# **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'])

##################################################

# **core/config.py**

import os import boto3 import json import logging from datetime import datetime from typing import Dict, List, Optional, Any, Union

# **Configure logging**

logger = logging.getLogger() logger.setLevel(logging.INFO)

class ConfigManager: """ Class to manage configuration and DynamoDB operations """ def **init**(self): self.dynamodb = boto3.resource('dynamodb') self.config\_table = self.dynamodb.Table(os.environ['CONFIG\_TABLE']) self.job\_status\_table = self.dynamodb.Table(os.environ['JOB\_STATUS\_TABLE']) self.table\_audit\_table = self.dynamodb.Table(os.environ['TABLE\_AUDIT\_TABLE']) self.processing\_metrics\_table = self.dynamodb.Table(os.environ['PROCESSING\_METRICS\_TABLE']) self.job\_control\_table = self.dynamodb.Table(os.environ['JOB\_CONTROL\_TABLE'])

def get\_active\_tables(self) -> List[Dict[str, Any]]:  
 """Get all active tables that should be processed"""  
 logger.info("Retrieving active tables from configuration")  
 response = self.config\_table.scan(  
 FilterExpression='active = :active\_value',  
 ExpressionAttributeValues={  
 ':active\_value': True  
 }  
 )  
   
 active\_tables = response['Items']  
 logger.info(f"Found {len(active\_tables)} active tables")  
 return active\_tables  
  
def get\_table\_config(self, table\_id: str) -> Dict[str, Any]:  
 """Get configuration for a specific table"""  
 logger.info(f"Retrieving configuration for table {table\_id}")  
 response = self.config\_table.get\_item(  
 Key={  
 'table\_id': table\_id  
 }  
 )  
   
 if 'Item' not in response:  
 raise Exception(f"Table configuration not found for {table\_id}")  
   
 return response['Item']  
  
def get\_job\_control(self) -> Dict[str, Any]:  
 """Get job control configuration"""  
 logger.info("Retrieving job control configuration")  
 response = self.job\_control\_table.get\_item(  
 Key={  
 'control\_id': 'ETL\_CONTROL'  
 }  
 )  
   
 if 'Item' not in response:  
 logger.warning("Job control configuration not found, using defaults")  
 return {  
 'enabled': True,  
 'include\_tables': [],  
 'exclude\_tables': []  
 }  
   
 return response['Item']  
  
def create\_job(self, job\_type: str = 'NORMAL') -> str:  
 """Create a new ETL job entry"""  
 job\_id = f"ETL-{datetime.now().strftime('%Y-%m-%d-%H-%M-%S')}"  
   
 self.job\_status\_table.put\_item(  
 Item={  
 'job\_id': job\_id,  
 'job\_status': 'RUNNING',  
 'job\_type': job\_type,  
 'start\_time': datetime.now().isoformat(),  
 'end\_time': None,  
 'error\_message': None,  
 'created\_at': datetime.now().isoformat()  
 }  
 )  
   
 logger.info(f"Created new job {job\_id} of type {job\_type}")  
 return job\_id  
  
def update\_job\_status(self, job\_id: str, status: str, error\_message: Optional[str] = None) -> None:  
 """Update job status in DynamoDB"""  
 update\_expression = "set job\_status = :s"  
 expression\_values = {  
 ':s': status  
 }  
   
 if status in ['COMPLETED', 'FAILED']:  
 update\_expression += ", end\_time = :e"  
 expression\_values[':e'] = datetime.now().isoformat()  
   
 if error\_message:  
 update\_expression += ", error\_message = :em"  
 expression\_values[':em'] = error\_message  
   
 self.job\_status\_table.update\_item(  
 Key={'job\_id': job\_id},  
 UpdateExpression=update\_expression,  
 ExpressionAttributeValues=expression\_values  
 )  
   
 logger.info(f"Updated job {job\_id} status to {status}")  
  
def update\_metrics(self, job\_id: str, table\_id: str, layer: str, record\_count: int) -> None:  
 """Update processing metrics in DynamoDB"""  
 self.processing\_metrics\_table.put\_item(  
 Item={  
 'job\_id': job\_id,  
 'table\_id': table\_id,  
 'layer': layer,  
 'record\_count': record\_count,  
 'timestamp': datetime.now().isoformat()  
 }  
 )  
   
 logger.info(f"Updated metrics for job {job\_id}, table {table\_id}, layer {layer}: {record\_count} records")  
  
def update\_table\_audit(self, job\_id: str, table\_id: str, status: str,   
 extraction\_timestamp: Optional[str] = None) -> None:  
 """Update table audit information"""  
 item = {  
 'table\_id': table\_id,  
 'last\_job\_id': job\_id,  
 'last\_job\_status': status,  
 'last\_job\_timestamp': datetime.now().isoformat()  
 }  
   
 if extraction\_timestamp:  
 item['last\_extraction\_timestamp'] = extraction\_timestamp  
   
 self.table\_audit\_table.put\_item(Item=item)  
 logger.info(f"Updated audit for table {table\_id}, job {job\_id}, status {status}")  
  
def get\_metrics(self, job\_id: str, table\_id: str, layer: str) -> int:  
 """Get record count metrics for a specific layer"""  
 response = self.processing\_metrics\_table.get\_item(  
 Key={  
 'job\_id': job\_id,  
 'table\_id': table\_id,  
 'layer': layer  
 }  
 )  
   
 if 'Item' not in response:  
 raise Exception(f"Metrics not found for job {job\_id}, table {table\_id}, layer {layer}")  
   
 return response['Item']['record\_count']

# **core/secrets.py**

import boto3 import json import logging from typing import Dict, Any

logger = logging.getLogger()

class SecretsManager: """Class to handle retrieving secrets from AWS Secrets Manager"""

def \_\_init\_\_(self):  
 self.client = boto3.client('secretsmanager')  
 self.\_secrets\_cache = {}  
  
def get\_secret(self, secret\_name: str) -> Dict[str, Any]:  
 """Get secret from AWS Secrets Manager with caching"""  
 if secret\_name in self.\_secrets\_cache:  
 return self.\_secrets\_cache[secret\_name]  
   
 try:  
 logger.info(f"Retrieving secret {secret\_name}")  
 response = self.client.get\_secret\_value(SecretId=secret\_name)  
 secret = json.loads(response['SecretString'])  
 self.\_secrets\_cache[secret\_name] = secret  
 return secret  
 except Exception as e:  
 logger.error(f"Error retrieving secret {secret\_name}: {str(e)}")  
 raise

# **core/notifications.py**

import boto3 import logging from typing import Optional

logger = logging.getLogger()

class NotificationManager: """Class to handle SNS notifications"""

def \_\_init\_\_(self):  
 self.sns\_client = boto3.client('sns')  
 self.topic\_arn = None  
  
def set\_topic\_arn(self, topic\_arn: str) -> None:  
 """Set the SNS topic ARN for notifications"""  
 self.topic\_arn = topic\_arn  
  
def send\_notification(self, subject: str, message: str) -> None:  
 """Send an SNS notification"""  
 if not self.topic\_arn:  
 logger.warning("Cannot send notification: SNS topic ARN not set")  
 return  
   
 try:  
 logger.info(f"Sending notification: {subject}")  
 self.sns\_client.publish(  
 TopicArn=self.topic\_arn,  
 Subject=subject,  
 Message=message  
 )  
 except Exception as e:  
 logger.error(f"Failed to send notification: {str(e)}")

# **core/data\_cleaner.py**

import boto3 import logging import snowflake.connector import json import os from typing import Dict, Any from core.secrets import SecretsManager from core.config import ConfigManager

logger = logging.getLogger()

class DataCleaner: """Class to handle cleanup of data in case of failures or reprocessing"""

def \_\_init\_\_(self):  
 self.s3\_client = boto3.client('s3')  
 self.config\_manager = ConfigManager()  
 self.secrets\_manager = SecretsManager()  
  
def \_extract\_bucket\_and\_prefix(self, s3\_path: str, job\_id: str) -> tuple:  
 """Extract bucket name and prefix from S3 path"""  
 parts = s3\_path.replace('s3://', '').split('/')  
 bucket = parts[0]  
 prefix = '/'.join(parts[1:]) + job\_id + '/'  
 return bucket, prefix  
  
def cleanup\_s3\_data(self, table\_config: Dict[str, Any], job\_id: str) -> None:  
 """Clean up data from S3 for a specific job"""  
 # Clean raw layer  
 logger.info(f"Cleaning up raw S3 data for job {job\_id}")  
 raw\_bucket, raw\_prefix = self.\_extract\_bucket\_and\_prefix(  
 table\_config['target\_s3\_raw\_path'], job\_id)  
   
 self.\_delete\_s3\_objects(raw\_bucket, raw\_prefix)  
   
 # Clean refined layer  
 logger.info(f"Cleaning up refined S3 data for job {job\_id}")  
 refined\_bucket, refined\_prefix = self.\_extract\_bucket\_and\_prefix(  
 table\_config['target\_s3\_refined\_path'], job\_id)  
   
 self.\_delete\_s3\_objects(refined\_bucket, refined\_prefix)  
  
def \_delete\_s3\_objects(self, bucket: str, prefix: str) -> None:  
 """Delete objects from S3 with given prefix"""  
 try:  
 # List objects  
 response = self.s3\_client.list\_objects\_v2(  
 Bucket=bucket,  
 Prefix=prefix  
 )  
   
 if 'Contents' not in response:  
 logger.info(f"No objects found in s3://{bucket}/{prefix}")  
 return  
   
 # Delete objects  
 objects = [{'Key': obj['Key']} for obj in response['Contents']]  
 self.s3\_client.delete\_objects(  
 Bucket=bucket,  
 Delete={'Objects': objects}  
 )  
   
 logger.info(f"Deleted {len(objects)} objects from s3://{bucket}/{prefix}")  
 except Exception as e:  
 logger.error(f"Error deleting S3 objects: {str(e)}")  
 raise  
  
def cleanup\_snowflake\_data(self, table\_config: Dict[str, Any], job\_id: str) -> None:  
 """Clean up data from Snowflake for a specific job"""  
 logger.info(f"Cleaning up Snowflake data for job {job\_id}")  
   
 try:  
 # Get Snowflake credentials  
 snowflake\_secret = self.secrets\_manager.get\_secret(  
 os.environ['SNOWFLAKE\_SECRET\_NAME'])  
   
 # Connect to Snowflake  
 conn = snowflake.connector.connect(  
 user=snowflake\_secret['username'],  
 password=snowflake\_secret['password'],  
 account=snowflake\_secret['account'],  
 warehouse=snowflake\_secret['warehouse'],  
 database=table\_config['target\_snowflake\_database'],  
 schema=table\_config['target\_snowflake\_schema']  
 )  
   
 cursor = conn.cursor()  
   
 # Delete data for this job  
 delete\_query = f"""  
 DELETE FROM {table\_config['target\_snowflake\_table']}   
 WHERE etl\_job\_id = '{job\_id}'  
 """  
   
 cursor.execute(delete\_query)  
   
 rows\_deleted = cursor.rowcount  
 conn.commit()  
   
 logger.info(f"Deleted {rows\_deleted} rows from Snowflake table {table\_config['target\_snowflake\_table']}")  
   
 cursor.close()  
 conn.close()  
 except Exception as e:  
 logger.error(f"Error cleaning up Snowflake data: {str(e)}")  
 raise  
  
def cleanup\_all(self, table\_id: str, job\_id: str) -> None:  
 """Clean up all data for a specific job and table"""  
 table\_config = self.config\_manager.get\_table\_config(table\_id)  
   
 # Cleanup S3  
 self.cleanup\_s3\_data(table\_config, job\_id)  
   
 # Cleanup Snowflake   
 self.cleanup\_snowflake\_data(table\_config, job\_id)  
   
 logger.info(f"Completed cleanup for job {job\_id}, table {table\_id}")

# **lambda\_functions/etl\_trigger.py**

import os import json import boto3 import logging from datetime import datetime from core.config import ConfigManager from core.notifications import NotificationManager

# **Configure logging**

logger = logging.getLogger() logger.setLevel(logging.INFO)

def lambda\_handler(event, context): """ Entry point Lambda function that validates if a new ETL job can be started """ try: logger.info("ETL trigger initiated")

# Initialize managers  
 config\_manager = ConfigManager()  
 notification\_manager = NotificationManager()  
 notification\_manager.set\_topic\_arn(os.environ['NOTIFICATION\_TOPIC'])  
   
 # Check if job control is enabled  
 job\_control = config\_manager.get\_job\_control()  
 if not job\_control.get('enabled', True):  
 logger.info("ETL jobs are disabled in job control configuration")  
 return {  
 'statusCode': 200,  
 'body': json.dumps('ETL jobs are currently disabled'),  
 'should\_continue': False  
 }  
   
 # Check for reprocess flag in the event  
 is\_reprocess = event.get('reprocess', False)  
 reprocess\_job\_id = event.get('job\_id') if is\_reprocess else None  
   
 if not is\_reprocess:  
 # Check if there's any job currently running  
 dynamodb = boto3.resource('dynamodb')  
 job\_status\_table = dynamodb.Table(os.environ['JOB\_STATUS\_TABLE'])  
   
 response = job\_status\_table.scan(  
 FilterExpression='job\_status = :status',  
 ExpressionAttributeValues={':status': 'RUNNING'}  
 )  
   
 if response['Items']:  
 logger.info("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'] and not is\_reprocess:  
 logger.warning("Previous job failed. Manual intervention required.")  
 notification\_manager.send\_notification(  
 'ETL Job Needs Attention',  
 '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 = reprocess\_job\_id if is\_reprocess else config\_manager.create\_job(  
 'REPROCESS' if is\_reprocess else 'NORMAL'  
 )  
   
 # Trigger Step Function execution  
 step\_function\_client = boto3.client('stepfunctions')  
 step\_function\_client.start\_execution(  
 stateMachineArn=os.environ['STEP\_FUNCTION\_ARN'],  
 name=job\_id.replace('-', ''), # Remove hyphens for step function name  
 input=json.dumps({  
 'job\_id': job\_id,  
 'reprocess': is\_reprocess,  
 'include\_tables': job\_control.get('include\_tables', []),  
 'exclude\_tables': job\_control.get('exclude\_tables', [])  
 })  
 )  
   
 logger.info(f"ETL job {job\_id} started successfully")  
   
 # If this is a reprocess job, send notification  
 if is\_reprocess:  
 notification\_manager.send\_notification(  
 f"ETL Reprocess Job Started: {job\_id}",  
 f"Reprocessing job {job\_id} has been initiated."  
 )  
   
 return {  
 'statusCode': 200,  
 'body': json.dumps(f'ETL job {job\_id} started successfully'),  
 'should\_continue': True,  
 'job\_id': job\_id  
 }  
  
except Exception as e:  
 logger.error(f"Error triggering ETL job: {str(e)}")  
 notification\_manager = NotificationManager()  
 notification\_manager.set\_topic\_arn(os.environ['NOTIFICATION\_TOPIC'])  
 notification\_manager.send\_notification(  
 'ETL Job Trigger Error',  
 f'Error occurred while triggering ETL job: {str(e)}'  
 )  
 raise

# **lambda\_functions/get\_active\_tables.py**

import os import json import boto3 import logging from core.config import ConfigManager

# **Configure logging**

logger = logging.getLogger() logger.setLevel(logging.INFO)

def lambda\_handler(event, context): """ Lambda function to get all active tables for ETL processing with inclusion/exclusion filters """ try: logger.info(f"Processing active tables for job: {event.get('job\_id')}")

# Get job parameters  
 job\_id = event['job\_id']  
 reprocess = event.get('reprocess', False)  
 include\_tables = event.get('include\_tables', [])  
 exclude\_tables = event.get('exclude\_tables', [])  
   
 # Initialize configuration manager  
 config\_manager = ConfigManager()  
   
 # Get all active tables  
 active\_tables = config\_manager.get\_active\_tables()  
   
 # Filter tables based on include/exclude lists  
 filtered\_tables = []  
 for table in active\_tables:  
 table\_id = table['table\_id']  
   
 # Skip if table is in exclude list  
 if table\_id in exclude\_tables:  
 logger.info(f"Skipping table {table\_id} (in exclude list)")  
 continue  
   
 # Skip if include list is provided and table is not in it  
 if include\_tables and table\_id not in include\_tables:  
 logger.info(f"Skipping table {table\_id} (not in include list)")  
 continue  
   
 filtered\_tables.append(table)  
   
 logger.info(f"Job {job\_id} will process {len(filtered\_tables)} tables")  
   
 # Return job\_id and tables for further processing  
 return {  
 'job\_id': job\_id,  
 'tables': filtered\_tables,  
 'reprocess': reprocess  
 }  
  
except Exception as e:  
 logger.error(f"Error getting active tables: {str(e)}")  
 raise

# **glue\_jobs/extract\_to\_raw.py**

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 json import os import datetime import logging from typing import Dict, Any, Tuple

# **Add custom modules to path**

sys.path.append('.')

# **Import custom modules**

try: from core.config import ConfigManager from core.secrets import SecretsManager from core.data\_cleaner import DataCleaner except ImportError: # Fall back implementation for Glue environment pass

# **Configure logging**

logger = logging.getLogger() logger.setLevel(logging.INFO) handler = logging.StreamHandler(sys.stdout) handler.setFormatter(logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')) logger.addHandler(handler)

def get\_postgres\_connection(hostname: str, port: int, database: str, username: str, password: str): """ Establish connection to PostgreSQL database """ try: conn = psycopg2.connect( host=hostname, port=port, database=database, user=username, password=password ) return conn except Exception as e: logger.error(f"Error connecting to PostgreSQL: {str(e)}") raise

def extract\_data(conn, table\_config: Dict[str, Any], job\_id: str, config\_manager) -> Tuple[pd.DataFrame, int]: """ Extract data from PostgreSQL table based on last extraction timestamp """ try: table\_id = table\_config['table\_id'] logger.info(f"Extracting data for table {table\_id}")

# Get the last successful extraction timestamp for this table  
 dynamodb = boto3.resource('dynamodb')  
 table\_audit\_table = dynamodb.Table(os.environ['TABLE\_AUDIT\_TABLE'])  
   
 response = table\_audit\_table.get\_item(  
 Key={  
 'table\_id': 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']  
 logger.info(f"Last extraction timestamp: {last\_extraction\_timestamp}")  
   
 # Prepare SQL query  
 cursor = conn.cursor()  
   
 if last\_extraction\_timestamp and table\_config.get('incremental\_column'):  
 # Incremental extraction  
 logger.info(f"Performing incremental extraction using column {table\_config['incremental\_column']}")  
 query = f"""  
 SELECT \* FROM {table\_config['source\_schema']}.{table\_config['source\_table']}  
 WHERE {table\_config['incremental\_column']} > '{last\_extraction\_timestamp}'  
 """  
 else:  
 # Full extraction  
 logger.info(f"Performing 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)  
 logger.info(f"Extracted {record\_count} records")  
   
 # Update processing metrics table  
 config\_manager.update\_metrics(job\_id, table\_id, 'source', record\_count)  
   
 # Get current timestamp for incremental loading  
 current\_timestamp = datetime.datetime.now().isoformat()  
   
 # Update table audit for extraction timestamp  
 if table\_config.get('incremental\_column') and record\_count > 0:  
 config\_manager.update\_table\_audit(  
 job\_id, table\_id, 'IN\_PROGRESS', current\_timestamp  
 )  
   
 # Close cursor  
 cursor.close()  
   
 return df, record\_count  
  
except Exception as e:  
 logger.error(f"Error extracting data: {str(e)}")  
 raise

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

logger.info(f"Starting extract to raw job for {table\_id}, job\_id: {job\_id}, reprocess: {reprocess}")  
  
try:  
 # Initialize Spark context  
 sc = SparkContext()  
 glueContext = GlueContext(sc)  
 spark = glueContext.spark\_session  
 job = Job(glueContext)  
 job.init(args['JOB\_NAME'], args)  
   
 # Initialize managers  
 config\_manager = ConfigManager()  
 secrets\_manager = SecretsManager()  
 data\_cleaner = DataCleaner()  
   
 # Get table configuration  
 table\_config = config\_manager.get\_table\_config(table\_id)  
   
 # If this is a reprocess job, clean up data first  
 if reprocess:  
 logger.info(f"Reprocessing job: cleaning up existing data for {job\_id}, table {table\_id}")  
 data\_cleaner.cleanup\_all(table\_id, job\_id)  
   
 # Get database credentials from AWS Secrets Manager  
 postgres\_secret = secrets\_manager.get\_secret(os.environ['POSTGRES\_SECRET\_NAME'])  
   
 # Connect to PostgreSQL  
 conn = get\_postgres\_connection(  
 postgres\_secret['host'],  
 postgres\_secret['port'],  
 table\_config['source\_database'],  
 postgres\_secret['username'],  
 postgres\_secret['password']  
 )  
   
 # Extract data  
 df, record\_count = extract\_data(conn, table\_config, job\_id, config\_manager)  
   
 # Close the connection  
 conn.close()  
   
 # Handle empty dataframe case  
 if record\_count == 0:  
 logger.info(f"No data to extract for table {table\_id}")  
 config\_manager.update\_metrics(job\_id, table\_id, 'raw', 0)  
 job.commit()  
 return  
   
 # Add metadata columns  
 from pyspark.sql.functions import current\_timestamp, lit  
   
 # Convert DataFrame to Spark DataFrame  
 spark\_df = spark.createDataFrame(df)  
   
 # Add audit columns  
 spark\_df = spark\_df.withColumn("etl\_job\_id", lit(job\_id))  
 spark\_df = spark\_df.withColumn("etl\_timestamp", current\_timestamp())  
   
 # Write to S3 in CSV format with job\_id in the path  
 s3\_output\_path = f"{table\_config['target\_s3\_raw\_path']}{job\_id}/"  
   
 logger.info(f"Writing {record\_count} records to S3 raw path: {s3\_output\_path}")  
   
 spark\_df.write.format("csv") \  
 .option("header", "true") \  
 .option("delimiter", ",") \  
 .mode("overwrite") \  
 .save(s3\_output\_path)  
   
 # Update processing metrics table for raw layer  
 config\_manager.update\_metrics(job\_id, table\_id, 'raw', record\_count)  
   
 logger.info(f"Successfully extracted data for table {table\_id} to raw layer")  
 job.commit()  
   
except Exception as e:  
 logger.error(f"Error in extract\_to\_raw job: {str(e)}")  
 raise

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

# **glue\_jobs/transform\_to\_refined.py**

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 import logging from typing import Dict, Any

# **Add custom modules to path**

sys.path.append('.')

# **Import custom modules**

try: from core.config import ConfigManager from core.data\_cleaner import DataCleaner except ImportError: # Fall back implementation for Glue environment pass

# **Configure logging**

logger = logging.getLogger() logger.setLevel(logging.INFO) handler = logging.StreamHandler(sys.stdout) handler.setFormatter(logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')) logger.addHandler(handler)

def transform\_data(df, table\_config: Dict[str, Any], job\_id: str): """ Apply transformations to the data based on table type and requirements """ from pyspark.sql.functions import col, when, current\_timestamp, lit

logger.info(f"Applying transformations for table {table\_config['table\_id']}")  
  
# Add audit columns if they don't exist  
if "etl\_job\_id" not in df.columns:  
 df = df.withColumn("etl\_job\_id", lit(job\_id))  
  
if "etl\_timestamp" not in df.columns:  
 df = df.withColumn("etl\_timestamp", current\_timestamp())  
  
# Table-specific transformations based on table type  
table\_type = table\_config.get('table\_type', 'TRANSACTIONAL')  
  
if table\_type == 'TRANSACTIONAL':  
 # Transformations for transactional tables  
 logger.info("Applying transactional table transformations")  
   
 # Validate numeric fields if needed  
 if 'amount' in df.columns:  
 df = df.withColumn("amount", col("amount").cast("decimal(18,2)"))  
   
 # Add payment status validation if needed  
 if 'payment\_status' in df.columns:  
 df = df.withColumn("payment\_status\_valid",   
 when(col("payment\_status").isin("COMPLETED", "PENDING", "FAILED", "REFUNDED"), True)  
 .otherwise(False))  
  
elif table\_type == 'REFERENCE':  
 # Transformations for reference tables  
 logger.info("Applying reference table transformations")  
   
 # Keep reference data as is, typically no transformations needed  
 pass  
  
# Add data quality validation columns if needed  
# ...  
  
return df

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

logger.info(f"Starting transform to refined job for {table\_id}, job\_id: {job\_id}, reprocess: {reprocess}")  
  
try:  
 # Initialize Spark context  
 sc = SparkContext()  
 glueContext = GlueContext(sc)  
 spark = glueContext.spark\_session  
 job = Job(glueContext)  
 job.init(args['JOB\_NAME'], args)  
   
 # Initialize managers  
 config\_manager = ConfigManager()  
   
 # Get table configuration  
 table\_config = config\_manager.get\_table\_config(table\_id)  
   
 # 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}/"  
   
 # Check if raw data exists