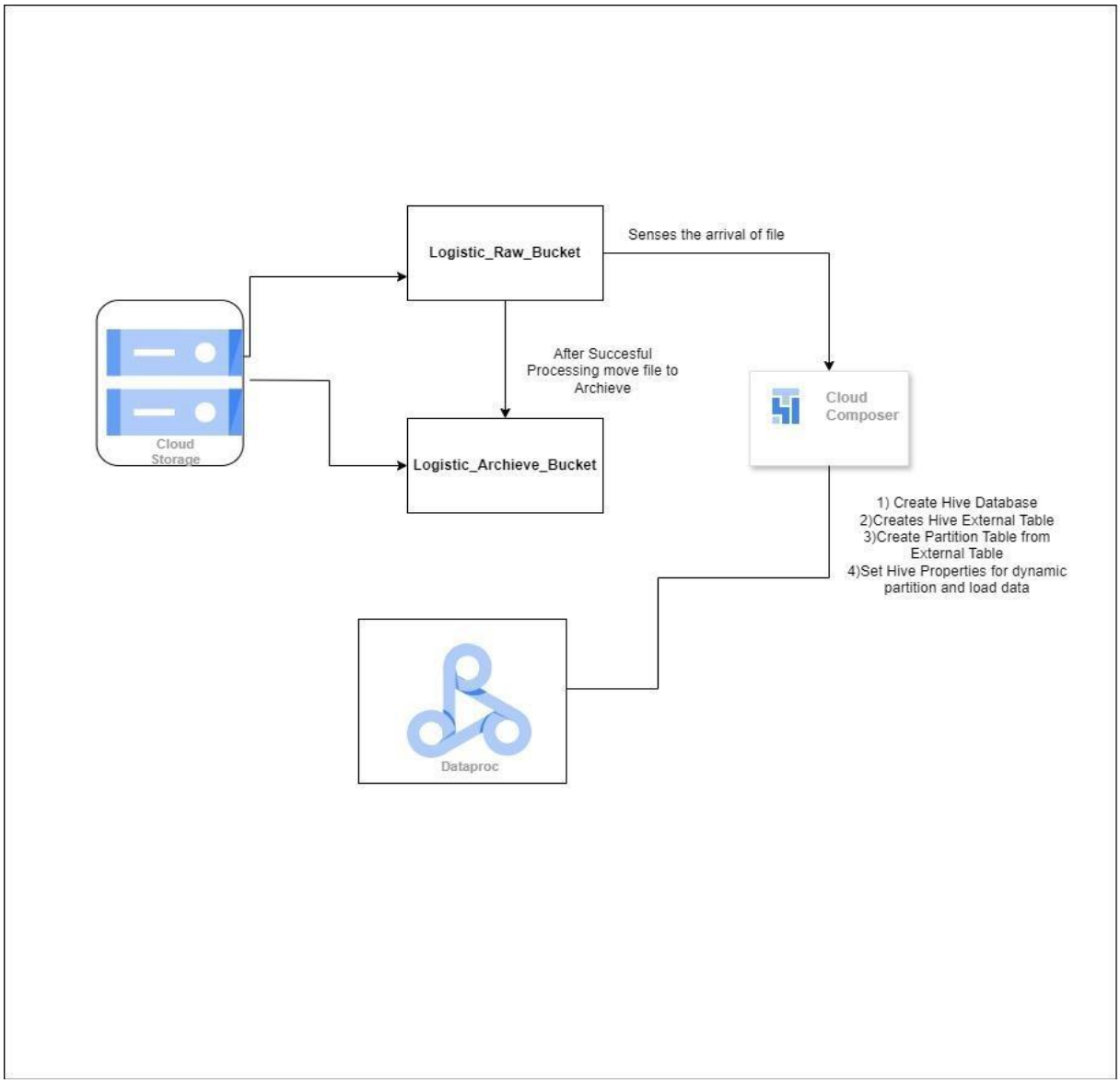
Maintaining Logistic Datawarehouse Project Using GCP

Project Architecture



Requirements

1. Create a Bucket in GCS for Logistic_Raw_Bucket and Logistic_Archieve_Bucket
2. Create DataProc Cluster in GCP and make sure you enable component gateway and check the YARN for observing, while creating cluster
3. Create Cloud Composer for airflow so that you can upload the airflow code in Dag Section

Airflow Code Explanation

1. Import the required dependencies like BashOperator, GCSObjectsWithPrefixExistenceSensor, DataprocSubmitJobOperator

   BashOperator : To move file from one bucket to another Bucket using linux command mv GCSObjectsWithPrefixExistenceSensor : For sensing the arrival of file in Logistic_Raw_Bucket DataprocSubmitJobOperator : Used to submit **Dataproc jobs** (such as **Spark**, **Hadoop**, or **Hive** jobs) from within your Airflow workflows.

2. Start with Default Args
3. Task 1 is for sensing the file in GCS where you will be giving task_id, bucket, prefix for bucket path, mode to check the file in bucket and poke_interval
4. Task 2 is creating Database if not exists and give the required Dataproc cluster name, region, project_id using DataprocSubmitJobOperator
5. Task 3 is Create hive external table in cluster with the required fields
6. Task 4 is set Hive properties for dynamic partition and load data
7. Task 5 is After successful processing and load move the file from Raw to archive using BashOperator

Observations :

Use of GCS Sensor for Dynamic Processing :

- The GCSObjectsWithPrefixExistenceSensor is used to monitor the Google Cloud Storage (GCS) bucket for files with a specific prefix (input_data/logistics_). This enables the DAG to trigger the pipeline dynamically based on the availability of new files in the GCS bucket, which is a useful approach for event-driven processing.

Automated Creation of Hive Database and Tables:

- The DAG ensures that the Hive database and tables are created if they do not exist. This makes the workflow resilient by not failing on subsequent runs and helps with easier setup during initial deployment.

Dynamic Partitioning in Hive:

- The set_hive_properties_and_load_partitioned task demonstrates the use of Hive's dynamic partitioning capabilities. It sets Hive properties to allow dynamic partitioning and inserts data into a partitioned table. This is an efficient way to manage and query large datasets by partitioning them based on a specified column (in this case, date).

Data Archival:

- The use of the BashOperator to move processed files to an archive location is a simple yet effective way to ensure data is not reprocessed on subsequent DAG runs. This practice is vital for maintaining a clean and organized data pipeline.

Clear Data Flow:

- The DAG tasks are arranged in a logical sequence: sensing new files, creating the database, defining the table, setting up partitions, and then archiving processed files. This helps in understanding the pipeline's flow and makes it easy to extend or modify the DAG in the future.

Resilience Against Failures:

- By specifying depends_on_past=False and email_on_failure=False, the DAG is configured to continue running without being halted by past failures, providing resilience to transient issues without manual intervention.

No Hardcoded Credentials or Sensitive Information:

- The DAG design adheres to good security practices by not including sensitive information (like credentials) directly in the code. It assumes that credentials are managed via environment variables or Airflow connections, which promotes better security management.

Optimized for Scalability:

- Using Google Cloud Dataproc for Hive tasks ensures the DAG can scale as the dataset grows. This cloud-native approach enables scalable and distributed data processing, avoiding the limitations of single-node setups.