Shirli home assessment

1. Pipeline design:

Requirements:

• The assignment is to develop a data processing pipeline that will run different data processing "tasks" (a "task" is a code that can be run) on raw data. This system should process the tasks and store the processes' results in a database or a data lake.

• The system should be run on demand and process different predefined "tasks".

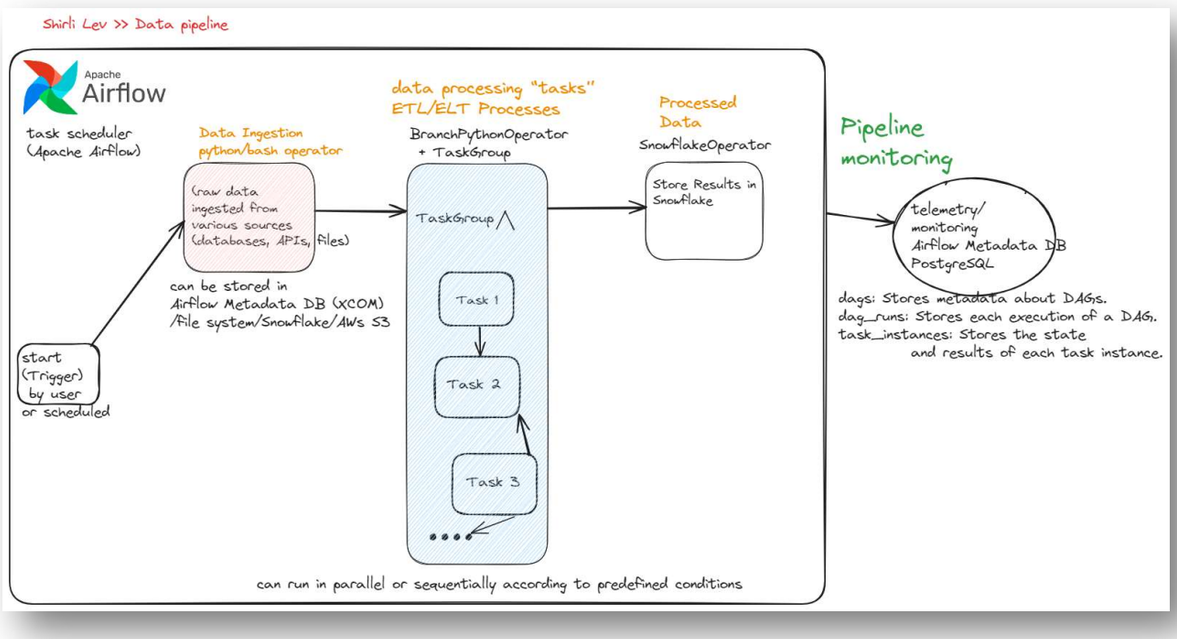
• One task can invoke other tasks.

• The system also produces telemetry on its runs (stores/updates the status of the runs).

*• **Pipeline monitoring**: Each execution should be monitored and store its **state** and **results**. This should be ready to be queried at any time.*

• You may use known orchestration services/frameworks.

a. Provide a diagram of the system (components/flows)



System Design:

1. **Data Ingestion**:
    - **PythonOperator** or **BashOperator** Fetches raw data.
2. **Task Grouping and Execution**:
    - ***(Optional) BranchPythonOperator**: Determines which tasks to run based on predefined conditions.*

- **TaskGroup**: Contains multiple tasks (Task1, Task2, Task3) that can run in parallel or sequentially.

3. **Result Storage**:
    - **SnowflakeOperator**: Stores the results of each task in Snowflake.
4. **Monitoring and Telemetry**:
    - **Airflow Database**: Uses PostgreSQL to store DAG runs, task instance states, and other telemetry data.

b. Provide the data model (including db. schema. Which database will you use? Why?)

Data Model and Database Schema

Database Choice:

- Airflow Metadata Database: PostgreSQL
- Results Storage: Snowflake

Schema for Airflow (PostgreSQL):

- dags: Stores metadata about DAGs.
- dag_runs: Stores each execution of a DAG.
- task_instances: Stores the state and results of each task instance.

Schema for Results (Snowflake):

tasks_results: Table to store results and telemetry of tasks.

- id: Primary key.
- task_id: Identifier for the task.
- dag_id: Identifier for the DAG.
- execution_date: Timestamp of the DAG run.
- status: Status of the task (success, failure, etc.).
- result: VARIANT column to store task result data.
- start_time: Timestamp when the task started.
- end_time: Timestamp when the task ended.
- logs: Any logs or additional information.

I believe Snowflake is an excellent choice for data storage and processing, especially for large-scale data pipelines, due to its scalability, performance, ease of use, and strong integration with various data processing tools like DBT and storage services like AWS S3.

As a cloud platform, Snowflake also supports semi-structured data formats such as JSON, XML and Parquet, making it suitable for handling raw data.

For orchestration, I have chosen Apache Airflow because it simplifies scheduling and monitoring workflows. It is great for batch processing, ETL pipelines, and data engineering workflows.

Airflow's task-based management, rich user interface for monitoring, and extensive integration options (DBT, AWS, Snowflake, PostgreSQL, MySQL) make it an ideal orchestration framework.

Additionally, Airflow fulfills the requirements of running the system on demand, processing different predefined tasks, and allowing one task to invoke other tasks.

The use of Airflow DAGs with Python Operators and Group Branch Operators allows for flexible task execution and dependencies. Tasks can invoke other tasks within Airflow using the TriggerDagRunOperator or by setting up dependencies.

Airflow's built-in metadata database (PostgreSQL) stores run statuses and basic telemetry.

Snowflake SQL Scripts

```
-- Create a database for our pipeline
CREATE DATABASE data_pipeline_db;
USE DATABASE data_pipeline_db;
-- Create a schema for our tables
CREATE SCHEMA pipeline_schema;
USE SCHEMA pipeline_schema;
-- Create tasks_results table
CREATE TABLE pipeline_schema.tasks_results (
    id INTEGER AUTOINCREMENT PRIMARY KEY,
    task_id STRING NOT NULL,
    dag_id STRING NOT NULL,
    execution_date TIMESTAMP NOT NULL,
    status STRING,
    result VARIANT, -- store semi-structured such as JSON, XML, Parquet
    start_time TIMESTAMP,
    end_time TIMESTAMP,
    logs STRING
);
```

c. An example of a query syntax to retrieve specific telemetry of a task that was run

```
SELECT task_id,
    dag_id,
    execution_date,
    status,
    result,
    start_time,
    end_time,
    logs
FROM tasks_results
WHERE task_id = 'task1'
```

AND execution_date > '2024-07-18 00:00:00';

2. *Coding skills assessment* *(not related to the above)*

a. You are asked to write a process that checks for the arrival of new files and loads them into their corresponding tables in the DB.

b. There are two types of files in the folder (See appendix below):

- *i. Objects_detection*
  1. *The format of the file name - objects_detection_[timestamp].json*
  2. *These files will hold streaming detection events that are sent from Mobileye's cars.*
  3. *Each file can hold one or more events.*
- *ii. Vehicles_status*
  1. *The format of the file name - vehicles_status_[timestamp].json*
  2. *These files will hold the latest status of each vehicle.*

c. You can choose whatever DB you wish to hold the received information, but you should take care of common queries that users can perform based on that information. You should also provide the code that configures the DB (creates table/defines scheme and so).

I am considering PostgreSQL and Snowflake for the database solution:
- PostgreSQL: Ideal for small-scale data. It offers a cost-effective, open-source solution with strong JSON support.
- Snowflake: Suitable for large-scale data due to its superior performance and scalability for handling large datasets efficiently.

I have tested the code locally without connecting to the database, and it works well.

Assumptions
Although this solution could be implemented on **AWS** (with files uploaded to S3 and a Lambda function triggered by S3 events), this is a code assessment, not an AWS exam.
Therefore, I assume the files are on the local file system and the data is not large-scale.
For this implementation, I use the `watchdog` library.

Options to Reduce CPU and Memory Usage
1. Load Files in Chunks with 'ijson':
   - Reduces memory usage by not loading the entire file into memory.
2. Concurrent Processing:
   - Multi-threading: For I/O-bound tasks, utilizes the same CPU core.
   - Multi-processing: For CPU-bound tasks, utilizes multiple CPU cores.
3. Batch Insertions:
   - Reduce overhead by inserting multiple rows into the database in one batch.
   - Store data temporarily in Python Pandas DataFrames for efficient manipulation.
4. Garbage Collection:
   - Manually trigger garbage collection to free up memory with the `gc` library.

All the code and readme files are available here:
https://github.com/shirlilev/mobileye-data-processor.git