Milestone 3 - Pipeline Orchestration, Visualization & AI-Powered Analytics

> **Deadline:** Monday 15 Dec. 11:59PM

Milestone Objective

Transform your data engineering pipeline into a production-ready orchestrated workflow using Apache Airflow, enable interactive data visualization, and implement an AI agent for natural-language SQL querying.

Note: All functions from Milestone 1 and 2 should be refactored and integrated into Airflow tasks.

Part 0: Docker Compose Setup

0.1 Airflow Integration

Add Apache Airflow services to your existing `docker-compose.yml` file:

- **Airflow Init** (initialize airflow)
- **Airflow Webserver** (UI access on port 8080)
- **Airflow Scheduler** (task orchestration)
- **Airflow Worker** (task execution) - if using CeleryExecutor
- **PostgreSQL** (Airflow metadata database - separate from data warehouse)
- **Redis** (for CeleryExecutor) - optional, can use LocalExecutor instead

0.2 Visualization Service

Add **ONE** of the following visualization services:

- **Streamlit** (recommended for simplicity)
- **Dash** (for more complex dashboards)
- **Apache Superset** (for enterprise-grade analytics)

Note: All services should be configured to work within the same Docker network for seamless communication.

Part 1: Airflow Pipeline Development

1.1 DAG Structure

Create a DAG named `stock_portfolio_pipeline_{teamname}` with the following task groups:

Stage 1: Data Cleaning & Integration

Convert all Milestone 1 functions into Airflow tasks:

1. **Task:** `clean_missing_values`
   - Handle missing values in daily_trade_prices
   - Use PythonOperator
2. **Task:** `detect_outliers`
   - Identify and handle outliers (>10% threshold)
   - Use PythonOperator
3. **Task:** `integrate_datasets`
   - Merge all datasets starting from trades.csv
   - Use PythonOperator
4. **Task:** `load_to_postgres`
   - Load cleaned data into PostgreSQL warehouse
   - Use PythonOperator or PostgresOperator

Stage 2: Encoding & Stream Preparation

Convert all Milestone 2 encoding functions into Airflow tasks:

1. **Task:** `prepare_streaming_data`
   - Extract 5% random sample for streaming
   - Save statistics for stream processing
   - Use PythonOperator
2. **Task:** `encode_categorical_data`
   - Apply encoding to all categorical columns in the remaining 95%
   - Generate lookup tables
   - Use PythonOperator

Stage 3: Kafka Streaming

Note: in this part you need to develop your tasks such that the consumer only starts when the producer is **DONE**, then consume using the `earliest` method, to avoid parallelism complexity.

1. **Task:** `start_kafka_producer`
   - Trigger Kafka producer script using BashOperator
   - Producer should run until the full streaming file is done the close.
   - End the produced messages with `EOS` message.
2. **Task:** `consume_and_process_stream`
   - Start consumer to process streamed records from the topic using the `earliest` mechanism and terminate the consuming loop after receiving `EOS`.
   - Apply encoding to each streamed row (once received) and finally save to `FINAL_STOCKS.csv`
   - Use PythonOperator
3. **Task**: `save_final_to_postgres`
   - Save the `FINAL_STOCKS.csv` to postgres
   - Use PythonOperator or PostgresOperator

Stage 4: Spark Analytics

1. **Task:** `initialize_spark_session`
   - Connect to Spark master node and create a spark session with the following name `M3_SPARK_APP_TEAM_NAME`
   - Use PythonOperator
2. **Task:** `run_spark_analytics`
   - Connect to the spark session
   - Read `FINAL_STOCKS.csv` into Spark DataFrame
   - Execute all Spark DataFrame operations from Milestone 2
   - Execute all Spark SQL queries from Milestone 2
   - Save each results from the spark functions to PostgreSQL analytics tables names `spark_analytics_{query number}`
   - Save each results from the spark sql to PostgreSQL analytics tables names `spark_sql_{query number}`
   - Use PythonOperator

Stage 5: Data Visualization

1. Task : `prepare_visualization`
   - Prepare any columns or aggregations needed for visualization, i.e. if you need extra columns or features, if you need to reverse any transformation made to correctly visualize, to revert any encoding or add any columns for initial values for encoding to help in the encoding
   - Use PythonOperator
2. **Task**: `start_visualization_service`

- Add one of the following
  - **Option A:** Launch Streamlit app using BashOperator

```
BashOperator(
    task_id='start_streamlit',
```

```
    bash_command='streamlit run /path/to/dashboard.py'
)
```

- **Option B:** Launch Dash app using PythonOperator

```
PythonOperator(
    task_id='start_dash',
    python_callable=run_dash_app
)
```

- **Option C:** Start Superset using DockerOperator or BashOperator

```
BashOperator(
    task_id='start_superset',
    bash_command='docker-compose up -d superset'
)
```

> Required visualizations will be mentioned later in the description

Stage 6: AI Agent Query Pipeline

Setup:

1. Create a volume for airflow called `agents`
2. Create a txt file in this volume and name it `user_query.txt` to simulate user input

Tasks:

- **Task:** `process_with_ai_agent`
  - Read natural language question from text file (e.g., `user_query.txt`) and save it to a variable
  - Create an ai agent with the following characteristics:
    - LLM: use a model (could be anything from `tinyllama` to any model you want, you can use ollama or huggingface for local models or any model api)
    - No tools needed
    - No memory needed.
    - Add an additional System Prompt to the model to instruct it to take the user prompt + column names from csv file and generate an SQL Query to answer the question.
  - Agent should:
    - Understand the natural language query
    - Take as input the csv column names as string
    - Convert the query to SQL
    - **APPEND** agent responses to a JSON file inside `agents` volume in the name `AGENT_LOGS.JSON` in the following form
      -
```
{
    "user_query": "<user_query>",
    "agent_response": "<agent_response>"
}
```
  - Use PythonOperator

BONUS:
You will be granted bonus points at a max of 5% on the project (combined) if you manage to implement one or more of the following (each point has different bonus weight):

1. Convert each stage into an airflow task group
2. Instead of reading the txt file and saving it to a variable, add a tool to the agent and instruct it to read the query file using this tool and answer it.
3. Furthermore, instead of reading the user query from a text file, implemented a user interfacing app (new parallel task) to help users interact with the ai agent (hint: you can streamlit here as well)
4. Add a tool to access the final stocks file from the database and execute the generated queries on it
5. Add a tool to save the output tables as csv files

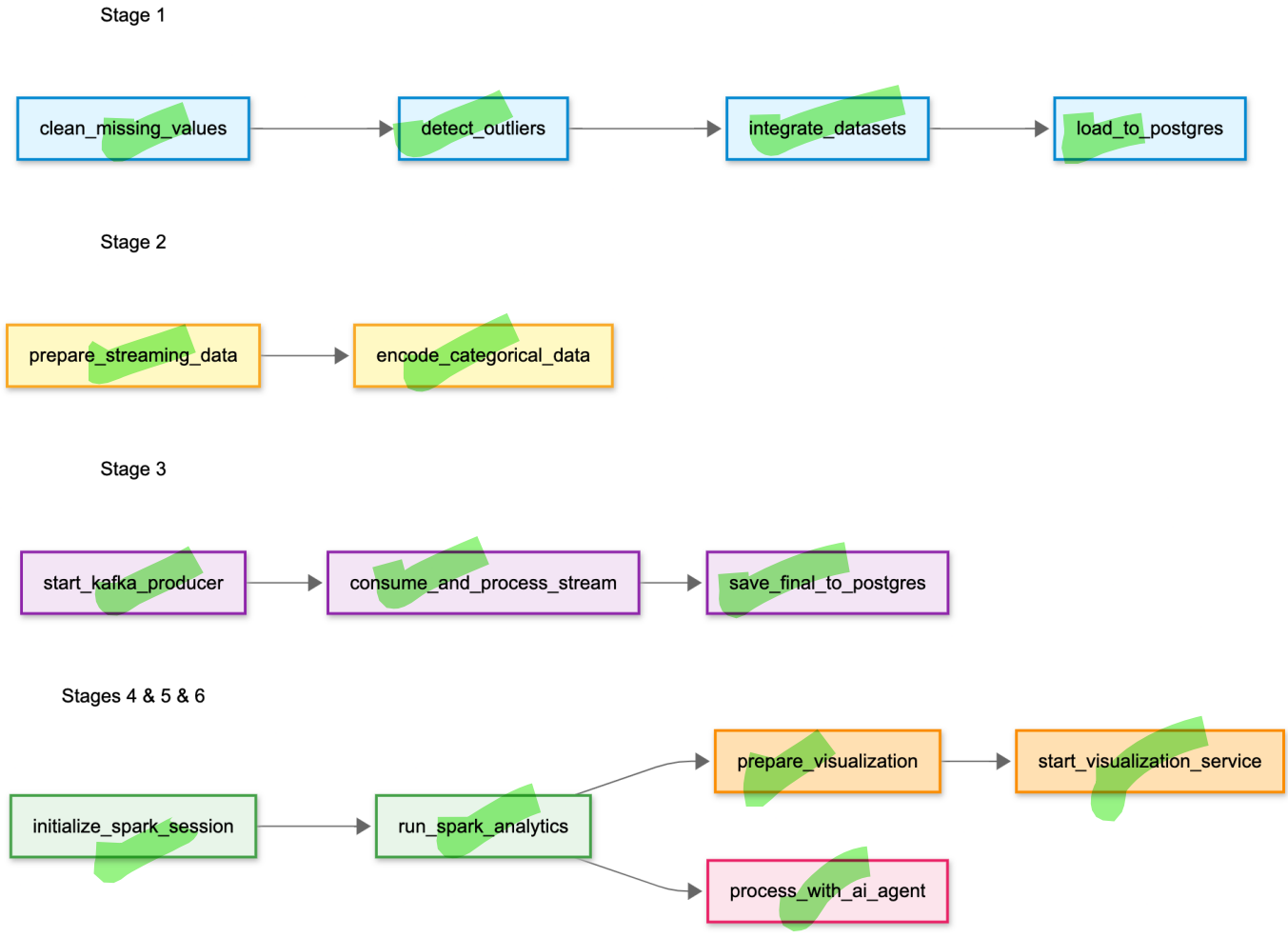
1.2 DAG Configuration

```
default_args = {
    'owner': 'data_engineering_team',
    'depends_on_past': False,
    'start_date': "<yesterday>", # change to use days ago
    'email_on_retry': False,
    'retries': 1
}

DAG(
    'stock_portfolio_pipeline',
    default_args=default_args,
    description='End-to-end stock portfolio analytics pipeline',
    schedule_interval='@daily',
    catchup=False,
    tags=['data-engineering', 'stocks', 'analytics'],
)
```

1.3 Task Dependencies

Define clear task dependencies using Airflow operators:



Part 2: Data Visualization Dashboard

2.1 Dashboard Requirements

Create an interactive dashboard that displays:

Core Visualizations (Mandatory)

1. **Trading Volume by Stock Ticker**
2. **Stock Price Trends by Sector**
3. **Buy vs Sell Transactions**

4. **Trading Activity by Day of Week**
5. **Customer Transaction Distribution**
6. **Top 10 Customers by Trade Amount**

Advanced Visualizations (Choose at least 2)

1. **Real-time Streaming Data Monitor** (updating as Kafka streams data)
2. **Portfolio Performance Metrics** (KPI cards)
3. **Sector Comparison Dashboard** (grouped bar charts)
4. **Holiday vs Non-Holiday Trading Patterns** (comparative analysis)
5. **Stock Liquidity Tier Analysis** (stacked area chart)

2.2 Dashboard Features

Your dashboard must include:

- **Filters:** Date range, stock ticker, sector, customer type
- **Interactive Elements:** Clickable charts for drill-down analysis
- ***BONUS***:
  - **Refresh Mechanism:** Ability to reload latest data from database
  - **Export Functionality:** Download visualizations as PNG/PDF (bonus)

2.3 Implementation Guidelines

If using Streamlit:

```
# Create dashboard.py
import streamlit as st
import pandas as pd
import plotly.express as px
from sqlalchemy import create_engine

# Connect to PostgreSQL
engine = create_engine('postgresql://user:pass@postgres:5432/stocks_db')

# Dashboard layout
st.title("Real-Time Stock Portfolio Analytics")
st.sidebar.header("Filters")

# Your visualization code here
```

If using Dash:

```
# Create dash_app.py
import dash
from dash import dcc, html, Input, Output
import plotly.graph_objs as go
import pandas as pd
from sqlalchemy import create_engine

app = dash.Dash(__name__)
engine = create_engine('postgresql://user:pass@postgres:5432/stocks_db')

# Dashboard layout and callbacks
```

If using Superset:

- Configure Superset to connect to PostgreSQL warehouse
- Create datasets from your analytics tables
- Build at least 6 charts covering the mandatory visualizations
- Combine charts into a dashboard
- Export dashboard configuration as JSON

Appendix A: AI Agent for Natural Language Querying

3.1 Agent Setup

Use can use any of the following setups

1. LangChain
2. LangGraph
3. CrewAI
4. Custom Agents using python classes

3.2 Query Implementation

Create a text file `user_query.txt` in your project directory with sample questions:

```
What was the total trading volume for technology stocks last month?
```

Appendix B: Testing & Validation

4.1 Pipeline Testing

1. **Run the complete DAG** and verify all tasks complete successfully
2. **Check task logs** in Airflow UI for any errors or warnings
3. **Validate data quality** at each pipeline stage

4.2 Visualization Testing

1. **Load dashboard** and verify all visualizations render correctly
2. **Test filters** and interactive elements
3. **Verify data accuracy** by comparing with database queries
4. **Check performance** with full dataset

Appendix C: Python Libraries

```
apache-airflow
pandas
numpy
psycopg2-binary
sqlalchemy
kafka-python-ng
pyspark
langchain
langchain-community
openai  # or google-generativeai, or langchain-ollama
streamlit  # or dash, plotly
python-dotenv
```

Deliverables

Create a new folder on your Google Drive for Milestone 3 and upload:

1. Airflow Implementation

- [ ] `stock_portfolio_pipeline.py` (complete DAG file)
- [ ] `dags/` folder with all task function modules
- [ ] Updated `docker-compose.yml` with Airflow and visualization services
- [ ] `.env` file with environment variables (API keys, DB credentials if exists)

2. Visualization

- [ ] Dashboard script (`dashboard.py` or `dash_app.py`)
- [ ] Screenshots of all visualizations in the dashboard (at least 6)
- [ ] If using Superset: dashboard configuration JSON file
- [ ] Documentation on how to access the dashboard (`VISUALIZATION.md`)

3. AI Agent

- [ ] `ai_agent.py` with agent implementation if in separate file
- [ ] `user_query.txt` with sample question
- [ ] `AGENT_LOGS.JSON` with agent outputs

4. Documentation

- [ ] Updated `README.md` with:
  - How to run the pipeline
  - How to access the Airflow UI
  - How to view the dashboard
  - How to test the AI agent
  - Architecture diagram (optional but recommended)

5. Screenshots

- [ ] Screenshots of Airflow DAG graph view
- [ ] Screenshots of successful DAG run

Resources

- Apache Airflow Documentation
- Streamlit Documentation
- LangChain SQL Tutorial