ordered in a single transaction. This histogram helps identify common order sizes, revealing whether customers typically buy single items or larger quantities.")
    pdf.add_image(os.path.join(image_dir, 'monetary_distribution.png'), 'Monetary Distribution',
                 "The 'Monetary Distribution' plot is a histogram that displays the distribution of total spending per order, showing the frequency of different spending levels by customers. It helps to understand the monetary value of transactions, indicating how much customers typically spend in a single purchase.")
```

```
# Section 2: Order Analysis
pdf.add_page()
pdf.chapter_title('Section 2: Order Analysis')
pdf.add_image(os.path.join(image_dir, 'recency_distribution.png'), 'Recency Distribution',
    "The 'Recency Distribution' plot is a histogram that illustrates the distribution of the number of days since customers last made a purchase. This plot provides insight into customer engagement by showing how recently customers have interacted with the business. The x-axis represents the number of days since the last purchase, while the y-axis shows the frequency of customers falling into each recency category. This analysis helps identify the proportion of recent versus infrequent customers, which can be useful for tailoring marketing and retention strategies.")
pdf.add_csv_as_table(os.path.join(image_dir, 'order_quantity_stats.csv'),
'Order Quantity Statistics',
    "The 'Order Quantity Statistics' table presents summary statistics for the quantity of items ordered per transaction. It provides key metrics such as the average, minimum, maximum, median, and quartiles (Q1 and Q3) of the total items ordered in each transaction. This table offers a comprehensive overview of order sizes, highlighting typical order quantities, the range of quantities ordered, and the distribution's central tendency and spread. This information can help in understanding customer purchasing patterns.")
pdf.add_image(os.path.join(image_dir, 'items_per_order_distribution.png'),
'Distribution of Items per Order',
    "The 'Distribution of Items per Order' plot is a histogram that visualizes the frequency of different quantities of items ordered in a single transaction. It shows how many transactions include specific numbers of items, providing insight into typical order sizes. This plot helps identify common order sizes, whether customers often purchase single items or multiple items, and the overall distribution of items per transaction.")

# Section 3: Pricing and Discount Analysis
pdf.add_page()
pdf.chapter_title('Section 3: Pricing and Discount Analysis')
pdf.add_image(os.path.join(image_dir, 'price_distribution.png'), 'Distribution of Selling Prices and List Prices',
    "The 'Distribution of Selling Prices and List Prices' plot is a histogram that shows the frequency distribution of both the selling prices and list prices of products. The plot helps to understand the range and distribution of prices at which products are sold (selling price) compared to their original or undiscounted prices (list price).")
pdf.add_image(os.path.join(image_dir, 'discount_percentage_distribution.png'),
'Distribution of Discount Percentages',
    "The 'Distribution of Discount Percentages' plot is a histogram that displays the frequency distribution of discount percentages applied to products. It provides insight into the range and prevalence of discounts offered on products. This plot helps to identify common discount levels and the extent of price reductions across the product range.")
pdf.add_image(os.path.join(image_dir, 'discount_percentage_by_category.png'),
'Discount Percentage by Category',
    "The 'Discount Percentage by Category' plot shows the variation in discount percentages across different product categories. This visualization helps to understand how discounting strategies differ between categories and which
```

```
categories offer higher or lower discounts.")  
    pdf.add_image(os.path.join(image_dir, 'discount_vs_ordered_quantity.png')),  
    'Scatter Plot of Discount Percentage vs. Ordered Quantity',  
        "The 'Scatter Plot of Discount Percentage vs. Ordered Quantity' shows the relationship between the discount percentage applied to products and the quantity ordered. It helps to analyze whether higher discounts correlate with higher sales volumes, providing insights into the effectiveness of discount strategies.")  
    pdf.add_image(os.path.join(image_dir,  
    'average_discounts_comparison_hist.png'), 'Comparison of Average Discounts for Popular and Less Popular Items',  
        "The 'Comparison of Average Discounts for Popular and Less Popular Items' plot is a histogram that compares the average discount percentages applied to popular and less popular items. The items are categorized based on their sales volume, with 'popular' items having sales above the median and 'less popular' items below. This plot helps to understand how discount strategies vary based on product popularity, showing the distribution of average discounts for each category. It provides insight into whether popular or less popular items receive larger discounts on average.")  
    pdf.add_image(os.path.join(image_dir,  
    'top_categories_unit_selling_prices.png'), 'Box Plot of Unit Selling Prices by Top Performing Product Categories',  
        "The 'Box Plot of Unit Selling Prices by Top Performing Product Categories' visualizes the range and distribution of unit selling prices across the top-performing product categories. This plot provides insight into the pricing strategies within these categories, highlighting price variations and the median price point.")  
  
# Section 4: Supply Chain Analysis  
pdf.add_page()  
pdf.chapter_title('Section 4: Supply Chain Analysis')  
pdf.add_image(os.path.join(image_dir, 'top_distributors_treemap.png'), 'Top Distributing Companies by Quantity Shipped',  
    "The 'Top Distributing Companies by Quantity Shipped' treemap visualizes the quantity of items shipped by each distributor. It provides a clear view of the most active distribution companies, showing their relative contribution to the total shipments. This visualization helps to understand the role and impact of different distributors in the supply chain.")  
  
# Section 5: Product Positioning  
pdf.add_page()  
pdf.chapter_title('Section 5: Product Positioning')  
pdf.add_image(os.path.join(image_dir, 'transaction_diversity.png'),  
    'Transaction Diversity',  
        "The 'Transaction Diversity' plot shows the diversity in transactions by displaying the number of unique items purchased in each transaction. It helps to understand whether customers are buying a wide variety of products or focusing on a few specific items.")  
    pdf.add_image(os.path.join(image_dir, 'item_frequencies.png'), 'Distribution of Item Frequencies',  
        "The 'Distribution of Item Frequencies' plot shows how frequently each product is purchased. This plot provides insight into product popularity and helps identify which products are commonly bought together.")  
    pdf.add_csv_as_table(os.path.join(image_dir,
```

```
'transaction_diversity_stats.csv'), 'Transaction Diversity Statistics',
    "The 'Transaction Diversity Statistics' table presents
summary statistics regarding the number of unique items purchased per transaction.
It provides key metrics such as the mean, median, minimum, maximum, and quartiles
(Q1 and Q3) of the unique items in each transaction. This table offers an overview
of how varied customer purchases are in terms of product diversity within a single
transaction.")
pdf.add_csv_as_table(os.path.join(image_dir, 'item_frequencies_stats.csv'),
'Item Frequency Statistics',
    "The 'Item Frequency Statistics' table provides summary
statistics on how frequently each item is purchased across all transactions. It
includes key metrics such as the mean, median, minimum, maximum, and quartiles (Q1
and Q3) for the frequency of item purchases. This table offers insights into the
popularity of different products, highlighting how often items are bought on
average, the range of purchase frequencies, and the distribution of these
frequencies.")
pdf.add_csv_as_paragraph(os.path.join(image_dir,
'combined_niche_products.csv'), 'Considered as Niche Products',
    "The 'Considered as Niche Products' section lists
products identified as niche based on specific criteria. These products are less
frequently purchased but may cater to specific customer segments or interests.
Understanding niche products helps in targeting niche markets and optimizing
inventory. Below is a block containing all Niche Products")

# Output the PDF to a file
pdf.output(pdf_path)
print('PDF report created successfully.')
```

Funciton to send the PDF by email:

```
def send_sales_report_email(report_path):
    subject = 'Sales Report'
    body = """
<html>
<body>
    <h2>Sales Report</h2>
    <p>Please find attached the latest sales report.</p>
    <br>
    <p>Best Regards,</p>
    <p>Your ETL Automation System</p>
</body>
</html>
"""

    to_email = ['rar37@mail.aub.edu', 'hmf31@mail.aub.edu']
    attachments = [report_path]

    msg = MIMEMultipart()
    msg['Subject'] = subject
    msg['From'] = 'youremail@example.com'
    msg['To'] = ', '.join(to_email)
    msg.attach(MIMEText(body, 'html'))
```

```

# Attach the sales report PDF
for attachment in attachments:
    part = MIMEBase('application', "octet-stream")
    with open(attachment, 'rb') as file:
        part.set_payload(file.read())
    encoders.encode_base64(part)
    part.add_header('Content-Disposition', 'attachment',
filename=os.path.basename(attachment))
    msg.attach(part)

send_email(to=to_email, subject=subject, html_content=body, files=
[report_path])

```

4.12 Upload Data to Github

In this section, we will detail how the `oracle_data` table is being pushed to github so that the user can access the laters data in the streamlit application (which will be discussed later)

An `upload_to_github.py` script was created in the `models` folder:

```

import base64
import requests

def upload_files_to_github(files, repo, token, message="Update files"):
    for local_path, repo_path in files.items():
        url = f"https://api.github.com/repos/{repo}/contents/{repo_path}"

        # Get the SHA of the existing file, if it exists
        response = requests.get(url, headers={"Authorization": f"token {token}"})
        if response.status_code == 200:
            sha = response.json()['sha']
        else:
            sha = None

        with open(local_path, "rb") as f:
            content = base64.b64encode(f.read()).decode()

        # Check if the content has changed
        if sha:
            existing_content = requests.get(url, headers={"Authorization": f"token {token}"})\
                .json()['content']
            if content == existing_content:
                print(f"No changes detected in {local_path}. Skipping upload.")
                continue

        data = {
            "message": message,
            "content": content,
            "branch": "main",
        }

```

```
if sha:
    data["sha"] = sha

headers = {
    "Authorization": f"token {token}",
    "Content-Type": "application/json"
}

response = requests.put(url, json=data, headers=headers)
response.raise_for_status()
```

4.13 ETL Task Functions

In this section, we will detail the task functions for each dataframe. Each subsection will cover the specific task functions used to perform the ETL steps (Extract, Transform, Load) for a particular dataframe. These functions will be created in an `etl_functions.py` file situated in the `scripts` folder and where the transformation pipelines will be imported from the `transform_pipelines.py` script

```
import pandas as pd
from datetime import datetime
import sys
import os
from sqlalchemy import inspect

from extract import read_data_from_excel, fetch_data_from_db_for_models
from transform import expand_column_to_rows
from transform_pipelines import (concatenate_Aramex_Cosmaline,
                                  convert_data_types_arandex_cosmaline,
                                  data_preprocessing_pipeline_ECOM_Data,
                                  data_preprocessing_pipeline_Aramex_Data,
                                  data_preprocessing_pipeline_Cosmaline_Data,
                                  data_preprocessing_pipeline_Cosmaline_Aramex_Data,
                                  data_preprocessing_pipeline_CreditCard_Data,
                                  data_preprocessing_pipeline_ERP_Data,
                                  data_preprocessing_pipeline_Oracle_Data)
from load import create_db_engine, upload_dataframe_to_temp_sql,
merge_data_from_temp_to_main_with_pk,
merge_data_from_temp_to_main_without_pk,cleanup_temp_tables
from reconciliation import execute_reconciliation_script
from email_notifications import success_email, failure_email
from models.generate_apriori import generate_apriori_results
from models.generate_fpgrowth import generate_fpgrowth_results
from models.transform_pipelines_for_model import preprocessing_pipeline_for_model,
preprocessing_pipeline_for_report
from models.visualizations import generate_all_visualizations
from models.generate_pdf import generate_pdf_report
from models.upload_to_github import upload_files_to_github
```

1 - **ecom_orders**:

The ECOM Data involves extracting order information from an Excel sheet, transforming the data to ensure consistency and accuracy, and finally loading the processed data into the database. The following subsections explain the ETL functions used for this datasource:

Extract ECOM Data This function reads the ECOM data from an Excel file and saves it as a CSV file.

```
def extract_ecom_data():
    df = read_data_from_excel('/opt/airflow/data/Final File.xlsx',
    sheet_name='Website ECOM Data')
    df.to_csv('/opt/airflow/data/ecom_data.csv', index=False)
```

Transform ECOM Data This function reads the CSV file, applies the data preprocessing pipeline, and saves the transformed data as a Parquet file.

```
def transform_ecom_data():
    df = pd.read_csv('/opt/airflow/data/ecom_data.csv')
    df = data_preprocessing_pipeline_ECOM_Data(df)
    df.to_parquet('/opt/airflow/data/transformed_ecom_data.parquet', index=False)
```

Load ECOM Data This function reads the transformed data from the Parquet file, uploads it to a temporary SQL table, and merges it into the main **ECOM_orders** table.

```
def load_ecom_data():
    engine = create_db_engine()
    df = pd.read_parquet('/opt/airflow/data/transformed_ecom_data.parquet')
    upload_dataframe_to_temp_sql(df, 'temp_ecom_orders', engine)
    merge_data_from_temp_to_main_with_pk('temp_ecom_orders', 'ECOM_orders',
    engine)
```

2 - **shippedandcollected_aramex**:

The Aramex Data involves extracting shipment information from an Excel sheet, transforming the data to ensure consistency and accuracy, and preparing it for further processing. The following subsections explain the functions used this datasource:

Extract Aramex Data This function reads the Aramex data from an Excel file and saves it as a CSV file.

```
def extract_aramex_data():
    df = read_data_from_excel('/opt/airflow/data/Final File.xlsx',
    sheet_name='Shipped & Collected - Aramex')
    df.to_csv('/opt/airflow/data/aramex_data.csv', index=False)
```

Transform Aramex Data This function reads the CSV file, merges it with the ECOM data, applies the data preprocessing pipeline, and saves the transformed data as a Parquet file.

```
def transform_aramex_data():
    df = pd.read_csv('/opt/airflow/data/aramex_data.csv')
    ecom_data = pd.read_parquet('/opt/airflow/data/transformed_ecom_data.parquet')
    df = data_preprocessing_pipeline_Aramex_Data(df, ecom_data)
    df.to_parquet('/opt/airflow/data/transformed_aramex_data.parquet',
    index=False)
```

Note: there is no load task here for Aramex because it will later be combined with Cosmaline data before being loaded into the database as a concatenated table.

3 - shippedandcollected_cosmaline:

The Cosmaline Data involves extracting shipment information from an Excel sheet, transforming the data to ensure consistency and accuracy, and preparing it for further processing. The following subsections explain the task functions used for this datasource:

Extract Cosmaline Data This function reads the Cosmaline data from an Excel file and saves it as a CSV file.

```
def extract_cosmaline_data():
    df = read_data_from_excel('/opt/airflow/data/Final File.xlsx',
    sheet_name='Shipped & Collected - Cosmaline')
    df.to_csv('/opt/airflow/data/cosmaline_data.csv', index=False)
```

Transform Cosmaline Data This function reads the CSV file, merges it with the ECOM data, applies the data preprocessing pipeline, and saves the transformed data as a Parquet file.

```
def transform_cosmaline_data():
    df = pd.read_csv('/opt/airflow/data/cosmaline_data.csv')
    ecom_data = pd.read_parquet('/opt/airflow/data/transformed_ecom_data.parquet')
    df = data_preprocessing_pipeline_Cosmaline_Data(df, ecom_data)
    df.to_parquet('/opt/airflow/data/transformed_cosmaline_data.parquet',
    index=False)
```

Note: there is no load task here for Cosmaline because it will later be combined with Aramex data before being loaded into the database as a concatenated table.

4 - shippedandcollected_aramex_cosmaline:

The concatenated table for Aramex and Cosmaline data involves combining the transformed shipment information from both sources, ensuring consistency and accuracy, and then loading the processed data into the database. The following subsections explain the task functions used for this dataframe:

Concatenate Aramex and Cosmaline Data This function reads the transformed data from Aramex and Cosmaline Parquet files, concatenates them, and saves the combined data as a new Parquet file.

```
def concatenate_aramex_cosmaline():
    aramex_data =
        pd.read_parquet('/opt/airflow/data/transformed_aramex_data.parquet')
    cosmaline_data =
        pd.read_parquet('/opt/airflow/data/transformed_cosmaline_data.parquet')
    aramex_data = convert_data_types_aramex_cosmaline(aramex_data)
    cosmaline_data = convert_data_types_aramex_cosmaline(cosmaline_data)
    combined_df = concatenate_Aramex_Cosmaline(aramex_data, cosmaline_data)
    combined_df.to_parquet('/opt/airflow/data/aramex_cosmaline_data.parquet',
                           index=False)
```

Load Concatenated Data This function reads the concatenated data from the Parquet file, uploads it to a temporary SQL table, and merges it into the main shippedandcollected_aramex_cosmaline table.

```
def load_concatenated_data():
    engine = create_db_engine()
    combined_df =
        pd.read_parquet('/opt/airflow/data/aramex_cosmaline_data.parquet')
        upload_dataframe_to_temp_sql(combined_df,
        'temp_shippedandcollected_aramex_cosmaline', engine)

    merge_data_from_temp_to_main_with_pk('temp_shippedandcollected_aramex_cosmaline',
    'shippedandcollected_aramex_cosmaline', engine)
```

5 - credit_card:

The Credit Card Data involves extracting transaction information from an Excel sheet, transforming the data to ensure consistency and accuracy, and finally loading the processed data into the database. The following subsections explain the task functions used for this datasource:

Extract Credit Card Data This function reads the credit card data from an Excel file and saves it as a CSV file.

```
def extract_credit_card_data():
    df = read_data_from_excel('/opt/airflow/data/Final File.xlsx',
    sheet_name='Collected - Credit Card')
    df.to_csv('/opt/airflow/data/credit_card_data.csv', index=False)
```

Transform Credit Card Data This function reads the CSV file, applies the data preprocessing pipeline, and saves the transformed data as a Parquet file.

```
def transform_credit_card_data():
    df = pd.read_csv('/opt/airflow/data/credit_card_data.csv')
```

```
df = data_preprocessing_pipeline_CreditCard_Data(df)
df.to_parquet('/opt/airflow/data/transformed_credit_card_data.parquet',
index=False)
```

Load Credit Card Data This function reads the transformed data from the Parquet file, uploads it to a temporary SQL table, and merges it into the main credit_card table while ensuring no duplicates.

```
def load_credit_card_data():
    engine = create_db_engine()
    df = pd.read_parquet('/opt/airflow/data/transformed_credit_card_data.parquet')
    upload_dataframe_to_temp_sql(df, 'temp_credit_card', engine)
    unique_columns = ['value_date', 'narrative', 'amount', 'currency']
    merge_data_from_temp_to_main_without_pk('temp_credit_card', 'credit_card',
    engine, unique_columns)
```

6 - erp_data:

The ERP Data involves extracting collection information from an Excel sheet, transforming the data to ensure consistency and accuracy, and finally loading the processed data into the database. The following subsections explain the task functions used for this datasource:

Extract ERP Data This function reads the ERP data from an Excel file and saves it as a CSV file.

```
def extract_erp_data():
    df = read_data_from_excel('/opt/airflow/data/Final File.xlsx',
    sheet_name='ERP-Oracle Collection')
    df.to_csv('/opt/airflow/data/erp_data.csv', index=False)
```

Transform ERP Data This function reads the CSV file, applies the data preprocessing pipeline, and saves the transformed data as a Parquet file.

```
def transform_erp_data():
    df = pd.read_csv('/opt/airflow/data/erp_data.csv')
    df = data_preprocessing_pipeline_ERP_Data(df)
    df.to_parquet('/opt/airflow/data/transformed_erp_data.parquet', index=False)
```

Load ERP Data This function reads the transformed data from the Parquet file, uploads it to a temporary SQL table, and merges it into the main erp_data table.

```
def load_erp_data():
    engine = create_db_engine()
    df = pd.read_parquet('/opt/airflow/data/transformed_erp_data.parquet')
    upload_dataframe_to_temp_sql(df, 'temp_erp_data', engine)
    merge_data_from_temp_to_main_with_pk('temp_erp_data', 'erp_data', engine)
```

7 - oracle_data:

The Oracle Data involves extracting order information from an Excel sheet, transforming the data to ensure consistency and accuracy, and finally loading the processed data into the database. The following subsections explain the task functions used for this datasource:

Extract Oracle Data This function reads the Oracle data from an Excel file and saves it as a CSV file.

```
def extract_oracle_data():
    df = read_data_from_excel('/opt/airflow/data/Final File.xlsx',
    sheet_name='Oracle Data')
    df.to_csv('/opt/airflow/data/oracle_data.csv', index=False)
```

Transform Oracle Data This function reads the CSV file, applies the data preprocessing pipeline, and saves the transformed data as a Parquet file.

```
def transform_oracle_data():
    df = pd.read_csv('/opt/airflow/data/oracle_data.csv')
    df = data_preprocessing_pipeline_Oracle_Data(df)
    df.to_parquet('/opt/airflow/data/transformed_oracle_data.parquet',
    index=False)
```

Load Oracle Data This function reads the transformed data from the Parquet file, uploads it to a temporary SQL table, and merges it into the main oracle_data table while ensuring no duplicates.

```
def load_oracle_data():
    engine = create_db_engine()
    df = pd.read_parquet('/opt/airflow/data/transformed_oracle_data.parquet')
    upload_dataframe_to_temp_sql(df, 'temp_oracle_data', engine)
    unique_columns = ['operating_unit_name', 'oracle_reference_order_number',
    'ecom_reference_order_number', 'order_type', 'order_type_name', 'ordered_item',
    'ordered_quantity', 'pricing_quantity_uom', 'unit_selling_price',
    'unit_list_price', 'ordered_date', 'customer_name', 'tax_code']
    merge_data_from_temp_to_main_without_pk('temp_oracle_data', 'oracle_data',
    engine, unique_columns)
```

8 - daily_rate:

The Daily Rate Data involves extracting exchange rate information from an Excel sheet and loading the processed data into the database. This data does not require any transformation steps. The following subsections explain the task functions used for this datasource:

Extract Daily Rate Data This function reads the daily rate data from an Excel file and saves it as a CSV file.

```
def extract_daily_rate_data():
    df = read_data_from_excel('/opt/airflow/data/Final File.xlsx',
    sheet_name='Daily Rate')
    df.to_csv('/opt/airflow/data/daily_rate_data.csv', index=False)
```

Load Daily Rate Data This function reads the data from the CSV file, uploads it to a temporary SQL table, and merges it into the main daily_rate table while ensuring no duplicates.

```
def load_daily_rate_data():
    engine = create_db_engine()
    df = pd.read_csv('/opt/airflow/data/daily_rate_data.csv')
    df.columns = [col.lower() for col in df.columns]
    upload_dataframe_to_temp_sql(df, 'temp_daily_rate', engine)
    inspector = inspect(engine)
    main_table_columns = [column['name'] for column in
    inspector.get_columns('daily_rate') if column['name'] != 'id']
    unique_columns = [col for col in df.columns if col in main_table_columns]
    merge_data_from_temp_to_main_without_pk('temp_daily_rate', 'daily_rate',
    engine, unique_columns)
```

9 - Reconciliation and Clean up functions

The reconciliation process in this ETL pipeline involves executing a series of SQL scripts to reconcile data across different sources. Each reconciliation function is responsible for executing a specific SQL script that performs the reconciliation task. The following sections provide the details of these functions.

```
def execute_cosmaline_reconciliation():
    execute_reconciliation_script('cosmaline_reconciliation.sql')

def execute_credit_card_reconciliaiton():
    execute_reconciliation_script('credit_card_reconciliaiton.sql')

def execute_ecom_orders_not_in_oracle_reconciliation():
    execute_reconciliation_script('ecom_orders_not_in_oracle_reconciliation.sql')

def execute_ecom_orders_not_in_shipping_reconciliation():
    execute_reconciliation_script('ecom_orders_not_in_shipping_reconciliation.sql')

def execute_ecom_reconciliation():
    execute_reconciliation_script('ecom_reconciliation.sql')

def execute_invalid_oracle_order_numbers_reconciliation():
    execute_reconciliation_script('invalid_oracle_order_numbers_reconciliation.sql')

def execute_invalid_shipping_order_numbers_reconciliation():
    execute_reconciliation_script('invalid_shipping_order_numbers_reconciliation.sql')
```

```
execute_reconciliation_script('invalid_shipping_order_numbers_reconciliation.sql')

def execute_token_reconciliation():
    execute_reconciliation_script('token_reconciliation.sql')

def cleanup_temp_tables_task():
    engine = create_db_engine()
    cleanup_temp_tables(engine)
```

10 - Business Analysis

Data Mining Models:

```
def extract_data_for_models():
    df = fetch_data_from_db_for_models()
    df.to_csv('/opt/airflow/data/data_for_model.csv')

def transform_data_for_model():
    df = pd.read_csv('/opt/airflow/data/data_for_model.csv')
    df = preprocessing_pipeline_for_model(df)
    df.to_parquet('/opt/airflow/data/data_for_model.parquet', index=False)

def generate_and_save_apriori_results():
    df = pd.read_parquet('/opt/airflow/data/data_for_model.parquet')
    apriori_results = generate_apriori_results(df)
    apriori_results.to_parquet('/opt/airflow/data/apriori_results.parquet',
index=False)

def generate_and_save_fpgrowth_results():
    df = pd.read_parquet('/opt/airflow/data/data_for_model.parquet')
    fpgrowth_results = generate_fpgrowth_results(df)
    fpgrowth_results.to_parquet('/opt/airflow/data/fpgrowth_results.parquet',
index=False)

def sanitize_column_names(df):
    """
    Rename DataFrame columns to be valid SQL identifiers by replacing spaces and
    special characters with underscores.
    """
    df.columns = [col.replace(' ', '_').replace('.', '').lower() for col in
df.columns]
    return df

def handle_inf_values(df):
    """
    Replaces inf and -inf values with NaN, and then replaces NaN with 0 in the
    DataFrame.
    """

    Parameters:
    df (DataFrame): The DataFrame to process.
```

```
Returns:  
DataFrame: The processed DataFrame with inf and NaN values handled.  
"""  
# Replace inf and -inf with NaN  
df.replace([float('inf'), float('-inf')], pd.NA, inplace=True)  
# Replace NaN with 0  
df.fillna(0, inplace=True)  
return df  
  
def round_float_columns(df, decimal_places=3):  
    float_cols = df.select_dtypes(include=['float64']).columns  
    df[float_cols] = df[float_cols].round(decimal_places)  
    return df  
  
def remove_duplicates(table_name, engine, columns, primary_key='id'):   
    column_list = ', '.join(columns)  
    # Exclude 'db_entry_time' and 'db_update_time' from duplicate check  
    duplicate_check_columns = [col for col in columns if col not in  
    ['db_entry_time', 'db_update_time']]  
  
    delete_duplicates_query = f"""  
    DELETE t1 FROM {table_name} t1  
    INNER JOIN {table_name} t2  
    WHERE  
        t1.{primary_key} < t2.{primary_key} AND  
        { ' AND '.join([f't1.{col} = t2.{col}' for col in  
    duplicate_check_columns]) };  
    """  
  
    with engine.connect() as conn:  
        conn.execute(delete_duplicates_query)  
        conn.execute("COMMIT;")  
  
def insert_apriori_results():  
    engine = create_db_engine()  
  
    # Insert Apriori results  
    apriori_results = pd.read_parquet('/opt/airflow/data/apriori_results.parquet')  
    apriori_results = sanitize_column_names(apriori_results)  
    apriori_results = handle_inf_values(apriori_results)  
    apriori_results = round_float_columns(apriori_results, decimal_places=3)  
    upload_dataframe_to_temp_sql(apriori_results, 'apriori_temp_results', engine,  
index=False)  
    columns = ['antecedents', 'consequents', 'antecedent_support',  
    'consequent_support', 'support', 'confidence', 'lift', 'leverage', 'conviction',  
    'zhangs_metric']  
    merge_data_from_temp_to_main_without_pk('apriori_temp_results',  
    'apriori_results', engine, columns)  
    remove_duplicates('apriori_results', engine, columns)  
  
def insert_fp_growth_results():  
    engine = create_db_engine()  
  
    # Insert FP-Growth results
```

```
fpgrowth_results =  
pd.read_parquet('/opt/airflow/data/fpgrowth_results.parquet')  
    fpgrowth_results = sanitize_column_names(fpgrowth_results)  
    fpgrowth_results = handle_inf_values(fpgrowth_results)  
    fpgrowth_results = round_float_columns(fpgrowth_results, decimal_places=3)  
    upload_dataframe_to_temp_sql(fpgrowth_results, 'fpgrowth_temp_results',  
    engine, index=False)  
    columns = ['antecedents', 'consequents', 'antecedent_support',  
'consequent_support', 'support', 'confidence', 'lift', 'leverage', 'conviction',  
'zhangs_metric']  
    merge_data_from_temp_to_main_without_pk('fpgrowth_temp_results',  
'fpgrowth_results', engine, columns)  
    remove_duplicates('fpgrowth_results', engine, columns)
```

EDA

```
def extract_data_for_report():  
    df = fetch_data_from_db_for_models()  
    df.to_csv('/opt/airflow/scripts/models/data/data_for_report.csv')  
  
def transform_data_for_report():  
    df = pd.read_csv('/opt/airflow/scripts/models/data/data_for_report.csv')  
    df = preprocessing_pipeline_for_report(df)  
    df_expanded = expand_column_to_rows(df, 'product_category')  
  
    df.to_parquet('/opt/airflow/scripts/models/data/transformed_data_for_report.parquet', index=False)  
  
    df_expanded.to_parquet('/opt/airflow/scripts/models/data/transformed_data_expanded_for_report.parquet', index=False)  
  
def create_pdf_report_task():  
    # Define the output directory and the path for the PDF report  
    output_dir = '/opt/airflow/scripts/models'  
    pdf_path = os.path.join(output_dir, 'sales_report.pdf')  
  
    # Call the function to generate the PDF report  
    generate_pdf_report(output_dir, pdf_path)  
    print(f"PDF report generated and saved to {pdf_path}")  
  
def generate_visualizations_task():  
    output_dir = '/opt/airflow/scripts/models/visuals'  
    transformed_data_path =  
    '/opt/airflow/scripts/models/data/transformed_data_for_report.parquet'  
    transformed_data_expanded_path =  
    '/opt/airflow/scripts/models/data/transformed_data_expanded_for_report.parquet'  
    df = pd.read_parquet(transformed_data_path)  
    df_expanded = pd.read_parquet(transformed_data_expanded_path)  
  
    generate_all_visualizations(df, df_expanded, output_dir)
```

```

print(f"Visualizations saved to {output_dir}")

def print_etl_done():
    print("ETL done")

```

upload data to github

```

def upload_files_to_github_task():
    files = {
        "/opt/airflow/scripts/models/data/transformed_data_for_report.parquet":
        "data/data_for_model.parquet",
        "/opt/airflow/scripts/models/data/transformed_data_expanded_for_report.parquet":
        "data/data_expanded_for_model.parquet"
    }
    repo = "Romanos-Rizk/AUB-capstone"
    token = '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L'
    message = "Update data files"

    upload_files_to_github(files, repo, token, message)

```

5. Airflow dag Implementation

The scripts explained in the previous sections are integrated into the Airflow automation software through a `dag.py` (the name is interchangeable) file. In this file, we import all the functions created in the scripts and use them to automate the ETL process for each dataframe. This setup allows for scheduling, monitoring, and managing the ETL workflow efficiently.

In this section, necessary modules such as pandas, DAG from airflow, PythonOperator, and datetime are imported. The `scripts` folder directory is also added to the system path to ensure that the custom scripts can be easily imported. Following that, all the ETL functions from the `extract.py`, `transform_pipleines.py` (that will import from `transform.py`), `load.py`, and `reconciliation.py` have been imported.

5.1 Importing the Necessary Modules

```

import pandas as pd
from airflow import DAG
from airflow.utils.email import send_email
from airflow.operators.bash_operator import BashOperator
from airflow.operators.python_operator import PythonOperator
from datetime import datetime
from airflow.operators.dummy_operator import DummyOperator
import sys
import os
from sqlalchemy import inspect

```

```
# Add the scripts directory to the system path
sys.path.append('/opt/airflow/scripts')

# Import your ETL functions
from extract import read_data_from_excel
from etl_functions import(extract_ecom_data,
                         transform_ecom_data,
                         load_ecom_data,
                         extract_aramex_data,
                         transform_aramex_data,
                         extract_cosmaline_data,
                         transform_cosmaline_data,
                         concatenate_aramex_cosmaline_data,
                         load_concatenated_data,
                         extract_credit_card_data,
                         transform_credit_card_data,
                         load_credit_card_data,
                         extract_erp_data,
                         transform_erp_data,
                         load_erp_data,
                         extract_oracle_data,
                         transform_oracle_data,
                         load_oracle_data,
                         extract_daily_rate_data,
                         load_daily_rate_data,
                         execute_cosmaline_reconciliation,
                         execute_credit_card_reconciliaiton,
                         execute_ecom_orders_not_in_oracle_reconciliation,
                         execute_ecom_orders_not_in_shipping_reconciliation,
                         execute_ecom_reconciliation,
                         execute_invalid_oracle_order_numbers_reconciliation,
                         execute_invalid_shipping_order_numbers_reconciliation,
                         execute_token_reconciliation,
                         cleanup_temp_tables_task
)
from load import create_db_engine, upload_dataframe_to_temp_sql,
merge_data_from_temp_to_main_with_pk,
merge_data_from_temp_to_main_without_pk,cleanup_temp_tables
from reconciliation import execute_reconciliation_script
from email_notifications import success_email, failure_email
```

5.2 Defining Default Arguments and the DAG Structure

In this section, the default arguments for the DAG are defined the DAG structure is created. The default arguments set the initial parameters for the DAG, such as the owner, start date, and number of retries. The DAG structure specifies the name, description, and schedule for the ETL process.

```
# Define the default arguments for the DAG
default_args = {
    'owner': 'airflow',
    'start_date': datetime(2024, 1, 1),
    'retries': 1,
    'email': ['rar37@mail.aub.edu', 'hmf31@mail.aub.edu'],
    'email_on_failure': False,
    'email_on_retry': False,
    'email_on_success': False,
    'on_failure_callback': failure_email,
}
```

Default Arguments:

- owner: The owner of the DAG, set to 'airflow'.
- start_date: The start date for the DAG, set to January 1, 2024.
- retries: The number of retries if a task fails, set to 1.
- email: The emails that are to be notified
- email_on_failure: notify by email in case of task failure, set to false
- email_on_retry: notify by email in case of task retry, set to false
- email_on_success: notify by email in case of task success, set to false
- on_failure_callback: custom notification by email, set to call the failure_email function in the `notifications.py` script

```
# Create the DAG
dag = DAG(
    'etl_dag',
    default_args=default_args,
    description='ETL DAG for Capstone Project',
    schedule_interval='@daily',
)
```

DAG Creation:

- dag_id: The unique identifier for the DAG, set to 'etl_dag'.
- default_args: The default arguments defined above.
- description: A brief description of the DAG, set to 'ETL DAG for Capstone Project'.
- schedule_interval: The schedule for the DAG, set to run daily using '@daily'.

5.3 Defining Task Python Operators

In this section, we will detail the corresponding Python operators for each dataframe. Each subsection will cover the specific Python operators that define Airflow tasks for the functions used to perform the ETL steps (Extract, Transform, Load) for a particular dataframe.

1 - `ecom_orders`:

The ECOM Data involves extracting order information from an Excel sheet, transforming the data to ensure consistency and accuracy, and finally loading the processed data into the database. The following subsections explain the python operators used for these steps and how they are implemented in Airflow using Python operators:

```
extract_ecom_task = PythonOperator(  
    task_id='extract_ecom_data',  
    python_callable=extract_ecom_data,  
    dag=dag,  
)
```

```
transform_ecom_task = PythonOperator(  
    task_id='transform_ecom_data',  
    python_callable=transform_ecom_data,  
    dag=dag,  
)
```

```
load_ecom_task = PythonOperator(  
    task_id='load_ecom_data',  
    python_callable=load_ecom_data,  
    dag=dag,  
)
```

2 - shippedandcollected_aramex:

The Aramex Data involves extracting shipment information from an Excel sheet, transforming the data to ensure consistency and accuracy, and preparing it for further processing. The following subsections explain the python operators used for these steps and how they are implemented in Airflow using Python operators:

```
extract_arameix_task = PythonOperator(  
    task_id='extract_arameix_data',  
    python_callable=extract_arameix_data,  
    dag=dag,  
)
```

```
transform_arameix_task = PythonOperator(  
    task_id='transform_arameix_data',  
    python_callable=transform_arameix_data,  
    dag=dag,  
)
```

Note: there is no load task here for Aramex because it will later be combined with Cosmaline data before being loaded into the database as a concatenated table.

3 - **shippedandcollected_cosmaline:**

The Cosmaline Data involves extracting shipment information from an Excel sheet, transforming the data to ensure consistency and accuracy, and preparing it for further processing. The following subsections explain the python operators used for these steps and how they are implemented in Airflow using Python operators:

```
extract_cosmaline_task = PythonOperator(  
    task_id='extract_cosmaline_data',  
    python_callable=extract_cosmaline_data,  
    dag=dag,  
)
```

```
transform_cosmaline_task = PythonOperator(  
    task_id='transform_cosmaline_data',  
    python_callable=transform_cosmaline_data,  
    dag=dag,  
)
```

Note: there is no load task here for Cosmaline because it will later be combined with Aramex data before being loaded into the database as a concatenated table.

4 - **shippedandcollected_aramex_cosmaline:**

The concatenated table for Aramex and Cosmaline data involves combining the transformed shipment information from both sources, ensuring consistency and accuracy, and then loading the processed data into the database. The following subsections explain the python operators used for these steps and how they are implemented in Airflow using Python operators:

```
concatenate_aramex_cosmaline_task = PythonOperator(  
    task_id='concatenate_aramex_cosmaline_data',  
    python_callable=concatenate_aramex_cosmaline_data,  
    dag=dag,  
)
```

```
load_concatenated_task = PythonOperator(  
    task_id='load_concatenated_data',  
    python_callable=load_concatenated_data,  
    dag=dag,  
)
```

5 - credit_card:

The Credit Card Data involves extracting transaction information from an Excel sheet, transforming the data to ensure consistency and accuracy, and finally loading the processed data into the database. The following subsections explain the python operators used for these steps and how they are implemented in Airflow using Python operators:

```
extract_credit_card_task = PythonOperator(  
    task_id='extract_credit_card_data',  
    python_callable=extract_credit_card_data,  
    dag=dag,  
)
```

```
transform_credit_card_task = PythonOperator(  
    task_id='transform_credit_card_data',  
    python_callable=transform_credit_card_data,  
    dag=dag,  
)
```

```
load_credit_card_task = PythonOperator(  
    task_id='load_credit_card_data',  
    python_callable=load_credit_card_data,  
    dag=dag,  
)
```

6 - erp_data:

The ERP Data involves extracting collection information from an Excel sheet, transforming the data to ensure consistency and accuracy, and finally loading the processed data into the database. The following subsections explain the python operators used for these steps and how they are implemented in Airflow using Python operators:

```
extract_erp_task = PythonOperator(  
    task_id='extract_erp_data',  
    python_callable=extract_erp_data,  
    dag=dag,  
)
```

```
transform_erp_task = PythonOperator(  
    task_id='transform_erp_data',  
    python_callable=transform_erp_data,
```

```
        dag=dag,  
    )
```

```
load_erp_task = PythonOperator(  
    task_id='load_erp_data',  
    python_callable=load_erp_data,  
    dag=dag,  
)
```

7 - oracle_data:

The Oracle Data involves extracting order information from an Excel sheet, transforming the data to ensure consistency and accuracy, and finally loading the processed data into the database. The following subsections explain the python operators used for these steps and how they are implemented in Airflow using Python operators:

```
``python extract_oracle_task = PythonOperator( task_id='extract_oracle_data',  
python_callable=extract_oracle_data, dag=dag, )
```

```
```python  
transform_oracle_task = PythonOperator(
 task_id='transform_oracle_data',
 python_callable=transform_oracle_data,
 dag=dag,
)
```

```
load_oracle_task = PythonOperator(
 task_id='load_oracle_data',
 python_callable=load_oracle_data,
 dag=dag,
)
```

## 8 - daily\_rate:

The Daily Rate Data involves extracting exchange rate information from an Excel sheet and loading the processed data into the database. This data does not require any transformation steps. The following subsections explain the python operators used for these steps and how they are implemented in Airflow using Python operators:

```
extract_daily_rate_task = PythonOperator(
 task_id='extract_daily_rate_data',
 python_callable=extract_daily_rate_data,
```

```
 dag=dag,
)
```

```
load_daily_rate_task = PythonOperator(
 task_id='load_daily_rate_data',
 python_callable=load_daily_rate_data,
 dag=dag,
)
```

## 9 - All Data Loaded Confirmation Task

```
all_data_loaded = PythonOperator(
 task_id='all_data_loaded',
 python_callable=lambda: print("All data loading tasks are completed."),
 dag=dag,
)
```

## 10 - Reconciliation and Clean up Tasks

The reconciliation process in this ETL pipeline involves executing a series of SQL scripts to reconcile data across different sources. Each reconciliation function is responsible for executing a specific SQL script that performs the reconciliation task. The following sections provide the details of these Python operators.

```
token_reconciliation_task = PythonOperator(
 task_id='execute_token_reconciliation',
 python_callable=execute_token_reconciliation,
 dag=dag,
)

cosmaline_reconciliation_task = PythonOperator(
 task_id='execute_cosmaline_reconciliation',
 python_callable=execute_cosmaline_reconciliation,
 dag=dag,
)

ecom_reconciliation_task = PythonOperator(
 task_id='execute_ecom_reconciliation',
 python_callable=execute_ecom_reconciliation,
 dag=dag,
)

credit_card_reconciliation_task = PythonOperator(
 task_id='execute_credit_card_reconciliaiton',
 python_callable=execute_credit_card_reconciliaiton,
 dag=dag,
)
```

```
ecom_orders_not_in_shipping_task = PythonOperator(
 task_id='execute_ecom_orders_not_in_shipping_reconciliation',
 python_callable=execute_ecom_orders_not_in_shipping_reconciliation,
 dag=dag,
)

ecom_orders_not_in_oracle_task = PythonOperator(
 task_id='execute_ecom_orders_not_in_oracle_reconciliation',
 python_callable=execute_ecom_orders_not_in_oracle_reconciliation,
 dag=dag,
)

invalid_shipping_order_numbers_task = PythonOperator(
 task_id='execute_invalid_shipping_order_numbers_reconciliation',
 python_callable=execute_invalid_shipping_order_numbers_reconciliation,
 dag=dag,
)

invalid_oracle_order_numbers_task = PythonOperator(
 task_id='execute_invalid_oracle_order_numbers_reconciliation',
 python_callable=execute_invalid_oracle_order_numbers_reconciliation,
 dag=dag,
)

cleanup_task = PythonOperator(
 task_id='cleanup_temp_tables',
 python_callable=cleanup_temp_tables_task,
 dag=dag,
)
```

## 11 - Send Email Notification Task:

This task is triggered when all the previous tasks are completed and is responsible of sending a success email notification to the specified stakeholders

```
Define a dummy operator to send a success email at the end of the DAG
all_completed = DummyOperator(
 task_id='all_completed',
 dag=dag,
 trigger_rule='all_success', # This task will be executed only if all upstream
 tasks succeed
 on_success_callback=success_email, # Success email callback for the final
 task
)
```

## 12 - Business Analysis Tasks:

```
extract_task_for_report = PythonOperator(
 task_id='extract_data_for_report',
 python_callable=extract_data_for_report,
 dag=dag,
)

transform_task_for_report = PythonOperator(
 task_id='transform_data_for_report',
 python_callable=transform_data_for_report,
 dag=dag,
)

visualize_task_for_report = PythonOperator(
 task_id='generate_visualizations_task',
 python_callable=generate_visualizations_task,
 dag=dag,
)

create_pdf_task_for_report = PythonOperator(
 task_id='create_pdf_report_task',
 python_callable=create_pdf_report_task,
 dag=dag,
)

send_report_task = PythonOperator(
 task_id='send_report_task',
 python_callable=send_sales_report_email,
 op_args=['/opt/airflow/scripts/models/sales_report.pdf'],
 dag=dag
)

upload_files_task = PythonOperator(
 task_id='upload_files_to_github',
 python_callable=upload_files_to_github_task,
 dag=dag
)
```

## 5.4 Defining Task Dependencies

The task dependencies in the Airflow DAG define the order in which tasks are executed. Here's a breakdown of how the tasks are dependent on each other:

### 1. ECOM Tasks:

- The ECOM data extraction task (`extract_ecom_task`) must complete before the transformation task (`transform_ecom_task`), which in turn must complete before the loading task (`load_ecom_task`).

### 2. Cosmaline, Aramex, and Oracle Tasks:

- The Cosmaline data extraction task (`extract_cosmaline_task`) is followed by its transformation task (`transform_cosmaline_task`).
- The Aramex data extraction task (`extract_aramex_task`) is followed by its transformation task (`transform_aramex_task`).
- The Oracle data extraction task (`extract_oracle_task`) is followed by its transformation task (`transform_oracle_task`), which is then followed by its loading task (`load_oracle_task`).

### 3. Dependencies on ECOM Transformation:

- Both the Cosmaline transformation task (`transform_cosmaline_task`) and the Aramex transformation task (`transform_aramex_task`) are dependent on the completion of the ECOM transformation task (`transform_ecom_task`).

### 4. Concatenation Tasks:

- Both the Cosmaline transformation task (`transform_cosmaline_task`) and the Aramex transformation task (`transform_aramex_task`) must complete before the concatenation task (`concatenate_aramex_cosmaline_task`).
- The concatenation task (`concatenate_aramex_cosmaline_task`) must complete before the loading task for the concatenated data (`load_concatenated_task`).

### 5. Load Dependencies:

- The loading of ECOM data (`load_ecom_task`) is a prerequisite for both the loading of the concatenated data (`load_concatenated_task`) and the loading of Oracle data (`load_oracle_task`).

### 6. Independent ERP, Credit Card, and Daily Rate Tasks:

- The ERP data tasks (extraction `extract_erp_task`, transformation `transform_erp_task`, and loading `load_erp_task`) are independent and follow their sequence.
- The Credit Card data tasks (extraction `extract_credit_card_task`, transformation `transform_credit_card_task`, and loading `load_credit_card_task`) follow their sequence independently.
- The Daily Rate data tasks (extraction `extract_daily_rate_task` and loading `load_daily_rate_task`) follow their sequence independently as they do not require transformation.

### 7. Completion of Data Loading:

- All the data loading tasks (`load_ecom_task`, `load_concatenated_task`, `load_oracle_task`, `load_erp_task`, `load_credit_card_task`, and `load_daily_rate_task`) must complete before the intermediary task (`all_data_loaded`).

### 8. Reconciliation Tasks:

- The intermediary task (`all_data_loaded`) must complete before any reconciliation tasks start. This ensures that all data is loaded before reconciliation begins.

### 9. Cleanup Task:

- All reconciliation tasks (`token_reconciliation_task`, `cosmaline_reconciliation_task`, `ecom_reconciliation_task`, `credit_card_reconciliation_task`, `ecom_orders_not_in_shipping_task`, `ecom_orders_not_in_oracle_task`, `invalid_shipping_order_numbers_task`, and `invalid_oracle_order_numbers_task`) must complete before the cleanup task (`cleanup_task`) is executed.

## 10. Business Analysis Tasks

- All the business analysis tasks shouls be performed after the reconcilaition processes

These dependencies ensure that tasks are executed in the correct order, maintaining data integrity and consistency throughout the ETL process.

```
ECOM tasks
extract_ecom_task >> transform_ecom_task >> load_ecom_task

Cosmaline, Aramex, and Oracle tasks
extract_cosmaline_task >> transform_cosmaline_task
extract_arameix_task >> transform_arameix_task
extract_oracle_task >> transform_oracle_task >> load_oracle_task

Transform tasks for Cosmaline and Aramex dependent on ECOM transformation
transform_ecom_task >> transform_cosmaline_task
transform_ecom_task >> transform_arameix_task

Concatenate tasks
transform_cosmaline_task >> concatenate_arameix_cosmaline_task
transform_arameix_task >> concatenate_arameix_cosmaline_task

Load Cosmaline and Aramex tasks are dependent on ECOM data load
load_ecom_task >> load_concatenated_task
load_ecom_task >> load_oracle_task

Concatenate and load concatenated data
concatenate_arameix_cosmaline_task >> load_concatenated_task

ERP, Credit Card, and Daily Rate tasks (independent of others)
extract_erp_task >> transform_erp_task >> load_erp_task
extract_credit_card_task >> transform_credit_card_task >> load_credit_card_task
extract_daily_rate_task >> load_daily_rate_task

All data loading tasks must complete before the intermediary task
[load_ecom_task, load_concatenated_task, load_oracle_task, load_erp_task,
load_credit_card_task, load_daily_rate_task] >> all_data_loaded

Intermediary task must complete before any reconciliation tasks start
all_data_loaded >> token_reconciliation_task
all_data_loaded >> cosmaline_reconciliation_task
all_data_loaded >> ecom_reconciliation_task
all_data_loaded >> credit_card_reconciliation_task
all_data_loaded >> ecom_orders_not_in_shipping_task
all_data_loaded >> ecom_orders_not_in_oracle_task
```

```
all_data_loaded >> invalid_shipping_order_numbers_task
all_data_loaded >> invalid_oracle_order_numbers_task

Task dependencies to ensure cleanup_task is done before starting model-related
tasks
[token_reconciliation_task, cosmaline_reconciliation_task,
ecom_reconciliation_task, credit_card_reconciliation_task,
ecom_orders_not_in_shipping_task, ecom_orders_not_in_oracle_task,
invalid_shipping_order_numbers_task, invalid_oracle_order_numbers_task] >>
print_etl_done_task

Model-related tasks start after cleanup_task
print_etl_done_task >> extract_for_model
extract_for_model >> transform_for_model
transform_for_model >> [generate_apriori_model, generate_fp_growth_model]
generate_apriori_model >> load_model_apriori_results
generate_fp_growth_model >> load_model_fp_growth_results

Report
print_etl_done_task >> extract_task_for_report >> transform_task_for_report >>
visualize_task_for_report >> create_pdf_task_for_report >> send_report_task

All tasks must complete before all_completed
[load_model_apriori_results, load_model_fp_growth_results, send_report_task] >>
cleanup_task
transform_task_for_report >> upload_files_task >> cleanup_task
cleanup_task >> all_completed
```

## 6. Software Testing

---

Software testing is a critical component of the development process, ensuring the reliability and correctness of the system. For this automation project, comprehensive testing has been implemented using pytest to validate the functionality and performance of the Airflow DAG system. The testing framework is structured to cover both unit tests and integration tests, organized within a dedicated **tests** folder in the root directory of the project.

The tests folder contains three sub folders:

- **unit:** This folder includes tests that validate individual components or functions in isolation, ensuring they work as expected.
- **Integration:** This folder contains tests that verify the interaction between multiple components, ensuring they work together correctly.
- **data:** This folder is used to store simulated data for testing purposes. Tests that need to read or write data can utilize this folder to manage test data effectively.

Command to test all test files in the root directory:

### 6.0 Pytest Setup:

## 6.0.1 Pytest Configuration File

Create a file in your project root directory named `pytest.ini` and include these commands in it. These commands will help have a less cluttered results view of the tests

```
[pytest]
filterwarnings =
 ignore::RuntimeWarning:airflow
 ignore::sqlalchemy.exc.RemovedIn20Warning
 ignore::FutureWarning:Mismatched null-like values
 ignore::*U.*mode is deprecated:DeprecationWarning
```

## 6.0.2 Conduct the Tests:

### Test All Test Files in the Root Directory:

```
pytest
```

### Test Specific Test Files in the Root Directory:

```
For unit tests:
pytest tests/unit/anyTestFile.py

For integration tests:
pytest tests/integration/anyTestFile.py
```

## 6.1 Unit Testing

Unit testing focuses on verifying the functionality of individual components or functions in isolation, ensuring that each part of the system performs as expected. In the Malia Project, pytest is used to create and run unit tests. This section provides detailed descriptions and code examples of the various unit tests implemented for the Airflow DAG system

### 1. Test: `test_extract.py`

The `test_extract.py` file contains unit tests for the `read_data_from_excel` function. This function is responsible for reading data from an Excel file. The tests cover the following scenarios:

1. **Successful Data Read:** We mock `pd.read_excel` to return a sample DataFrame. We then check if the function correctly calls `pd.read_excel` and returns the expected DataFrame.
2. **Exception Handling:** We mock `pd.read_excel` to raise an exception and check if the function correctly handles the exception and returns `None`.

```
import pytest
from unittest.mock import patch, MagicMock
import pandas as pd
import sys
import os

Adjusting the script path to include the extract module
script_path = os.path.abspath(os.path.join(os.path.dirname(__file__),
'../../scripts'))
sys.path.append(script_path)
from extract import read_data_from_excel

Test case for successful data read
def test_read_data_from_excel_success():
 # Sample data to be returned by the mock
 sample_data = pd.DataFrame({
 'column1': [1, 2],
 'column2': ['a', 'b']
 })

 # Patch the pd.read_excel method to return sample_data
 with patch('pandas.read_excel', return_value=sample_data) as mock_read_excel:
 filepath = 'dummy_path.xlsx'
 sheet_name = 'Sheet1'
 result = read_data_from_excel(filepath, sheet_name)

 # Assertions
 mock_read_excel.assert_called_once_with(filepath, sheet_name=sheet_name)
 assert result.equals(sample_data)

Test case for exception handling
def test_read_data_from_excel_failure():
 # Patch the pd.read_excel method to raise an exception
 with patch('pandas.read_excel', side_effect=Exception("File not found")) as mock_read_excel:
 filepath = 'non_existent_file.xlsx'
 result = read_data_from_excel(filepath)

 # Assertions
 mock_read_excel.assert_called_once_with(filepath, sheet_name='Website ECOM Data')
 assert result is None
```

## Explanation

### test\_read\_data\_from\_excel\_success

- This test simulates a successful read operation from an Excel file.
- It uses `patch` to mock the `pd.read_excel` function to return a predefined sample DataFrame.
- It asserts that `pd.read_excel` is called once with the correct parameters.
- It verifies that the function returns the expected DataFrame.

### test\_read\_data\_from\_excel\_failure

- This test simulates a failure in reading an Excel file.
- It uses `patch` to mock the `pd.read_excel` function to raise an exception.
- It asserts that `pd.read_excel` is called once with the correct parameters.
- It verifies that the function handles the exception correctly and returns `None`.

These tests ensure that the `read_data_from_excel` function behaves correctly under both normal and exceptional conditions.

## 2. Test: `test_transform_general.py`

The `test_transform_general.py` file contains unit tests for various data transformation functions. These functions perform common data cleaning tasks such as formatting columns, removing missing values, and eliminating duplicates. The tests cover the following functions:

- `lower_columns`
- `uppercase_columns`
- `remove_missing_values`
- `remove_duplicates`
- `rename_columns_to_lowercase`

```
...
test_lower_columns:
 -Purpose: Verify that the lower_columns function correctly strips whitespace, converts text to lowercase, and capitalizes the first letter of each word in the specified columns.
 -Setup: Create a DataFrame with varying cases and whitespace.
 -Assertion: Check if the function output matches the expected DataFrame.

test_uppercase_columns:
 -Purpose: Verify that the uppercase_columns function correctly strips whitespace and converts text to uppercase in the specified columns.
 -Setup: Create a DataFrame with varying cases and whitespace.
 -Assertion: Check if the function output matches the expected DataFrame.

test_remove_missing_values:
 -Purpose: Verify that the remove_missing_values function correctly removes rows with any missing values.
 -Setup: Create a DataFrame with some missing values.
 -Assertion: Check if the function output matches the expected DataFrame with rows containing missing values removed.

test_remove_duplicates:
 -Purpose: Verify that the remove_duplicates function correctly removes duplicate rows, keeping the first occurrence.
 -Setup: Create a DataFrame with duplicate rows.
 -Assertion: Check if the function output matches the expected DataFrame with duplicates removed.

test_rename_columns_to_lowercase:
```

```
-Purpose: Verify that the rename_columns_to_lowercase function correctly renames columns to lowercase.
-Setup: Create a DataFrame with uppercase column names.
-Assertion: Check if the function output matches the expected DataFrame with lowercase column names.
...

import pytest
import pandas as pd
import sys
import os

Add the scripts directory to the system path
script_path = os.path.abspath(os.path.join(os.path.dirname(__file__),
'../../scripts'))
sys.path.append(script_path)

from transform import lower_columns, uppercase_columns, remove_missing_values,
remove_duplicates, rename_columns_to_lowercase

def test_lower_columns():
 input_data = pd.DataFrame({
 'Name': ['Romanos', 'HADIL', 'ali'],
 'Age': [25, 30, 35]
 })
 columns = ['Name']
 expected_output = pd.DataFrame({
 'Name': ['Romanos', 'Hadil', 'Ali'],
 'Age': [25, 30, 35]
 })
 result = lower_columns(input_data, columns)
 assert result.equals(expected_output)

def test_uppercase_columns():
 input_data = pd.DataFrame({
 'Name': ['Romanos', 'hadil', 'ali'],
 'Age': [25, 30, 35]
 })
 columns = ['Name']
 expected_output = pd.DataFrame({
 'Name': ['ROMANOS', 'HADIL', 'ALI'],
 'Age': [25, 30, 35]
 })
 result = uppercase_columns(input_data, columns)
 assert result.equals(expected_output)

def test_remove_missing_values():
 input_data = pd.DataFrame({
 'Name': ['Romanos', None, 'Ali'],
 'Age': [25, 30, None]
 })
 expected_output = pd.DataFrame({
 'Name': ['Romanos'],
 'Age': [25]
 })
```

```
 })
 expected_output['Age'] = expected_output['Age'].astype(float) # Ensure
matching data types
 result = remove_missing_values(input_data)
 assert result.equals(expected_output)

def test_remove_duplicates():
 input_data = pd.DataFrame({
 'Name': ['Romanos', 'Hadir', 'Romanos'],
 'Age': [25, 30, 25]
 })
 expected_output = pd.DataFrame({
 'Name': ['Romanos', 'Hadir'],
 'Age': [25, 30]
 })
 result = remove_duplicates(input_data)
 assert result.equals(expected_output)

def test_rename_columns_to_lowercase():
 input_data = pd.DataFrame({
 'NAME': ['Romanos', 'Hadir', 'Ali'],
 'AGE': [25, 30, 35]
 })
 expected_output = pd.DataFrame({
 'name': ['Romanos', 'Hadir', 'Ali'],
 'age': [25, 30, 35]
 })
 result = rename_columns_to_lowercase(input_data)
 assert result.equals(expected_output)
```

## Explanation

### test\_lower\_columns

- This test checks if the `lower_columns` function correctly trims and capitalizes the specified columns in a DataFrame.
- It uses sample data with extra spaces and mixed cases.
- It asserts that the function returns a DataFrame with the specified columns properly trimmed and capitalized.

### test\_uppercase\_columns

- This test checks if the `uppercase_columns` function correctly trims and converts the specified columns to uppercase in a DataFrame.
- It uses sample data with extra spaces and mixed cases.
- It asserts that the function returns a DataFrame with the specified columns in uppercase.

### test\_remove\_missing\_values

- This test checks if the `remove_missing_values` function correctly removes rows with any missing values.
- It uses sample data with some missing values.

- It asserts that the function returns a DataFrame with rows containing missing values removed.

#### test\_remove\_duplicates

- This test checks if the `remove_duplicates` function correctly removes duplicate rows.
- It uses sample data with duplicate rows.
- It asserts that the function returns a DataFrame with duplicate rows removed.

#### test\_rename\_columns\_to\_lowercase

- This test checks if the `rename_columns_to_lowercase` function correctly renames all columns to lowercase.
- It uses sample data with columns in uppercase.
- It asserts that the function returns a DataFrame with all column names in lowercase.

These tests ensure that the data transformation functions behave correctly and handle common data cleaning tasks effectively.

### 3. Test: `test_transform_erp.py`

The `test_transform_erp.py` file contains unit tests for the ERP data transformation functions. These functions are used to convert data types and extract tokens from descriptions in ERP data. The tests cover the following functions:

- `convert_data_types_ERP_Data`
- `extract_token_ERP_Data`

```
import pytest
import pandas as pd
import sys
import os
from pandas.testing import assert_frame_equal

Add the scripts directory to the system path
script_path = os.path.abspath(os.path.join(os.path.dirname(__file__),
'../../scripts'))
sys.path.append(script_path)

from transform import convert_data_types_ERP_Data, extract_token_ERP_Data

def test_convert_data_types_ERP_Data():
 input_data = pd.DataFrame({
 'CUSTOMER_NUMBER': [12345, 67890],
 'OtherColumn': ['A', 'B']
 })
 expected_output = input_data.astype({
 'CUSTOMER_NUMBER': 'object'
 })
 result = convert_data_types_ERP_Data(input_data)
 assert_frame_equal(result, expected_output, check_dtype=True)

def test_extract_token_ERP_Data():
```

```

input_data = pd.DataFrame({
 'description': ['Token 12345', 'No Number Here', 'Token 67890']
})
expected_output = pd.DataFrame({
 'description': ['12345', 'No Number Here', '67890']
})
result = extract_token_ERP_Data(input_data, 'description')
assert_frame_equal(result, expected_output, check_dtype=True)

```

## Explanation

### test\_convert\_data\_types\_ERP\_Data

- This test checks if the `convert_data_types_ERP_Data` function correctly converts the data type of the `CUSTOMER_NUMBER` column to `object`.
- It uses sample data with numerical `CUSTOMER_NUMBER` values.
- It asserts that the function returns a DataFrame with the `CUSTOMER_NUMBER` column data type converted to `object`.

### test\_extract\_token\_ERP\_Data

- This test checks if the `extract_token_ERP_Data` function correctly extracts numerical tokens from the `description` column.
- It uses sample data with descriptions containing numerical tokens and some without.
- It asserts that the function returns a DataFrame with the numerical tokens extracted from the descriptions.

These tests ensure that the ERP data transformation functions behave correctly and handle the specific data manipulation tasks required for ERP data.

## 4. Test: `test_transform_ecom.py`

The `test_transform_ecom.py` file contains unit tests for the e-commerce data transformation functions. These functions are used to convert data types and rename columns for website e-commerce data. The tests cover the following functions:

- `convert_data_types_ECOM_Data`
- `rename_columns_for_website_ecom_data`

```

import pytest
import pandas as pd
import sys
import os
from pandas.testing import assert_frame_equal

Add the scripts directory to the system path
script_path = os.path.abspath(os.path.join(os.path.dirname(__file__),
'../../scripts'))
sys.path.append(script_path)

```

```
from transform import convert_data_types_ECOM_Data,
rename_columns_for_website_ecom_data

def test_convert_data_types_ECOM_Data():
 input_data = pd.DataFrame({
 'Order Number': [123, 456],
 'Shipper Name': ['DHL', 'FedEx'],
 'Created At': ['2023-01-01', '2023-01-02'],
 'Delivered At': ['2023-01-03', '2023-01-04'],
 'Billing Type': ['Prepaid', 'COD'],
 'Amount': [100.0, 200.0],
 'Currency': ['USD', 'EUR'],
 'Country': ['US', 'DE'],
 'AWB': [789, 1011]
 })
 expected_output = input_data.astype({
 'Order Number': 'object',
 'Shipper Name': 'object',
 'Created At': 'datetime64[ns]',
 'Delivered At': 'datetime64[ns]',
 'Billing Type': 'object',
 'Amount': 'float64',
 'Currency': 'object',
 'Country': 'object',
 'AWB': 'object'
 })
 result = convert_data_types_ECOM_Data(input_data)
 assert_frame_equal(result, expected_output, check_dtype=True)

def test_rename_columns_for_website_ecom_data():
 input_data = pd.DataFrame({
 'Order Number': [123, 456],
 'Shipper Name': ['DHL', 'FedEx'],
 'Created At': ['2023-01-01', '2023-01-02'],
 'Delivered At': ['2023-01-03', '2023-01-04'],
 'Billing Type': ['Prepaid', 'COD'],
 'Amount': [100.0, 200.0],
 'Currency': ['USD', 'EUR'],
 'Country': ['US', 'DE'],
 'AWB': [789, 1011]
 })
 expected_output = input_data.rename(columns={
 'Order Number': 'order_number',
 'Shipper Name': 'shipper_name',
 'Created At': 'created_at',
 'Delivered At': 'delivered_at',
 'Billing Type': 'billing_type',
 'Amount': 'amount',
 'Currency': 'currency',
 'Country': 'country',
 'AWB': 'AWB'
 })
 result = rename_columns_for_website_ecom_data(input_data)
 assert_frame_equal(result, expected_output, check_dtype=True)
```

## Explanation

### test\_convert\_data\_types\_ECOM\_Data

- This test checks if the `convert_data_types_ECOM_Data` function correctly converts the data types of various columns in the e-commerce data.
- It uses sample data with mixed data types for columns such as `Order Number`, `Created At`, and `Amount`.
- It asserts that the function returns a DataFrame with the specified columns converted to the correct data types.

### test\_rename\_columns\_for\_website\_ecom\_data

- This test checks if the `rename_columns_for_website_ecom_data` function correctly renames the columns to a standardized format.
- It uses sample data with columns in mixed formats.
- It asserts that the function returns a DataFrame with columns renamed to a consistent lowercase format.

These tests ensure that the e-commerce data transformation functions behave correctly and handle the specific data manipulation tasks required for e-commerce data.

## 5. Test: `test_transform_cosmaline.py`

The `test_transform_cosmaline.py` file contains unit tests for the Cosmaline data transformation functions. These functions perform various data manipulation tasks specific to Cosmaline data, such as converting data types, renaming columns, adding new columns, and merging data. The tests cover the following functions:

- `convert_data_types_Cosmaline_Data`
- `rename_columns_for_Cosmaline_data`
- `add_cod_amount_column_Cosmaline_data`
- `add_new_columns_Cosmaline_Data`
- `add_AWB_Cosmaline_Data`
- `convert_data_types_aramec_cosmaline`

```
import pytest
import pandas as pd
import sys
import os
from pandas.testing import assert_frame_equal

Add the scripts directory to the system path
script_path = os.path.abspath(os.path.join(os.path.dirname(__file__),
'../../scripts'))
sys.path.append(script_path)

from transform import (
 convert_data_types_Cosmaline_Data,
 rename_columns_for_Cosmaline_data,
```

```
add_cod_amount_column_Cosmaline_data,
add_new_columns_Cosmaline_Data,
add_AWB_Cosmaline_Data,
convert_data_types_aramex_cosmaline
})

def test_convert_data_types_Cosmaline_Data():
 input_data = pd.DataFrame({
 'Currency': ['USD', 'LBP'],
 'Driver_Delivery_date': ['2023-01-01', '2023-01-02'],
 'Amount': [100.0, 200.0],
 'OrderNo': [123, 456]
 })
 expected_output = input_data.astype({
 'Currency': 'object',
 'Driver_Delivery_date': 'datetime64[ns]',
 'Amount': 'float64',
 'OrderNo': 'object'
 })
 result = convert_data_types_Cosmaline_Data(input_data)
 assert_frame_equal(result, expected_output, check_dtype=True)

def test_rename_columns_for_Cosmaline_data():
 input_data = pd.DataFrame({
 'Driver_Delivery_date': ['2023-01-01', '2023-01-02'],
 'OrderNo': [123, 456],
 'Amount': [100.0, 200.0],
 'Currency': ['USD', 'LBP']
 })
 expected_output = input_data.rename(columns={
 'Driver_Delivery_date': 'Delivery_Date',
 'OrderNo': 'order_number',
 'Amount': 'O_CODAmount',
 'Currency': 'CODCurrency'
 })
 result = rename_columns_for_Cosmaline_data(input_data)
 assert_frame_equal(result, expected_output, check_dtype=True)

def test_add_cod_amount_column_Cosmaline_data():
 input_data = pd.DataFrame({
 'CODCurrency': ['USD', 'LBP'],
 'O_CODAmount': [100.0, 200.0]
 })
 expected_output = input_data.copy()
 expected_output['CODAmount'] = [0.0, 200.0]
 result = add_cod_amount_column_Cosmaline_data(input_data)
 assert_frame_equal(result, expected_output, check_dtype=True)

def test_add_new_columns_Cosmaline_Data():
 input_data = pd.DataFrame({
 'order_number': [123, 456]
 })
 expected_output = input_data.copy()
 expected_output['ShprNo'] = pd.NA
```

```

expected_output['Aramex_Inv_date'] = pd.NA
expected_output['TOKENNO'] = 'Shipped With Cosmaline'
result = add_new_columns_Cosmaline_Data(input_data)
assert_frame_equal(result, expected_output, check_dtype=True)

def test_add_AWB_Cosmaline_Data():
 cosmaline_data = pd.DataFrame({
 'order_number': [123, 456],
 'OtherColumn': ['A', 'B']
 })
 ecom_data = pd.DataFrame({
 'order_number': [123, 456],
 'AWB': [789, 1011]
 })
 expected_output = pd.DataFrame({
 'order_number': [123, 456],
 'OtherColumn': ['A', 'B'],
 'AWB': [789, 1011]
 })
 result = add_AWB_Cosmaline_Data(cosmaline_data, ecom_data)
 assert_frame_equal(result, expected_output, check_dtype=True)

def test_convert_data_types_aramecosmaline():
 input_data = pd.DataFrame({
 'AWB': [123, 456],
 'ShprNo': ['ABC', 'DEF'],
 'Delivery_Date': ['2023-01-01', '2023-01-02'],
 'CODAmount': [100.0, 200.0],
 'CODCurrency': ['USD', 'LBP'],
 'O_CODAmount': [150.0, 250.0],
 'Aramex_Inv_date': ['2023-01-03', '2023-01-04'],
 'TOKENNO': [5678, 91011],
 'order_number': [1122, 3344]
 })
 expected_output = input_data.astype({
 'AWB': 'str',
 'ShprNo': 'object',
 'Delivery_Date': 'datetime64[ns]',
 'CODAmount': 'float32',
 'CODCurrency': 'object',
 'O_CODAmount': 'float32',
 'Aramex_Inv_date': 'datetime64[ns]',
 'TOKENNO': 'object',
 'order_number': 'object'
 })
 result = convert_data_types_aramecosmaline(input_data)
 assert_frame_equal(result, expected_output, check_dtype=True)

```

## Explanation

### test\_convert\_data\_types\_Cosmaline\_Data

- This test checks if the `convert_data_types_Cosmaline_Data` function correctly converts the data types of various columns in Cosmaline data.
- It uses sample data with mixed data types for columns such as `Currency`, `Driver_Delivery_date`, and `Amount`.
- It asserts that the function returns a DataFrame with the specified columns converted to the correct data types.

#### `test_rename_columns_for_Cosmaline_data`

- This test checks if the `rename_columns_for_Cosmaline_data` function correctly renames the columns to a standardized format.
- It uses sample data with columns in mixed formats.
- It asserts that the function returns a DataFrame with columns renamed to a consistent format.

#### `test_add_cod_amount_column_Cosmaline_data`

- This test checks if the `add_cod_amount_column_Cosmaline_data` function correctly adds a `CODAmount` column based on the existing `O_CODAmount` and `CODCurrency` columns.
- It uses sample data with `O_CODAmount` and `CODCurrency` columns.
- It asserts that the function returns a DataFrame with the `CODAmount` column added correctly.

#### `test_add_new_columns_Cosmaline_Data`

- This test checks if the `add_new_columns_Cosmaline_Data` function correctly adds new columns (`ShprNo`, `Aramex_Inv_date`, and `TOKENNO`) to the Cosmaline data.
- It uses sample data with an `order_number` column.
- It asserts that the function returns a DataFrame with the new columns added and initialized with default values.

#### `test_add_AWB_Cosmaline_Data`

- This test checks if the `add_AWB_Cosmaline_Data` function correctly merges the `AWB` column from the e-commerce data into the Cosmaline data based on the `order_number`.
- It uses sample data with matching `order_number` columns in both Cosmaline and e-commerce data.
- It asserts that the function returns a DataFrame with the `AWB` column correctly merged.

#### `test_convert_data_types_aramec_cosmaline`

- This test checks if the `convert_data_types_aramec_cosmaline` function correctly converts the data types of various columns in the Aramex Cosmaline data.
- It uses sample data with mixed data types for columns such as `AWB`, `Delivery_Date`, and `CODAmount`.
- It asserts that the function returns a DataFrame with the specified columns converted to the correct data types.

These tests ensure that the Cosmaline data transformation functions behave correctly and handle the specific data manipulation tasks required for Cosmaline data.

## 6. Test: `test_transform_aramec.py`

The `test_transform_aramec.py` file contains unit tests for the Aramex data transformation functions. These functions perform various data manipulation tasks specific to Aramex data, such as converting data types,

renaming columns, updating currency values, cleaning tokens, adding order numbers, and concatenating data. The tests cover the following functions:

- `convert_data_types_Aramex_Data`
- `rename_columns_for_website_Aramex_data`
- `update_cod_currency_Aramex_data`
- `clean_tokens_by_date_Aramex_Data`
- `add_order_number_Aramex_Data`
- `convert_data_types_aramex_cosmaline`
- `concatenate_Aramex_Cosmaline`

```
import pytest
import pandas as pd
import sys
import os
from pandas.testing import assert_frame_equal

Add the scripts directory to the system path
script_path = os.path.abspath(os.path.join(os.path.dirname(__file__),
'../../scripts'))
sys.path.append(script_path)

from transform import (
 convert_data_types_Aramex_Data,
 rename_columns_for_website_Aramex_data,
 update_cod_currency_Aramex_data,
 clean_tokens_by_date_Aramex_Data,
 add_order_number_Aramex_Data,
 convert_data_types_aramex_cosmaline,
 concatenate_Aramex_Cosmaline,
)

from transform_pipelines import data_preprocessing_pipeline_Aramex_Data

def test_convert_data_types_Aramex_Data():
 input_data = pd.DataFrame({
 'ShprNo': [123, 456],
 'HAWB': [789, 1011],
 'Delivery_Date': ['2023-01-01', '2023-01-02'],
 'CODAmount': [100.0, 200.0],
 'O_CODAmount': [150.0, 250.0],
 'Aramex_Inv_date': ['2023-01-03', '2023-01-04'],
 'TOKENNO': [5678, 91011]
 })
 expected_output = input_data.astype({
 'ShprNo': 'object',
 'HAWB': 'object',
 'Delivery_Date': 'datetime64[ns]',
 'CODAmount': 'float64',
 'O_CODAmount': 'float64',
 'Aramex_Inv_date': 'datetime64[ns]',
 'TOKENNO': 'object'
```

```
 })
 result = convert_data_types_Aramex_Data(input_data)
 assert result.equals(expected_output)

def test_rename_columns_for_website_Aramex_data():
 input_data = pd.DataFrame({
 'HAWB': [123, 456],
 'Value': [789, 1011]
 })
 expected_output = input_data.rename(columns={'HAWB': 'AWB'})
 result = rename_columns_for_website_Aramex_data(input_data)
 assert result.equals(expected_output)

def test_update_cod_currency_Aramex_data():
 input_data = pd.DataFrame({
 'CODAmount': [100.0, 0.0],
 'CODCurrency': ['', '']
 })
 expected_output = pd.DataFrame({
 'CODAmount': [100.0, 0.0],
 'CODCurrency': ['LBP', '']
 })
 result = update_cod_currency_Aramex_data(input_data)
 assert result.equals(expected_output)

def test_clean_tokens_by_date_Aramex_Data():
 input_data = pd.DataFrame({
 'Aramex_Inv_date': ['2023-01-01', '2023-01-01', '2023-01-02'],
 'TOKENNO': [1111, 2222, 3333]
 })
 expected_output = pd.DataFrame({
 'Aramex_Inv_date': ['2023-01-01', '2023-01-01', '2023-01-02'],
 'TOKENNO': [1111, 1111, 3333]
 })
 result = clean_tokens_by_date_Aramex_Data(input_data, 'Aramex_Inv_date',
 'TOKENNO')
 assert result.equals(expected_output)

def test_add_order_number_Aramex_Data():
 aramex_data = pd.DataFrame({
 'AWB': ['123', '456'],
 'OtherColumn': ['A', 'B']
 })
 ecom_data = pd.DataFrame({
 'AWB': ['123', '456'],
 'order_number': [789, 1011]
 })
 expected_output = pd.DataFrame({
 'AWB': ['123', '456'],
 'OtherColumn': ['A', 'B'],
 'order_number': [789, 1011]
 })
 result = add_order_number_Aramex_Data(aramex_data, ecom_data)
 assert result.equals(expected_output)
```

```

def test_convert_data_types_aramelex_cosmaline():
 input_data = pd.DataFrame({
 'AWB': [123, 456],
 'ShprNo': ['ABC', 'DEF'],
 'Delivery_Date': ['2023-01-01', '2023-01-02'],
 'CODAmount': [100.0, 200.0],
 'CODCurrency': ['USD', 'LBP'],
 'O_CODAmount': [150.0, 250.0],
 'Aramex_Inv_date': ['2023-01-03', '2023-01-04'],
 'TOKENNO': [5678, 91011],
 'order_number': [1122, 3344]
 })
 expected_output = input_data.astype({
 'AWB': 'str',
 'ShprNo': 'object',
 'Delivery_Date': 'datetime64[ns]',
 'CODAmount': 'float32',
 'CODCurrency': 'object',
 'O_CODAmount': 'float32',
 'Aramex_Inv_date': 'datetime64[ns]',
 'TOKENNO': 'object',
 'order_number': 'object'
 })
 result = convert_data_types_aramelex_cosmaline(input_data)
 assert result.equals(expected_output)

def test_concatenate_Aramex_Cosmaline():
 df1 = pd.DataFrame({
 'Column1': [1, 2],
 'Column2': ['A', 'B']
 })
 df2 = pd.DataFrame({
 'Column1': [3, 4],
 'Column2': ['C', 'D']
 })
 expected_output = pd.DataFrame({
 'Column1': [1, 2, 3, 4],
 'Column2': ['A', 'B', 'C', 'D']
 })
 result = concatenate_Aramex_Cosmaline(df1, df2)
 assert result.equals(expected_output)

```

## Explanation

- **test\_convert\_data\_types\_Aramelex\_Data**

- This test checks if the `convert_data_types_Aramelex_Data` function correctly converts the data types of various columns in Aramelex data.
- It uses sample data with mixed data types for columns such as `ShprNo`, `HAWB`, and `Delivery_Date`.

- It asserts that the function returns a DataFrame with the specified columns converted to the correct data types.

- **test\_rename\_columns\_for\_website\_Aramex\_data**

- This test checks if the `rename_columns_for_website_Aramex_data` function correctly renames the `HAWB` column to `AWB`.
- It uses sample data with columns in mixed formats.
- It asserts that the function returns a DataFrame with the `HAWB` column renamed to `AWB`.

- **test\_update\_cod\_currency\_Aramex\_data**

- This test checks if the `update_cod_currency_Aramex_data` function correctly updates the `CODCurrency` column based on the `CODAmount`.
- It uses sample data with `CODAmount` and empty `CODCurrency` columns.
- It asserts that the function returns a DataFrame with the `CODCurrency` column updated correctly.

- **test\_clean\_tokens\_by\_date\_Aramex\_Data**

- This test checks if the `clean_tokens_by_date_Aramex_Data` function correctly cleans the `TOKENNO` column based on the `Aramex_Inv_date`.
- It uses sample data with multiple tokens for the same date.
- It asserts that the function returns a DataFrame with the `TOKENNO` column cleaned by date.

- **test\_add\_order\_number\_Aramex\_Data**

- This test checks if the `add_order_number_Aramex_Data` function correctly merges the `order_number` column from the e-commerce data into the Aramex data based on the `AWB`.
- It uses sample data with matching `AWB` columns in both Aramex and e-commerce data.
- It asserts that the function returns a DataFrame with the `order_number` column correctly merged.

- **test\_convert\_data\_types\_aramecosmaline**

- This test checks if the `convert_data_types_aramecosmaline` function correctly converts the data types of various columns in the Aramex Cosmaline data.
- It uses sample data with mixed data types for columns such as `AWB`, `Delivery_Date`, and `CODAmount`.
- It asserts that the function returns a DataFrame with the specified columns converted to the correct data types.

- **test\_concatenate\_Aramex\_Cosmaline**

- This test checks if the `concatenate_Aramex_Cosmaline` function correctly concatenates two DataFrames.
- It uses sample data with two DataFrames having the same columns.
- It asserts that the function returns a single DataFrame with the rows from both input DataFrames concatenated.

These tests ensure that the Aramex data transformation functions behave correctly and handle the specific data manipulation tasks required for Aramex data.

## 7. Test: test\_transform\_oracle.py

The `test_transform_oracle.py` file contains unit tests for the Oracle data transformation functions. These functions perform various data manipulation tasks specific to Oracle data, such as removing time from datetime values, renaming columns, and converting data types. The tests cover the following functions:

- `remove_time_from_datetime_Oracle_Data`
- `rename_columns_for_Oracle_data`
- `convert_data_types_Oracle_Data`

```
import pytest
import pandas as pd
import sys
import os
from pandas.testing import assert_frame_equal

Add the scripts directory to the system path
script_path = os.path.abspath(os.path.join(os.path.dirname(__file__),
'../../scripts'))
sys.path.append(script_path)

from transform import (
 remove_time_from_datetime_Oracle_Data,
 rename_columns_for_Oracle_data,
 convert_data_types_Oracle_Data
)

def test_remove_time_from_datetime_Oracle_Data():
 input_data = pd.DataFrame({
 'date_column': ['2023-01-01 10:00:00', '2023-01-02 15:30:00']
 })
 expected_output = pd.DataFrame({
 'date_column': pd.to_datetime(['2023-01-01', '2023-01-02'])
 })
 result = remove_time_from_datetime_Oracle_Data(input_data, 'date_column')
 assert_frame_equal(result, expected_output, check_dtype=True)

def test_rename_columns_for_Oracle_data():
 input_data = pd.DataFrame({
 'OPERATING_UNIT_NAME': ['Unit1', 'Unit2'],
 'ORACLE_REFERENCE_ORDER_NUMBER': [123, 456],
 'ECOM_REFERENCE_ORDER_NUMBER': ['ORD001', 'ORD002'],
 'ORDER_TYPE': ['Type1', 'Type2'],
 'ORDER_TYPE_NAME': ['Type Name1', 'Type Name2'],
 'ORDERED_ITEM': ['Item1', 'Item2'],
 'ORDERED_QUANTITY': [1, 2],
 'PRICING_QUANTITY_UOM': ['UOM1', 'UOM2'],
 'UNIT_SELLING_PRICE (after discount without vat)': [100.0, 200.0],
 'UNIT_LIST_PRICE(original price before discount without vat)': [150.0,
250.0],
 'ORDERED_DATE': ['2023-01-01', '2023-01-02'],
 'CUSTOMER_NAME': ['Customer1', 'Customer2'],
 })
```

```
'TAX_CODE': ['TAX1', 'TAX2']
})
expected_output = input_data.rename(columns={
 'OPERATING_UNIT_NAME': 'operating_unit_name',
 'ORACLE_REFERENCE_ORDER_NUMBER': 'oracle_reference_order_number',
 'ECOM_REFERENCE_ORDER_NUMBER': 'ecom_reference_order_number',
 'ORDER_TYPE': 'order_type',
 'ORDER_TYPE_NAME': 'order_type_name',
 'ORDERED_ITEM': 'ordered_item',
 'ORDERED_QUANTITY': 'ordered_quantity',
 'PRICING_QUANTITY_UOM': 'pricing_quantity_uom',
 'UNIT_SELLING_PRICE (after discount without vat)': 'unit_selling_price',
 'UNIT_LIST_PRICE(original price before discount without vat)': 'unit_list_price',
 'ORDERED_DATE': 'ordered_date',
 'CUSTOMER_NAME': 'customer_name',
 'TAX_CODE': 'tax_code'
})
result = rename_columns_for_Oracle_data(input_data)
assert_frame_equal(result, expected_output, check_dtype=True)

def test_convert_data_types_Oracle_Data():
 input_data = pd.DataFrame({
 'operating_unit_name': ['Unit1', 'Unit2'],
 'oracle_reference_order_number': [123, 456],
 'ecom_reference_order_number': ['ORD001', 'ORD002'],
 'order_type': ['Type1', 'Type2'],
 'order_type_name': ['Type Name1', 'Type Name2'],
 'ordered_item': ['Item1', 'Item2'],
 'ordered_quantity': [1, 2],
 'pricing_quantity_uom': ['UOM1', 'UOM2'],
 'unit_selling_price': [100.0, 200.0],
 'unit_list_price': [150.0, 250.0],
 'ordered_date': ['2023-01-01', '2023-01-02'],
 'customer_name': ['Customer1', 'Customer2'],
 'tax_code': ['TAX1', 'TAX2']
 })
 expected_output = input_data.astype({
 'operating_unit_name': 'object',
 'oracle_reference_order_number': 'int32',
 'ecom_reference_order_number': 'object',
 'order_type': 'object',
 'order_type_name': 'object',
 'ordered_item': 'object',
 'ordered_quantity': 'int32',
 'pricing_quantity_uom': 'object',
 'unit_selling_price': 'float32',
 'unit_list_price': 'float32',
 'ordered_date': 'datetime64[ns]',
 'customer_name': 'object',
 'tax_code': 'object'
 })
 expected_output['ordered_date'] =
 pd.to_datetime(expected_output['ordered_date'])
```

```
result = convert_data_types_Oracle_Data(input_data)
assert_frame_equal(result, expected_output, check_dtype=True)
```

## Explanation

- **test\_remove\_time\_from\_datetime\_Oracle\_Data**

- This test checks if the `remove_time_from_datetime_Oracle_Data` function correctly removes the time portion from datetime values in the specified column.
- It uses sample data with datetime values including time.
- It asserts that the function returns a DataFrame with the time portion removed from the datetime values.

- **test\_rename\_columns\_for\_Oracle\_data**

- This test checks if the `rename_columns_for_Oracle_data` function correctly renames the columns to a standardized format.
- It uses sample data with columns in mixed formats.
- It asserts that the function returns a DataFrame with columns renamed to a consistent format.

- **test\_convert\_data\_types\_Oracle\_Data**

- This test checks if the `convert_data_types_Oracle_Data` function correctly converts the data types of various columns in Oracle data.
- It uses sample data with mixed data types for columns such as `operating_unit_name`, `ordered_quantity`, and `ordered_date`.
- It asserts that the function returns a DataFrame with the specified columns converted to the correct data types.

These tests ensure that the Oracle data transformation functions behave correctly and handle the specific data manipulation tasks required for Oracle data.

## 8. Test: `test_transform_creditCard.py`

The `test_transform_creditCard.py` file contains unit tests for the Credit Card data transformation functions. These functions perform data type conversions specific to credit card transaction data. The test covers the following function:

- `convert_data_types_Credit_Card`

```
import pytest
import pandas as pd
import sys
import os
from pandas.testing import assert_frame_equal

Add the scripts directory to the system path
script_path = os.path.abspath(os.path.join(os.path.dirname(__file__),
'../../scripts'))
sys.path.append(script_path)
```

```

from transform import convert_data_types_Credit_Card

def test_convert_data_types_Credit_Card():
 input_data = pd.DataFrame({
 'Value_date': ['2023-01-01', '2023-01-02'],
 'Narrative': ['Transaction 1', 'Transaction 2'],
 'Amount': [100.0, 200.0],
 'Currency': ['USD', 'EUR']
 })
 expected_output = input_data.astype({
 'Value_date': 'datetime64[ns]',
 'Narrative': 'str',
 'Amount': 'float32',
 'Currency': 'str'
 })
 result = convert_data_types_Credit_Card(input_data)
 assert_frame_equal(result, expected_output, check_dtype=True)

```

## Explanation

- **test\_convert\_data\_types\_Credit\_Card**

- This test checks if the `convert_data_types_Credit_Card` function correctly converts the data types of various columns in credit card transaction data.
- It uses sample data with mixed data types for columns such as `Value_date`, `Narrative`, `Amount`, and `Currency`.
- It asserts that the function returns a DataFrame with the specified columns converted to the correct data types.

This test ensures that the Credit Card data transformation functions behave correctly and handle the specific data manipulation tasks required for credit card transaction data.

## 9. Test: `test_reconciliation.py`

The `test_reconciliation.py` file contains unit tests for the reconciliation script execution functions. These functions handle the execution of SQL reconciliation scripts and manage database sessions. The tests cover the following functions:

- `execute_reconciliation_script`

```

import pytest
from unittest.mock import patch, MagicMock, mock_open
import sqlalchemy as sa
import sys
import os

Add the scripts directory to the system path
script_path = os.path.abspath(os.path.join(os.path.dirname(__file__),
'../../scripts'))
sys.path.append(script_path)

```

```
from reconciliation import execute_reconciliation_script

Mocking the create_db_engine function from load.py
@patch('reconciliation.create_db_engine')
@patch('reconciliation.sessionmaker')
@patch('builtins.open', new_callable=mock_open, read_data='SELECT * FROM test_table;')
def test_execute_reconciliation_script(mock_open, mock_sessionmaker,
mock_create_db_engine):
 # Mock engine and session
 mock_engine = MagicMock()
 mock_create_db_engine.return_value = mock_engine

 mock_session = MagicMock()
 mock_sessionmaker.return_value = MagicMock(return_value=mock_session)

 # Call the function
 execute_reconciliation_script('dummy_script.sql')

 # Check that create_db_engine was called
 mock_create_db_engine.assert_called_once()

 # Check that sessionmaker was called with the mock engine
 mock_sessionmaker.assert_called_once_with(bind=mock_engine)

 # Check that the SQL script was read from the file
 mock_open.assert_called_once_with('/opt/airflow/scripts/dummy_script.sql',
'r')

 # Extract the executed SQL statement text
 executed_sql = mock_session.execute.call_args[0][0].text.strip()
 expected_sql = sa.text('SELECT * FROM test_table').text.strip()

 # Check that the SQL statements were executed
 assert executed_sql == expected_sql

 # Check that the session commit was called
 mock_session.commit.assert_called_once()

 # Check that the session was closed
 mock_session.close.assert_called_once()

Test case for exception handling
@patch('reconciliation.create_db_engine')
@patch('reconciliation.sessionmaker')
@patch('builtins.open', new_callable=mock_open, read_data='SELECT * FROM test_table;')
def test_execute_reconciliation_script_exception(mock_open, mock_sessionmaker,
mock_create_db_engine):
 # Mock engine and session
 mock_engine = MagicMock()
 mock_create_db_engine.return_value = mock_engine
```

```
mock_session = MagicMock()
mock_sessionmaker.return_value = MagicMock(return_value=mock_session)

Configure the session to raise an exception on execute
mock_session.execute.side_effect = Exception('Execution error')

Call the function
execute_reconciliation_script('dummy_script.sql')

Check that the session rollback was called
mock_session.rollback.assert_called_once()

Check that the session was closed
mock_session.close.assert_called_once()

if __name__ == '__main__':
 pytest.main()
```

## Explanation

- **test\_execute\_reconciliation\_script**
  - This test checks if the `execute_reconciliation_script` function correctly executes a SQL script and manages the database session.
  - It uses `patch` to mock the `create_db_engine`, `sessionmaker`, and `open` functions.
  - It mocks the database engine and session, ensuring the SQL script is read and executed correctly.
  - It asserts that the `create_db_engine` function is called once.
  - It asserts that the `sessionmaker` function is called with the mock engine.
  - It asserts that the SQL script is read from the file.
  - It checks that the correct SQL statement is executed.
  - It asserts that the session is committed and closed properly.
- **test\_execute\_reconciliation\_script\_exception**
  - This test checks if the `execute_reconciliation_script` function correctly handles exceptions during SQL script execution.
  - It uses `patch` to mock the `create_db_engine`, `sessionmaker`, and `open` functions.
  - It mocks the database engine and session, configuring the session to raise an exception on execute.
  - It asserts that the session rollback is called when an exception occurs.
  - It asserts that the session is closed properly after an exception.

These tests ensure that the reconciliation script execution functions behave correctly and handle the specific tasks required for executing SQL scripts and managing database sessions.

## 10. Test: `test_email_notifications.py`

The `test_email_notifications.py` file contains unit tests for the email notification functions. These functions are responsible for sending email notifications when tasks succeed or fail in the Airflow DAG system. The tests cover the following functions:

- `format_failure_email_body`
- `format_success_email_body`
- `success_email`
- `failure_email`

```
import pytest
from unittest.mock import patch, MagicMock
import sys
import os
script_path = os.path.abspath(os.path.join(os.path.dirname(__file__),
'../../scripts'))
sys.path.append(script_path)

import pytest
from unittest.mock import patch, MagicMock, mock_open
import importlib

Mock environment variables to bypass Airflow initialization issues
@pytest.fixture(autouse=True)
def mock_airflow_env(monkeypatch):
 monkeypatch.setenv('AIRFLOW__CORE__SQLALCHEMY_CONN',
'sqlite:///tmp/airflow.db')
 monkeypatch.setenv('AIRFLOW__CORE__LOAD_EXAMPLES', 'False')
 monkeypatch.setenv('AIRFLOW__CORE__EXECUTOR', 'SequentialExecutor')

Mocking the send_email function from Airflow
@pytest.fixture(autouse=True)
def mock_airflow_imports(monkeypatch):
 mock_send_email = MagicMock()
 monkeypatch.setattr('airflow.utils.email.send_email', mock_send_email)

 # Dynamically import the email_notifications module after mocking
 global email_notifications
 email_notifications = importlib.import_module('email_notifications')

def test_format_failure_email_body():
 task_instance_mock = MagicMock()
 task_instance_mock.task_id = 'dummy_task'
 task_instance_mock.dag_id = 'dummy_dag'
 task_instance_mock.execution_date = '2023-01-01'
 task_instance_mock.log_url = 'http://dummy_log_url'

 expected_body = f"""
<html>
<body>
 <h2>ETL Automated Process Notification</h2>
 <p>Task: dummy_task</p>
 <p>Status: Failed</p>
 <p>DAG: dummy_dag</p>
 <p>Execution Date: 2023-01-01</p>
 <p>Log URL: http://dummy_log_url</p>

```

```


<p>Best Regards,</p>
<p>Your ETL Automation System</p>
</body>
</html>
"""

actual_body =
email_notifications.format_failure_email(task_instance_mock, 'Failed')
assert actual_body.strip() == expected_body.strip()

def test_format_success_email():
 expected_body = f"""
<html>
<body>
 <h2>ETL Automated Process Notification</h2>
 <p>The ETL process has completed successfully.</p>
 <p>All tasks have been executed without any issues.</p>

 <p>Best Regards,</p>
 <p>Your ETL Automation System</p>
</body>
</html>
"""

 actual_body = email_notifications.format_success_email()
 assert actual_body.strip() == expected_body.strip()

Mocking the send_email function for success_email
@patch('email_notifications.send_email')
def test_success_email(mock_send_email):
 context = {}
 email_notifications.success_email(context)

 subject = 'ETL Process Success'
 body = email_notifications.format_success_email_body()
 to_email = ['rar37@mail.aub.edu', 'hmf31@mail.aub.edu']

 mock_send_email.assert_called_once_with(to=to_email, subject=subject,
 html_content=body)

Mocking the send_email function for failure_email
@patch('email_notifications.send_email')
def test_failure_email(mock_send_email):
 task_instance_mock = MagicMock()
 task_instance_mock.task_id = 'dummy_task'
 task_instance_mock.dag_id = 'dummy_dag'
 task_instance_mock.execution_date = '2023-01-01'
 task_instance_mock.log_url = 'http://dummy_log_url'

 context = {'task_instance': task_instance_mock}
 email_notifications.failure_email(context)

 subject = 'Airflow Task dummy_task Failed'
 body = email_notifications.format_failure_email_body(task_instance_mock,
 'Failed')
```

```
to_email = ['rar37@mail.aub.edu', 'hmf31@mail.aub.edu']

mock_send_email.assert_called_once_with(to=to_email, subject=subject,
html_content=body)
```

## Explanation

- **test\_format\_failure\_email\_body**

- This test checks if the `format_failure_email_body` function correctly formats the email body for a task failure notification.
- It uses a mock task instance with attributes such as `task_id`, `dag_id`, `execution_date`, and `log_url`.
- It asserts that the function returns the expected HTML formatted email body.

- **test\_format\_success\_email\_body**

- This test checks if the `format_success_email_body` function correctly formats the email body for a success notification.
- It asserts that the function returns the expected HTML formatted email body for a successful ETL process.

- **test\_success\_email**

- This test checks if the `success_email` function correctly sends a success email using the `send_email` function from Airflow.
- It uses `patch` to mock the `send_email` function.
- It asserts that the `send_email` function is called once with the correct subject, body, and recipient email addresses.

- **test\_failure\_email**

- This test checks if the `failure_email` function correctly sends a failure email using the `send_email` function from Airflow.
- It uses `patch` to mock the `send_email` function.
- It asserts that the `send_email` function is called once with the correct subject, body, and recipient email addresses.

These tests ensure that the email notification functions behave correctly and handle the specific tasks required for sending task success and failure notifications.

## 6.2 Integration Testing

Integration testing focuses on verifying the interactions between different components of the system to ensure they work together correctly. In the Malia Project, pytest is used to create and run integration tests. These tests are designed to validate the end-to-end functionality of the Airflow DAG system by testing the complete workflow, including data extraction, transformation, loading, and reconciliation processes.

### 1. Test: `test_transform_oracle_pipeline.py`

The `test_transform_oracle_pipeline.py` file contains an integration test for the Oracle data transformation pipeline. This pipeline processes Oracle data through a series of transformation steps, ensuring that the data is correctly formatted and ready for further analysis or integration. The test covers the following function:

- `data_preprocessing_pipeline_Oracle_Data`

```
import pytest
import pandas as pd
import os
import sys

script_path = os.path.abspath(os.path.join(os.path.dirname(__file__),
'../../scripts'))
sys.path.append(script_path)
from transform_pipelines import data_preprocessing_pipeline_Oracle_Data

@pytest.fixture
def sample_oracle_data():
 df = pd.DataFrame({
 'OPERATING_UNIT_NAME': ['Unit1', 'Unit2', 'Unit3'],
 'ORACLE_REFERENCE_ORDER_NUMBER': [12345, 67890, 13579],
 'ECOM_REFERENCE_ORDER_NUMBER': [111, 222, 333],
 'ORDER_TYPE': ['Type1', 'Type2', 'Type3'],
 'ORDER_TYPE_NAME': ['Name1', 'Name2', 'Name3'],
 'ORDERED_ITEM': ['Item1', 'Item2', 'Item3'],
 'ORDERED_QUANTITY': [10, 20, 30],
 'PRICING_QUANTITY_UOM': ['UOM1', 'UOM2', 'UOM3'],
 'UNIT_SELLING_PRICE (after discount without vat)': [100.5, 200.75,
300.00],
 'UNIT_LIST_PRICE(original price before discount without vat)': [120.5,
220.75, 320.00],
 'ORDERED_DATE': ['2023-01-01 10:00:00', '2023-01-02 11:00:00', '2023-01-03
12:00:00'],
 'CUSTOMER_NAME': ['Customer1', 'Customer2', 'Customer3'],
 'TAX_CODE': ['TAX1', 'TAX2', 'TAX3']
 })
 df['ECOM_REFERENCE_ORDER_NUMBER'] =
 df['ECOM_REFERENCE_ORDER_NUMBER'].astype('str')
 return df

@pytest.fixture
def expected_output_oracle_data():
 expected_df = pd.DataFrame({
 'operating_unit_name': ['Unit1', 'Unit2', 'Unit3'],
 'oracle_reference_order_number': pd.Series([12345, 67890, 13579],
dtype='int32'),
 'ecom_reference_order_number': pd.Series(['111', '222', '333'],
dtype='object'),
 'order_type': ['Type1', 'Type2', 'Type3'],
 'order_type_name': ['Name1', 'Name2', 'Name3'],
 'ordered_item': ['Item1', 'Item2', 'Item3'],
 })
```

```

'ordered_quantity': pd.Series([10, 20, 30], dtype='int32'),
'pricing_quantity_uom': ['UOM1', 'UOM2', 'UOM3'],
'unit_selling_price': pd.Series([100.5, 200.75, 300.00], dtype='float32'),
'unit_list_price': pd.Series([120.5, 220.75, 320.00], dtype='float32'),
'ordered_date': pd.to_datetime(['2023-01-01', '2023-01-02', '2023-01-
03']),
'customer_name': ['Customer1', 'Customer2', 'Customer3'],
'tax_code': ['TAX1', 'TAX2', 'TAX3']
})
return expected_df

def test_data_preprocessing_pipeline_oracle_data(sample_oracle_data,
expected_output_oracle_data):
 transformed_data = data_preprocessing_pipeline_Oracle_Data(sample_oracle_data)

 # Ensure the data types match
 transformed_data['oracle_reference_order_number'] =
 transformed_data['oracle_reference_order_number'].astype('int32')
 transformed_data['ordered_quantity'] =
 transformed_data['ordered_quantity'].astype('int32')
 transformed_data['ecom_reference_order_number'] =
 transformed_data['ecom_reference_order_number'].astype('object')

 try:
 pd.testing.assert_frame_equal(transformed_data,
expected_output_oracle_data)
 except AssertionError as e:
 print("AssertionError:", e)
 print("Transformed DataFrame:\n", transformed_data)
 print("Transformed DataFrame Types:\n", transformed_data.dtypes)
 print("Expected DataFrame:\n", expected_output_oracle_data)
 print("Expected DataFrame Types:\n", expected_output_oracle_data.dtypes)
 raise

```

## Explanation

- **test\_data\_preprocessing\_pipeline\_oracle\_data**
  - This test verifies the end-to-end functionality of the `data_preprocessing_pipeline_Oracle_Data` function.
  - It uses sample Oracle data with various columns including `OPERATING_UNIT_NAME`, `ORACLE_REFERENCE_ORDER_NUMBER`, `ECOM_REFERENCE_ORDER_NUMBER`, `ORDER_TYPE`, `ORDERED_DATE`, etc.
  - The test checks if the pipeline correctly processes the data, converting data types and renaming columns as required.
  - The test asserts that the transformed data matches the expected output DataFrame in both values and data types.

This test ensures that the Oracle data transformation pipeline functions correctly, handling the specific data manipulation tasks required for Oracle data.

## 2. Test: `test_transform_erp_pipelines.py`

The `test_transform_erp_pipelines.py` file contains an integration test for the ERP data transformation pipeline. This pipeline processes ERP data through a series of transformation steps, ensuring that the data is correctly formatted and ready for further analysis or integration. The test covers the following function:

- `data_preprocessing_pipeline_ERP_Data`

```
import pytest
import pandas as pd
import os
import sys
script_path = os.path.abspath(os.path.join(os.path.dirname(__file__),
'../../scripts'))
sys.path.append(script_path)
from transform_pipelines import data_preprocessing_pipeline_ERP_Data

@pytest.fixture
def sample_erp_data():
 df = pd.DataFrame({
 'CUSTOMER_NUMBER': [123, 456, 789],
 'CUSTOMER_NAME': ['Acme Corp', 'Widget Works', 'Gizmo Co'],
 'RECEIPT_CLASS': ['PREMIUM', 'Standard', 'basic'],
 'CURRENCY_CODE': ['usd', 'eur', 'GBP'],
 'COMMENTS': ['Payment for Token A123', 'Refund Token B456', 'Charge for
account']
 })
 return df

@pytest.fixture
def expected_output_erp_data():
 expected_df = pd.DataFrame({
 'CUSTOMER_NUMBER': ['123', '456', '789'],
 'CUSTOMER_NAME': ['Acme corp', 'Widget works', 'Gizmo co'],
 'RECEIPT_CLASS': ['Premium', 'Standard', 'Basic'],
 'CURRENCY_CODE': ['USD', 'EUR', 'GBP'],
 'COMMENTS': ['A123', 'B456', 'Charge for account']
 })
 expected_df['CUSTOMER_NUMBER'] = expected_df['CUSTOMER_NUMBER'].astype('str')
 return expected_df

def test_data_preprocessing_pipeline_erp_data(sample_erp_data,
expected_output_erp_data):
 transformed_data = data_preprocessing_pipeline_ERP_Data(sample_erp_data)

 # Ensure CUSTOMER_NUMBER is string in both dataframes
 transformed_data['CUSTOMER_NUMBER'] =
 transformed_data['CUSTOMER_NUMBER'].astype('str')

 pd.testing.assert_frame_equal(transformed_data, expected_output_erp_data)
```

## Explanation

- **test\_data\_preprocessing\_pipeline\_erp\_data**

- This test verifies the end-to-end functionality of the `data_preprocessing_pipeline_ERP_Data` function.
- It uses sample ERP data with various columns including `CUSTOMER_NUMBER`, `CUSTOMER_NAME`, `RECEIPT_CLASS`, `CURRENCY_CODE`, and `COMMENTS`.
- The test checks if the pipeline correctly processes the data by converting data types, trimming spaces, standardizing text formats, and extracting tokens from comments.
- The test asserts that the transformed data matches the expected output DataFrame in both values and data types.

This test ensures that the ERP data transformation pipeline functions correctly, handling the specific data manipulation tasks required for ERP data.

### 3. Test: `test_transform_ecom_pipeline.py`

The `test_transform_ecom_pipeline.py` file contains an integration test for the e-commerce data transformation pipeline. This pipeline processes e-commerce data through a series of transformation steps, ensuring that the data is correctly formatted and ready for further analysis or integration. The test covers the following function:

- `data_preprocessing_pipeline_ECOM_Data`

```
import pytest
import pandas as pd
import os
import sys
script_path = os.path.abspath(os.path.join(os.path.dirname(__file__), '../..../scripts'))
sys.path.append(script_path)
from transform_pipelines import data_preprocessing_pipeline_ECOM_Data

@pytest.fixture
def sample_ecom_data():
 return pd.DataFrame({
 'Order Number': ['123', '124', '125'],
 'Shipper Name': ['FedEx', 'UPS', 'DHL'],
 'Created At': ['2023-01-01 10:00:00', '2023-01-02 11:00:00', '2023-01-03 12:00:00'],
 'Delivered At': ['2023-01-02 10:00:00', '2023-01-03 11:00:00', '2023-01-04 12:00:00'],
 'Billing Type': ['Prepaid', 'Postpaid', 'prepaid'],
 'Amount': ['100.50', '200.75', '300.00'],
 'Currency': ['usd', 'eur', 'gbp'],
 'Country': ['us', 'fr', 'de'],
 'AWB': ['123-456', '234-567', '345-678']
 })

@pytest.fixture
def expected_output_ecom_data():
 return pd.DataFrame({
 'order_number': ['123', '124', '125'],
 'customer_name': ['Customer A', 'Customer B', 'Customer C'],
 'receipt_class': ['Class A', 'Class B', 'Class C'],
 'currency_code': ['USD', 'EUR', 'GBP'],
 'comments': ['Comments 1', 'Comments 2', 'Comments 3']
 })
```

```

 'shipper_name': ['Fedex', 'Ups', 'Dhl'],
 'created_at': pd.to_datetime(['2023-01-01 10:00:00', '2023-01-02
11:00:00', '2023-01-03 12:00:00']),
 'delivered_at': pd.to_datetime(['2023-01-02 10:00:00', '2023-01-03
11:00:00', '2023-01-04 12:00:00']),
 'billing_type': ['Prepaid', 'Postpaid', 'Prepaid'],
 'amount': [100.50, 200.75, 300.00],
 'currency': ['USD', 'EUR', 'GBP'],
 'country': ['US', 'FR', 'DE'],
 'AWB': ['123-456', '234-567', '345-678']
 })
}

def test_data_preprocessing_pipeline_ecom_data(sample_ecom_data,
expected_output_ecom_data):
 transformed_data = data_preprocessing_pipeline_ECOM_Data(sample_ecom_data)
 pd.testing.assert_frame_equal(transformed_data, expected_output_ecom_data)

```

## Explanation

- **test\_data\_preprocessing\_pipeline\_ecom\_data**
  - This test verifies the end-to-end functionality of the `data_preprocessing_pipeline_ECOM_Data` function.
  - It uses sample e-commerce data with various columns including `Order Number`, `Shipper Name`, `Created At`, `Delivered At`, `Billing Type`, `Amount`, `Currency`, `Country`, and `AWB`.
  - The test checks if the pipeline correctly processes the data by trimming spaces, converting data types, standardizing text formats, and renaming columns.
  - The test asserts that the transformed data matches the expected output DataFrame in both values and data types.

This test ensures that the e-commerce data transformation pipeline functions correctly, handling the specific data manipulation tasks required for e-commerce data.

## 4. Test: `test_transform_creditCard_pipelines.py`

The `test_transform_creditCard_pipelines.py` file contains an integration test for the credit card data transformation pipeline. This pipeline processes credit card data through a series of transformation steps, ensuring that the data is correctly formatted and ready for further analysis or integration. The test covers the following function:

- `data_preprocessing_pipeline_CreditCard_Data`

```

import pytest
import pandas as pd
import os
import sys
script_path = os.path.abspath(os.path.join(os.path.dirname(__file__),
'../../scripts'))
sys.path.append(script_path)
from transform_pipelines import data_preprocessing_pipeline_CreditCard_Data

```

```

@pytest.fixture
def sample_credit_card_data():
 return pd.DataFrame({
 'Value_date': ['2023-01-01', '2023-01-02', '2023-01-03'],
 'Narrative': ['payment', 'refund', 'chargeback'],
 'Amount': ['100.50', '200.75', '300.00'],
 'Currency': ['usd', 'eur', 'gbp']
 })

@pytest.fixture
def expected_output_credit_card_data():
 return pd.DataFrame({
 'value_date': pd.to_datetime(['2023-01-01', '2023-01-02', '2023-01-03']),
 'narrative': ['PAYMENT', 'REFUND', 'CHARGEBACK'],
 'amount': pd.Series([100.50, 200.75, 300.00], dtype='float32'),
 'currency': ['USD', 'EUR', 'GBP']
 })

def test_data_preprocessing_pipeline_credit_card_data(sample_credit_card_data,
expected_output_credit_card_data):
 transformed_data =
 data_preprocessing_pipeline_CreditCard_Data(sample_credit_card_data)
 pd.testing.assert_frame_equal(transformed_data,
expected_output_credit_card_data)

```

## Explanation

- **test\_data\_preprocessing\_pipeline\_credit\_card\_data**
  - This test verifies the end-to-end functionality of the `data_preprocessing_pipeline_CreditCard_Data` function.
  - It uses sample credit card data with various columns including `Value_date`, `Narrative`, `Amount`, and `Currency`.
  - The test checks if the pipeline correctly processes the data by converting data types, trimming spaces, and standardizing text formats.
  - The test asserts that the transformed data matches the expected output DataFrame in both values and data types.

This test ensures that the credit card data transformation pipeline functions correctly, handling the specific data manipulation tasks required for credit card data.

## 5. Test: `test_transform_cosmaline_pipeline.py`

The `test_transform_cosmaline_pipeline.py` file contains an integration test for the Cosmaline data transformation pipeline. This pipeline processes Cosmaline data through a series of transformation steps, ensuring that the data is correctly formatted and ready for further analysis or integration. The test covers the following functions:

- `data_preprocessing_pipeline_ECOM_Data`
- `data_preprocessing_pipeline_Cosmaline_Data`

```
import pytest
import pandas as pd
import os
import sys
script_path = os.path.abspath(os.path.join(os.path.dirname(__file__),
'../../scripts'))
sys.path.append(script_path)
from transform_pipelines import data_preprocessing_pipeline_ECOM_Data,
data_preprocessing_pipeline_Cosmaline_Data

@pytest.fixture
def sample_ecom_data():
 return pd.DataFrame({
 'Order Number': ['123', '124', '125'],
 'Shipper Name': ['Fedex', 'Ups', 'Dhl'],
 'Created At': ['2023-01-01 10:00:00', '2023-01-02 11:00:00', '2023-01-03
12:00:00'],
 'Delivered At': ['2023-01-02 10:00:00', '2023-01-03 11:00:00', '2023-01-04
12:00:00'],
 'Billing Type': ['Prepaid', 'Postpaid', 'Prepaid'],
 'Amount': [100.50, 200.75, 300.00],
 'Currency': ['USD', 'EUR', 'GBP'],
 'Country': ['US', 'FR', 'DE'],
 'AWB': ['123-456', '234-567', '345-678']
 })

@pytest.fixture
def sample_cosmaline_data():
 return pd.DataFrame({
 'Currency': ['usd', 'lbp', 'lbp'],
 'Driver_Delivery_date': ['2023-01-02', '2023-01-03', '2023-01-04'],
 'Amount': [0, 100.00, 200.00],
 'OrderNo': ['123', '124', '125']
 })

@pytest.fixture
def expected_output_cosmaline_data():
 expected_df = pd.DataFrame({
 'CODCurrency': ['USD', 'LBP', 'LBP'],
 'Delivery_Date': pd.to_datetime(['2023-01-02', '2023-01-03', '2023-01-
04']),
 'O_CODAmount': pd.Series([0, 100.00, 200.00], dtype='float32'),
 'order_number': ['123', '124', '125'],
 'CODAmount': pd.Series([0, 100.00, 200.00], dtype='float32'),
 'ShprNo': pd.NA,
 'Aramex_Inv_date': pd.to_datetime([pd.NaT, pd.NaT, pd.NaT]),
 'TOKENNO': 'Shipped With Cosmaline',
 'AWB': ['123-456', '234-567', '345-678']
 })
 return expected_df

def test_data_preprocessing_pipeline_cosmaline_data(sample_cosmaline_data,
sample_ecom_data, expected_output_cosmaline_data):
```

```

First preprocess the ECOM data
processed_ecom_data = data_preprocessing_pipeline_ECOM_Data(sample_ecom_data)

Then preprocess the Cosmaline data
transformed_cosmaline_data =
data_preprocessing_pipeline_Cosmaline_Data(sample_cosmaline_data,
processed_ecom_data)

pd.testing.assert_frame_equal(transformed_cosmaline_data,
expected_output_cosmaline_data)

```

## Explanation

- **test\_data\_preprocessing\_pipeline\_cosmaline\_data**
  - This test verifies the end-to-end functionality of the `data_preprocessing_pipeline_Cosmaline_Data` function.
  - It uses sample e-commerce and Cosmaline data with various columns including `Order Number`, `Currency`, `Driver_Delivery_date`, and `Amount`.
  - The test checks if the pipeline correctly processes the e-commerce data first and then uses it to further process the Cosmaline data.
  - It asserts that the transformed Cosmaline data matches the expected output DataFrame in both values and data types.

This test ensures that the Cosmaline data transformation pipeline functions correctly, handling the specific data manipulation tasks required for Cosmaline data.

## 6. Test: `test_transform_arameix_pipeline.py`

The `test_transform_arameix_pipeline.py` file contains an integration test for the Aramex data transformation pipeline. This pipeline processes Aramex data through a series of transformation steps, ensuring that the data is correctly formatted and ready for further analysis or integration. The test covers the following functions:

- `data_preprocessing_pipeline_ECOM_Data`
- `data_preprocessing_pipeline_Aramex_Data`

```

import pytest
import pandas as pd
import os
import sys
script_path = os.path.abspath(os.path.join(os.path.dirname(__file__),
'../../scripts'))
sys.path.append(script_path)
from transform_pipelines import data_preprocessing_pipeline_ECOM_Data,
data_preprocessing_pipeline_Aramex_Data

@pytest.fixture
def sample_ecom_data():
 return pd.DataFrame({
 'Order Number': ['123', '124', '125'],

```

```
'Shipper Name': ['Fedex', 'Ups', 'Dhl'],
'Created At': ['2023-01-01 10:00:00', '2023-01-02 11:00:00', '2023-01-03
12:00:00'],
'Delivered At': ['2023-01-02 10:00:00', '2023-01-03 11:00:00', '2023-01-04
12:00:00'],
'Billing Type': ['Prepaid', 'Postpaid', 'Prepaid'],
'Amount': [100.50, 200.75, 300.00],
'Currency': ['USD', 'EUR', 'GBP'],
'Country': ['US', 'FR', 'DE'],
'AWB': ['123-456', '234-567', '345-678']
})

@pytest.fixture
def sample_aramex_data():
 return pd.DataFrame({
 'ShprNo': ['S1', 'S2', 'S3'],
 'HAWB': ['123-456', '234-567', '345-678'],
 'Delivery_Date': ['2023-01-02', '2023-01-03', '2023-01-04'],
 'CODAmount': [150.00, 250.00, 350.00],
 'O_CODAmount': [100.00, 200.00, 300.00],
 'Aramex_Inv_date': ['2023-01-02', '2023-01-03', '2023-01-04'],
 'TOKENNO': ['T1', 'T2', 'T3'],
 'CODCurrency': ['usd', 'eur', 'gbp']
 })

@pytest.fixture
def expected_output_aramex_data():
 return pd.DataFrame({
 'ShprNo': ['S1', 'S2', 'S3'],
 'AWB': ['123-456', '234-567', '345-678'],
 'Delivery_Date': pd.to_datetime(['2023-01-02', '2023-01-03', '2023-01-
04']),
 'CODAmount': pd.Series([150.00, 250.00, 350.00], dtype='float32'),
 'O_CODAmount': pd.Series([100.00, 200.00, 300.00], dtype='float32'),
 'Aramex_Inv_date': pd.to_datetime(['2023-01-02', '2023-01-03', '2023-01-
04']),
 'TOKENNO': ['T1', 'T2', 'T3'],
 'CODCurrency': ['LBP', 'LBP', 'LBP'],
 'order_number': ['123', '124', '125']
 })

def test_data_preprocessing_pipeline_aramex_data(sample_aramex_data,
sample_ecom_data, expected_output_aramex_data):
 # First preprocess the ECOM data
 processed_ecom_data = data_preprocessing_pipeline_ECOM_Data(sample_ecom_data)

 # Then preprocess the Aramex data
 transformed_aramex_data =
 data_preprocessing_pipeline_Aramex_Data(sample_aramex_data, processed_ecom_data)

 pd.testing.assert_frame_equal(transformed_aramex_data,
expected_output_aramex_data)
```

## Explanation

- **test\_data\_preprocessing\_pipeline\_aramelex\_data**
  - This test verifies the end-to-end functionality of the `data_preprocessing_pipeline_Aramex_Data` function.
  - It uses sample e-commerce and Aramex data with various columns including `Order Number`, `ShprNo`, `HAWB`, `Delivery Date`, `CODAmount`, and `O_CODAmount`.
  - The test checks if the pipeline correctly processes the e-commerce data first and then uses it to further process the Aramex data.
  - It asserts that the transformed Aramex data matches the expected output DataFrame in both values and data types.

This test ensures that the Aramex data transformation pipeline functions correctly, handling the specific data manipulation tasks required for Aramex data.

## 7. Test: `test_transform_aramelex_cosmaline_pipeline.py`

The `test_transform_aramelex_cosmaline_pipeline.py` file contains an integration test for the combined Aramex and Cosmaline data transformation pipeline. This pipeline processes both Aramex and Cosmaline data through a series of transformation steps, ensuring that the data is correctly formatted and concatenated for further analysis or integration. The test covers the following functions:

- `data_preprocessing_pipeline_ECOM_Data`
- `data_preprocessing_pipeline_Aramex_Data`
- `data_preprocessing_pipeline_Cosmaline_Data`
- `concatenate_Aramex_Cosmaline`
- `data_preprocessing_pipeline_Cosmaline_Aramex_Data`

```
import pytest
import pandas as pd
import os
import sys
script_path = os.path.abspath(os.path.join(os.path.dirname(__file__),
'../../scripts'))
sys.path.append(script_path)
from transform import concatenate_Aramex_Cosmaline

from transform_pipelines import (
 data_preprocessing_pipeline_ECOM_Data,
 data_preprocessing_pipeline_Aramex_Data,
 data_preprocessing_pipeline_Cosmaline_Data,
 data_preprocessing_pipeline_Cosmaline_Aramex_Data
)

@pytest.fixture
def sample_ecom():
 return pd.DataFrame({
 'Order Number': ['123', '124', '125'],
 'Shipper Name': ['Fedex', 'Ups', 'Dhl'],
 'Created At': ['2023-01-01 10:00:00', '2023-01-02 11:00:00', '2023-01-03']
 })
```

```
12:00:00'],
 'Delivered At': ['2023-01-02 10:00:00', '2023-01-03 11:00:00', '2023-01-04
12:00:00'],
 'Billing Type': ['Prepaid', 'Postpaid', 'Prepaid'],
 'Amount': [100.50, 200.75, 300.00],
 'Currency': ['USD', 'EUR', 'GBP'],
 'Country': ['US', 'FR', 'DE'],
 'AWB': ['123-456', '234-567', '345-678']
})
}

@pytest.fixture
def sample_aramex_data():
 return pd.DataFrame({
 'ShprNo': ['S1', 'S2', 'S3'],
 'HAWB': ['123-456', '234-567', '345-678'],
 'Delivery_Date': ['2023-01-02', '2023-01-03', '2023-01-04'],
 'CODAmount': [150.00, 250.00, 350.00],
 'O_CODAmount': [100.00, 200.00, 300.00],
 'Aramex_Inv_date': ['2023-01-02', '2023-01-03', '2023-01-04'],
 'TOKENNO': ['T1', 'T2', 'T3'],
 'CODCurrency': ['usd', 'eur', 'gbp']
 })

@pytest.fixture
def sample_cosmaline_data():
 return pd.DataFrame({
 'Currency': ['usd', 'lbp', 'lbp'],
 'Driver_Delivery_date': ['2023-01-02', '2023-01-03', '2023-01-04'],
 'Amount': [0, 100.00, 200.00],
 'OrderNo': ['123', '124', '125']
 })

@pytest.fixture
def expected_output_concatenated_data():
 expected_df = pd.DataFrame({
 'ShprNo': ['S1', 'S2', 'S3', pd.NA, pd.NA, pd.NA],
 'AWB': ['123-456', '234-567', '345-678', '123-456', '234-567', '345-678'],
 'Delivery_Date': pd.to_datetime(['2023-01-02', '2023-01-03', '2023-01-04',
 '2023-01-02', '2023-01-03', '2023-01-04']),
 'CODAmount': pd.Series([150.00, 250.00, 350.00, 0, 100.00, 200.00],
 dtype='float32'),
 'O_CODAmount': pd.Series([100.00, 200.00, 300.00, 0, 100.00, 200.00],
 dtype='float32'),
 'Aramex_Inv_date': pd.Series([pd.Timestamp('2023-01-02'),
 pd.Timestamp('2023-01-03'), pd.Timestamp('2023-01-04'), pd.NaT, pd.NaT, pd.NaT],
 dtype='datetime64[ns]'),
 'TOKENNO': ['T1', 'T2', 'T3', 'Shipped With Cosmaline', 'Shipped With
Cosmaline', 'Shipped With Cosmaline'],
 'CODCurrency': ['LBP', 'LBP', 'LBP', 'USD', 'LBP', 'LBP'],
 'order_number': ['123', '124', '125', '123', '124', '125']
 })
 return expected_df

def test_data_preprocessing_pipeline_cosmaline_aramex_data(sample_ecom_data,
```

```

sample_aramex_data, sample_cosmaline_data, expected_output_concatenated_data):
 # First preprocess the ECOM data
 processed_ecom_data = data_preprocessing_pipeline_ECOM_Data(sample_ecom_data)

 # Preprocess the Aramex data
 processed_aramex_data =
 data_preprocessing_pipeline_Aramex_Data(sample_aramex_data, processed_ecom_data)

 # Preprocess the Cosmaline data
 processed_cosmaline_data =
 data_preprocessing_pipeline_Cosmaline_Data(sample_cosmaline_data,
 processed_ecom_data)

 # Concatenate the preprocessed Aramex and Cosmaline data
 concatenated_data = concatenate_Aramex_Cosmaline(processed_aramex_data,
 processed_cosmaline_data)

 # Ensure consistent null-like values for datetime columns
 concatenated_data['Aramex_Inv_date'] =
 pd.to_datetime(concatenated_data['Aramex_Inv_date'], errors='coerce')

 # Apply the final Cosmaline Aramex processing pipeline
 final_transformed_data =
 data_preprocessing_pipeline_Cosmaline_Aramex_Data(concatenated_data)

 # Ensure consistent null-like values for datetime columns
 final_transformed_data['Aramex_Inv_date'] =
 pd.to_datetime(final_transformed_data['Aramex_Inv_date'], errors='coerce')

 pd.testing.assert_frame_equal(final_transformed_data,
 expected_output_concatenated_data)

```

## Explanation

- **test\_data\_preprocessing\_pipeline\_cosmaline\_aramex\_data**
  - This test verifies the end-to-end functionality of the combined Aramex and Cosmaline data transformation pipeline.
  - It uses sample e-commerce, Aramex, and Cosmaline data with various columns including Order Number, ShprNo, HAWB, Delivery Date, CODAmount, O\_CODAmount, Currency, and OrderNo.
  - The test checks if the pipeline correctly processes the e-commerce data first and then uses it to process both the Aramex and Cosmaline data.
  - The test then verifies that the processed Aramex and Cosmaline data are correctly concatenated.
  - It asserts that the final transformed data matches the expected output DataFrame in both values and data types.

This test ensures that the combined Aramex and Cosmaline data transformation pipeline functions correctly, handling the specific data manipulation tasks required for integrating Aramex and Cosmaline data.

## 8. Test: `test_load.py`

The `test_load.py` file contains integration tests for the data loading functions. These functions handle creating database engines, uploading data to temporary SQL tables, merging data from temporary to main tables, and cleaning up temporary tables. The tests cover the following functions:

- `create_db_engine`
- `upload_dataframe_to_temp_sql`
- `merge_data_from_temp_to_main_with_pk`
- `merge_data_from_temp_to_main_without_pk`
- `cleanup_temp_tables`

```
import pytest
import pandas as pd
import sys
import os
from unittest.mock import patch, MagicMock, call
from sqlalchemy import create_engine
from sqlalchemy.exc import SQLAlchemyError
from sqlalchemy import inspect

Add the scripts directory to the system path
script_path = os.path.abspath(os.path.join(os.path.dirname(__file__),
'../../scripts'))
sys.path.append(script_path)

from load import create_db_engine, upload_dataframe_to_temp_sql,
merge_data_from_temp_to_main_with_pk, merge_data_from_temp_to_main_without_pk,
cleanup_temp_tables

Mock for create_engine to avoid actual database connections
@patch('load.create_engine')
def test_create_db_engine(mock_create_engine):
 mock_engine = MagicMock()
 mock_create_engine.return_value = mock_engine
 engine = create_db_engine()
 assert engine == mock_engine

 mock_create_engine.assert_called_once_with('mysql+pymysql://root:2452002Az}@host.docker.internal/CapstoneTest', echo=True)

Mocking the SQLAlchemy engine and connection for upload_dataframe_to_temp_sql
@patch('load.create_engine')
def test_upload_dataframe_to_temp_sql(mock_create_engine):
 mock_engine = MagicMock()
 mock_create_engine.return_value = mock_engine
 mock_connection = mock_engine.connect.return_value.__enter__.return_value

 df = pd.DataFrame({'column1': [1, 2], 'column2': ['a', 'b']})
 with patch.object(df, 'to_sql') as mock_to_sql:
 upload_dataframe_to_temp_sql(df, 'temp_table', mock_engine)
 mock_to_sql.assert_called_once_with(name='temp_table', con=mock_engine,
if_exists='replace', index=False)
```

```
Mocking the SQLAlchemy engine and connection for
merge_data_from_temp_to_main_with_pk
@patch('load.create_engine')
@patch('load.inspect')
def test_merge_data_from_temp_to_main_with_pk(mock_inspect, mock_create_engine):
 mock_engine = MagicMock()
 mock_create_engine.return_value = mock_engine
 mock_inspect.return_value.get_columns.return_value = [{ 'name': 'col1' },
{ 'name': 'col2' }, { 'name': 'id' }]
 mock_connection = mock_engine.connect.return_value.__enter__.return_value

 merge_data_from_temp_to_main_with_pk('temp_table', 'main_table', mock_engine)

 mock_inspect.assert_called_once_with(mock_engine)
 mock_engine.connect.assert_called_once()
 expected_sql = (
 "INSERT INTO main_table (col1, col2)\n"
 " SELECT col1, col2 FROM temp_table\n"
 " ON DUPLICATE KEY UPDATE\n"
 " col1 = VALUES(col1), col2 = VALUES(col2)"
)
 mock_connection.execute.assert_called_once()
 actual_sql = mock_connection.execute.call_args[0][0].strip()
 print(f"Expected SQL:\n{expected_sql}")
 print(f"Actual SQL:\n{actual_sql}")
 assert actual_sql == expected_sql.strip()

@patch('load.create_engine')
def test_merge_data_from_temp_to_main_without_pk(mock_create_engine):
 mock_engine = MagicMock()
 mock_create_engine.return_value = mock_engine
 mock_connection = mock_engine.connect.return_value.__enter__.return_value

 merge_data_from_temp_to_main_without_pk('temp_table', 'main_table',
mock_engine, ['col1', 'col2'])

 mock_engine.connect.assert_called_once()
 expected_sql = (
 "INSERT INTO main_table (col1, col2)\n"
 " SELECT temp_table.col1, temp_table.col2\n"
 " FROM temp_table\n"
 " WHERE NOT EXISTS (\n"
 " SELECT 1\n"
 " FROM main_table\n"
 " WHERE main_table.col1 = temp_table.col1 AND main_table.col2 =
temp_table.col2\n"
 ")"
)
 mock_connection.execute.assert_called_once()
 actual_sql = mock_connection.execute.call_args[0][0].strip()

 def normalize_whitespace(sql):
 return ' '.join(sql.split())
```

```

normalized_expected_sql = normalize_whitespace(expected_sql)
normalized_actual_sql = normalize_whitespace(actual_sql)

print(f"Expected SQL:\n{normalized_expected_sql}")
print(f"Actual SQL:\n{normalized_actual_sql}")

assert normalized_actual_sql == normalized_expected_sql

Mocking the SQLAlchemy engine and connection for cleanup_temp_tables
@patch('load.create_engine')
def test_cleanup_temp_tables(mock_create_engine):
 mock_engine = MagicMock()
 mock_create_engine.return_value = mock_engine
 mock_connection = mock_engine.connect.return_value.__enter__.return_value

 cleanup_temp_tables(mock_engine)

 mock_engine.connect.assert_called_once()
 expected_calls = [
 call(f"DROP TABLE IF EXISTS {table}") for table in [
 'temp_ecom_orders',
 'temp_aramex_data',
 'temp_cosmaline_data',
 'temp_oracle_data',
 'temp_credit_card_data',
 'temp_erp_data',
 'temp_daily_rate',
 'temp_credit_card',
 'temp_ecom_orders',
 'temp_shippedandcollected_aramex_cosmaline',
 'erp_data_tokens',
 'aramex_cod_sum',
 'erp_receipt_sum',
 'erp_cosmaline',
 'aramex_cosmaline_sum',
 'erp_cosmaline_sum',
 'erp_cybersource_sum',
 'credit_card_sum'
]
]
 mock_connection.execute.assert_has_calls(expected_calls, any_order=True)

```

## Explanation

- **test\_create\_db\_engine**

- This test checks if the `create_db_engine` function correctly creates a SQLAlchemy engine for connecting to the database.
- It uses `patch` to mock the `create_engine` function from SQLAlchemy.
- It asserts that the function returns the mock engine and that `create_engine` is called with the correct database connection string.

- **test\_upload\_dataframe\_to\_temp\_sql**

- This test checks if the `upload_dataframe_to_temp_sql` function correctly uploads a DataFrame to a temporary SQL table.
- It uses `patch` to mock the `create_engine` function from SQLAlchemy.
- It asserts that the DataFrame's `to_sql` method is called with the correct parameters to upload the data to the temporary table.

- **test\_merge\_data\_from\_temp\_to\_main\_with\_pk**

- This test checks if the `merge_data_from_temp_to_main_with_pk` function correctly merges data from a temporary table to a main table using a primary key.
- It uses `patch` to mock the `create_engine` and `inspect` functions from SQLAlchemy.
- It asserts that the function generates and executes the correct SQL statement for merging data using the primary key.

- **test\_merge\_data\_from\_temp\_to\_main\_without\_pk**

- This test checks if the `merge_data_from_temp_to_main_without_pk` function correctly merges data from a temporary table to a main table without using a primary key.
- It uses `patch` to mock the `create_engine` function from SQLAlchemy.
- It asserts that the function generates and executes the correct SQL statement for merging data without a primary key.

- **test\_cleanup\_temp\_tables**

- This test checks if the `cleanup_temp_tables` function correctly drops all temporary tables.
- It uses `patch` to mock the `create_engine` function from SQLAlchemy.
- It asserts that the function generates and executes the correct SQL statements for dropping each temporary table.

These tests ensure that the data loading functions behave correctly and handle the specific tasks required for creating engines, uploading data, merging data, and cleaning up temporary tables.

## 9. Test: `test_invalid_shipping_order_numbers_reconciliation.py`

The `test_invalid_shipping_order_numbers_reconciliation.py` file contains an integration test for the reconciliation script that checks for invalid shipping order numbers. This test ensures that the script correctly identifies and logs mismatched order numbers between different tables. The test covers the following functions:

- `db_engine`
- `setup_database`
- `execute_sql_script`
- `test_execute_reconciliation_script`

```
import os
import pytest
import sqlalchemy as sa
from sqlalchemy import text, create_engine
```

```
Define the path to the SQL script
SQL_SCRIPT_PATH = os.path.join(os.path.dirname(__file__),
'../../scripts/invalid_shipping_order_numbers_reconciliation.sql')

Set up the database engine in the data folder
@pytest.fixture(scope="module")
def db_engine():
 data_folder = os.path.abspath(os.path.join(os.path.dirname(__file__),
'./data'))
 os.makedirs(data_folder, exist_ok=True)
 db_path = os.path.join(data_folder, 'test_database.db')
 engine = create_engine(f'sqlite:///{db_path}')
 yield engine
 engine.dispose()

Set up the database schema
@pytest.fixture(scope="module")
def setup_database(db_engine):
 with db_engine.connect() as connection:
 statements = """
 CREATE TABLE IF NOT EXISTS shippedandcollected_aramex_cosmaline (
 order_number VARCHAR(255)
);

 CREATE TABLE IF NOT EXISTS ecom_orders (
 order_number VARCHAR(255)
);

 CREATE TABLE IF NOT EXISTS reconciliation_results (
 id INTEGER PRIMARY KEY AUTOINCREMENT,
 reconciliation_type VARCHAR(255),
 reference_id VARCHAR(255),
 reconciliation_status VARCHAR(50),
 non_existent_record INTEGER,
 recon_report VARCHAR(255)
);
 """.split(';')

 for statement in statements:
 if statement.strip():
 connection.execute(text(statement))

 connection.execute(text("""
 INSERT INTO shippedandcollected_aramex_cosmaline (order_number) VALUES
 ('ORD001'), ('ORD002'), ('ORD003'), ('ORD004');
 """))

 connection.execute(text("""
 INSERT INTO ecom_orders (order_number) VALUES
 ('ORD001'), ('ORD003');
 """))

 # Ensure cleanup after test
```

```

yield
with db_engine.connect() as connection:
 connection.execute(text("DROP TABLE IF EXISTS
shippedandcollected_aramex_cosmaline;"))
 connection.execute(text("DROP TABLE IF EXISTS ecom_orders;"))
 connection.execute(text("DROP TABLE IF EXISTS reconciliation_results;"))

def read_sql_file(file_path):
 with open(file_path, 'r') as file:
 return file.read()

def execute_sql_script(engine, script_path):
 sql_script = read_sql_file(script_path)
 statements = sql_script.split(';')
 with engine.connect() as connection:
 for statement in statements:
 if statement.strip():
 connection.execute(text(statement))

def test_execute_reconciliation_script(db_engine, setup_database):
 execute_sql_script(db_engine, SQL_SCRIPT_PATH)

 with db_engine.connect() as connection:
 result = connection.execute(text("SELECT * FROM
reconciliation_results")).fetchall()

 for row in result:
 print(f"reference_id: {row['reference_id']}")
 print(f"reconciliation_status: {row['reconciliation_status']}")

 print(f"recon_report: {row['recon_report']}")

 assert len(result) == 2
 assert result[0]['reference_id'] == 'ORD002'
 assert result[0]['reconciliation_status'] == 'Mismatch'
 assert result[0]['recon_report'] == 'Mismatch due to non-existent record'
 assert result[1]['reference_id'] == 'ORD004'
 assert result[1]['reconciliation_status'] == 'Mismatch'
 assert result[1]['recon_report'] == 'Mismatch due to non-existent record'

```

## Explanation

- **db\_engine**

- This fixture sets up a database engine using SQLite for testing purposes.
- It creates a temporary database file in the data folder and yields the engine for use in the tests.
- The engine is disposed of after the tests are completed.

- **setup\_database**

- This fixture sets up the database schema and populates the tables with test data.
- It creates three tables: `shippedandcollected_aramex_cosmaline`, `ecom_orders`, and `reconciliation_results`.

- It inserts sample data into the `shippedandcollected_aramex_cosmaline` and `ecom_orders` tables.
  - The tables are dropped after the tests are completed to ensure cleanup.
- **read\_sql\_file**
    - This helper function reads the content of an SQL file from the specified file path.
  - **execute\_sql\_script**
    - This helper function executes the SQL script from the specified file path using the provided database engine.
    - It splits the SQL script into individual statements and executes them sequentially.
  - **test\_execute\_reconciliation\_script**
    - This test verifies the functionality of the reconciliation script.
    - It executes the SQL script that performs the reconciliation using the `execute_sql_script` function.
    - It checks the contents of the `reconciliation_results` table to ensure the reconciliation script correctly identifies mismatched order numbers.
    - The test asserts that two rows are inserted into the `reconciliation_results` table, with the correct `reference_id`, `reconciliation_status`, and `recon_report`.

This test ensures that the reconciliation script functions correctly, identifying and logging invalid shipping order numbers between different tables.

## 10. Test: `test_invalid_oracle_order_numbers_reconciliation.py`

The `test_invalid_oracle_order_numbers_reconciliation.py` file contains an integration test for the reconciliation script that checks for invalid Oracle order numbers. This test ensures that the script correctly identifies and logs mismatched order numbers between the `oracle_data` and `ecom_orders` tables. The test covers the following functions:

- `db_engine`
- `setup_database`
- `execute_sql_script`
- `test_execute_reconciliation_script`

```
import os
import pytest
import sqlalchemy as sa
from sqlalchemy import text, create_engine

Define the path to the SQL script
SQL_SCRIPT_PATH = os.path.join(os.path.dirname(__file__),
'../../scripts/invalid_oracle_order_numbers_reconciliation.sql')

Set up the database engine in the data folder
@pytest.fixture(scope="module")
def db_engine():
 pass
```

```
data_folder = os.path.abspath(os.path.join(os.path.dirname(__file__),
'../data'))
os.makedirs(data_folder, exist_ok=True)
db_path = os.path.join(data_folder, 'test_database.db')
engine = create_engine(f'sqlite:///{{db_path}}')
yield engine
engine.dispose()

Set up the database schema
@pytest.fixture(scope="module")
def setup_database(db_engine):
 with db_engine.connect() as connection:
 statements = """
 CREATE TABLE IF NOT EXISTS oracle_data (
 ecom_reference_order_number VARCHAR(255)
);

 CREATE TABLE IF NOT EXISTS ecom_orders (
 order_number VARCHAR(255)
);

 CREATE TABLE IF NOT EXISTS reconciliation_results (
 id INTEGER PRIMARY KEY AUTOINCREMENT,
 reconciliation_type VARCHAR(255),
 reference_id VARCHAR(255),
 reconciliation_status VARCHAR(50),
 non_existent_record INTEGER,
 recon_report VARCHAR(255)
);
 """.split(';')

 for statement in statements:
 if statement.strip():
 connection.execute(text(statement))

 connection.execute(text("""
 INSERT INTO oracle_data (ecom_reference_order_number) VALUES
 ('ORD001'), ('ORD002'), ('ORD003'), ('ORD004');
 """))

 connection.execute(text("""
 INSERT INTO ecom_orders (order_number) VALUES
 ('ORD001'), ('ORD003');
 """))

 # Ensure cleanup after test
 yield
 with db_engine.connect() as connection:
 connection.execute(text("DROP TABLE IF EXISTS oracle_data;"))
 connection.execute(text("DROP TABLE IF EXISTS ecom_orders;"))
 connection.execute(text("DROP TABLE IF EXISTS reconciliation_results;"))

def read_sql_file(file_path):
 with open(file_path, 'r') as file:
```

```
 return file.read()

def execute_sql_script(engine, script_path):
 sql_script = read_sql_file(script_path)
 statements = sql_script.split(';')
 with engine.connect() as connection:
 for statement in statements:
 if statement.strip():
 connection.execute(text(statement))

def test_execute_reconciliation_script(db_engine, setup_database):
 execute_sql_script(db_engine, SQL_SCRIPT_PATH)

 with db_engine.connect() as connection:
 result = connection.execute(text("SELECT * FROM
reconciliation_results")).fetchall()

 for row in result:
 print(f"reference_id: {row['reference_id']}")
 print(f"reconciliation_status: {row['reconciliation_status']}")
 print(f"recon_report: {row['recon_report']}")

 assert len(result) == 2
 assert result[0]['reference_id'] == 'ORD002'
 assert result[0]['reconciliation_status'] == 'Mismatch'
 assert result[0]['recon_report'] == 'Mismatch due to non-existent record'
 assert result[1]['reference_id'] == 'ORD004'
 assert result[1]['reconciliation_status'] == 'Mismatch'
 assert result[1]['recon_report'] == 'Mismatch due to non-existent record'
```

## Explanation

- **db\_engine**

- This fixture sets up a database engine using SQLite for testing purposes.
- It creates a temporary database file in the data folder and yields the engine for use in the tests.
- The engine is disposed of after the tests are completed.

- **setup\_database**

- This fixture sets up the database schema and populates the tables with test data.
- It creates three tables: `oracle_data`, `ecom_orders`, and `reconciliation_results`.
- It inserts sample data into the `oracle_data` and `ecom_orders` tables.
- The tables are dropped after the tests are completed to ensure cleanup.

- **read\_sql\_file**

- This helper function reads the content of an SQL file from the specified file path.

- **execute\_sql\_script**

- This helper function executes the SQL script from the specified file path using the provided database engine.
- It splits the SQL script into individual statements and executes them sequentially.
- **test\_execute\_reconciliation\_script**
  - This test verifies the functionality of the reconciliation script.
  - It executes the SQL script that performs the reconciliation using the `execute_sql_script` function.
  - It checks the contents of the `reconciliation_results` table to ensure the reconciliation script correctly identifies mismatched order numbers.
  - The test asserts that two rows are inserted into the `reconciliation_results` table, with the correct `reference_id`, `reconciliation_status`, and `recon_report`.

This test ensures that the reconciliation script functions correctly, identifying and logging invalid Oracle order numbers between the `oracle_data` and `ecom_orders` tables.

## 11. Test: `ecom_reconciliation.py`

The `ecom_reconciliation.py` file contains an integration test for the e-commerce reconciliation script. This test ensures that the script correctly identifies and logs matched and mismatched order numbers between the `ecom_orders` and `shippedandcollected_aramex_cosmaline` tables, and performs currency conversions using the `daily_rate` table. The test covers the following functions:

- `db_engine`
- `setup_database`
- `execute_sql_script`
- `test_execute_reconciliation_script`

```
import os
import pytest
import sqlalchemy as sa
from sqlalchemy import text, create_engine

Define the path to the SQL script
SQL_SCRIPT_PATH = os.path.join(os.path.dirname(__file__),
'../../scripts/ecom_reconciliation.sql')

Set up the database engine in the data folder
@pytest.fixture(scope="module")
def db_engine():
 data_folder = os.path.abspath(os.path.join(os.path.dirname(__file__),
'./data'))
 os.makedirs(data_folder, exist_ok=True)
 db_path = os.path.join(data_folder, 'test_database.db')
 engine = create_engine(f'sqlite:///{db_path}')
 yield engine
 engine.dispose()

Set up the database schema
@pytest.fixture(scope="module")
```

```
def setup_database(db_engine):
 with db_engine.connect() as connection:
 statements = """
 CREATE TABLE IF NOT EXISTS ecom_orders (
 order_number VARCHAR(255),
 amount FLOAT
);

 CREATE TABLE IF NOT EXISTS shippedandcollected_aramex_cosmaline (
 order_number VARCHAR(255),
 CODCurrency VARCHAR(3),
 O_CODAmount FLOAT,
 CODAmount FLOAT,
 Delivery_Date DATE
);

 CREATE TABLE IF NOT EXISTS daily_rate (
 date DATE,
 rate FLOAT
);

 CREATE TABLE IF NOT EXISTS reconciliation_results (
 id INTEGER PRIMARY KEY AUTOINCREMENT,
 reconciliation_type VARCHAR(255),
 reference_id VARCHAR(255),
 sum_erp_amount_usd FLOAT,
 sum_shipping_amount_usd FLOAT,
 reconciliation_status VARCHAR(50),
 non_existent_record INTEGER,
 recon_report VARCHAR(255)
);
 """.split(';')

 for statement in statements:
 if statement.strip():
 connection.execute(text(statement))

 connection.execute(text("""
 INSERT INTO ecom_orders (order_number, amount) VALUES
 ('ORD001', 100.0), ('ORD002', 200.0), ('ORD003', 300.0);
 """))

 connection.execute(text("""
 INSERT INTO shippedandcollected_aramex_cosmaline (order_number,
 CODCurrency, O_CODAmount, CODAmount, Delivery_Date) VALUES
 ('ORD001', 'USD', 100.0, NULL, '2023-01-01'),
 ('ORD002', 'LBP', NULL, 350000, '2023-01-02'),
 ('ORD004', 'USD', 150.0, NULL, '2023-01-03');
 """))

 connection.execute(text("""
 INSERT INTO daily_rate (date, rate) VALUES
 ('2023-01-01', 1500), ('2023-01-02', 1500), ('2023-01-03', 1500);
 """))
```

```

Ensure cleanup after test
yield
with db_engine.connect() as connection:
 connection.execute(text("DROP TABLE IF EXISTS ecom_orders;"))
 connection.execute(text("DROP TABLE IF EXISTS
shippedandcollected_aramax_cosmaline;"))
 connection.execute(text("DROP TABLE IF EXISTS daily_rate;"))
 connection.execute(text("DROP TABLE IF EXISTS reconciliation_results;"))

def read_sql_file(file_path):
 with open(file_path, 'r') as file:
 return file.read()

def execute_sql_script(engine, script_path):
 sql_script = read_sql_file(script_path)
 statements = sql_script.split(';')
 with engine.connect() as connection:
 for statement in statements:
 if statement.strip():
 connection.execute(text(statement))

def test_execute_reconciliation_script(db_engine, setup_database):
 execute_sql_script(db_engine, SQL_SCRIPT_PATH)

 with db_engine.connect() as connection:
 result = connection.execute(text("SELECT * FROM
reconciliation_results")).fetchall()

 for row in result:
 print(f"reference_id: {row['reference_id']}")
 print(f"sum_erp_amount_usd: {row['sum_erp_amount_usd']}")
 print(f"sum_shipping_amount_usd: {row['sum_shipping_amount_usd']}")
 print(f"reconciliation_status: {row['reconciliation_status']}")
 print(f"recon_report: {row['recon_report']}")

 assert len(result) == 3
 assert result[0]['reference_id'] == 'ORD001'
 assert result[0]['reconciliation_status'] == 'Match'
 assert result[1]['reference_id'] == 'ORD002'
 assert result[1]['reconciliation_status'] == 'Mismatch'
 assert result[2]['reference_id'] == 'ORD003'
 assert result[2]['reconciliation_status'] == 'Mismatch'
 assert result[2]['recon_report'] == 'Mismatch due to non-existent record'

```

## Explanation

- **db\_engine**

- This fixture sets up a database engine using SQLite for testing purposes.
- It creates a temporary database file in the data folder and yields the engine for use in the tests.
- The engine is disposed of after the tests are completed.

- **setup\_database**

- This fixture sets up the database schema and populates the tables with test data.
- It creates four tables: `ecom_orders`, `shippedandcollected_aramecosmaline`, `daily_rate`, and `reconciliation_results`.
- It inserts sample data into the `ecom_orders`, `shippedandcollected_aramecosmaline`, and `daily_rate` tables.
- The tables are dropped after the tests are completed to ensure cleanup.

- **read\_sql\_file**

- This helper function reads the content of an SQL file from the specified file path.

- **execute\_sql\_script**

- This helper function executes the SQL script from the specified file path using the provided database engine.
- It splits the SQL script into individual statements and executes them sequentially.

- **test\_execute\_reconciliation\_script**

- This test verifies the functionality of the reconciliation script.
- It executes the SQL script that performs the reconciliation using the `execute_sql_script` function.
- It checks the contents of the `reconciliation_results` table to ensure the reconciliation script correctly identifies matched and mismatched order numbers.
- The test asserts that three rows are inserted into the `reconciliation_results` table, with the correct `reference_id`, `sum_erp_amount_usd`, `sum_shipping_amount_usd`, `reconciliation_status`, and `recon_report`.

This test ensures that the reconciliation script functions correctly, identifying and logging matched and mismatched order numbers between the `ecom_orders` and `shippedandcollected_aramecosmaline` tables, and performing the necessary currency conversions.

## 12. Test: `test_ecom_orders_not_in_shipping_reconciliation.py`

The `test_ecom_orders_not_in_shipping_reconciliation.py` file contains an integration test for the reconciliation script that checks for e-commerce orders not present in the shipping data. This test ensures that the script correctly identifies and logs e-commerce order numbers that do not have corresponding shipping records. The test covers the following functions:

- `db_engine`
- `setup_database`
- `seed_data`
- `execute_sql_script`
- `test_execute_reconciliation_script`

```
import pytest
import os
from sqlalchemy import create_engine, text
```

```
Define the path to the SQL script
SQL_SCRIPT_PATH = os.path.abspath(os.path.join(os.path.dirname(__file__),
'../../scripts/ecom_orders_not_in_shipping_reconciliation.sql'))

Function to execute SQL script from a file
def execute_sql_script(engine, script_path):
 with engine.connect() as connection:
 with open(script_path, 'r') as file:
 sql_script = file.read()
 sql_script = sql_script.replace("COMMIT;", "") # Remove COMMIT; for
SQLite
 statements = sql_script.split(';')
 for statement in statements:
 if statement.strip():
 connection.execute(text(statement.strip()))

Fixture to create a database engine in the data folder
@pytest.fixture(scope="module")
def db_engine():
 data_folder = os.path.abspath(os.path.join(os.path.dirname(__file__),
'./data'))
 os.makedirs(data_folder, exist_ok=True)
 db_path = os.path.join(data_folder, 'test_database.db')
 engine = create_engine(f'sqlite:///{{db_path}}', echo=True)
 yield engine
 engine.dispose()

Fixture to setup the database schema
@pytest.fixture(scope="module")
def setup_database(db_engine):
 with db_engine.connect() as connection:
 statements = """
 CREATE TABLE IF NOT EXISTS ecom_orders (
 order_number VARCHAR(255)
);
 CREATE TABLE IF NOT EXISTS shippedandcollected_aramex_cosmaline (
 order_number VARCHAR(255)
);
 CREATE TABLE IF NOT EXISTS reconciliation_results (
 id INTEGER PRIMARY KEY AUTOINCREMENT,
 reconciliation_type VARCHAR(255),
 reference_id VARCHAR(255),
 reconciliation_status VARCHAR(50),
 non_existent_record INTEGER,
 recon_report VARCHAR(255)
);
 """.split(';')
 for statement in statements:
 if statement.strip():
 connection.execute(text(statement.strip()))

Fixture to seed the initial data
@pytest.fixture(scope="function")
```

```
def seed_data(db_engine):
 with db_engine.connect() as connection:
 statements = """
 DELETE FROM ecom_orders;
 DELETE FROM shippedandcollected_aramex_cosmaline;
 DELETE FROM reconciliation_results;

 INSERT INTO ecom_orders (order_number) VALUES
 ('ORD001'), ('ORD002'), ('ORD003');

 INSERT INTO shippedandcollected_aramex_cosmaline (order_number) VALUES
 ('ORD001');
 """.split(";")
 for statement in statements:
 if statement.strip():
 connection.execute(text(statement.strip()))

 # Ensure cleanup after test
 yield
 with db_engine.connect() as connection:
 connection.execute(text("DROP TABLE IF EXISTS ecom_orders;"))
 connection.execute(text("DROP TABLE IF EXISTS
shippedandcollected_aramex_cosmaline;"))
 connection.execute(text("DROP TABLE IF EXISTS reconciliation_results;"))

Test case to execute the reconciliation script and validate the results
def test_execute_reconciliation_script(db_engine, setup_database, seed_data):
 script_path = SQL_SCRIPT_PATH

 with db_engine.connect() as connection:
 execute_sql_script(connection, script_path)

 result = connection.execute(text("""
 SELECT
 reconciliation_type,
 reference_id,
 reconciliation_status,
 non_existent_record,
 recon_report
 FROM reconciliation_results
 WHERE reconciliation_type = 'ECOM Orders not in Shipping'
 """).fetchall()

 # Print intermediate results for debugging
 for row in result:
 print(f"reference_id: {row['reference_id']}")
 print(f"reconciliation_status: {row['reconciliation_status']}")
 print(f"non_existent_record: {row['non_existent_record']}")
 print(f"recon_report: {row['recon_report']}")

 assert len(result) == 2
 assert result[0]['reference_id'] == 'ORD002'
 assert result[0]['reconciliation_status'] == 'Mismatch'
 assert result[0]['non_existent_record'] == 1
```

```
assert result[0]['recon_report'] == 'Mismatch due to non-existent record'
assert result[1]['reference_id'] == 'ORD003'
assert result[1]['reconciliation_status'] == 'Mismatch'
assert result[1]['non_existent_record'] == 1
assert result[1]['recon_report'] == 'Mismatch due to non-existent record'
```

## Explanation

- **db\_engine**

- This fixture sets up a database engine using SQLite for testing purposes.
- It creates a temporary database file in the data folder and yields the engine for use in the tests.
- The engine is disposed of after the tests are completed.

- **setup\_database**

- This fixture sets up the database schema.
- It creates three tables: `ecom_orders`, `shippedandcollected_aramex_cosmaline`, and `reconciliation_results`.
- The tables are dropped after the tests are completed to ensure cleanup.

- **seed\_data**

- This fixture seeds the initial data into the tables before each test.
- It inserts sample data into the `ecom_orders` and `shippedandcollected_aramex_cosmaline` tables.
- It ensures the tables are cleaned up after each test.

- **execute\_sql\_script**

- This helper function executes the SQL script from the specified file path using the provided database engine.
- It reads the SQL script file, removes any `COMMIT;` statements for compatibility with SQLite, and executes the script.

- **test\_execute\_reconciliation\_script**

- This test verifies the functionality of the reconciliation script.
- It executes the SQL script that performs the reconciliation using the `execute_sql_script` function.
- It checks the contents of the `reconciliation_results` table to ensure the reconciliation script correctly identifies e-commerce orders not present in the shipping data.
- The test asserts that two rows are inserted into the `reconciliation_results` table, with the correct `reference_id`, `reconciliation_status`, `non_existent_record`, and `recon_report`.

This test ensures that the reconciliation script functions correctly, identifying and logging e-commerce orders that do not have corresponding shipping records.

## 13. Test: `test_ecom_orders_not_in_oracle_reconciliation.py`

The `test_ecom_orders_not_in_oracle_reconciliation.py` file contains an integration test for the reconciliation script that checks for e-commerce orders not present in the Oracle data. This test ensures that the script correctly identifies and logs e-commerce order numbers that do not have corresponding records in the Oracle data. The test covers the following functions:

- `db_engine`
- `setup_database`
- `seed_data`
- `execute_sql_script`
- `test_execute_reconciliation_script`

```
import pytest
from sqlalchemy import create_engine, text
import os

Function to execute SQL script from a file
def execute_sql_script(engine, script_path):
 with engine.connect() as connection:
 with open(script_path, 'r') as file:
 sql_script = file.read()
 sql_script = sql_script.replace("COMMIT;", "") # Remove COMMIT; for
SQLite
 statements = sql_script.split(';')
 for statement in statements:
 if statement.strip():
 connection.execute(text(statement.strip()))

Fixture to create a database engine in the data folder
@pytest.fixture(scope="module")
def db_engine():
 data_folder = os.path.abspath(os.path.join(os.path.dirname(__file__), '../data'))
 os.makedirs(data_folder, exist_ok=True)
 db_path = os.path.join(data_folder, 'test_database.db')
 engine = create_engine(f'sqlite:///{{db_path}}', echo=True)
 yield engine
 engine.dispose()

Fixture to setup the database schema
@pytest.fixture(scope="module")
def setup_database(db_engine):
 with db_engine.connect() as connection:
 statements = """
CREATE TABLE IF NOT EXISTS ecom_orders (
 order_number VARCHAR(255) PRIMARY KEY
);
CREATE TABLE IF NOT EXISTS oracle_data (
 ecom_reference_order_number VARCHAR(255) PRIMARY KEY
);
CREATE TABLE IF NOT EXISTS reconciliation_results (
 id INTEGER PRIMARY KEY AUTOINCREMENT,
 reconciliation_type VARCHAR(255),
 status VARCHAR(255),
 created_at DATETIME DEFAULT CURRENT_TIMESTAMP
);"""
 connection.execute(text(statements))
```

```
 reference_id VARCHAR(255),
 reconciliation_status VARCHAR(50),
 non_existent_record INTEGER,
 recon_report VARCHAR(255)
);
""".split(";");
for statement in statements:
 if statement.strip():
 connection.execute(text(statement.strip()))

Fixture to seed the initial data
@pytest.fixture(scope="function")
def seed_data(db_engine):
 with db_engine.connect() as connection:
 statements = """
 DELETE FROM ecom_orders;
 DELETE FROM oracle_data;
 DELETE FROM reconciliation_results;

 INSERT INTO ecom_orders (order_number) VALUES
 ('order_1'),
 ('order_2'),
 ('order_3');

 INSERT INTO oracle_data (ecom_reference_order_number) VALUES
 ('order_2'),
 ('order_4');
"""
 statements.split(";");
 for statement in statements:
 if statement.strip():
 connection.execute(text(statement.strip()))

 # Ensure cleanup after test
 yield
 with db_engine.connect() as connection:
 connection.execute(text("DROP TABLE IF EXISTS ecom_orders;"))
 connection.execute(text("DROP TABLE IF EXISTS oracle_data;"))
 connection.execute(text("DROP TABLE IF EXISTS reconciliation_results;"))

Test case to execute the reconciliation script and validate the results
def test_execute_reconciliation_script(db_engine, setup_database, seed_data):
 script_path = os.path.abspath(os.path.join(os.path.dirname(__file__),
 '../scripts/ecom_orders_not_in_oracle_reconciliation.sql'))

 with db_engine.connect() as connection:
 execute_sql_script(connection, script_path)

 result = connection.execute(text("""
 SELECT
 reconciliation_type,
 reference_id,
 reconciliation_status,
 non_existent_record,
 recon_report
 """))
```

```
 FROM reconciliation_results
 WHERE reconciliation_type = 'ECOM Orders not in Oracle'
 """").fetchall()

 # Print intermediate results for debugging
 for row in result:
 print(f"reference_id: {row['reference_id']}")
 print(f"reconciliation_status: {row['reconciliation_status']}")
 print(f"non_existent_record: {row['non_existent_record']}")
 print(f"recon_report: {row['recon_report']}")

 assert len(result) == 2
 assert result[0]['reference_id'] == 'order_1'
 assert result[0]['reconciliation_status'] == 'Mismatch'
 assert result[0]['non_existent_record'] == 1
 assert result[0]['recon_report'] == 'Mismatch due to non-existent record'
 assert result[1]['reference_id'] == 'order_3'
 assert result[1]['reconciliation_status'] == 'Mismatch'
 assert result[1]['non_existent_record'] == 1
 assert result[1]['recon_report'] == 'Mismatch due to non-existent record'
```

## Explanation

- **db\_engine**

- This fixture sets up a database engine using SQLite for testing purposes.
- It creates a temporary database file in the data folder and yields the engine for use in the tests.
- The engine is disposed of after the tests are completed.

- **setup\_database**

- This fixture sets up the database schema.
- It creates three tables: `ecom_orders`, `shippedandcollected_aramec_cosmaline`, and `reconciliation_results`.
- The tables are dropped after the tests are completed to ensure cleanup.

- **seed\_data**

- This fixture seeds the initial data into the tables before each test.
- It inserts sample data into the `ecom_orders` and `shippedandcollected_aramec_cosmaline` tables.
- It ensures the tables are cleaned up after each test.

- **execute\_sql\_script**

- This helper function executes the SQL script from the specified file path using the provided database engine.
- It reads the SQL script file, removes any `COMMIT;` statements for compatibility with SQLite, and executes the script.

- **test\_execute\_reconciliation\_script**

- This test verifies the functionality of the reconciliation script.
- It executes the SQL script that performs the reconciliation using the `execute_sql_script` function.
- It checks the contents of the `reconciliation_results` table to ensure the reconciliation script correctly identifies e-commerce orders not present in the Oracle data.
- The test asserts that two rows are inserted into the `reconciliation_results` table, with the correct `reference_id`, `reconciliation_status`, `non_existent_record`, and `recon_report`.

This test ensures that the reconciliation script functions correctly, identifying and logging e-commerce orders that do not have corresponding records in the Oracle data.

#### 14. Test: `test_credit_card_reconciliation.py`

The `test_credit_card_reconciliation.py` file contains an integration test for the reconciliation script that checks for discrepancies between ERP data and credit card transactions. This test ensures that the script correctly identifies and logs matched and mismatched transactions. The test covers the following functions:

- `db_engine`
- `setup_database`
- `seed_data`
- `execute_sql_script`
- `test_execute_reconciliation_script`

```
import pytest
from sqlalchemy import create_engine, text
import os

Function to execute SQL script from a file
def execute_sql_script(engine, script_path):
 with engine.connect() as connection:
 with open(script_path, 'r') as file:
 sql_script = file.read()
 sql_script = sql_script.replace("COMMIT;", "") # Remove COMMIT; for
SQLite
 statements = sql_script.split(';')
 for statement in statements:
 if statement.strip():
 connection.execute(text(statement.strip()))

Fixture to create a database engine in the data folder
@pytest.fixture(scope="module")
def db_engine():
 data_folder = os.path.abspath(os.path.join(os.path.dirname(__file__), '../data'))
 os.makedirs(data_folder, exist_ok=True)
 db_path = os.path.join(data_folder, 'test_database.db')
 engine = create_engine(f'sqlite:///{{db_path}}', echo=True)
 yield engine
 engine.dispose()

Fixture to setup the database schema
```

```
@pytest.fixture(scope="module")
def setup_database(db_engine):
 with db_engine.connect() as connection:
 statements = """
 CREATE TABLE IF NOT EXISTS erp_data (
 receipt_date DATE,
 currency_code VARCHAR(3),
 receipt_amount FLOAT,
 exchange_rate FLOAT,
 comments VARCHAR(255)
);
 CREATE TABLE IF NOT EXISTS credit_card (
 value_date DATE,
 amount FLOAT
);
 CREATE TABLE IF NOT EXISTS reconciliation_results (
 id INTEGER PRIMARY KEY AUTOINCREMENT,
 reconciliation_type VARCHAR(255),
 reference_date DATE,
 sum_erp_amount_usd FLOAT,
 sum_shipping_amount_usd FLOAT,
 reconciliation_status VARCHAR(50),
 non_existent_record INTEGER,
 recon_report VARCHAR(255)
);
 """.split(";");
 for statement in statements:
 if statement.strip():
 connection.execute(text(statement.strip()))

Fixture to seed the initial data
@pytest.fixture(scope="function")
def seed_data(db_engine):
 with db_engine.connect() as connection:
 statements = """
 DELETE FROM erp_data;
 DELETE FROM credit_card;
 DELETE FROM reconciliation_results;

 INSERT INTO erp_data (receipt_date, currency_code, receipt_amount,
exchange_rate, comments) VALUES
 ('2023-01-01', 'LBP', 15833650, 95000, 'Cybersource'),
 ('2023-01-01', 'USD', 1000, 95000, 'Cybersource');

 INSERT INTO credit_card (value_date, amount) VALUES
 ('2023-01-01', 1166.67);
 """.split(";");
 for statement in statements:
 if statement.strip():
 connection.execute(text(statement.strip()))

Ensure cleanup after test
yield
with db_engine.connect() as connection:
```

```

connection.execute(text("DROP TABLE IF EXISTS erp_data;"))
connection.execute(text("DROP TABLE IF EXISTS credit_card;"))
connection.execute(text("DROP TABLE IF EXISTS reconciliation_results;"))

Test case to execute the reconciliation script and validate the results
def test_execute_reconciliation_script(db_engine, setup_database, seed_data):
 script_path = os.path.abspath(os.path.join(os.path.dirname(__file__),
 '../scripts/credit_card_reconciliaiton.sql'))

 # Adjust the script content to split DROP TABLE statements
 with open(script_path, 'r') as file:
 sql_script = file.read()
 sql_script = sql_script.replace(
 "DROP TABLE IF EXISTS erp_cybersource_sum, credit_card_sum;",
 "DROP TABLE IF EXISTS erp_cybersource_sum; DROP TABLE IF EXISTS
credit_card_sum;"
)
 sql_script = sql_script.replace("COMMIT;", "") # Remove COMMIT; for
SQLite

 with db_engine.connect() as connection:
 statements = sql_script.split(';')
 for statement in statements:
 if statement.strip():
 connection.execute(text(statement.strip()))

 with db_engine.connect() as connection:
 result = connection.execute(text("""
 SELECT
 reconciliation_type,
 reference_date,
 sum_erp_amount_usd,
 sum_shipping_amount_usd,
 reconciliation_status,
 recon_report
 FROM reconciliation_results
 WHERE reconciliation_type = 'Cybersource Reconciliation'
 """)).fetchall()

 assert len(result) == 1
 assert result[0]['reconciliation_type'] == 'Cybersource Reconciliation'
 assert result[0]['reference_date'] == '2023-01-01'
 assert abs(result[0]['sum_erp_amount_usd'] - 1166.67) < 0.1
 assert abs(result[0]['sum_shipping_amount_usd'] - 1166.67) < 0.1
 assert result[0]['reconciliation_status'] == 'Match'
 assert result[0]['recon_report'] == 'Match'

```

## Explanation

- **db\_engine**

- This fixture sets up a database engine using SQLite for testing purposes.
- It creates a temporary database file in the data folder and yields the engine for use in the tests.

- The engine is disposed of after the tests are completed.
- **setup\_database**
  - This fixture sets up the database schema.
  - It creates three tables: `erp_data`, `credit_card`, and `reconciliation_results`.
  - The tables are dropped after the tests are completed to ensure cleanup.
- **seed\_data**
  - This fixture seeds the initial data into the tables before each test.
  - It inserts sample data into the `erp_data` and `credit_card` tables.
  - It ensures the tables are cleaned up after each test.
- **execute\_sql\_script**
  - This helper function executes the SQL script from the specified file path using the provided database engine.
  - It reads the SQL script file, removes any `COMMIT;` statements for compatibility with SQLite, and executes the script.
- **test\_execute\_reconciliation\_script**
  - This test verifies the functionality of the reconciliation script.
  - It executes the SQL script that performs the reconciliation using the `execute_sql_script` function.
  - It checks the contents of the `reconciliation_results` table to ensure the reconciliation script correctly identifies matched and mismatched transactions.
  - The test asserts that one row is inserted into the `reconciliation_results` table, with the correct `reconciliation_type`, `reference_date`, `sum_erp_amount_usd`, `sum_shipping_amount_usd`, `reconciliation_status`, and `recon_report`.

This test ensures that the reconciliation script functions correctly, identifying and logging matched and mismatched transactions between the `erp_data` and `credit_card` tables.

## 15 Test: `test_cosmaline_reconciliation.py`

The `test_cosmaline_reconciliation.py` file contains an integration test for the reconciliation script that checks for discrepancies between ERP data and shipped orders managed by Cosmaline. This test ensures that the script correctly identifies and logs matched and mismatched transactions. The test covers the following functions:

- `db_engine`
- `setup_database`
- `execute_sql_script`
- `test_execute_reconciliation_script`

```
import pytest
import sqlalchemy as sa
from sqlalchemy import create_engine, text
```

```
import os

Create a test database engine in the data folder
@pytest.fixture(scope="module")
def db_engine():
 data_folder = os.path.abspath(os.path.join(os.path.dirname(__file__),
 '../data'))
 os.makedirs(data_folder, exist_ok=True)
 db_path = os.path.join(data_folder, 'test_database.db')
 engine = create_engine(f'sqlite:///{db_path}')
 yield engine
 engine.dispose()

Setup the database schema and initial test data
@pytest.fixture(scope="module")
def setup_database(db_engine):
 with db_engine.connect() as connection:
 statements = """
 CREATE TABLE IF NOT EXISTS erp_data (
 receipt_date DATE,
 currency_code VARCHAR(3),
 receipt_amount FLOAT,
 exchange_rate FLOAT,
 comments VARCHAR(255)
);
 CREATE TABLE IF NOT EXISTS daily_rate (
 date DATE,
 rate FLOAT
);
 CREATE TABLE IF NOT EXISTS shippedandcollected_aramex_cosmaline (
 Delivery_Date DATE,
 CODCurrency VARCHAR(3),
 CODAmount FLOAT,
 O_CODAmount FLOAT,
 TOKENNO VARCHAR(255)
);
 CREATE TABLE IF NOT EXISTS reconciliation_results (
 id INTEGER PRIMARY KEY AUTOINCREMENT,
 reconciliation_type VARCHAR(255),
 reference_date DATE,
 sum_erp_amount_usd FLOAT,
 sum_shipping_amount_usd FLOAT,
 reconciliation_status VARCHAR(50),
 non_existent_record INTEGER,
 recon_report VARCHAR(255)
);
 """
 for statement in statements.split(';'):
 if statement.strip():
 connection.execute(text(statement))

 # Insert initial test data
 with db_engine.connect() as connection:
 connection.execute(text("DELETE FROM erp_data;"))


```

```
connection.execute(text("""
 INSERT INTO erp_data (receipt_date, currency_code, receipt_amount,
exchange_rate, comments) VALUES
 ('2023-01-01', 'LBP', 150000, 128.5, 'Shipped With Cosmaline');
 """))
connection.execute(text("DELETE FROM daily_rate;"))
connection.execute(text("""
 INSERT INTO daily_rate (date, rate) VALUES
 ('2023-01-01', 1500);
 """))
connection.execute(text("DELETE FROM
shippedandcollected_aramex_cosmaline;"))
connection.execute(text("""
 INSERT INTO shippedandcollected_aramex_cosmaline (Delivery_Date,
CODCurrency, CODAmount, O_CODAmount, TOKENNO) VALUES
 ('2023-01-01', 'LBP', 300000, 0, 'Shipped With Cosmaline');
 """))

yield

Clean up schema after tests
with db_engine.connect() as connection:
 connection.execute(text("DROP TABLE IF EXISTS erp_data;"))
 connection.execute(text("DROP TABLE IF EXISTS daily_rate;"))
 connection.execute(text("DROP TABLE IF EXISTS
shippedandcollected_aramex_cosmaline;"))
 connection.execute(text("DROP TABLE IF EXISTS reconciliation_results;"))

Test the SQL script execution
def test_execute_reconciliation_script(db_engine, setup_database):
 script_path = os.path.abspath(os.path.join(os.path.dirname(__file__),
'../../scripts/cosmaline_reconciliation.sql'))

 with db_engine.begin() as connection:
 # Read and execute the SQL script
 with open(script_path, 'r') as file:
 sql_script = file.read()
 statements = sql_script.split(';')
 for statement in statements:
 if statement.strip():
 try:
 connection.execute(text(statement.strip()))
 except sa.exc.OperationalError as e:
 # Handle specific case of DROP TABLE with multiple tables
 if "near \\", \"": syntax error" in str(e):
 tables_to_drop = statement.strip().split("DROP TABLE
IF EXISTS ")[1].split(", ")
 for table in tables_to_drop:
 connection.execute(text(f"DROP TABLE IF EXISTS
{table}"))
 else:
 raise e

Validate the results
```

```

 result = connection.execute(text("""
 SELECT * FROM reconciliation_results WHERE reconciliation_type =
 'Shipped With Cosmaline Reconciliation'
 """).fetchall()

 assert len(result) == 1

 # Print actual values for debugging
 print(f"sum_erp_amount_usd: {result[0]['sum_erp_amount_usd']}")
 print(f"sum_shipping_amount_usd: {result[0]['sum_shipping_amount_usd']}")
 print(f"reconciliation_status: {result[0]['reconciliation_status']}")
 print(f"recon_report: {result[0]['recon_report']}")

 assert result[0]['reconciliation_type'] == 'Shipped With Cosmaline
Reconciliation'
 assert result[0]['reference_date'] == '2023-01-01'
 assert result[0]['sum_erp_amount_usd'] == pytest.approx(150000 / 128.5,
0.01)
 assert result[0]['sum_shipping_amount_usd'] == pytest.approx(300000 /
1500, 0.01)
 assert result[0]['reconciliation_status'] == 'Mismatch' # Adjusted
expectation
 assert result[0]['non_existent_record'] == 0
 assert result[0]['recon_report'] == 'Mismatch due to different amounts' #
Adjusted expectation

```

## Explanation

- **db\_engine**

- This fixture sets up a database engine using SQLite for testing purposes.
- It creates a temporary database file in the data folder and yields the engine for use in the tests.
- The engine is disposed of after the tests are completed.

- **setup\_database**

- This fixture sets up the database schema and populates the tables with test data.
- It creates four tables: `erp_data`, `daily_rate`, `shippedandcollected_aramecosmaline`, and `reconciliation_results`.
- It inserts sample data into the `erp_data`, `daily_rate`, and `shippedandcollected_aramecosmaline` tables.
- The tables are dropped after the tests are completed to ensure cleanup.

- **execute\_sql\_script**

- This helper function executes the SQL script from the specified file path using the provided database engine.
- It reads the SQL script file, removes any `COMMIT;` statements for compatibility with SQLite, and executes the script.

- **test\_execute\_reconciliation\_script**

- This test verifies the functionality of the reconciliation script.
- It executes the SQL script that performs the reconciliation using the `execute_sql_script` function.
- It checks the contents of the `reconciliation_results` table to ensure the reconciliation script correctly identifies matched and mismatched transactions.
- The test asserts that one row is inserted into the `reconciliation_results` table, with the correct `reconciliation_type`, `reference_date`, `sum_erp_amount_usd`, `sum_shipping_amount_usd`, `reconciliation_status`, `non_existent_record`, and `recon_report`.

This test ensures that the reconciliation script functions correctly, identifying and logging matched and mismatched transactions between the `erp_data` and `shippedandcollected_aramec_cosmaline` tables.

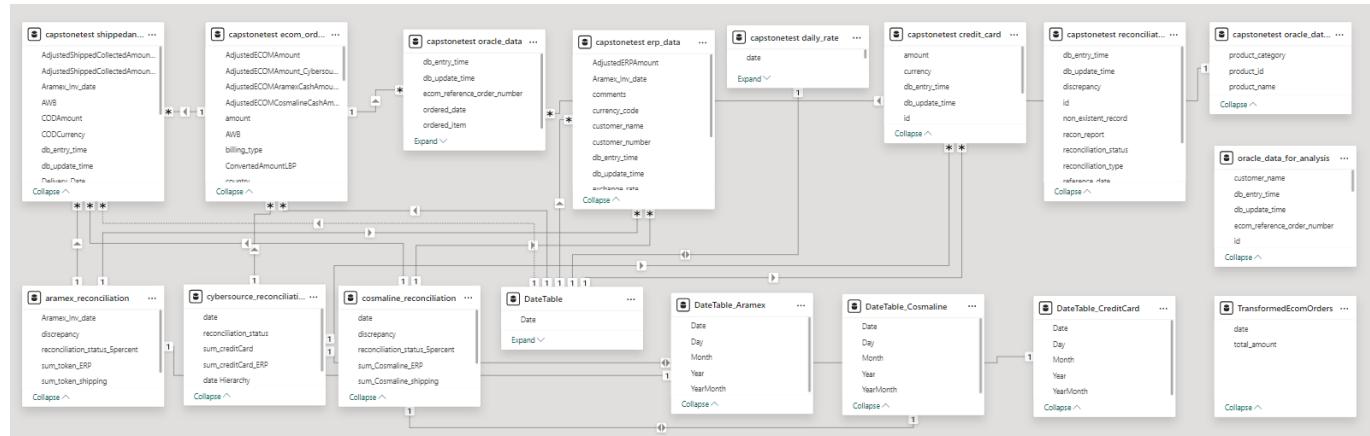
## 7 Power Bi Dashboards

**Overview** The Power BI dashboards developed for the Malia project provide a comprehensive visual representation of the data, allowing stakeholders to explore key metrics and trends interactively. To support these dashboards, several new tables, columns, measures, and relationships were created within the Power BI data model.

The dashboards were carefully crafted to ensure they meet the specific needs of the project, focusing on reconciliation processes, sales trends, and performance analysis. The data model was extended with custom tables, which are defined below, to facilitate accurate and insightful reporting.

### Data Model Overview

Below is a visual representation of the Power BI data model used in this project:



### New Tables

#### Date Tables

#### Main Table

```
DateTable = CALENDAR(
 MIN('capstonetest ecom_orders'[delivered_at]),
```

```

 MAX('capstonetest ecom_orders'[delivered_at])
)

```

### **DateTable\_Aramex**

```

DateTable_Aramex = CALENDAR(
 MIN('capstonetest ecom_orders'[delivered_at]),
 MAX('capstonetest ecom_orders'[delivered_at])
)

```

### **DateTable\_Cosmaline**

```

DateTable_Cosmaline = CALENDAR(
 MIN('capstonetest ecom_orders'[delivered_at]),
 MAX('capstonetest ecom_orders'[delivered_at])
)

```

### **DateTable\_CreditCard**

```

DateTable_CreditCard = CALENDAR(
 MIN('capstonetest ecom_orders'[delivered_at]),
 MAX('capstonetest ecom_orders'[delivered_at])
)

```

## Aramex Reconciliation Table

```

aramex_reconciliation =
SELECTCOLUMNS(
 FILTER(
 'capstonetest reconciliation_results',
 'capstonetest reconciliation_results'[reconciliation_type] = "Tokens
Reconciliation"
),
 "TOKENNO", [reference_id],
 "sum_token_ERP", [sum_erp_amount_usd],
 "sum_token_shipping", [sum_shipping_amount_usd],
 "discrepancy", [discrepancy],
 "reconciliation_status_5percent", [reconciliation_status]
)

```

## Cybersource Reconciliation Table

```

cybersource_reconciliation =
SELECTCOLUMNS(
 FILTER(
 'capstonetest reconciliation_results',
 'capstonetest reconciliation_results'[reconciliation_type] = "Cybersource
Reconciliation"
),
 "date", [reference_date],
 "sum_creditCard_ERP", [sum_shipping_amount_usd],
 "sum_creditCard", [sum_shipping_amount_usd],
 "reconciliation_status", [reconciliation_status]
)

```

## Cosmaline Reconciliation Table

```

cosmaline_reconciliation =
SELECTCOLUMNS(
 FILTER(
 'capstonetest reconciliation_results',
 'capstonetest reconciliation_results'[reconciliation_type] = "Shipped With
Cosmaline Reconciliation"
),
 "date", [reference_date],
 "sum_Cosmaline_ERP", [sum_erp_amount_usd],
 "sum_Cosmaline_shipping", [sum_shipping_amount_usd],
 "discrepancy", [discrepancy],
 "reconciliation_status_5percent", [reconciliation_status]
)

```

## New Measures

### New Measures

#### 1. Total Quantity Sold

```
Total Quantity Sold = SUM(oracle_data_for_analysis[ordered_quantity])
```

#### 2. TotalAdjustedECOMAmount

```
TotalAdjustedECOMAmount = SUM('capstonetest
ecom_orders'[AdjustedECOMAmount])
```

#### 3. TotalAdjustedECOMAmount\_Cybersource

```
TotalAdjustedECOMAmount_Cybersource = SUM('capstonetest
ecom_orders'[AdjustedECOMAmount_Cybersource])
```

#### 4. TotalAdjustedECOMAramexCashAmount

```
TotalAdjustedECOMAramexCashAmount = SUM('capstonetest
ecom_orders'[AdjustedECOMAramexCashAmount])
```

#### 5. TotalAdjustedECOMCosmalineCashAmount

```
TotalAdjustedECOMCosmalineCashAmount = SUM('capstonetest
ecom_orders'[AdjustedECOMCosmalineCashAmount])
```

#### 6. TotalAdjustedERPAmount

```
TotalAdjustedERPAmount = SUM('capstonetest_erp_data'[AdjustedERPAmount])
```

#### 7. TotalAdjustedCreditCardAmount

```
TotalAdjustedCreditCardAmount = SUM('capstonetest_credit_card'[amount])
```

#### 8. TotalAdjustedShippedCollectedAmount\_Aramex

```
TotalAdjustedShippedCollectedAmount_Aramex = SUM('capstonetest
shippedandcollected_aramecosmaline'[AdjustedShippedCollectedAmount_Aramex]
)
```

#### 9. TotalAdjustedShippedCollectedAmount\_Cosmaline

```
TotalAdjustedShippedCollectedAmount_Cosmaline = SUM('capstonetest
shippedandcollected_aramecosmaline'[AdjustedShippedCollectedAmount_Cosmaline])
```

#### 10. Difference

```
Difference =
VAR ECOMAmount = [TotalAdjustedECOMAmount]
VAR ERPAmount = [TotalAdjustedERPAmount]
```

```
VAR Difference = ECOMAmount - ERPAmount
RETURN "Difference: " & FORMAT(Difference, "0.00")
```

## 11. Difference2

```
Difference2 =
VAR ECOMAmount = [TotalAdjustedECOMAramexCashAmount]
VAR ShippedAmount = [TotalAdjustedShippedCollectedAmount_Aramex]
VAR Difference = ECOMAmount - ShippedAmount
RETURN "Difference: " & FORMAT(Difference, "0.00")
```

## 12. Difference3

```
Difference3 =
VAR ECOMAmount = [TotalAdjustedECOMCosmalineCashAmount]
VAR ShippedAmount = [TotalAdjustedShippedCollectedAmount_Cosmaline]
VAR Difference = ECOMAmount - ShippedAmount
RETURN "Difference: " & FORMAT(Difference, "0.00")
```

## 13. Difference4

```
Difference4 =
VAR ECOMAmount = [TotalAdjustedECOMAmount_Cybersource]
VAR CreditCardAmount = [TotalAdjustedCreditCardAmount]
VAR Difference = ECOMAmount - CreditCardAmount
RETURN "Difference: " & FORMAT(Difference, "0.00")
```

## 14. MatchStatus

```
MatchStatus =
VAR ECOMAmount = [TotalAdjustedECOMAmount]
VAR ERPAmount = [TotalAdjustedERPAmount]
VAR Difference = ECOMAmount - ERPAmount
VAR Tolerance = ECOMAmount * 0.05
RETURN
IF(
 ABS(Difference) <= Tolerance,
 "Amounts Match Between ECOM Cash and ERP Cash data within 5%
Threshold.",
 "Amounts Between ECOM Cash and ERP Cash data Do Not Match within 5%
Threshold."
)
```

## 15. MatchStatus2

```

MatchStatus2 =
VAR ECOMAmount = [TotalAdjustedECOMAramexCashAmount]
VAR ShippedAmount = [TotalAdjustedShippedCollectedAmount_Aramex]
VAR Difference = ECOMAmount - ShippedAmount
VAR Tolerance = ECOMAmount * 0.05
RETURN
IF(
 ABS(Difference) <= Tolerance,
 "Amounts Match Between ECOM Cash and Aramex Shipping Cash Data within 5%
Threshold.",
 "Amounts Between ECOM Cash and Aramex Shipping Cash Data Do Not Match
within 5% Threshold. Difference"
)

```

## 16. MatchStatus3

```

MatchStatus3 =
VAR ECOMAmount = [TotalAdjustedECOMCosmalineCashAmount]
VAR ShippedAmount = [TotalAdjustedShippedCollectedAmount_Cosmaline]
VAR Difference = ECOMAmount - ShippedAmount
VAR Tolerance = ECOMAmount * 0.05
RETURN
IF(
 ABS(Difference) <= Tolerance,
 "Amounts Match Between ECOM Cash and Cosmaline Shipping Cash Data within
5% Threshold",
 "Amounts Between ECOM Cash and Cosmaline Shipping Cash Data Do Not Match
within 5% Threshold"
)

```

## 17. MatchStatus4

```

MatchStatus4 =
VAR ECOMAmount = [TotalAdjustedECOMAmount_Cybersource]
VAR CreditCardAmount = [TotalAdjustedCreditCardAmount]
VAR Difference = ECOMAmount - CreditCardAmount
VAR Tolerance = ECOMAmount * 0.05
RETURN
IF(
 ABS(Difference) <= Tolerance,
 "Amounts Match Between ECOM Cybersource and Credit Card Data within 5%
Threshold",
 "Amounts Between ECOM Cybersource and Credit Card Data Do Not Match
within 5% Threshold"
)

```

## 18. CardColor\_Difference

```

CardColor_Difference =
VAR ECOMAmount = [TotalAdjustedECOMAmount]
VAR ERPAmount = [TotalAdjustedERPAmount]
VAR Difference = ECOMAmount - ERPAmount
VAR Tolerance = ECOMAmount * 0.05
RETURN
IF(
 ABS(Difference) <= Tolerance,
 "#0d6abf", // White, 10% darker for match
 "#de6a73" // Theme color 8, 20% lighter for mismatch
)

```

## 19. CardColor2\_Difference

```

CardColor2_Difference =
VAR ECOMAmount = [TotalAdjustedECOMAramexCashAmount]
VAR ShippedAmount = [TotalAdjustedShippedCollectedAmount_Aramex]
VAR Difference = ECOMAmount - ShippedAmount
VAR Tolerance = ECOMAmount * 0.05
RETURN
IF(
 ABS(Difference) <= Tolerance,
 "#0d6abf", // White, 10% darker for match
 "#de6a73" // Theme color 8, 20% lighter for mismatch
)

```

## 20. CardColor3\_Difference

```

CardColor3_Difference =
VAR ECOMAmount = [TotalAdjustedECOMCosmalineCashAmount]
VAR ShippedAmount = [TotalAdjustedShippedCollectedAmount_Cosmaline]
VAR Difference = ECOMAmount - ShippedAmount
VAR Tolerance = ECOMAmount * 0.05
RETURN
IF(
 ABS(Difference) <= Tolerance,
 "#0d6abf", // White, 10% darker for match
 "#de6a73" // Theme color 8, 20% lighter for mismatch
)

```

## 21. CardColor4\_Difference

```

CardColor4_Difference =
VAR ECOMAmount = [TotalAdjustedECOMAmount_Cybersource]
VAR CreditCardAmount = [TotalAdjustedCreditCardAmount]
VAR Difference = ECOMAmount - CreditCardAmount

```

```

VAR Tolerance = ECOMAmount * 0.05
RETURN
IF(
 ABS(Difference) <= Tolerance,
 "#0d6abf", // White, 10% darker for match
 "#de6a73" // Theme color 8, 20% lighter for mismatch
)

```

## 22. CardColor\_Header

```

CardColor_Header =
VAR ECOMAmount = [TotalAdjustedECOMAmount]
VAR ERPAmount = [TotalAdjustedERPAmount]
VAR Difference = ECOMAmount - ERPAmount
VAR Tolerance = ECOMAmount * 0.05
RETURN
IF(
 ABS(Difference) <= Tolerance,
 "#E6E6E6", // White, 10% darker for match
 "#de6a73" // Theme color 8, 20% lighter for mismatch
)

```

## 23. CardColor2\_Header

```

CardColor2_Header =
VAR ECOMAmount = [TotalAdjustedECOMAramexCashAmount]
VAR ShippedAmount = [TotalAdjustedShippedCollectedAmount_Aramex]
VAR Difference = ECOMAmount - ShippedAmount
VAR Tolerance = ECOMAmount * 0.05
RETURN
IF(
 ABS(Difference) <= Tolerance,
 "#E6E6E6", // White, 10% darker for match
 "#de6a73" // Theme color 8, 20% lighter for mismatch
)

```

## 24. CardColor3\_Header

```

CardColor3_Header =
VAR ECOMAmount = [TotalAdjustedECOMCosmalineCashAmount]
VAR ShippedAmount = [TotalAdjustedShippedCollectedAmount_Cosmaline]
VAR Difference = ECOMAmount - ShippedAmount
VAR Tolerance = ECOMAmount * 0.05
RETURN
IF(
 ABS(Difference) <= Tolerance,
 "#E6E6E6", // White, 10% darker for match
)

```

```
 "#de6a73" // Theme color 8, 20% lighter for mismatch
)
```

## 25. CardColor4\_Header

```
CardColor4_Header =
VAR ECOMAmount = [TotalAdjustedECOMAmount_Cybersource]
VAR CreditCardAmount = [TotalAdjustedCreditCardAmount]
VAR Difference = ECOMAmount - CreditCardAmount
VAR Tolerance = ECOMAmount * 0.05
RETURN
IF(
 ABS(Difference) <= Tolerance,
 "#E6E6E6", // White, 10% darker for match
 "#de6a73" // Theme color 8, 20% lighter for mismatch
)
```

# 8 Streamlit Application

The Streamlit application developed for the Malia project allows users to select and customize model parameters for Apriori and FP-Growth models. Users can choose between a full MySQL `oracle_data` table or upload their own data, modify thresholds for support, confidence, and lift, and generate association rules with visual insights. The application also includes an Exploratory Data Analysis (EDA) and provides explanations for better understanding of the models and metrics used.

Full code:

```
import pandas as pd
import ast
import re
import matplotlib.pyplot as plt
import seaborn as sns
import squarify
import streamlit as st
import seaborn as sns
from mlxtend.frequent_patterns import apriori, fpgrowth, association_rules
from PIL import Image
import requests
from io import BytesIO
import plotly.express as px

Set the page layout to wide mode
st.set_page_config(layout="wide")

Local file path for the image
#coverImage_path =
```

```
"C:/Users/Lenovo/AIRFLOW_DOCKER_1/streamlit/resources/5BannerImage.jpg"
coverImage_path = "https://raw.githubusercontent.com/Romanos-Rizk/AUB-
capstone/main/resources/5BannerImage.jpg"

Fetch the image data from the URL
response = requests.get(coverImage_path)
coverImage = Image.open(BytesIO(response.content))
Open the image file directly
#coverImage = Image.open(coverImage_path)

Example credentials (replace with more secure storage in production)
credentials = {
 "romanos": "romanos",
 "user2": "mypassword",
}

Initialize session state
if "logged_in" not in st.session_state:
 st.session_state["logged_in"] = False

if "current_page" not in st.session_state:
 st.session_state["current_page"] = "Login"

def login_page():
 # Add some padding to the top to centralize the content vertically
 st.markdown(
 """
 <style>
 .centered-content {
 padding-top: 150px; /* Adjust this value to control the amount of
white space at the top */
 }
 </style>
 """,
 unsafe_allow_html=True
)

 # Create two columns with a 2/3 and 1/3 ratio
 col1, col2 = st.columns([2, 1])

 # Display the company image in the left column, covering the entire column
 with col1:
 st.markdown('<div class="centered-content">', unsafe_allow_html=True) # Add the padding CSS
 st.image(coverImage, use_column_width=True)

 # Display the login form in the right column with additional padding
 with col2:
 st.markdown('<div class="centered-content">', unsafe_allow_html=True) # Add the padding CSS
 st.title("Login")

 # Username and Password input with unique keys
 username = st.text_input("Username", key="login_username")
```

```
password = st.text_input("Password", type="password",
key="login_password")

if st.button("Login"):
if credentials.get(username) == password:
st.session_state["logged_in"] = True
st.session_state["current_page"] = "Upload"
st.experimental_rerun() # Force a rerun to redirect immediately
else:
st.error("Invalid username or password")

st.markdown('</div>', unsafe_allow_html=True) # Close the centered
content div

if st.button("Login"):
 if credentials.get(username) == password:
 st.session_state["logged_in"] = True
 st.session_state["current_page"] = "Upload"
 else:
 st.error("Invalid username or password")

Redirect to the correct page based on session state
if st.session_state.get("logged_in"):
if st.session_state.get("current_page") == "Upload":
upload_page() # Render the Upload page if logged in and ready
elif st.session_state.get("current_page") == "Home":
home_page() # Render Home page after file upload is complete
else:
If not logged in, stay on the login page
login_page()

#-----
```

---

```
Preprocessing functions provided by you
def convert_strings_to_lists(df):
 df['product_category'] = df['product_category'].apply(lambda x:
ast.literal_eval(x))
 return df

def remove_product_not_found(df):
 df = df[df['product_name'] != 'Product Not Found']
 return df

def preprocess_category_list(category_list):
 return [re.sub(r'\.\(\)\]', '', category) for category in category_list]

def apply_preprocess_category_list(df):
 df['product_category'] =
df['product_category'].apply(preprocess_category_list)
 return df

def convert_lists_to_strings(df):
```

```
df['product_category'] = df['product_category'].apply(lambda x: ', '.join(x))
if x else 'Unknown')
return df

def remove_higher_selling_price(df):
 df = df[df['unit_selling_price'] <= df['unit_list_price']]
 return df

def calculate_total_sales(df):
 df['total_sales_with_discount'] = df['ordered_quantity'] *
df['unit_selling_price']
 df['total_sales_without_discount'] = df['ordered_quantity'] *
df['unit_list_price']
 return df

def calculate_discount_percentage(df, unit_list_price_col, unit_selling_price_col,
new_col_name='discount_percentage'):
 df[new_col_name] = (df[unit_list_price_col] - df[unit_selling_price_col]) /
df[unit_list_price_col] * 100
 return df

def expand_column_to_rows(df, column_to_expand, delimiter=', ',
new_col_name=None):
 if new_col_name is None:
 new_col_name = column_to_expand

 df_expanded = df.assign(**{new_col_name:
df[column_to_expand].str.split(delimiter)}).explode(new_col_name)
 return df_expanded

def preprocessing_pipeline_for_report(df):
 df = convert_strings_to_lists(df)
 df = remove_product_not_found(df)
 df = apply_preprocess_category_list(df)
 df = convert_lists_to_strings(df)
 df = remove_higher_selling_price(df)
 df = calculate_total_sales(df)
 df = calculate_discount_percentage(df, 'unit_list_price',
'unit_selling_price')
 return df

def upload_page():
 # Create three columns with a 1/3, 0.05 for divider, and 2/3 ratio
 col1, divider, col2 = st.columns([1, 0.05, 2])

 # Left Column: File Upload (1/3)
 with col1:
 st.markdown("<h1 style='color: #007bff;'>Upload Your Data</h1>",
unsafe_allow_html=True)

 file = st.file_uploader(
 "Please upload a CSV or Excel file",
 type=['csv', 'xlsx'],
 help="Make sure your file includes the required columns for Market
```

```
Basket Analysis."
)

Preprocessing and File Handling inside the first column
if file is not None:
 try:
 # Read the uploaded file into a DataFrame
 if file.name.endswith('.csv'):
 df = pd.read_csv(file)
 else:
 df = pd.read_excel(file)

 st.success("File uploaded successfully!")

 # Apply preprocessing pipeline
 data = preprocessing_pipeline_for_report(df)

 # Create the expanded DataFrame
 data_expanded = expand_column_to_rows(data, 'product_category')

 st.write("Preprocessing complete! Here's a preview of your data:")
 st.write(data.head())

 # Save 'data' and 'data_expanded' to session state so they can be
 used in other pages
 st.session_state['data'] = data
 st.session_state['data_expanded'] = data_expanded

 # Apply styling to make the "Proceed to Home Page" button larger
 st.markdown(
 """
 <style>
 .big-button {
 display: inline-block;
 padding: 1rem 2rem;
 font-size: 1.2rem;
 font-weight: bold;
 color: white;
 background-color: #4CAF50;
 border: none;
 border-radius: 10px;
 cursor: pointer;
 text-align: center;
 }
 .big-button:hover {
 background-color: #45a049;
 }
 </style>
 """", unsafe_allow_html=True
)

 # # Use Streamlit's st.button with CSS styling applied
 # if st.button("Proceed to Home Page", key="proceed_button"):
 # # Set both session states to ensure the flow moves forward
```

```
st.session_state["data_uploaded"] = True
st.session_state["current_page"] = "Home"
st.experimental_rerun() # Force a rerun to navigate to the
home page

except Exception as e:
st.error(f"An error occurred while processing the file: {e}")

Use Streamlit's st.button with CSS styling applied
if st.button("Proceed to Home Page", key="proceed_button"):
 # Update session state instead of forcing a rerun
 st.session_state["data_uploaded"] = True
 st.session_state["current_page"] = "Home"

except Exception as e:
 st.error(f"An error occurred while processing the file: {e}")

Middle Column: Vertical Line
with divider:
 st.markdown(
 """
<style>
.vertical-line {
 border-left: 2px solid #ccc;
 height: 100vh;
 margin: 0 auto;
}
</style>
<div class="vertical-line"></div>
""",
 unsafe_allow_html=True
)

Right Column: Data Format Guidelines and SQL Query Explanation (2/3)
with col2:
 st.markdown("<h1 style='color: #007bff;'>Data Format Guidelines for Market
Basket Analysis</h1>", unsafe_allow_html=True)

 st.markdown("""
Welcome to the **data upload page**. The dataset you upload will be used
for performing a comprehensive **Market Basket Analysis**.

To ensure the analysis is accurate and insightful, please upload
transactional data representing the purchases made by customers.

The required dataset can be sourced from the **`oracle_data`** and
`oracle_data_product_name` tables, which are available in the
`capstonetest` database.
""")

 st.markdown("""
The uploaded dataset should contain the following key columns:
- **Order Number**: A unique identifier for each transaction.
- **Item**: The specific item ordered.
- **Quantity**: The quantity of the item ordered.
""")
```

```
- **Price**: The selling price of the item.
- **Date**: The date when the order was placed.
- **Category**: The category to which the item belongs.
"""")

The SQL Query Reveal Button
if st.button("Need help with querying the data? Click here"):
 st.markdown("")
    ```sql  
SELECT  
    oracle_data.operating_unit_name,  
    oracle_data.ecom_reference_order_number,  
    oracle_data.ordered_item,  
    oracle_data.ordered_quantity,  
    oracle_data.unit_selling_price,  
    oracle_data.unit_list_price,  
    oracle_data.ordered_date,  
    oracle_data.tax_code,  
    oracle_data_product_name.product_id,  
    oracle_data_product_name.product_name,  
    oracle_data_product_name.product_category  
FROM  
    oracle_data  
JOIN  
    oracle_data_product_name  
ON  
    oracle_data.ordered_item = oracle_data_product_name.product_id  
WHERE  
    oracle_data.ordered_date BETWEEN 'start_date' AND 'end_date';  
```
```

Please replace `start\_date` and `end\_date` with your desired date range to filter the data accordingly.  
"""")

---

---

```
Home Page Function
def home_page():
 # Main Title
 st.markdown(
 "<h1 style='font-size:40px; text-align: center; color:#007bff;'>AUB x
Malia Market Basket Analysis Application</h1>",
 unsafe_allow_html=True
)

 # Key Features Section
 st.markdown("")
 <div style="background-color:#f9f9f9; padding:20px; border-radius:10px;
margin-bottom:20px;">
 <h2 style='color:#007bff;'>Key Features</h2>
 <p style="font-size:20px; line-height:1.6;">
```

This application provides an interactive interface for performing market basket analysis using the **Apriori** and **FP-Growth** algorithms.

It helps you uncover hidden patterns in customer purchases, enabling better product placement, promotions, and inventory management.

```
</p>
<ul style="font-size:20px; line-height:1.6;">
 Data Exploration: Get an overview of your transaction data with comprehensive Exploratory Data Analysis (EDA).
 Model Building: Generate association rules using Apriori and FP-Growth models.
 Insights and Recommendations: Analyze the generated rules and gain actionable business insights.

</div>
""", unsafe_allow_html=True)
```

# How to Use Section

```
st.markdown("""
<div style="background-color:#f0f8ff; padding:20px; border-radius:10px; margin-bottom:20px;">
 <h2 style='color:#007bff;'>How to Use</h2>
 <p style="font-size:20px; line-height:1.6;">
 Follow these steps to utilize the application:
 </p>
 <ol style="font-size:20px; line-height:1.6;">
 Data Page: Explore and understand your transaction data.
 Model Configuration: Set parameters and build your models.
 Results and Insights: View and analyze the generated rules and insights.

 <p style="font-size:20px;">
 We hope you find this application valuable for uncovering meaningful insights from your sales data. For further assistance, please refer to the documentation or contact support.
 </p>
</div>
""", unsafe_allow_html=True)
```

# Resources Section

```
st.markdown("""
<div style="background-color:#f9f9f9; padding:20px; border-radius:10px; margin-bottom:20px;">
 <h2 style='color:#007bff;'>Additional Resources</h2>
 <p style="font-size:20px; line-height:1.6;">
 For more information on market basket analysis, visit:
 </p>
 <ul style="font-size:20px; line-height:1.6;">
 Market Basket Analysis - Wikipedia
 Association Rule Learning - Wikipedia

</div>
""", unsafe_allow_html=True)
```

```

 </div>
 """", unsafe_allow_html=True)

Footer Section
st.markdown("""
<div style="font-size:16px; color:gray; text-align:right; margin-top:30px;">
 <p>To Malia Group, By AUB Students</p>
</div>
""", unsafe_allow_html=True)

#-----

```

```
def data_overview_page():
 # Main Title
 st.markdown(
 "<h1 style='font-size:40px; text-align: center; color:#007bff;'>Data
Overview</h1>",
 unsafe_allow_html=True
)

 # Display introduction and dataset description with larger text
 st.markdown("""
<div style="font-size:24px;">
 This page provides a detailed overview of the e-commerce order
transactions data used in the Market Basket Analysis application.
 It includes information about orders, products, pricing, and quantities
that are essential for uncovering patterns in customer purchasing behavior.
 Below are some key statistics about the dataset:
</div>
""", unsafe_allow_html=True)

 # Calculate key statistics
 num_rows = st.session_state["data"].shape[0]
 num_columns = st.session_state["data"].shape[1]
 num_orders = st.session_state["data"]["ecom_reference_order_number"].nunique()
 num_unique_items = st.session_state["data"]["ordered_item"].nunique()
 avg_basket_value =
 st.session_state["data"].groupby("ecom_reference_order_number")
 ["total_sales_with_discount"].sum().mean()
 avg_items_per_transaction = st.session_state["data"]
 ["ordered_quantity"].mean()
 avg_items_per_order =
 st.session_state["data"].groupby("ecom_reference_order_number")
 ["ordered_quantity"].sum().mean()

 # Additional metrics
 total_revenue = st.session_state["data"]["total_sales_with_discount"].sum()
 top_product_category = st.session_state["data_expanded"]
```

```
["product_category"].value_counts().idxmax()
avg_discount = st.session_state["data"]["discount_percentage"].mean()

First row of tiles
col1, col2, col3 = st.columns(3)

with col1:
 st.markdown(f"""
 <div style="text-align: center; padding: 20px; margin: 10px; background-color: #f0f8ff; border-radius: 10px; box-shadow: 0px 4px 10px rgba(0,0,0,0.1);">
 <h1 style="font-size: 48px; color: #007bff;">{num_rows}</h1>
 <p style="font-size: 18px;">Total Rows</p>
 </div>
 """", unsafe_allow_html=True)

with col2:
 st.markdown(f"""
 <div style="text-align: center; padding: 20px; margin: 10px; background-color: #f9f9f9; border-radius: 10px; box-shadow: 0px 4px 10px rgba(0,0,0,0.1);">
 <h1 style="font-size: 48px; color: #007bff;">{num_columns}</h1>
 <p style="font-size: 18px;">Total Columns</p>
 </div>
 """", unsafe_allow_html=True)

with col3:
 st.markdown(f"""
 <div style="text-align: center; padding: 20px; margin: 10px; background-color: #f0f8ff; border-radius: 10px; box-shadow: 0px 4px 10px rgba(0,0,0,0.1);">
 <h1 style="font-size: 48px; color: #007bff;">{num_orders}</h1>
 <p style="font-size: 18px;">Total Orders</p>
 </div>
 """", unsafe_allow_html=True)

Second row of tiles
col4, col5, col6 = st.columns(3)

with col4:
 st.markdown(f"""
 <div style="text-align: center; padding: 20px; margin: 10px; background-color: #f9f9f9; border-radius: 10px; box-shadow: 0px 4px 10px rgba(0,0,0,0.1);">
 <h1 style="font-size: 48px; color: #007bff;">{num_unique_items}</h1>
 <p style="font-size: 18px;">Unique Items</p>
 </div>
 """", unsafe_allow_html=True)

with col5:
 st.markdown(f"""
 <div style="text-align: center; padding: 20px; margin: 10px; background-color: #f0f8ff; border-radius: 10px; box-shadow: 0px 4px 10px rgba(0,0,0,0.1);">
 <h1 style="font-size: 48px; color: #007bff;">${total_revenue:,.2f}</h1>
 <p style="font-size: 18px;">Total Revenue</p>
 </div>
 """", unsafe_allow_html=True)
```

```
with col6:
 st.markdown(f"""
 <div style="text-align: center; padding: 20px; margin: 10px; background-color: #f9f9f9; border-radius: 10px; box-shadow: 0px 4px 10px rgba(0,0,0,0.1);">
 <h1 style="font-size: 48px; color: #007bff;">${avg_basket_value:.2f}</h1>
 </div>
 """, unsafe_allow_html=True)

Third row of tiles for additional metrics
col7, col8, col9 = st.columns(3)

with col7:
 st.markdown(f"""
 <div style="text-align: center; padding: 20px; margin: 10px; background-color: #f0f8ff; border-radius: 10px; box-shadow: 0px 4px 10px rgba(0,0,0,0.1);">
 <h1 style="font-size: 48px; color: #007bff;">{avg_discount:.2f}%</h1>
 <p style="font-size: 18px;">Avg Discount</p>
 </div>
 """, unsafe_allow_html=True)

with col8:
 st.markdown(f"""
 <div style="text-align: center; padding: 20px; margin: 10px; background-color: #f9f9f9; border-radius: 10px; box-shadow: 0px 4px 10px rgba(0,0,0,0.1);">
 <h1 style="font-size: 48px; color: #007bff;">{top_product_category}</h1>
 <p style="font-size: 18px;">Top Product Category</p>
 </div>
 """, unsafe_allow_html=True)

with col9:
 st.markdown(f"""
 <div style="text-align: center; padding: 20px; margin: 10px; background-color: #f0f8ff; border-radius: 10px; box-shadow: 0px 4px 10px rgba(0,0,0,0.1);">
 <h1 style="font-size: 48px; color: #007bff;">
{avg_items_per_transaction:.2f}</h1>
 <p style="font-size: 18px;">Avg Items per Transaction</p>
 </div>
 """, unsafe_allow_html=True)

#-----

import plotly.express as px

def eda_page():
st.title("Exploratory Data Analysis (EDA)")
```

```
Create a tab layout for the EDA categories
tabs = st.tabs(["Sales Analysis", "Order Analysis", "Pricing and Discount
Analysis", "Supply Chain Analysis", "Product Positioning Analysis"])

Sales Analysis Tab
with tabs[0]:
 st.markdown("""
 <div style="background-color: #f0f8ff; padding: 20px; border-radius: 10px;
box-shadow: 0px 4px 10px rgba(0,0,0,0.1);">
 <h2 style="text-align: center;">Sales Analysis Dashboard</h2>
 <p style="font-size:18px; line-height:1.6;">
 The Sales Analysis Dashboard provides a comprehensive overview of
sales performance across various metrics,
 including revenue distribution, customer purchasing behaviors, and
monthly sales trends. By analyzing the visualizations, you can identify the most
 significant product categories, understand the distribution of order
quantities, and explore the monetary distribution of transactions.
 </p>
 </div>
 """", unsafe_allow_html=True)

 st.markdown("

", unsafe_allow_html=True) # Add white space between
title and visualizations

 # First row: Revenue Contribution by Product Category and Distribution of
Order Quantities
 col1, col2 = st.columns(2)

 with col1:
 # Revenue Contribution by Product Category (sorted by total sales
descending)
 category_sales =
st.session_state["data_expanded"].groupby('product_category')
['total_sales_with_discount'].sum().reset_index().sort_values(by='total_sales_with
_discount', ascending=False)

 fig = px.bar(
 category_sales,
 x='product_category',
 y='total_sales_with_discount',
 labels={'product_category': 'Product Category',
'total_sales_with_discount': 'Total Sales'},
 title="Revenue Contribution by Product Category"
)
 fig.update_layout(xaxis_tickangle=-45, title_x=0.3,
title_font_size=20)
 fig.update_xaxes(showgrid=False)
 fig.update_yaxes(showgrid=False)
 st.plotly_chart(fig)

 with col2:
 # Distribution of Order Quantities
 fig = px.histogram(
 st.session_state["data"],
```

```
 x='ordered_quantity',
 nbins=30,
 title="Distribution of Order Quantities",
 labels={'ordered_quantity': 'Order Quantity'},
)
fig.update_layout(title_x=0.35, title_font_size=20)
fig.update_xaxes(showgrid=False)
fig.update_yaxes(showgrid=False)
st.plotly_chart(fig)

st.markdown("

", unsafe_allow_html=True) # Add white space between rows

Second row: Monetary Distribution and Monthly Sales Trend (Filtered for Jan, Feb, and March)
col3, col4 = st.columns(2)

with col3:
 # Monetary Distribution
 monetary =
st.session_state["data"].groupby('ecom_reference_order_number')
['total_sales_with_discount'].sum().reset_index()

 fig = px.histogram(
 monetary,
 x='total_sales_with_discount',
 nbins=30,
 title="Monetary Distribution",
 labels={'total_sales_with_discount': 'Total Spend'},
)
 fig.update_layout(title_x=0.4, title_font_size=20)
 fig.update_xaxes(showgrid=False)
 fig.update_yaxes(showgrid=False)
 st.plotly_chart(fig)

with col4:
 # Monthly Sales Trend as Line Chart
 df = st.session_state["data"]
 df['ordered_date'] = pd.to_datetime(df['ordered_date'],
errors='coerce')
 df['year_month'] = df['ordered_date'].dt.to_period('M').astype(str) # Convert Period to string

 monthly_sales = df.groupby('year_month')
['total_sales_with_discount'].sum().reset_index()

 # Create a line chart instead of a bar chart
 fig = px.line(
 monthly_sales,
 x='year_month',
 y='total_sales_with_discount',
 labels={'year_month': 'Month', 'total_sales_with_discount': 'Total Sales with Discount'},
 title="Monthly Sales Trend"
```

```
)

 fig.update_layout(
 xaxis_tickangle=-45,
 title_x=0.4,
 title_font_size=20
)

 # Plot the line chart
 st.plotly_chart(fig)

 st.markdown("

", unsafe_allow_html=True) # Add white space between
rows

 # Third row: Top 10 Best-Selling Products (full width, larger height)
 # Top 10 Best-Selling Products (last plot)
 top_products = st.session_state["data"][st.session_state["data"]
['product_name'] != 'Product Not Found']\br/> .groupby('product_name')['ordered_quantity']\br/> .sum().sort_values(ascending=False).head(10)

 fig = px.bar(
 top_products,
 x=top_products.index,
 y=top_products.values,
 labels={'x': 'Product Name', 'y': 'Total Quantity Sold'},
 title="Top 10 Best-Selling Products"
)
 fig.update_layout(xaxis_tickangle=-45, title_x=0.45, title_font_size=20,
height=700)
 fig.update_xaxes(showgrid=False)
 fig.update_yaxes(showgrid=False)
 st.plotly_chart(fig, use_container_width=True) # Full width for this plot

 # Order Analysis Tab
 with tabs[1]:
 st.markdown("""
 <div style='background-color: #f0f8ff; padding: 20px; border-radius: 10px;
box-shadow: 0px 4px 10px rgba(0,0,0,0.1);'
 <h2 style="text-align: center;">Order Analysis Dashboard</h2>
 <p style="font-size:18px; line-height:1.6;">
 The Order Analysis Dashboard provides detailed insights into order
frequency, quantity distributions, and customer behavior patterns.
 The visualizations below will help identify recency distributions,
statistics about order quantities, and how many items are typically included in
each order.
 </p>
 </div>
 """", unsafe_allow_html=True)

 st.markdown("

", unsafe_allow_html=True) # Add white space between
title and visualizations
```

```
First row: Recency Distribution and Order Quantity Statistics
col1, col2 = st.columns(2)

with col1:
 # Recency Distribution
 df = st.session_state["data"]
 df['ordered_date'] = pd.to_datetime(df['ordered_date'],
errors='coerce')
 df['recency'] = (df['ordered_date'].max() -
df['ordered_date']).dt.days
 rfm = pd.DataFrame({'recency':
df.groupby('ecom_reference_order_number')['recency'].min()})
 fig = px.histogram(
 rfm,
 x='recency',
 nbins=30,
 title="Recency Distribution",
 labels={'recency': 'Days Since Last Purchase'},
)
 fig.update_layout(title_x=0.35, title_font_size=20)
 fig.update_xaxes(showgrid=False)
 fig.update_yaxes(showgrid=False)
 st.plotly_chart(fig)

with col2:
 # Order Quantity Statistics
 order_quantity_stats =
st.session_state["data"].groupby('ecom_reference_order_number')[['ordered_quantity']].sum()
 summary_stats = order_quantity_stats.describe()

 stats = {
 'Statistic': ['Average', 'Minimum', 'Maximum', 'Q1 (25th percentile)', 'Median (50th percentile)', 'Q3 (75th percentile)'],
 'Value': [
 summary_stats['mean'],
 summary_stats['min'],
 summary_stats['max'],
 summary_stats['25%'],
 summary_stats['50%'],
 summary_stats['75%']
]
 }
 stats_df = pd.DataFrame(stats)

 st.markdown("<h3 style='text-align: center;'>Order Quantity Statistics</h3>", unsafe_allow_html=True)
 st.table(stats_df)

 st.markdown("

", unsafe_allow_html=True) # Add white space between rows
Second row: Distribution of Items per Order
```

```
col3, _ = st.columns([2, 1]) # First column with a larger width

with col3:
 # Distribution of Items per Order
 order_quantity_stats =
st.session_state["data"].groupby('ecom_reference_order_number')[['ordered_quantity']].sum()

 fig = px.histogram(
 order_quantity_stats,
 x=order_quantity_stats,
 nbins=30,
 title="Distribution of Items per Order",
 labels={'x': 'Number of Items per Order'},
)
 fig.update_layout(title_x=0.35, title_font_size=20)
 fig.update_xaxes(showgrid=False)
 fig.update_yaxes(showgrid=False)
 st.plotly_chart(fig)

Pricing and Discount Analysis Tab
with tabs[2]:
 st.markdown("""
 <div style='background-color: #f0f8ff; padding: 20px; border-radius: 10px;
 box-shadow: 0px 4px 10px rgba(0,0,0,0.1);'>
 <h2 style="text-align: center;">Pricing and Discount Analysis
 Dashboard</h2>
 <p style="font-size:18px; line-height:1.6;">
 This dashboard focuses on analyzing the pricing strategies and
 discount applications across different products and categories.
 Explore the distribution of selling and list prices, the impact of
 discounts on sales, and how discounts vary across product categories and
 popularity.
 </p>
 </div>
 """", unsafe_allow_html=True)

 st.markdown("

", unsafe_allow_html=True) # Add white space between
 title and visualizations

 # First row: Distribution of Selling Prices and Distribution of Discount
 Percentages
 col1, col2 = st.columns(2)

 with col1:
 # Distribution of Selling Prices and List Prices
 fig = px.histogram(
 st.session_state["data"],
 x=["unit_selling_price", "unit_list_price"],
 nbins=20,
 title="Distribution of Selling Prices and List Prices",
 labels={'x': 'Price'},
)
 fig.update_layout(title_x=0.35, title_font_size=20, barmode='overlay')
```

```
fig.update_xaxes(showgrid=False)
fig.update_yaxes(showgrid=False)
st.plotly_chart(fig)

with col2:
 # Distribution of Discount Percentages
 st.session_state["data"]['discount_percentage'] =
(st.session_state["data"]['unit_list_price'] - st.session_state["data"]
['unit_selling_price']) / st.session_state["data"]['unit_list_price'] * 100

 fig = px.histogram(
 st.session_state["data"],
 x='discount_percentage',
 nbins=20,
 title="Distribution of Discount Percentages",
 labels={'discount_percentage': 'Discount Percentage'},
)
 fig.update_layout(title_x=0.35, title_font_size=20)
 fig.update_xaxes(showgrid=False)
 fig.update_yaxes(showgrid=False)
 st.plotly_chart(fig)

st.markdown("

", unsafe_allow_html=True) # Add white space between
rows

Second row: Discount Percentage by Category and Scatter Plot of Discount
Percentage vs. Ordered Quantity
col3, col4 = st.columns(2)

with col3:
 # Discount Percentage by Category
 fig = px.box(
 st.session_state["data_expanded"],
 x='product_category',
 y='discount_percentage',
 title="Discount Percentage by Category",
 labels={'product_category': 'Product Category',
 'discount_percentage': 'Discount Percentage'},
)
 fig.update_layout(title_x=0.35, title_font_size=20)
 fig.update_xaxes(showgrid=False, tickangle=-90)
 fig.update_yaxes(showgrid=False)
 st.plotly_chart(fig)

with col4:
 # Scatter Plot of Discount Percentage vs. Ordered Quantity
 fig = px.scatter(
 st.session_state["data"],
 x='discount_percentage',
 y='ordered_quantity',
 title="Scatter Plot of Discount Percentage vs. Ordered Quantity",
 labels={'discount_percentage': 'Discount Percentage',
 'ordered_quantity': 'Ordered Quantity'},
)
```

```
fig.update_layout(title_x=0.20, title_font_size=20)
fig.update_xaxes(showgrid=False)
fig.update_yaxes(showgrid=False)
st.plotly_chart(fig)

st.markdown("

", unsafe_allow_html=True) # Add white space between rows

Third row: Comparison of Average Discounts for Popular and Less Popular Items and Box Plot of Unit Selling Prices by Top Performing Product Categories
col5, col6 = st.columns(2)

with col5:
 # Comparison of Average Discounts for Popular and Less Popular Items
 item_popularity = st.session_state["data"].groupby('product_name')[['ordered_quantity']].sum()
 median_popularity = item_popularity.median()
 st.session_state["data"]['popularity'] = st.session_state["data"]['product_name'].map(lambda x: 'Popular' if item_popularity[x] > median_popularity else 'Less Popular')
 average_discounts = st.session_state["data"].groupby('product_name')['discount_percentage'].mean()
 st.session_state["data"]['average_discount'] = st.session_state["data"]['product_name'].map(average_discounts)

 comparison_df = st.session_state["data"][[['product_name', 'popularity', 'average_discount']]].drop_duplicates()

 fig = px.histogram(
 comparison_df,
 x='average_discount',
 color='popularity',
 title="Comparison of Average Discounts for Popular and Less Popular Items",
 labels={'average_discount': 'Average Discount Percentage'},
 nbins=20,
)
 fig.update_layout(title_x=0.20, title_font_size=20)
 fig.update_xaxes(showgrid=False)
 fig.update_yaxes(showgrid=False)
 st.plotly_chart(fig)

with col6:
 # Box Plot of Unit Selling Prices by Top Performing Product Categories
 top_categories =
 st.session_state["data_expanded"].groupby('product_category')[['total_sales_with_discount']].sum().sort_values(ascending=False).head(10).index.to_list()
 top_categories_df = st.session_state["data_expanded"]
 [st.session_state["data_expanded"]['product_category'].isin(top_categories)]

 fig = px.box(
 top_categories_df,
 x='product_category',
```

```
 y='unit_selling_price',
 title="Box Plot of Unit Selling Prices by Top Performing Product
Categories",
 labels={'product_category': 'Product Category',
'unit_selling_price': 'Unit Selling Price'},
)
fig.update_layout(title_x=0.2, title_font_size=20)
fig.update_xaxes(tickangle=-90, showgrid=False)
fig.update_yaxes(showgrid=False)
st.plotly_chart(fig)

with tabs[3]:
 st.markdown("""

Supply Chain Analysis Dashboard

The Supply Chain Analysis Dashboard provides insights into the
distribution activities and supply chain efficiency.
Explore the top-performing distributors, shipment quantities, and
their relative contributions to the supply chain.

""", unsafe_allow_html=True)

 st.markdown("

", unsafe_allow_html=True) # Add white space between
title and visualizations

Bar Chart of Top Distributing Companies by Quantity Shipped
top_distributors = st.session_state["data"].groupby('operating_unit_name')
['ordered_quantity'].sum().sort_values(ascending=False).head(10)

Create a bar chart using Plotly
fig = px.bar(
 top_distributors.reset_index(),
 x='operating_unit_name',
 y='ordered_quantity',
 labels={'operating_unit_name': 'Distributing Company',
'ordered_quantity': 'Quantity Shipped'},
 title='Top Distributing Companies by Quantity Shipped'
)

Customize the layout of the chart
fig.update_layout(
 title_x=0.2, # Center the title
 title_font_size=20,
 xaxis_tickangle=-45, # Rotate x-axis labels for better readability
 xaxis_title_font=dict(size=15),
 yaxis_title_font=dict(size=15),
 showlegend=False, # Hide legend
 plot_bgcolor='rgba(0,0,0,0)', # Transparent background
 paper_bgcolor='rgba(0,0,0,0)', # Transparent paper background
 margin=dict(l=100, r=100, t=50, b=50) # Add margins to center the
plot
```

```
)\n\n # Center the plot on the page\n st.markdown("<div style='text-align: center;'>", unsafe_allow_html=True)\n st.plotly_chart(fig, use_container_width=False)\n st.markdown("</div>", unsafe_allow_html=True)\n\n with tabs[4]:\n st.markdown("""\n <div style='background-color: #f0f8ff; padding: 20px; border-radius: 10px;\n box-shadow: 0px 4px 10px rgba(0,0,0,0.1);'\n <h2 style="text-align: center;">Product Positioning Dashboard</h2>\n <p style="font-size:18px; line-height:1.6;">\n The Product Positioning Dashboard provides insights into the diversity\n and frequency of items purchased in transactions.\n It explores customer purchasing behaviors and highlights niche\n products that may have unique market opportunities.\n </p>\n </div>\n """, unsafe_allow_html=True)\n\n st.markdown("

", unsafe_allow_html=True) # Add white space between\n title and visualizations\n\n # First row: Distribution of the Number of Items per Transaction and\n Distribution of Item Frequencies\n col1, col2 = st.columns(2)\n\n with col1:\n # Distribution of the Number of Items per Transaction\n basket = (st.session_state["data"]\n .groupby(['ecom_reference_order_number', 'product_name'])\n ['product_name']\n .count().unstack().reset_index().fillna(0)\n .set_index('ecom_reference_order_number'))\n basket_sets = basket.applymap(lambda x: 1 if x > 0 else 0)\n unique_items_per_transaction = basket_sets.sum(axis=1)\n\n fig = px.histogram(\n unique_items_per_transaction,\n nbins=30,\n title="Distribution of the Number of Items per Transaction",\n labels={'value': 'Number of Items', 'count': 'Frequency'})\n)\n fig.update_layout(\n title_x=0.2,\n title_font_size=20,\n xaxis=dict(showgrid=False),\n yaxis=dict(showgrid=False),\n plot_bgcolor='rgba(0,0,0,0)')\n)\n st.plotly_chart(fig, use_container_width=True)\n\n with col2:
```

```
Distribution of Item Frequencies
item_frequencies = basket_sets.sum(axis=0)

fig = px.histogram(
 item_frequencies,
 nbins=30,
 title="Distribution of Item Frequencies",
 labels={'value': 'Frequency of Items', 'count': 'Number of Items'}
)
fig.update_layout(
 title_x=0.2,
 title_font_size=20,
 xaxis=dict(showgrid=False),
 yaxis=dict(showgrid=False),
 plot_bgcolor='rgba(0,0,0,0)'
)
st.plotly_chart(fig, use_container_width=True)

st.markdown("

", unsafe_allow_html=True) # Add white space between
rows

Second row: Transaction Diversity Statistics and Item Frequencies
Statistics
col3, col4 = st.columns(2)

with col3:
 # Transaction Diversity Statistics
 stats = unique_items_per_transaction.describe()
 stats_df = pd.DataFrame(stats, columns=['Value']).reset_index()
 stats_df.columns = ['Statistic', 'Value']
 st.markdown("<h3 style='text-align: center;'>Transaction Diversity
Statistics</h3>", unsafe_allow_html=True)
 st.table(stats_df)

with col4:
 # Item Frequencies Statistics
 stats = item_frequencies.describe()
 stats_df = pd.DataFrame(stats, columns=['Value']).reset_index()
 stats_df.columns = ['Statistic', 'Value']
 st.markdown("<h3 style='text-align: center;'>Item Frequencies
Statistics</h3>", unsafe_allow_html=True)
 st.table(stats_df)

st.markdown("

", unsafe_allow_html=True) # Add white space between
rows

Third row: Considered as Niche Products
st.markdown("<h3 style='text-align: center;'>Products Considered
Niche</h3>", unsafe_allow_html=True)

sales_volume = st.session_state["data"].groupby('product_name')
['ordered_quantity'].sum().reset_index()
niche_products = sales_volume[sales_volume['ordered_quantity'] <
sales_volume['ordered_quantity'].quantile(0.25)]
```

```
purchase_frequency = st.session_state["data"].groupby('product_name')
['ecom_reference_order_number'].nunique().reset_index()
niche_purchase_frequency =
purchase_frequency[purchase_frequency['ecom_reference_order_number'] <
purchase_frequency['ecom_reference_order_number'].quantile(0.25)]

combined_niche_products =
set(niche_products['product_name']).intersection(
 set(niche_purchase_frequency['product_name']))
)

combined_niche_products_df = pd.DataFrame({
 'Product Name': list(combined_niche_products)
})

st.table(combined_niche_products_df)

#-----

def run_model(df, model_choice, min_support, min_confidence, min_lift):
 # Preprocess the data into a basket format
 basket = (df
 .groupby(['ecom_reference_order_number', 'product_name'])
 ['product_name']
 .count().unstack().reset_index().fillna(0)
 .set_index('ecom_reference_order_number'))
 basket_sets = basket.applymap(lambda x: 1 if x > 0 else 0)

 # Choose between Apriori or FP-Growth
 if model_choice == "Apriori":
 frequent_itemsets = apriori(basket_sets, min_support=min_support,
use_colnames=True)
 elif model_choice == "FP-Growth":
 frequent_itemsets = fpgrowth(basket_sets, min_support=min_support,
use_colnames=True)

 # Check if frequent itemsets are found
 if frequent_itemsets.empty:
 return None

 # Generate association rules based on confidence and lift thresholds
 rules = association_rules(frequent_itemsets, metric="confidence",
min_threshold=min_confidence)
 filtered_rules = rules[rules['lift'] >= min_lift]

 # Format the antecedents and consequents as strings for better readability
 filtered_rules['antecedents'] = filtered_rules['antecedents'].apply(lambda x:
'', '.join(list(x)))
 filtered_rules['consequents'] = filtered_rules['consequents'].apply(lambda x:
'', '.join(list(x)))

 # Reset the index and return the rules
```

```
filtered_rules.reset_index(drop=True, inplace=True)
return filtered_rules

Define the model configuration page function correctly
def model_config_page():

 st.markdown(
 "


```

```
Show the selected parameters in a summary box
st.markdown(f"""
 <div style='background-color: #eaf2f8; padding: 10px; border-radius: 8px;'>
 <h4 style="text-align: center;">Selected Parameters</h4>
 <p>Model: {model_choice}</p>
 <p>Minimum Support: {min_support}</p>
 <p>Minimum Confidence: {min_confidence}</p>
 <p>Minimum Lift: {min_lift}</p>
 </div>
""", unsafe_allow_html=True)

Run Model Button with real-time feedback
st.markdown("
", unsafe_allow_html=True)
if st.button("Run Model", help="Run the selected algorithm with the specified parameters"):
 with st.spinner("Running the model..."):
 result = run_model(st.session_state['data'], model_choice,
min_support, min_confidence, min_lift)
 if result is None or result.empty:
 st.error("No rules found with the given combination of parameters.
Please adjust the thresholds.")
 else:
 st.success(f"Model run successfully! Number of rules:
{len(result)}")
 st.write(result)
 st.session_state['results'] = result # Store results in session state

Footer
st.markdown("""
 <div style="position: fixed; bottom: 10px; right: 10px; font-size:16px;
color:gray;">
 To Malia Group, By AUB Students
 </div>
""", unsafe_allow_html=True)

#-----

```

---

```
def model_insights_page():
 # Header section
 st.markdown("""
 <div style='background-color: #f0f8ff; padding: 20px; border-radius: 10px;
box-shadow: 0px 4px 10px rgba(0,0,0,0.1);'>
 <h2 style="text-align: center;">Results and Visualizations</h2>
 <p style="font-size:18px; line-height:1.6; text-align: center;">
 Explore the results of the Market Basket Analysis. Visualize key
association rules and filter them by antecedents.
 Additionally, examine the distribution of support, confidence, and
lift for a deeper understanding of the generated rules.
 </p>
 </div>
""")
```

```
</div>
"""", unsafe_allow_html=True)

st.markdown("
", unsafe_allow_html=True) # Add white space

Load the generated rules
if 'results' in st.session_state:
 results = st.session_state['results']

First Row: Scatter Plot and Explanation
col1, col2 = st.columns([3, 2]) # Wider column for plot, narrower for
description

with col1:
 st.markdown("<h3 style='text-align: center;'>Scatter Plot of Support
and Confidence</h3>", unsafe_allow_html=True)
 def plot_support_confidence(results):
 fig = px.scatter(
 results,
 x='support', y='confidence',
 size='lift', color='lift',
 title="Support vs Confidence",
 labels={'support': 'Support', 'confidence': 'Confidence'},
 hover_data=['antecedents', 'consequents'],
 color_continuous_scale='Viridis'
)
 fig.update_traces(marker=dict(line=dict(width=2,
color='DarkSlateGrey')))
 fig.update_layout(title_x=0.5, plot_bgcolor='rgba(0,0,0,0)')
 st.plotly_chart(fig, use_container_width=True)

 plot_support_confidence(results)

with col2:
 st.markdown("""
<div style='font-size:18px;'>

The scatter plot visualizes the association rules with the x-
axis representing the support, and the y-axis representing the confidence of each
rule.
 The size and color of the points represent the lift of the rule.

Use this plot to identify significant association rules
that can help in better decision-making and marketing strategies.
</div>
""", unsafe_allow_html=True)

 st.markdown("<hr>", unsafe_allow_html=True) # Separator

Second Row: Support, Confidence, and Lift Distributions (3 plots in one
row)
 st.markdown("<h3 style='text-align: center;'>Distributions of Support,
Confidence, and Lift</h3>", unsafe_allow_html=True)
 col3, col4, col5 = st.columns(3)

 with col3:
```

```
 st.markdown("<h4 style='text-align: center;'>Support
Distribution</h4>", unsafe_allow_html=True)
 fig, ax = plt.subplots(figsize=(6, 4))
 sns.histplot(results['support'], bins=20, kde=True, ax=ax)
 ax.set_xlabel('Support')
 ax.set_ylabel('Frequency')
 ax.grid(False)
 st.pyplot(fig)

 with col4:
 st.markdown("<h4 style='text-align: center;'>Confidence
Distribution</h4>", unsafe_allow_html=True)
 fig, ax = plt.subplots(figsize=(6, 4))
 sns.histplot(results['confidence'], bins=20, kde=True, ax=ax)
 ax.set_xlabel('Confidence')
 ax.set_ylabel('Frequency')
 ax.grid(False)
 st.pyplot(fig)

 with col5:
 st.markdown("<h4 style='text-align: center;'>Lift Distribution</h4>",
unsafe_allow_html=True)
 fig, ax = plt.subplots(figsize=(6, 4))
 sns.histplot(results['lift'], bins=20, kde=True, ax=ax)
 ax.set_xlabel('Lift')
 ax.set_ylabel('Frequency')
 ax.grid(False)
 st.pyplot(fig)

st.markdown("

", unsafe_allow_html=True) # White space before the
filtered rules section
 st.markdown("<hr>", unsafe_allow_html=True) # Separator

Filter by Antecedents
st.subheader("Filter Rules by Antecedents")
all_antecedents = sorted(set(results['antecedents']))
selected_antecedents = st.multiselect("Select Antecedents:",
all_antecedents)

Filter the results based on antecedents
if selected_antecedents:
 filtered_results =
results[results['antecedents'].isin(selected_antecedents)]
else:
 filtered_results = results

Display filtered rules using an accordion-style layout
st.subheader("Association Rules")
if filtered_results.empty:
 st.write("No rules found with the selected antecedents.")
else:
 for index, row in filtered_results.iterrows():
```

```
 with st.expander(f"Rule {index + 1}: {row['antecedents']} -> {row['consequents']}"):
 st.markdown(f"**Support:** {row['support']:.3f}")
 st.markdown(f"**Confidence:** {row['confidence']:.3f}")
 st.markdown(f"**Lift:** {row['lift']:.3f}")
 st.markdown(f"**Leverage:** {row.get('leverage', 'N/A'):.3f}")
 st.markdown(f"**Conviction:** {row.get('conviction', 'N/A'):.3f}")
 st.markdown(f"**Zhang's Metric:** {row.get('zhangs_metric', 'N/A'):.3f}")

 else:
 st.write("Please run the model first to generate results.")

 st.markdown("

", unsafe_allow_html=True) # White space

Footer
st.markdown("""
<div style="position: fixed; bottom: 10px; right: 10px; font-size:16px;
color:gray;">
 To Malia Group, By AUB Students
</div>
""", unsafe_allow_html=True)
```

---

```
#-----

def render_page():
 # Only show the navigation bar after login and data upload
 if st.session_state.get("logged_in") and
 st.session_state.get("data_uploaded"):
 # Define the navigation options for the main app
 navigation_options = ["Home", "Data Overview", "EDA", "Model Configuration", "Model Insights"]

 # Ensure the current page is valid (must be one of the navigation options)
 if st.session_state["current_page"] not in navigation_options:
 st.session_state["current_page"] = "Home" # Fallback to Home if invalid page

 # Custom CSS for the navigation bar
 st.markdown(
 """
<style>
.nav-container {
 display: flex;
 justify-content: space-evenly;
 background-color: #f8f9fa;
 padding: 10px;
 border-radius: 10px;
 margin-bottom: 20px;
 border: 1px solid #ddd;

```

```
 }
 .nav-button {
 width: 100%;
 background-color: #007bff;
 color: white;
 font-weight: bold;
 padding: 15px;
 border-radius: 0;
 text-align: center;
 cursor: pointer;
 text-decoration: none;
 transition: background-color 0.3s ease;
 border: none;
 }
 .nav-button:hover {
 background-color: #0056b3;
 }
 .nav-button.active {
 background-color: #0056b3;
 }
 .block-container {
 padding-top: 0px;
 padding-bottom: 0px;
 }
</style>
"""
,
unsafe_allow_html=True
)

Custom navigation bar
st.markdown('<div class="nav-container">', unsafe_allow_html=True)
col1, col2, col3, col4, col5 = st.columns([1, 1, 1, 1, 1])

Navigation without rerun
with col1:
 if st.button("Home", key="home_button"):
 st.session_state["current_page"] = "Home"

with col2:
 if st.button("Data Overview", key="overview_button"):
 st.session_state["current_page"] = "Data Overview"

with col3:
 if st.button("EDA", key="eda_button"):
 st.session_state["current_page"] = "EDA"

with col4:
 if st.button("Model Configuration", key="config_button"):
 st.session_state["current_page"] = "Model Configuration"

with col5:
 if st.button("Model Insights", key="insights_button"):
 st.session_state["current_page"] = "Model Insights"
```

```
st.markdown('</div>', unsafe_allow_html=True)

Route to the correct page based on session state
if st.session_state["current_page"] == "Home":
 home_page()
elif st.session_state["current_page"] == "Data Overview":
 data_overview_page()
elif st.session_state["current_page"] == "EDA":
 eda_page()
elif st.session_state["current_page"] == "Model Configuration":
 model_config_page()
elif st.session_state["current_page"] == "Model Insights":
 model_insights_page()

else:
 # If not logged in, show login page
 if not st.session_state.get("logged_in"):
 login_page()
 # If logged in but data not uploaded, show upload page
 elif not st.session_state.get("data_uploaded"):
 upload_page()

Ensure session state is initialized
if "logged_in" not in st.session_state:
 st.session_state["logged_in"] = False

if "data_uploaded" not in st.session_state:
 st.session_state["data_uploaded"] = False

if "current_page" not in st.session_state:
 st.session_state["current_page"] = "Login" # Default page is Login until
logged in

Render the appropriate page based on the session state
render_page()
```

## 9 Conclusion

---

This documentation provides an overview of the key components and processes implemented in the Malia project. By integrating various tools and methodologies, we have established a robust framework for data analysis and model customization. This ensures that the project is well-positioned to deliver valuable insights and supports ongoing and future work in e-commerce performance analysis.