### Documentation for Earthquake Data Ingestion and Transformation Pipeline

#### ****Project Files:****

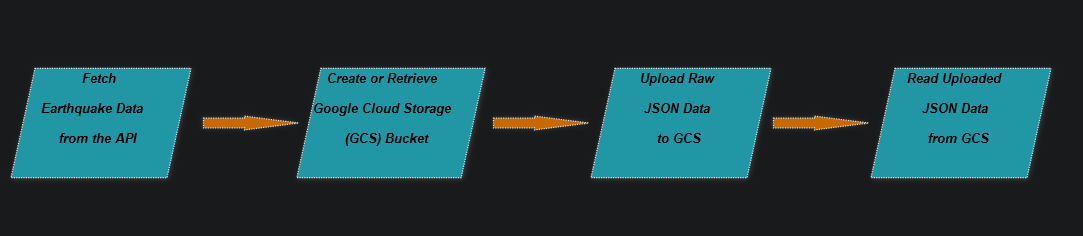
1. dataproc\_load\_historical\_data\_as\_parquet.py
2. dynamic\_methods.py
3. util.py
4. test\_dataproc.py

**Libarary Used:**

* **pyspark:** Used for distributed data processing.
* **google.cloud.storage:** For GCS operations.
* **google.cloud.bigquery:** For BigQuery interactions.
* **pytest:** For unit testing and validation.
* **json:** For JSON data parsing.
* **datetime:** For date handling and string formatting.
* **logging:** For logging process information and errors.

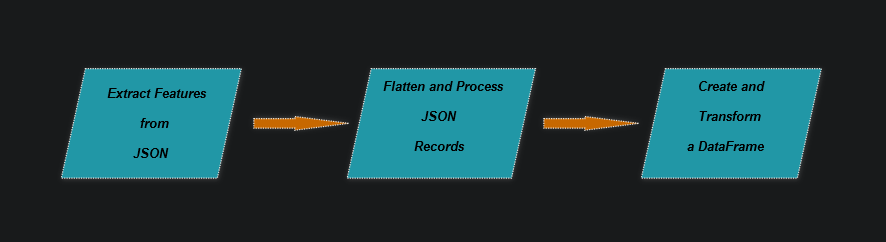
#### ****Step 1 to 4: Fetching Data and Uploading to Google Cloud Storage (GCS)****

1. **Fetch Earthquake Data from the API**
   * **Description**: Fetch historical earthquake data from the USGS Earthquake API for the past month in GeoJSON format.
   * **Implementation**:
     + URL: https://earthquake.usgs.gov/earthquakes/feed/v1.0/summary/all\_month.geojson
     + Use the request\_url utility function to send a request and retrieve the data.
   * **Logging**: Log whether the data fetch is successful or fails.
2. **Create or Retrieve a Google Cloud Storage (GCS) Bucket**
   * **Description**: Check if a bucket exists in the specified Google Cloud project. If not, create it.
   * **Implementation**:
     + Bucket name: earthquake\_analysis\_by\_hp\_24
     + Use the create\_bucket utility function to handle bucket creation or retrieval.
3. **Upload Raw JSON Data to GCS**
   * **Description**: Upload the fetched earthquake data as a raw JSON file to the "landing" area in the GCS bucket.
   * **Implementation**:
     + JSON file name: historical\_data\_<YYYYMMDD>.json
     + Folder path: pyspark\_dataproc\_testing/landing/
     + Use the upload\_to\_gcs utility function.
   * **Logging**: Log a confirmation when the file upload is successful.
4. **Read Uploaded JSON Data from GCS**
   * **Description**: Read the raw JSON file stored in GCS to ensure its integrity and availability for further processing.
   * **Implementation**:
     + Use the read\_json\_from\_gcs utility function to fetch the JSON data from GCS.
   * **Logging**: Log if the data read operation is successful or if there are issues.



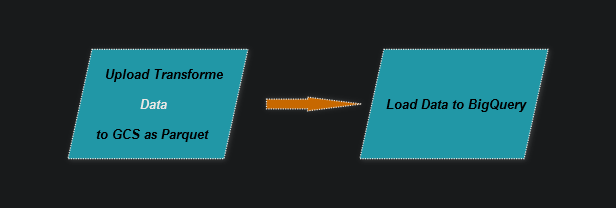
#### ****Step 5 to 7: Processing and Transforming Data****

1. **Extract Features from JSON**
   * **Description**: Extract the 'features' key from the JSON data retrieved from GCS, which contains earthquake records.
   * **Implementation**:
     + Check if the features key exists in the JSON data. If not, log an error.
   * **Output**: List of earthquake records.
2. **Flatten and Process JSON Records**
   * **Description**: Flatten nested JSON structures and transform the earthquake records into a clean tabular format.
   * **Implementation**:
     + Use the DataExtractor class to extract and flatten JSON properties for each earthquake record.
   * **Output**: A list of flattened and cleaned earthquake records.
3. **Create and Transform a DataFrame**
   * **Description**: Convert the processed records into a PySpark DataFrame and apply necessary transformations.
   * **Implementation**:
     + Use the DataProcessor class:
       - earthquake\_df\_creation: Create a DataFrame from the records.
       - earthquake\_transformation\_df: Apply transformations to the DataFrame (e.g., renaming columns, formatting timestamps).
   * **Logging**: Log when the DataFrame is created and successfully transformed.



#### ****Step 8 to 9: Uploading Transformed Data to GCS and BigQuery****

1. **Upload Transformed Data to GCS as Parquet**
   * **Description**: Write the transformed earthquake DataFrame to GCS in Parquet format for optimized storage and processing.
   * **Implementation**:
     + File name: flattened\_and\_transformed\_historical\_data\_<YYYYMMDD>.parquet
     + Folder path: pyspark\_dataproc\_testing/Silver/parquet/
     + Use the write\_df\_to\_gcs\_parquet utility function.
   * **Logging**: Log the success of the Parquet file upload.
2. **Load Data to BigQuery**
   * **Description**: Load the Parquet file from GCS into a BigQuery table for further analysis.
   * **Implementation**:
     + BigQuery table: <project\_id>.earthquake\_analysis.new\_flattened\_historical\_data\_by\_parquet
     + Define the schema explicitly using BigQuery's SchemaField class.
     + Use the load\_parquet\_data\_to\_bigquery\_from\_gcs utility function to load data.
   * **Logging**: Log the successful load into BigQuery.



#### ****Step 10: Export for Unit Testing****

1. **Convert DataFrame to JSON for Unit Testing**
   * **Description**: Convert the transformed DataFrame to a JSON format for unit testing purposes and upload it to GCS.
   * **Implementation**:
     + Convert the DataFrame to JSON using PySpark's toJSON method and collect the results.
     + Upload the JSON data to GCS using the upload\_to\_gcs utility function.
     + Folder path: pyspark\_dataproc\_testing/Silver/earthquake\_df\_to\_json/<YYYYMMDD>/
     + File name: earthquake\_df\_json\_data\_<YYYYMMDD>.json
   * **Logging**: Log when the JSON file is successfully uploaded for testing.

### Optimization Techniques in the Provided Code

Code demonstrates several optimization techniques to improve performance, maintainability, and scalability. Here's a breakdown:

#### ****1. Modularization with Utility Functions****

* **Techniques Used**:
  + Separation of concerns by using utility functions (request\_url, create\_bucket, upload\_to\_gcs, etc.) and dynamic methods (DataExtractor, DataProcessor) from external modules (util.py and dynamic\_methods.py).
  + This improves **reusability** and makes the codebase easier to test and maintain.

#### ****2. Efficient Use of Google Cloud Services****

* **Bucket and GCS Optimizations**:
  + Avoids repeatedly creating buckets by checking or reusing existing ones (create\_bucket ensures this). This minimizes redundant API calls, saving time and resources.
  + Uses **folder paths** in GCS for better organization of files, which simplifies navigation and querying of historical data.
* **BigQuery Optimization**:
  + Defines a schema explicitly when loading data into BigQuery. This prevents BigQuery from inferring the schema, which can be computationally expensive and error-prone.
  + Writes transformed data directly to **Parquet format**, a columnar storage format. Parquet is highly efficient for analytical workloads in BigQuery, especially when querying specific columns.

#### ****3. Handling Large Data Efficiently****

* **Techniques Used**:
  + Reads JSON data directly from GCS instead of working with local files, which avoids unnecessary disk I/O and leverages GCS's scalable infrastructure.
  + Uses Apache Spark's distributed processing capabilities (via DataProcessor methods) to handle large datasets efficiently.
  + Converts transformed DataFrame to **Parquet** for GCS upload, which:
    - Reduces data size due to compression.
    - Optimizes query performance for downstream BigQuery analysis.

#### ****4. Transformation Pipeline with Apache Spark****

* **Techniques Used**:
  + Implements a clear transformation pipeline:
    1. **Flattening and extracting** JSON data using DataExtractor.flatten\_json.
    2. Performing **data cleansing and formatting** using DataProcessor.earthquake\_transformation\_df.
* **Advantages**:
  + The pipeline minimizes intermediate data storage and processing overhead.
  + It leverages Spark’s in-memory computation for faster data processing.

#### ****5. Timestamp-Based Data Partitioning****

* **Techniques Used**:
  + Dynamically names files and folders using the current date (datetime.now().strftime('%Y%m%d')).
  + This enables time-based partitioning of data in GCS and BigQuery, optimizing retrieval and storage costs for time-series analysis.

#### ****6. Logging for Debugging and Monitoring****

* **Techniques Used**:
  + Uses Python's logging library to provide detailed logs at every critical step.
  + Logs provide useful context, including errors, successful uploads, and processing status, facilitating faster debugging and real-time monitoring.

#### ****7. Efficient Error Handling****

* **Techniques Used**:
  + Includes a try-except block to catch and log errors during execution. This ensures smooth handling of unexpected issues without crashing the pipeline.

#### ****8. Dynamic and Flexible Input/Output****

* **Techniques Used**:
  + Processes raw earthquake data dynamically from an external API, making the pipeline robust to changes in the source data.
  + Dynamically generates file names and paths, allowing the code to scale seamlessly across multiple runs without overwriting previous data.

#### ****9. PySpark Write Optimization****

* **Techniques Used**:
  + Writes the DataFrame directly to GCS in Parquet format using Spark’s efficient I/O operations.
  + Appends the date to the folder path to enable incremental data storage.

#### ****10. Testing with JSON Serialization****

* **Techniques Used**:
  + Converts the DataFrame to JSON format for testing (earthquake\_df.toJSON().collect()).
  + This ensures that the data can be validated before being used in downstream applications, improving pipeline reliability.

Testing Documentation

**1. Purpose**:  
It is a validation tests to ensure that the data is processed correctly.

**2. Libraries Used**:

* **PySpark**: For distributed data processing and Spark SQL operations.
* **Google Cloud BigQuery**: For reading and writing data from/to BigQuery.
* **pytest**: For unit testing and validation of data transformations.
* **json**: For parsing JSON data read from GCS.
* **datetime**: For working with dates and formatting strings.
* **logging**: For logging test and process information.

**3. Process Flow**:

* A Spark session is initialized.
* Data is read from BigQuery and GCS as DataFrames.
* Several tests are executed:
  1. **Row Count Match**: Verifies that the row count in the earthquake JSON data matches the BigQuery data.
  2. **Column Name Match**: Verifies that the column names in the earthquake JSON data match those in BigQuery.
  3. **Data Type Validation**: Ensures that the data types in the earthquake DataFrame match the expected schema from BigQuery.
  4. **Final Goal Validation**: Compares a sample of both DataFrames to ensure the final data is correct.
  5. **Cell-by-Cell String Comparison**: Performs a detailed row-by-row comparison of string columns to ensure data consistency between the Earthquake DataFrame and the BigQuery table.

**4. Pytest Commands**:  
To run the tests, use the following command in your terminal:

>> **pytest test\_dataproc.py -vv**

The -vv option provides a verbose output to help with debugging and understanding test results.