LOW LEVEL DOCUMENTATION -2

PROJECT-2

**Infrastructure Setup:**

S3, Lambda, DynamoDB, Glue

To go on with the project First we have created a S3 Bucket with the name groupno6 then we have created the folders like:

* Data/
* Logs/
* Results/
* Temp\_dir/

Create an Admin full access IAM role “Glue\_AdminFullAccessRole”

* 1. Go to IAM 🡪 Roles 🡪 Create role
  2. Select Type of trusted entity : AWS Service 🡪Service=Glue 🡪 Next
  3. Search policy 🡪 AdministratorAccess 🡪 Next:Tags 🡪 Next:Review
  4. Role Name : Glue\_AdminFullAccessRole 🡪 Create Role

We have created a database named **groupno6** and added tables using Aws glue Job and uploaded the data files in the S3 bucket with the names

* Project2-avro.avro
* Project2-parquet.parquet

Once the data files have been uploaded in the S3 bucket the Lambda function is triggered

Then through the Lambda function we check for the data file format and start the respective Aws glue job -- Avro/Parquet

In aws glue job we get the data files from the source which is S3 bucket and we transform it and give the processed data to the target which is DynamoDB tables

* We are creating 3 DynamoDB tables   
  + **Account Master**
  + **Transaction table**
  + **Source System Master**
* In the **Account Master** we enter records and give the acc\_no as partition key
* Acc\_no
* Acc\_name
* Acc\_desc
* Acc\_type
* In the **Source System Master** we enter records and give the System\_id as partition key
* System\_id
* System\_name
* In the **Transaction table** we enter records and give the txn\_id as partition key
* txn\_id
* system\_id
* acc\_no

once the glue job has started running the data files from the data location in the S3 is transferred to the archived location in the S3 bucket and when it has finished and succeeded the data is entered into the DynamoDB table ( [group6-transaction\_table](https://us-east-1.console.aws.amazon.com/dynamodbv2/home?region=us-east-1#table?name=group6-transaction_table) )

**AWS Lambda Function:**  
The AWS Lambda function code performs the following tasks:

1. It imports the necessary libraries, including `json`, `urllib.parse`, and `boto3`.

2. The `start\_glue\_job()` function is defined, which starts an AWS Glue job using the provided job name, file URI, and object key.

3. The `lambda\_handler()` function is defined, which serves as the entry point for the Lambda function.

4. The function receives an event and context as parameters, representing the Lambda event triggering the function and the runtime information.

5. The bucket name and object key are extracted from the event using the provided structure.

6. A file URI is constructed using the bucket name and object key, in the format `s3://bucket\_name/object\_key`.

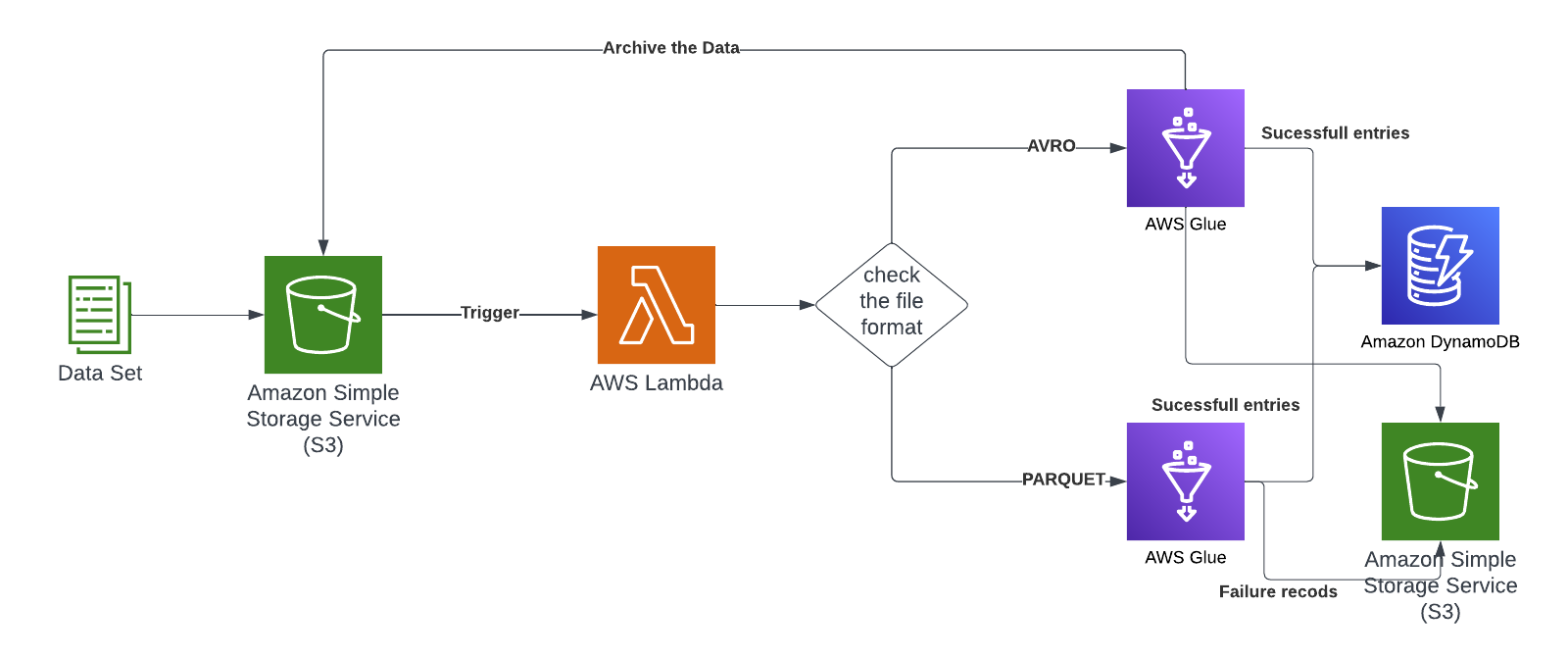
7. The file URI is printed for debugging or logging purposes.

8. The function checks the file URI to determine the file type. If it contains the string 'parquet', it starts the AWS Glue job named 'group6 parq' using the `start\_glue\_job()` function. If it contains the string 'avro', it starts the AWS Glue job named 'group6 avro'. Otherwise, it prints an unsupported file type message.

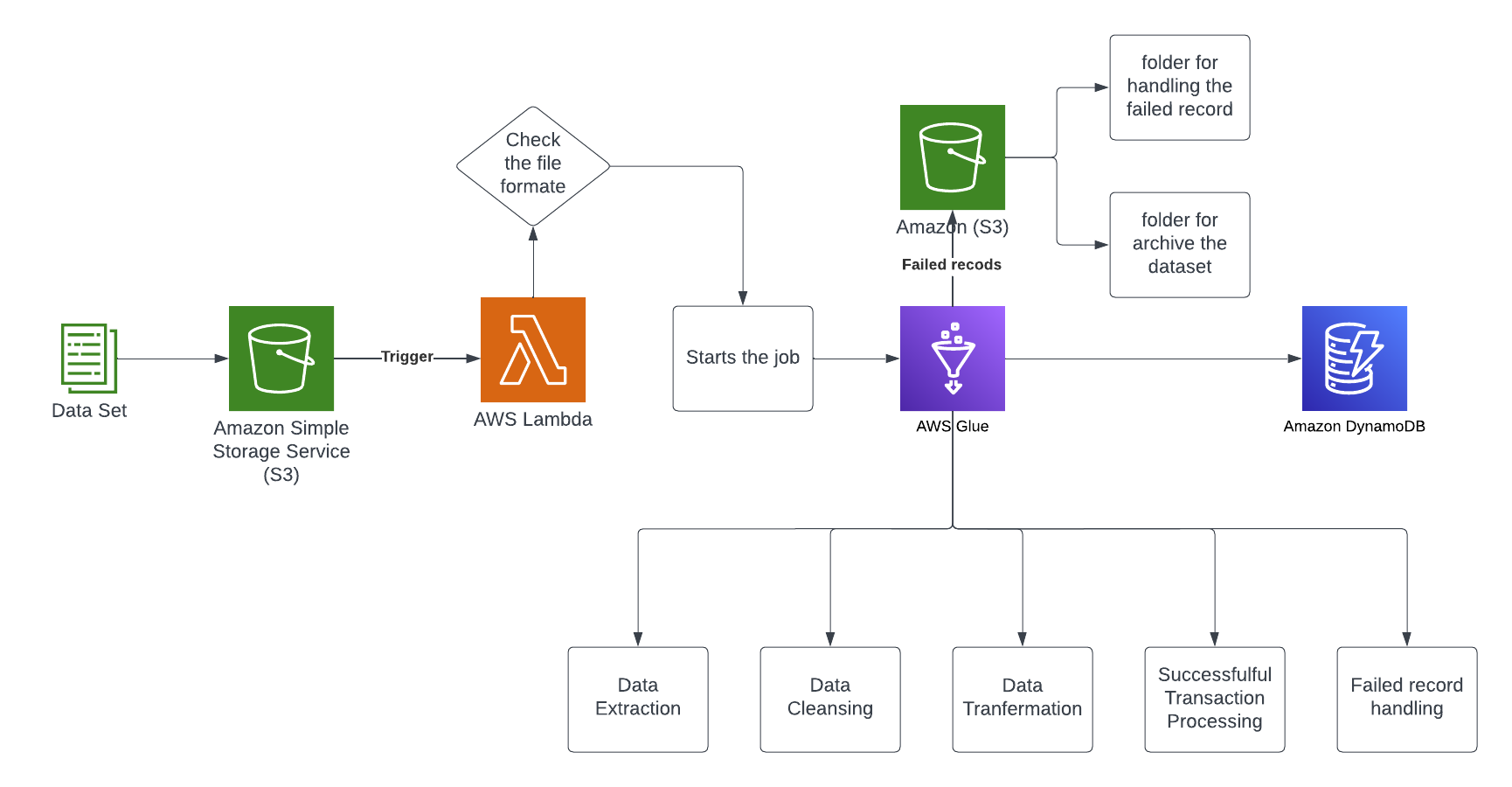
9. The function returns a response with a status code of 200 and a JSON body with the message 'Hello from Lambda!' (this response can be customized as needed).

In summary, the Lambda function is triggered by an S3 event. It extracts the bucket name and object key from the event, constructs a file URI, determines the file type based on the URI, and starts the corresponding AWS Glue job.

**Infrastructure Setup Block Diagram:**



**Data Processing Block Diagram:**



**AWS glue Avro**

The provided code is an AWS Glue job written in Python. Here's a brief explanation of its functionality:

The necessary libraries and modules are imported, including sys, AWS Glue components (awsglue.transforms, awsglue.utils, awsglue.context, awsglue.job), Spark components (pyspark.context, pyspark.sql.functions, pyspark.sql.window), and uuid.

The script retrieves the arguments passed to the job, including the job name, file path, and key, using getResolvedOptions().

Spark and Glue contexts are initialized.

The code creates a dynamic frame S3bucket\_node1 by reading the data from an S3 bucket specified by the file\_path argument.

The dynamic frame is converted to a DataFrame using the toDF() method.

The Boto3 library is imported to perform S3 operations.

Source and destination paths for S3 operations are defined, including the source bucket, source key, destination bucket, and destination key.

A new S3 client is created using the default AWS credentials.

The file is copied from the source to the destination using the copy\_object() method of the S3 client.

The original file is deleted from the source folder using the delete\_object() method of the S3 client.

A generate\_unique\_id() function is defined to generate a unique ID using the uuid library.

An example usage of the generate\_unique\_id() function generates a unique ID stored in id\_int.

Data transformation operations are performed on the DataFrame to create a new DataFrame result\_df. These operations include column renaming, data manipulation, and column splitting.

A window specification is created using Window.orderBy().

The result\_df DataFrame is further transformed and columns are selected using select().

The resulting DataFrame is converted back to a DynamicFrame using DynamicFrame.fromDF().

The script writes the DynamicFrame to a DynamoDB table specified by the connection options using glueContext.write\_dynamic\_frame.from\_options().

The job is committed using job.commit(), indicating the successful execution of the job.

In summary, the code reads data from an S3 bucket, performs data transformation operations on the data, copies the file to a destination location, deletes the original file, generates a unique ID, and writes the transformed data to a DynamoDB table.

**AWS glue parquet**

This AWS Glue job code performs the following tasks:

1. It imports the necessary libraries and modules required for the job, such as `sys`, AWS Glue components (`awsglue.transforms`, `awsglue.utils`, `awsglue.context`, `awsglue.job`), Spark components (`pyspark.context`, `pyspark.sql.functions`, `pyspark.sql.window`), and `uuid`.

2. The `generate\_unique\_id()` function is defined, which generates a unique ID using the `uuid` library.

3. The script retrieves the arguments passed to the job, including the job name, file path, and key, using `getResolvedOptions()`.

4. Spark and Glue contexts are initialized.

5. The code creates a dynamic frame `S3bucket\_node1` by reading the data from an S3 bucket specified by the `file\_path` argument. The data is in the Parquet format.

6. The dynamic frame is converted to a DataFrame using the `toDF()` method.

7. The Boto3 library is imported to perform S3 operations.

8. Source and destination paths for S3 operations are defined, including the source bucket, source key, destination bucket, and destination key.

9. A new S3 client is created using the default AWS credentials.

10. The file is copied from the source to the destination using the `copy\_object()` method of the S3 client.

11. The `generate\_unique\_id()` function is called to generate a unique ID stored in `id\_int`.

12. A window specification is created using `Window.orderBy()`.

13. The DataFrame is transformed using `selectExpr()` to perform column renaming and data manipulation. The transformed data is stored in the `result\_df` DataFrame.

14. The `result\_df` DataFrame is further transformed by adding a new column `txn\_id` using `F.concat()`, `F.lit()`, `F.row\_number()`, and `over(window)`. The column is cast to an integer.

15. The column order in the `result\_df` DataFrame is rearranged to place `txn\_id` as the first column.

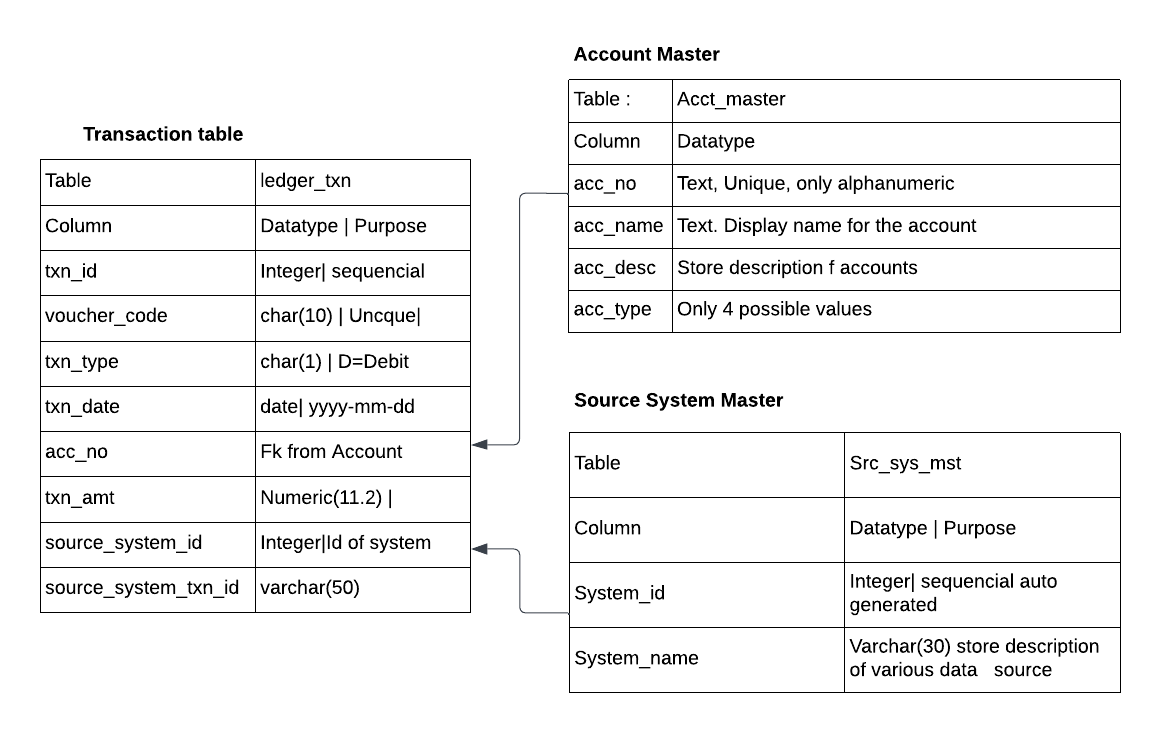
16. The `result\_df` DataFrame is converted back to a DynamicFrame using `DynamicFrame.fromDF()`.

17. The script writes the DynamicFrame to a DynamoDB table specified by the connection options using `glueContext.write\_dynamic\_frame.from\_options()`.

18. The original file is deleted from the source folder using the `delete\_object()` method of the S3 client.

19. The job is committed using `job.commit()`, indicating the successful execution of the job.In summary, the code reads data from an S3 bucket in Parquet format, performs data transformation operations, copies the file to a destination location, generates a unique ID, writes the transformed data to a DynamoDB table, and deletes the original file from the source folder.

**DynamoDB Table structure**



**Processing Steps:**

* For Parquet Files:
* Read the Parquet file from the S3 bucket.
* Extract the necessary fields (trnrefid, code, tdate, trn\_amount, vat, excise\_duty).
* Generate voucher code from trnrefid.
* Validate the transaction date to ensure it's not a future date.
* Create three separate transaction entries for each transaction:
* Credit "Product Sale" with trn\_amount.
* Credit "Value Added Tax" with vat.
* Credit "Excise Duty" with excise\_duty.
* Post the transactions to the ledger\_txn table.
* If any transaction fails, create a separate file for failed records.
* For Avro Files:
* Read the Avro file from the S3 bucket.
* Extract the necessary fields (Tran\_ref\_id, Transaction\_dt, amt, gst, custom\_duty).
* Generate voucher code from Tran\_ref\_id.
* Validate the transaction date to ensure it's not a future date.
* Create three separate transaction entries for each transaction:
* Credit "Product Sale" with amt.
* Credit an appropriate account (not specified in the requirements) with gst.
* Credit "Custom Duty" with custom\_duty.
* Post the transactions to the ledger\_txn table.
* If any transaction fails, create a separate file for failed records.

**Error Handling and Logging:**

* Implement robust error handling mechanisms to capture and handle exceptions during data processing.
* Use logging frameworks (e.g., AWS Cloud execution flow, identify issues, and aid in troubleshooting.

**Automation Considerations:**

* Set up AWS CloudWatch Events to trigger the Lambda function at the end of each day when new files arrive in the S3 bucket.
* Configure Lambda to handle file processing and posting of transactions automatically.
* Schedule periodic checks for failed transactions and reload the data if needed.

This provides overview of the system design based on the given requirements. It's important to consider additional factors, such as security, performance optimization, and scalability, when implementing the system in a production environment.

While the Low-Level Design (LLD) provides a comprehensive plan for implementing the system, it's important to note that the actual implementation may require additional adjustments and optimizations. These modifications are influenced by specific requirements, resource limitations, and adherence to best practices. It's crucial to adapt the implementation accordingly to ensure an effective and efficient system.