### **Data pipeline prototype**

### **ETL Pipeline Steps:**

1. **Extract**: Data is extracted from a source, which could be an API, a file, or a database.
2. **Transform**: Data is cleaned, formatted, or aggregated to meet your requirements.
3. **Load**: The transformed data is loaded into a cloud data warehouse (e.g., Amazon Redshift or Athena).

### **Architecture Components:**

1. **AWS S3** — For storing raw data and transformed data.
2. **AWS Lambda** — For extracting and transforming data in real-time or batch mode.
3. **AWS Glue** — For performing more complex transformations and loading data into Redshift or another data warehouse.
4. **Amazon Redshift (or other data warehouse)** — For storing the final structured data.

### **Step 1: Data Extraction**

Let’s assume we are extracting data from an S3 bucket or an API. We can use AWS Lambda to do this.

#### **Lambda Function: Extracting Data from an API**

1. **Lambda Python Script**: Libraries Require are requests, boto3

* **Lambda Code to Extract Data**:

| import json import requests import boto3 from datetime import datetime  # Set up S3 client s3\_client = boto3.client('s3')  # Define the S3 bucket where the raw data will be stored bucket\_name = "your-s3-bucket-name"  def lambda\_handler(event, context):  # Example API URL to fetch data (replace with your actual API endpoint)  api\_url = "https://api.example.com/data"    # Make an API request  response = requests.get(api\_url)  data = response.json() # Assuming the API returns JSON    # Generate a unique file name with a timestamp  timestamp = datetime.now().strftime('%Y-%m-%d\_%H-%M-%S')  file\_name = f"raw\_data\_{timestamp}.json"    # Save data to S3  s3\_client.put\_object(  Bucket=bucket\_name,  Key=file\_name,  Body=json.dumps(data)  )    return {  'statusCode': 200,  'body': json.dumps('Data extracted and stored successfully')  } |
| --- |

* **Deploy this Lambda function** in AWS and set it to trigger periodically (e.g., every 3 hour) using an **Amazon CloudWatch Events** rule.

### **Step 2: Data Transformation**

In this step, you can use AWS Glue or an additional Lambda function to transform the data.

#### **AWS Glue Job for Data Transformation**

1. **AWS Glue Setup**:

* Create a Glue Crawler to catalog the data stored in the S3 bucket (this will make it easier to query).
* Create a Glue Job (Python script) to transform the data.

*Glue Scripts for transforming data into s3*

| import boto3 import pandas as pd from pyspark.sql import SparkSession from pyspark.sql.functions import col import json  # Initialize the Spark session (Glue provides this automatically in the environment) spark = SparkSession.builder.appName("GlueETL").getOrCreate()  # Set up S3 client (optional, in case you want to interact directly with S3) s3\_client = boto3.client('s3') bucket\_name = "Bucket-name"  # Define paths for input and output data in S3 input\_path = "s3://{}/raw\_data/".format(bucket\_name) output\_path = "s3://{}/transformed\_data/".format(bucket\_name)  def glue\_job():  # Read the raw data from S3 into a PySpark DataFrame  # Glue automatically infers the schema of JSON files, so no need to define it explicitly  raw\_data\_df = spark.read.json(input\_path)    # Perform transformations using PySpark DataFrame operations  # Example: Drop rows with missing values in any column  transformed\_df = raw\_data\_df.dropna()   # Example: Add a new column (e.g., 'processed\_timestamp' with current timestamp)  from pyspark.sql.functions import current\_timestamp  transformed\_df = transformed\_df.withColumn("processed\_timestamp", current\_timestamp())   # Write the transformed DataFrame back to S3 (as JSON)  transformed\_df.write.json(output\_path)   # Optionally: You can also save it as other formats such as CSV, Parquet, or Avro  # transformed\_df.write.parquet("s3://your-s3-bucket-name/transformed\_data\_parquet/")   return {  'statusCode': 200,  'body': 'Transformation and data writing to S3 completed successfully'  }  # Run the glue job (in AWS Glue, this would run within the Glue job environment) if \_\_name\_\_ == "\_\_main\_\_":  glue\_job() |
| --- |

**Note**: The code above is using Glue’s Apache Spark environment. we can also use Python for lighter transformations.

### **Step 3: Data Loading**

Once the data is transformed, it needs to be loaded into a data warehouse. For simplicity, let’s assume you’re loading it into **Amazon Redshift**.

#### **Lambda Function for Loading Data into Redshift**

1. **Create an IAM Role** that allows your Lambda function to write to Redshift.
2. **Lambda Code to Load Data into Redshift**:

| import psycopg2 import boto3  # Redshift connection details REDSHIFT\_HOST = "XXX.redshift.amazonaws.com" REDSHIFT\_PORT = 5439 REDSHIFT\_USER = "user" REDSHIFT\_PASSWORD = "password" REDSHIFT\_DB = "db-name" REDSHIFT\_TABLE = "table"  # Set up the S3 client to retrieve the transformed data s3\_client = boto3.client('s3') bucket\_name = "bucket-name" file\_name = "transformed\_data.json"  def lambda\_handler(event, context):  # Get the S3 object containing the transformed data  s3\_client.download\_file(bucket\_name, file\_name, '/tmp/transformed\_data.json')   # Connect to Redshift  conn = psycopg2.connect(  dbname=REDSHIFT\_DB,  user=REDSHIFT\_USER,  password=REDSHIFT\_PASSWORD,  host=REDSHIFT\_HOST,  port=REDSHIFT\_PORT  )    cursor = conn.cursor()    # Define the SQL COPY command to load data into Redshift  copy\_sql = f"""  COPY {REDSHIFT\_TABLE}  FROM 's3://{bucket\_name}/{file\_name}'  IAM\_ROLE 'arn:aws:iam::your-account-id:role/your-redshift-role'  JSON 'auto';  """    # Execute the COPY command  cursor.execute(copy\_sql)    # Commit and close the connection  conn.commit()  cursor.close()  conn.close()   return {  'statusCode': 200,  'body': 'Data loaded into Redshift successfully'  } |
| --- |

### **Step 4: Automating and Orchestrating**

* **AWS Lambda** can be triggered by events like new data landing in S3 (via S3 Event Notifications).
* **AWS Glue** jobs can be scheduled to run periodically or triggered by an S3 event , And can be scheduled using Step Functions based on how Glue should execute
* **Amazon CloudWatch logs** can be used to monitor the execution and failure of Lambda functions or Glue jobs.
* Airflow for orchestration as well in terms of heavy data processing or multi cloud and various on prem orchestration

### **Step 5: Monitoring and Error Handling**

1. **CloudWatch Logs**: Set up CloudWatch logs to capture Lambda function and Glue jobs outputs.
2. **Error Handling**: You should have error handling in your Lambda functions to retry or notify you in case of failure. For example, using AWS SNS for alerts.
3. AWS SNS notifications can be used for Glue job and Lambda failure