Rystad Energy report 2023

Approach:

The problem statement requires the integration of data from multiple sources using open-source technologies. The task is to develop a Python-based ETL/ELT pipeline that can collect data from at least two public APIs and integrate it into a single database or data warehouse. The pipeline should be able to handle both structured and unstructured data and should be scalable and efficient. The pipeline should be designed in a modular and reusable way so that it can be easily adapted to different data sources or systems.

To solve this problem, I used Python, open-source libraries like requests, pandas, SQLAlchemy, and sqlalchemy-utils to create an ETL/ELT pipeline. I started by defining API endpoints for the OpenWeatherMap API and NewsAPI. I used API keys to authenticate the requests and obtain data from the APIs. I defined a function to fetch weather data for a given city and another function to fetch news data for a given country. These functions use the requests library to send HTTP requests and receive responses in JSON format.

After obtaining the data from the APIs, I transformed it into a pandas DataFrame and then loaded it into a PostgreSQL database using SQLAlchemy. I defined a function to transform and load data into the database. This function creates a connection to the database using create_engine() and then loads the DataFrame into a table using to_sql(). I used the if_exists parameter to specify that the data should be appended to the table if it already exists.

To make the pipeline scalable and efficient, I used modular and reusable code. I defined separate functions for fetching data from the APIs and transforming and loading data into the database. I used SQLAlchemy to handle the database connection and loading of data. I also used sqlalchemy-utils to handle database existence and creation.

Design decision:

I chose PostgreSQL as the database for this pipeline because it is a popular open-source database that is highly scalable, reliable, and efficient. I also chose to use SQLAlchemy as the Object-related mapping because it is a powerful and flexible library that can handle complex

data transformations and queries. I used pandas to handle the data as a DataFrame, which is a convenient data structure for handling structured data.

To handle API rate limits and errors gracefully, I added error handling to the functions that fetch data from the APIs. If the API returns an error, the function prints an informative message to the console and moves on to the next data source.

To expose a REST API endpoint that can query the integrated data and return the results in a JSON format, I would use Flask, which is a popular web framework for Python. I would define an API endpoint that queries the database using SQLAlchemy and returns the results as JSON.

Limitation:
One limitation of this solution is that it only handles data from two APIs. If there were more data sources to integrate, the code would need to be modified to handle them.

Another limitation is that the solution assumes that the data from the APIs is in a compatible format. If the format changes, the code may need to be updated to handle the new format.

Potential improvements:
To improve this solution, I would add more error handling to handle edge cases and unexpected errors. I would also add more data validation to ensure that the data is of the expected format before loading it into the database. I would also add more functionality to the REST API endpoint to support filtering and sorting of the data based on different criteria. I would also consider using a NoSQL database to handle unstructured data.