# **End-to-End ETL Pipeline: Aurora PostgreSQL to Snowflake**

## **Overview**

This document outlines an end-to-end ETL pipeline that extracts payment system data from Aurora PostgreSQL databases, processes it through various AWS services, and loads it into Snowflake. The pipeline includes comprehensive auditing, error handling, restart capabilities, and notifications.

## **Architecture Diagram**

┌────────────────┐ ┌───────────────┐ ┌───────────────┐ ┌───────────────┐ ┌────────────┐  
│ Amazon │ │ │ │ │ │ │ │ │  
│ EventBridge ├─────►│ AWS Lambda ├─────►│ AWS Step ├─────►│ AWS Glue ├─────►│ Snowflake │  
│ (Scheduler) │ │ (Trigger) │ │ Functions │ │ Jobs │ │ │  
└────────────────┘ └───────────────┘ └───────────────┘ └───────┬───────┘ └────────────┘  
 │ │  
 │ │  
 ┌───────▼───────┐ ┌───────▼───────┐  
 │ AWS DynamoDB │ │ Amazon S3 │  
 │ (Config/Audit)│ │ (Data Storage)│  
 └───────────────┘ └───────────────┘  
 │  
 │  
 ┌───────▼───────┐  
 │ Amazon SNS │  
 │ (Notifications│  
 └───────────────┘

## **Data Flow**

1. Data is extracted from Aurora PostgreSQL (multiple databases/schemas)
2. Loaded to S3 raw layer (CSV format)
3. Transformed and loaded to S3 refined layer (Parquet format)
4. Finally loaded to Snowflake tables

## **Components**

### **1. AWS Services Used**

* **Amazon EventBridge**: Triggers the pipeline hourly
* **AWS Lambda**: Initial validation and orchestration
* **AWS Step Functions**: Main workflow orchestration
* **AWS Glue**: ETL processing jobs
* **Amazon S3**: Data storage in raw and refined layers
* **AWS DynamoDB**: Configuration and audit tracking
* **Amazon SNS**: Notifications for success/failure
* **AWS Secrets Manager**: Credential storage and management

### **2. Database Tables**

#### **DynamoDB Tables**

* **ConfigTable**: Stores configuration for each source table
* **JobStatusTable**: Tracks job execution status
* **TableAuditTable**: Records audit information for each table
* **ProcessingMetricsTable**: Stores processing metrics

## **Implementation Details**

### **1. Configuration Management**

The ConfigTable in DynamoDB stores the following information for each source table:

{  
 "table\_id": "payments\_transactions",  
 "source\_database": "payments\_db",  
 "source\_schema": "public",  
 "source\_table": "transactions",  
 "target\_s3\_raw\_path": "s3://raw-layer/payments/transactions/",  
 "target\_s3\_refined\_path": "s3://refined-layer/payments/transactions/",  
 "target\_snowflake\_database": "PAYMENTS",  
 "target\_snowflake\_schema": "PUBLIC",  
 "target\_snowflake\_table": "TRANSACTIONS",  
 "primary\_key\_columns": ["transaction\_id"],  
 "incremental\_column": "last\_updated\_timestamp",  
 "active": true  
}

### **2. Pipeline Components**

#### **2.1 EventBridge Rule**

{  
 "name": "hourly-etl-trigger",  
 "schedule": "cron(0 \* \* \* ? \*)",  
 "target": {  
 "id": "etl-trigger-lambda",  
 "arn": "arn:aws:lambda:region:account:function:etl-trigger-lambda"  
 }  
}

#### **2.2 ETL Trigger Lambda**

# **1. ETL Trigger Lambda Function**

import boto3 import json import os from datetime import datetime

def lambda\_handler(event, context): """ Entry point Lambda function that validates if a new ETL job can be started """ # Initialize DynamoDB client dynamodb = boto3.resource('dynamodb') job\_status\_table = dynamodb.Table(os.environ['JOB\_STATUS\_TABLE'])

# Check if there's any job currently running  
response = job\_status\_table.scan(  
 FilterExpression='job\_status = :status',  
 ExpressionAttributeValues={':status': 'RUNNING'}  
)  
  
if response['Items']:  
 print("Another job is currently running. Skipping this execution.")  
 return {  
 'statusCode': 200,  
 'body': json.dumps('ETL job skipped - another job is currently running'),  
 'should\_continue': False  
 }  
  
# Check if there's any failed job that needs fixing  
response = job\_status\_table.scan(  
 FilterExpression='job\_status = :status',  
 ExpressionAttributeValues={':status': 'FAILED'}  
)  
  
if response['Items']:  
 print("Previous job failed. Manual intervention required before starting a new job.")  
 # Send SNS notification  
 sns\_client = boto3.client('sns')  
 sns\_client.publish(  
 TopicArn=os.environ['NOTIFICATION\_TOPIC'],  
 Subject='ETL Job Needs Attention',  
 Message='A previous ETL job has failed and requires manual intervention before starting a new job.'  
 )  
 return {  
 'statusCode': 200,  
 'body': json.dumps('ETL job skipped - previous job failed'),  
 'should\_continue': False  
 }  
  
# Create a new job entry  
job\_id = f"ETL-{datetime.now().strftime('%Y-%m-%d-%H-%M-%S')}"  
job\_status\_table.put\_item(  
 Item={  
 'job\_id': job\_id,  
 'job\_status': 'RUNNING',  
 'start\_time': datetime.now().isoformat(),  
 'end\_time': None,  
 'error\_message': None  
 }  
)  
  
# Trigger Step Function execution  
step\_function\_client = boto3.client('stepfunctions')  
step\_function\_client.start\_execution(  
 stateMachineArn=os.environ['STEP\_FUNCTION\_ARN'],  
 name=job\_id,  
 input=json.dumps({'job\_id': job\_id})  
)  
  
return {  
 'statusCode': 200,  
 'body': json.dumps(f'ETL job {job\_id} started successfully'),  
 'should\_continue': True,  
 'job\_id': job\_id  
}

# **2. Step Function Definition (AWS Step Functions - JSON)**

step\_function\_definition = { "Comment": "ETL Pipeline for Aurora PostgreSQL to Snowflake", "StartAt": "GetActiveTables", "States": { "GetActiveTables": { "Type": "Task", "Resource": "arn:aws:lambda:region:account:function:get-active-tables", "Next": "ProcessTables" }, "ProcessTables": { "Type": "Map", "ItemsPath": "$.tables", "Parameters": { "table.$": "$$.Map.Item.Value", "job\_id.$": "$.job\_id" }, "Iterator": { "StartAt": "ExtractFromSource", "States": { "ExtractFromSource": { "Type": "Task", "Resource": "arn:aws:states:::glue:startJobRun.sync", "Parameters": { "JobName": "extract-to-raw", "Arguments": { "--job\_id.$": "$.job\_id", "--table\_id.$": "$.table.table\_id" } }, "Catch": [{ "ErrorEquals": ["States.ALL"], "Next": "HandleFailure" }], "Next": "ValidateRawExtract" }, "ValidateRawExtract": { "Type": "Task", "Resource": "arn:aws:lambda:region:account:function:validate-extract", "Parameters": { "job\_id.$": "$.job\_id", "table\_id.$": "$.table.table\_id", "layer": "raw" }, "Catch": [{ "ErrorEquals": ["States.ALL"], "Next": "HandleFailure" }], "Next": "TransformToRefined" }, "TransformToRefined": { "Type": "Task", "Resource": "arn:aws:states:::glue:startJobRun.sync", "Parameters": { "JobName": "transform-to-refined", "Arguments": { "--job\_id.$": "$.job\_id", "--table\_id.$": "$.table.table\_id" } }, "Catch": [{ "ErrorEquals": ["States.ALL"], "Next": "HandleFailure" }], "Next": "ValidateRefinedTransform" }, "ValidateRefinedTransform": { "Type": "Task", "Resource": "arn:aws:lambda:region:account:function:validate-transform", "Parameters": { "job\_id.$": "$.job\_id", "table\_id.$": "$.table.table\_id", "layer": "refined" }, "Catch": [{ "ErrorEquals": ["States.ALL"], "Next": "HandleFailure" }], "Next": "LoadToSnowflake" }, "LoadToSnowflake": { "Type": "Task", "Resource": "arn:aws:states:::glue:startJobRun.sync", "Parameters": { "JobName": "load-to-snowflake", "Arguments": { "--job\_id.$": "$.job\_id", "--table\_id.$": "$.table.table\_id" } }, "Catch": [{ "ErrorEquals": ["States.ALL"], "Next": "HandleFailure" }], "Next": "ValidateSnowflakeLoad" }, "ValidateSnowflakeLoad": { "Type": "Task", "Resource": "arn:aws:lambda:region:account:function:validate-load", "Parameters": { "job\_id.$": "$.job\_id", "table\_id.$": "$.table.table\_id", "layer": "snowflake" }, "Catch": [{ "ErrorEquals": ["States.ALL"], "Next": "HandleFailure" }], "Next": "UpdateTableAudit" }, "UpdateTableAudit": { "Type": "Task", "Resource": "arn:aws:lambda:region:account:function:update-table-audit", "Parameters": { "job\_id.$": "$.job\_id", "table\_id.$": "$.table.table\_id", "status": "SUCCESS" }, "End": true }, "HandleFailure": { "Type": "Task", "Resource": "arn:aws:lambda:region:account:function:handle-failure", "Parameters": { "job\_id.$": "$.job\_id", "table\_id.$": "$.table.table\_id", "error.$": "$.error" }, "Next": "CleanupFailedData" }, "CleanupFailedData": { "Type": "Task", "Resource": "arn:aws:lambda:region:account:function:cleanup-failed-data", "Parameters": { "job\_id.$": "$.job\_id", "table\_id.$": "$.table.table\_id" }, "Next": "UpdateTableAuditFailure" }, "UpdateTableAuditFailure": { "Type": "Task", "Resource": "arn:aws:lambda:region:account:function:update-table-audit", "Parameters": { "job\_id.$": "$.job\_id", "table\_id.$": "$.table.table\_id", "status": "FAILED" }, "End": true } } }, "Next": "CheckProcessStatus" }, "CheckProcessStatus": { "Type": "Task", "Resource": "arn:aws:lambda:region:account:function:check-process-status", "Parameters": { "job\_id.$": "$.job\_id" }, "Next": "FinalizeJob" }, "FinalizeJob": { "Type": "Task", "Resource": "arn:aws:lambda:region:account:function:finalize-job", "Parameters": { "job\_id.$": "$.job\_id" }, "End": true } } }

# **3. Get Active Tables Lambda Function**

def get\_active\_tables(event, context): """ Lambda function to get all active tables for ETL processing """ # Get job\_id from the input event job\_id = event['job\_id']

# Initialize DynamoDB client  
dynamodb = boto3.resource('dynamodb')  
config\_table = dynamodb.Table(os.environ['CONFIG\_TABLE'])  
  
# Get all active tables  
response = config\_table.scan(  
 FilterExpression='active = :active\_value',  
 ExpressionAttributeValues={':active\_value': True}  
)  
  
tables = response['Items']  
  
# Return job\_id and tables for further processing  
return {  
 'job\_id': job\_id,  
 'tables': tables  
}

# **4. AWS Glue Job - Extract to Raw**

import sys from awsglue.transforms import \* from awsglue.utils import getResolvedOptions from pyspark.context import SparkContext from awsglue.context import GlueContext from awsglue.job import Job import boto3 import psycopg2 import pandas as pd from io import StringIO import datetime

def get\_postgres\_connection(hostname, port, database, username, password): """ Establish connection to PostgreSQL database """ conn = psycopg2.connect( host=hostname, port=port, database=database, user=username, password=password ) return conn

def extract\_data(conn, table\_config, job\_id): """ Extract data from PostgreSQL table based on last extraction timestamp """ table\_audit\_table = boto3.resource('dynamodb').Table(os.environ['TABLE\_AUDIT\_TABLE'])

# Get the last successful extraction timestamp for this table  
response = table\_audit\_table.get\_item(  
 Key={  
 'table\_id': table\_config['table\_id']  
 }  
)  
  
last\_extraction\_timestamp = None  
if 'Item' in response and 'last\_extraction\_timestamp' in response['Item']:  
 last\_extraction\_timestamp = response['Item']['last\_extraction\_timestamp']  
  
# Prepare SQL query  
cursor = conn.cursor()  
  
if last\_extraction\_timestamp and table\_config.get('incremental\_column'):  
 # Incremental extraction  
 query = f"""  
 SELECT \* FROM {table\_config['source\_schema']}.{table\_config['source\_table']}  
 WHERE {table\_config['incremental\_column']} > '{last\_extraction\_timestamp}'  
 """  
else:  
 # Full extraction  
 query = f"""  
 SELECT \* FROM {table\_config['source\_schema']}.{table\_config['source\_table']}  
 """  
  
# Execute the query and fetch data  
cursor.execute(query)  
  
# Get column names  
col\_names = [desc[0] for desc in cursor.description]  
  
# Fetch all rows  
rows = cursor.fetchall()  
  
# Create a pandas DataFrame  
df = pd.DataFrame(rows, columns=col\_names)  
  
# Count records  
record\_count = len(df)  
  
# Update processing metrics table  
metrics\_table = boto3.resource('dynamodb').Table(os.environ['PROCESSING\_METRICS\_TABLE'])  
metrics\_table.put\_item(  
 Item={  
 'job\_id': job\_id,  
 'table\_id': table\_config['table\_id'],  
 'layer': 'source',  
 'record\_count': record\_count,  
 'timestamp': datetime.datetime.now().isoformat()  
 }  
)  
  
# Close cursor  
cursor.close()  
  
return df, record\_count

def main(): # Get job parameters args = getResolvedOptions(sys.argv, ['JOB\_NAME', 'job\_id', 'table\_id']) job\_id = args['job\_id'] table\_id = args['table\_id']

# Initialize Spark context  
sc = SparkContext()  
glueContext = GlueContext(sc)  
spark = glueContext.spark\_session  
job = Job(glueContext)  
job.init(args['JOB\_NAME'], args)  
  
# Get table configuration  
dynamodb = boto3.resource('dynamodb')  
config\_table = dynamodb.Table(os.environ['CONFIG\_TABLE'])  
  
response = config\_table.get\_item(  
 Key={  
 'table\_id': table\_id  
 }  
)  
  
table\_config = response['Item']  
  
# Get database credentials from AWS Secrets Manager  
secrets\_client = boto3.client('secretsmanager')  
postgres\_secret = secrets\_client.get\_secret\_value(  
 SecretId=os.environ['POSTGRES\_SECRET\_NAME']  
)  
  
postgres\_credentials = json.loads(postgres\_secret['SecretString'])  
  
# Connect to PostgreSQL  
conn = get\_postgres\_connection(  
 postgres\_credentials['host'],  
 postgres\_credentials['port'],  
 table\_config['source\_database'],  
 postgres\_credentials['username'],  
 postgres\_credentials['password']  
)  
  
# Extract data  
df, record\_count = extract\_data(conn, table\_config, job\_id)  
  
# Close the connection  
conn.close()  
  
# Convert DataFrame to Spark DataFrame  
spark\_df = spark.createDataFrame(df)  
  
# Write to S3 in CSV format  
s3\_output\_path = f"{table\_config['target\_s3\_raw\_path']}{job\_id}/"  
  
spark\_df.write.format("csv") \  
 .option("header", "true") \  
 .option("delimiter", ",") \  
 .mode("overwrite") \  
 .save(s3\_output\_path)  
  
# Update processing metrics table for raw layer  
metrics\_table = boto3.resource('dynamodb').Table(os.environ['PROCESSING\_METRICS\_TABLE'])  
metrics\_table.put\_item(  
 Item={  
 'job\_id': job\_id,  
 'table\_id': table\_id,  
 'layer': 'raw',  
 'record\_count': record\_count,  
 'timestamp': datetime.datetime.now().isoformat()  
 }  
)  
  
job.commit()

if **name** == "**main**": main()

# **5. Validation Lambda Function**

def validate\_extract(event, context): """ Lambda function to validate extract counts between source and target """ job\_id = event['job\_id'] table\_id = event['table\_id'] layer = event['layer']

# Initialize DynamoDB client  
dynamodb = boto3.resource('dynamodb')  
metrics\_table = dynamodb.Table(os.environ['PROCESSING\_METRICS\_TABLE'])  
  
# Get source record count  
source\_response = metrics\_table.get\_item(  
 Key={  
 'job\_id': job\_id,  
 'table\_id': table\_id,  
 'layer': 'source'  
 }  
)  
  
# Get target record count  
target\_response = metrics\_table.get\_item(  
 Key={  
 'job\_id': job\_id,  
 'table\_id': table\_id,  
 'layer': layer  
 }  
)  
  
source\_count = source\_response['Item']['record\_count']  
target\_count = target\_response['Item']['record\_count']  
  
# Compare counts  
if source\_count != target\_count:  
 raise Exception(f"Validation failed for {layer} layer. Source count: {source\_count}, Target count: {target\_count}")  
  
return {  
 'job\_id': job\_id,  
 'table\_id': table\_id,  
 'validation': 'SUCCESS',  
 'source\_count': source\_count,  
 'target\_count': target\_count  
}

# **6. AWS Glue Job - Transform to Refined**

import sys from awsglue.transforms import \* from awsglue.utils import getResolvedOptions from pyspark.context import SparkContext from awsglue.context import GlueContext from awsglue.job import Job import boto3 import datetime

def main(): # Get job parameters args = getResolvedOptions(sys.argv, ['JOB\_NAME', 'job\_id', 'table\_id']) job\_id = args['job\_id'] table\_id = args['table\_id']

# Initialize Spark context  
sc = SparkContext()  
glueContext = GlueContext(sc)  
spark = glueContext.spark\_session  
job = Job(glueContext)  
job.init(args['JOB\_NAME'], args)  
  
# Get table configuration  
dynamodb = boto3.resource('dynamodb')  
config\_table = dynamodb.Table(os.environ['CONFIG\_TABLE'])  
  
response = config\_table.get\_item(  
 Key={  
 'table\_id': table\_id  
 }  
)  
  
table\_config = response['Item']  
  
# Read data from S3 raw layer  
s3\_input\_path = f"{table\_config['target\_s3\_raw\_path']}{job\_id}/"  
s3\_output\_path = f"{table\_config['target\_s3\_refined\_path']}{job\_id}/"  
  
# Read CSV data  
df = spark.read.format("csv") \  
 .option("header", "true") \  
 .option("inferSchema", "true") \  
 .load(s3\_input\_path)  
  
# Transform data (add any necessary transformations here)  
# For example, we might add a timestamp column  
from pyspark.sql.functions import current\_timestamp, lit  
  
df = df.withColumn("etl\_timestamp", current\_timestamp())  
df = df.withColumn("etl\_job\_id", lit(job\_id))  
  
# Write to S3 in Parquet format  
df.write.format("parquet") \  
 .mode("overwrite") \  
 .save(s3\_output\_path)  
  
# Get record count  
record\_count = df.count()  
  
# Update processing metrics table  
metrics\_table = boto3.resource('dynamodb').Table(os.environ['PROCESSING\_METRICS\_TABLE'])  
metrics\_table.put\_item(  
 Item={  
 'job\_id': job\_id,  
 'table\_id': table\_id,  
 'layer': 'refined',  
 'record\_count': record\_count,  
 'timestamp': datetime.datetime.now().isoformat()  
 }  
)  
  
job.commit()

if **name** == "**main**": main()

# **7. AWS Glue Job - Load to Snowflake**

import sys from awsglue.transforms import \* from awsglue.utils import getResolvedOptions from pyspark.context import SparkContext from awsglue.context import GlueContext from awsglue.job import Job import boto3 import snowflake.connector import datetime

def main(): # Get job parameters args = getResolvedOptions(sys.argv, ['JOB\_NAME', 'job\_id', 'table\_id']) job\_id = args['job\_id'] table\_id = args['table\_id']

# Initialize Spark context  
sc = SparkContext()  
glueContext = GlueContext(sc)  
spark = glueContext.spark\_session  
job = Job(glueContext)  
job.init(args['JOB\_NAME'], args)  
  
# Get table configuration  
dynamodb = boto3.resource('dynamodb')  
config\_table = dynamodb.Table(os.environ['CONFIG\_TABLE'])  
  
response = config\_table.get\_item(  
 Key={  
 'table\_id': table\_id  
 }  
)  
  
table\_config = response['Item']  
  
# Get Snowflake credentials from AWS Secrets Manager  
secrets\_client = boto3.client('secretsmanager')  
snowflake\_secret = secrets\_client.get\_secret\_value(  
 SecretId=os.environ['SNOWFLAKE\_SECRET\_NAME']  
)  
  
snowflake\_credentials = json.loads(snowflake\_secret['SecretString'])  
  
# Read Parquet data from S3 refined layer  
s3\_input\_path = f"{table\_config['target\_s3\_refined\_path']}{job\_id}/"  
  
df = spark.read.format("parquet").load(s3\_input\_path)  
  
# Set up Snowflake connection options  
sfOptions = {  
 "sfURL": snowflake\_credentials['url'],  
 "sfUser": snowflake\_credentials['username'],  
 "sfPassword": snowflake\_credentials['password'],  
 "sfDatabase": table\_config['target\_snowflake\_database'],  
 "sfSchema": table\_config['target\_snowflake\_schema'],  
 "sfWarehouse": snowflake\_credentials['warehouse'],  
 "sfRole": snowflake\_credentials['role']  
}  
  
# Write to Snowflake  
df.write \  
 .format("snowflake") \  
 .options(\*\*sfOptions) \  
 .option("dbtable", table\_config['target\_snowflake\_table']) \  
 .mode("append") \  
 .save()  
  
# Connect to Snowflake to get count  
conn = snowflake.connector.connect(  
 user=snowflake\_credentials['username'],  
 password=snowflake\_credentials['password'],  
 account=snowflake\_credentials['account'],  
 warehouse=snowflake\_credentials['warehouse'],  
 database=table\_config['target\_snowflake\_database'],  
 schema=table\_config['target\_snowflake\_schema']  
)  
  
cursor = conn.cursor()  
cursor.execute(f"SELECT COUNT(\*) FROM {table\_config['target\_snowflake\_table']} WHERE etl\_job\_id = '{job\_id}'")  
record\_count = cursor.fetchone()[0]  
cursor.close()  
conn.close()  
  
# Update processing metrics table  
metrics\_table = boto3.resource('dynamodb').Table(os.environ['PROCESSING\_METRICS\_TABLE'])  
metrics\_table.put\_item(  
 Item={  
 'job\_id': job\_id,  
 'table\_id': table\_id,  
 'layer': 'snowflake',  
 'record\_count': record\_count,  
 'timestamp': datetime.datetime.now().isoformat()  
 }  
)  
  
job.commit()

if **name** == "**main**": main()

# **8. Handle Failure Lambda Function**

def handle\_failure(event, context): """ Lambda function to handle failures in the ETL process """ job\_id = event['job\_id'] table\_id = event['table\_id'] error = event['error']

# Update job status  
dynamodb = boto3.resource('dynamodb')  
job\_status\_table = dynamodb.Table(os.environ['JOB\_STATUS\_TABLE'])  
  
response = job\_status\_table.update\_item(  
 Key={'job\_id': job\_id},  
 UpdateExpression="set job\_status = :s, error\_message = :e",  
 ExpressionAttributeValues={  
 ':s': 'FAILED',  
 ':e': error  
 },  
 ReturnValues="UPDATED\_NEW"  
)  
  
# Send SNS notification  
sns\_client = boto3.client('sns')  
sns\_client.publish(  
 TopicArn=os.environ['NOTIFICATION\_TOPIC'],  
 Subject=f'ETL Job {job\_id} Failed',  
 Message=f'ETL job {job\_id} failed processing table {table\_id}. Error: {error}'  
)  
  
return {  
 'job\_id': job\_id,  
 'table\_id': table\_id,  
 'status': 'FAILED',  
 'error': error  
}

# **9. Cleanup Failed Data Lambda Function**

def cleanup\_failed\_data(event, context): """ Lambda function to clean up data from failed jobs """ job\_id = event['job\_id'] table\_id = event['table\_id']

# Get table configuration  
dynamodb = boto3.resource('dynamodb')  
config\_table = dynamodb.Table(os.environ['CONFIG\_TABLE'])  
  
response = config\_table.get\_item(  
 Key={  
 'table\_id': table\_id  
 }  
)  
  
table\_config = response['Item']  
  
# Initialize S3 client  
s3\_client = boto3.client('s3')  
  
# Delete data from S3 raw layer  
raw\_bucket = table\_config['target\_s3\_raw\_path'].split('/')[2]  
raw\_prefix = '/'.join(table\_config['target\_s3\_raw\_path'].split('/')[3:]) + job\_id + '/'  
  
response = s3\_client.list\_objects\_v2(  
 Bucket=raw\_bucket,  
 Prefix=raw\_prefix  
)  
  
if 'Contents' in response:  
 for obj in response['Contents']:  
 s3\_client.delete\_object(  
 Bucket=raw\_bucket,  
 Key=obj['Key']  
 )  
  
# Delete data from S3 refined layer  
refined\_bucket = table\_config['target\_s3\_refined\_path'].split('/')[2]  
refined\_prefix = '/'.join(table\_config['target\_s3\_refined\_path'].split('/')[3:]) + job\_id + '/'  
  
response = s3\_client.list\_objects\_v2(  
 Bucket=refined\_bucket,  
 Prefix=refined\_prefix  
)  
  
if 'Contents' in response:  
 for obj in response['Contents']:  
 s3\_client.delete\_object(  
 Bucket=refined\_bucket,  
 Key=obj['Key']  
 )  
  
# Delete data from Snowflake (if needed)  
secrets\_client = boto3.client('secretsmanager')  
snowflake\_secret = secrets\_client.get\_secret\_value(  
 SecretId=os.environ['SNOWFLAKE\_SECRET\_NAME']  
)  
  
snowflake\_credentials = json.loads(snowflake\_secret['SecretString'])