**Pipeline Design Assignment:**

**a. Diagram of the system:**

attached

**b. Data model.**

User/Trigger/API: The task suppose to run on demand, which means either a user will start the step function, another step function will call the step function using an API call, or any AWS service will trigger the step function.

Step Functions: There is a step function for each different task (N tasks in total). The tasks are supposed to be divided by the runtime of the Glue ETL job and the raw data logic division (data type, creation time, data source, etc).

Start: The start of the step function.

State1 - Glue ETL job (1): Glue ETL job 1 is accessing the raw data that is stored in S3 buckets (extract). Every step function will trigger a different Glue ETL job function, which will contain a task code. The Glue ETL job function will access different raw data based on logical division (see 'S3 Storage - raw data').

After the Glue ETL job accesses its corresponding data, it will start to process (transform) it and will save the results of the process in the current state (state1) output of the step function.

State 1 will output the processed data in the state output to State 2's Glue ETL job.

S3 Storage - raw data: where the raw data is stored. The S3 needs to be divided based on logical division (data type, creation time, data source, etc.). This is for preventing data from being processed multiple times and for organization and simplicity. Ideally, every step function should access its own S3 bucket of data, but it depends on how the data is organized in the S3 (not a part of the question).

State2 - Glue ETL job (2): Glue ETL job 2 is for storing the processed data from Glue ETL job 1. State2 will connect its input state (state2) to State1 output state, so it will take the processed data as an input for the Glue ETL job.

Glue ETL job 2 will store (load) the processed data in Aurora DB in a logical way (see 'Aurora DB - processed and telemetry data').

Aurora DB - processed and telemetry data: I have decided to use Aurora DB because it has high performance (read replicas performance advanced algorithms), scalability (and auto-scalability), reliability, and resiliency (i.e., multi-az, continues backups). Also, Aurora is good at handling high volumes of data. In addition, Aurora is a SQL DB, which is good for structured data requirements and complex queries.

My DB scheme will look like this: the data will be stored in a logical way: two tables for each task - one for the processed data and one for the telemetry data. The process data will include the results of the processing task/the processed data. The telemetry data will include: status, duration, timestamp, how much data has been processed, errors, etc.

For the processed data table – table name: ProcessDataTaskN, description: <description>, columns: DataID (primary key – unique identifier for each processed data), TaskID, Timestamp, and ProcessedData, indexing (for improved querying performance): Timestamp.

For the telemetry data table: table name: TelemetryDataTaskN, description: <description>, columns: TelemetryID (primary key – unique identifier for each telemetry data), TaskID, StartTimestamp, EndTimestamp, Duration, Status, NumberOfChanges, etc, indexing (for improved querying performance): StartTimestamp.

Client/Application (querying): any client/application can query the data from the Aurora DB, either the processed data (the results) or the telemetry data.

End: the end of the step function.

CloudWatch Metrics and Logs - Glue ETL Jobs: We will create a cloudwatch dashboard for each type of Glue ETL job: one for the processing job and one for the storing job. Glue jobs automatically generate CloudWatch metrics such as number of execution runs, jobs success and failures, etc, and CloudWatch logs such as errors, etc.

CloudWatch Dashboards: We can create dashboards in CloudWatch from the logs and metrics we gather for each job type. In the dashboard, we will create four things: graph of all the metrics of the Glue ETL job for general information about the job, graph of all logs of Glue ETL job for troubleshooting and inspecting the job, alarms based on the metrics for notifying the administrators if something happened, and CloudWatch Log Insight for querying real-time information about the job (like the current state of the job).

User/Administrator (monitoring/alerted/querying): user/administrator can query, monitor, and get alerted about the data pipelines. They can monitor using the CloudWatch graphs, get alerts from the alerts, and query real-time information using CloudWatch Log Insight.

API call for invoking another task: If needed, we can define an API call from certain Glue ETL jobs to AWS for starting another step function/"task" (i.e. using boto3 in Python).

Notes:

1. Using two different ETL jobs for processing and storing: My approach is to store the processed data together when all the processing is done, instead of storing every single unit of processed data after it has been processed.

Resiliency: I think this approach is more resilient because, if the first Glue ETL job fails, it will be hard to keep track of which data has been processed and stored and which has not.

Simplicity: it is more simple to clearly see every state in the step function, and every Glue ETL job has its own rule. It's also easier to monitor and troubleshoot the data pipeline in this approach.

1. Glue ETL job instead of lambada:

Glue is more suitable for this use case because it is a service designated for ETL and therefore will do it more smoothly and efficiently. Furthermore, the Glue job can handle a high volume of data, and his time exceed limit is 48 hours (against 15 minutes in lambada).

1. Saving the telemetry data in the Aurora DB instead of using the telemetry data generated for CloudWatch:

Queries: Aurora has more complicated SQL queries.

Simplicity: the monitoring data is being separated from the telemetry data, making it simpler.

Retention: CloudWatch has retention for the data he gets, and Aurora does not.

But, it is still possible to query CloudWatch Log Insight for real-time and short-lived data, like the current state of the data pipeline (which is mentioned in the assignment).

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

SELECT \*

FROM TelemetryDataTask5

WHERE StartTimestamp >= '2024-03-05 00:00:00'

  AND EndTimestamp <= '2024-03-05 23:59:59'

ORDER BY StartTimestamp DESC;

In this query, I will retrieve all the columns of the telemetry rows from the ‘TelemetryDataTask5’ table (the telemetry data of task 5) that occurred in 3.5.2024, ordered by their start time (for convenience).