

# EShop Data Lake Project

## Sources

A python script running on local machine handles data ingestion from the following sources:

- **SQL Server:**  
Data is extracted from SQL Server tables using the `extract_from_sqlserver` function. This leverages the `pyodbc` library, which establishes a connection using a Windows-authenticated connection string. Queries such as `SELECT * FROM CustomerTable` can retrieve structured data for analysis.
- **CSV Files:**  
Raw transactional data, like purchase history, is read using the `extract_from_csv` function, which uses `pandas.read_csv`. This efficiently parses flat files and converts them into a DataFrame for processing.
- **JSON Files:**  
Hierarchical and semi-structured data is processed via the `extract_from_json` function. The `json.load` and `pandas.json_normalize` methods are employed to flatten nested JSON into a tabular format suitable for storage and querying or `upload_json_to_s3` method was used which sent the file to s3 directly without converting to csv.
- **Parquet files:**  
Read using `pandas.read_parquet`, which efficiently loads the columnar data into a DataFrame. Parquet files are optionally transformed (e.g., cleansing or type conversions) using the same methods as CSV or JSON data. The processed Parquet data can be uploaded by converting it into s3 for easier handling.

## Ingestion Tools

To process data from these sources python script was embedded in the AWS Lambda function, which was triggered when a file was uploaded to eshopdataraw s3 bucket. The file was then streamed through and converted into csv format for better data analytics and ease for Amazon Glue crawler to create tables. Then, the file was uploaded to another s3 bucket eshopdatadadb. The following tools and libraries are utilized:

1. **pyodbc:**  
Facilitates connection to SQL Server, allowing the script to execute queries and extract data efficiently. This is particularly useful for accessing relational databases in real-time.
2. **pandas:**  
Serves as the core data manipulation library. It parses CSV and JSON files into DataFrame structures, enabling seamless data transformation and cleansing. Its versatility allows for handling both tabular and hierarchical data formats.
3. **boto3:**  
AWS SDK for Python, `boto3`, is used to interact with Amazon S3. It allows uploading processed DataFrames as CSVs and JSON objects to the specified S3 bucket. Raw

data is categorized into folders (e.g., `eshopdataraw.csv` for CSV, `productdata.json` for JSON).

## **Analytics:**

**AWS Glue** was used for Transforming data further operations for analytics

- Cleansing raw data (removing nulls, standardizing formats).
- Transforming data (e.g., converting JSON to Parquet for efficient querying).
- Writing processed data back to S3 in a structured manner for analytics.

## **Querying Data with AWS Athena**

- Athena was used to perform serverless SQL querying directly on processed S3 data.
- It was used for
  - Joining and combining transaction logs with customer data.
  - Analysing clickstream data as a function of customer data.
  - Mapping clickstream to product data.

## **Amazon QuickSight**

We utilized Amazon QuickSight to create interactive dashboards that visualized query results directly from AWS Athena. These visual were designed to monitor performance of the ecommerce website eshop. Multiple tables were joined as two datasets. Then, on these datasets, analytics were performed visually, like which customers are spending the highest by mapping customer and purchase data.

## **Data Handling of Multiple formats:**

Using Pandas and pyodbc library, Json and CSV, data was processed as part of Lambda function script.

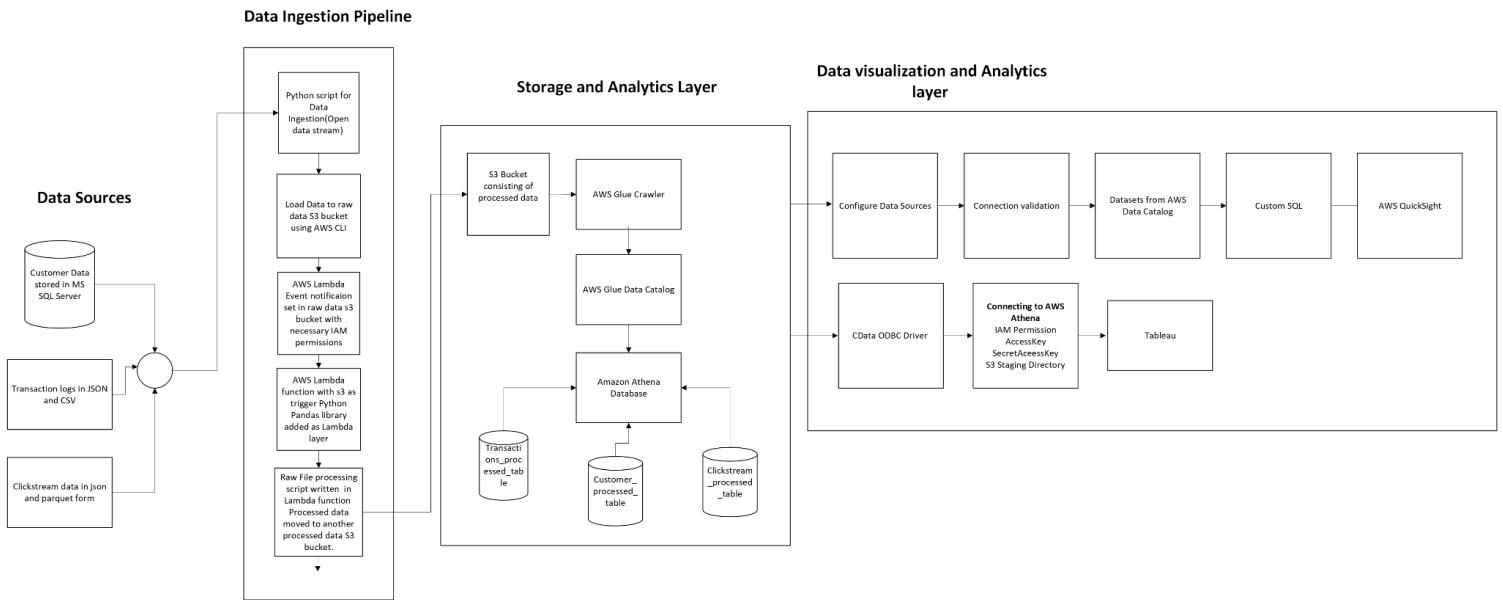
AWS Glue's schema registry and Apache Spark's(which is embedded in AWS Glue) flexibility will manage Parquet, CSV, JSON, and SQL data transformations.

## **Scalability:**

Amazon S3 is the data lake, offering virtually unlimited storage with high durability. Its ability to support multi-format datasets allowed me to scale data ingestion without the need for manual tasks. I optimized AWS Athena's performance by partitioning data in S3 and converting it into efficient formats like Parquet. This will improve execution times, even with large data. Glue Crawlers automat the schema detection and data cataloging process for datasets stored in

Amazon S3, for formats like JSON, CSV, and Parquet. This will make sure that as data sources grow in complexity and size, the metadata remained consistent and query-ready.

## Architecture diagram for Datalake:



### Task 1:

A python script was run on local machine to get data from following sources:

- Customer data from a MySQL database.
- Transaction data from a CSV file located on local machine.
- Product information from a JSON file on local machine.

The following functions were used to load data into Amazon S3 in the ETL pipeline:

`load_to_s3(data, s3_bucket, s3_key):`

This function was used to upload data to S3. The data argument is the data that needs to be stored in s3.

The function converts the DataFrame into a CSV string using `dataframe.to_csv(index=False)` and then uploads it to the specified S3 bucket and key (path) using `s3_client.put_object()`. The `s3_key` represents the file path and name in the S3 bucket, ensuring that data is organized properly within the appropriate folder (either raw or processed).

`upload_json_to_s3(local_file_path, s3_bucket, s3_key):`

This function was specifically designed for uploading JSON files to S3. It reads the content of a local JSON file (specified by `local_file_path`) using `open(local_file_path, 'r')`, then reads the entire file into a string.

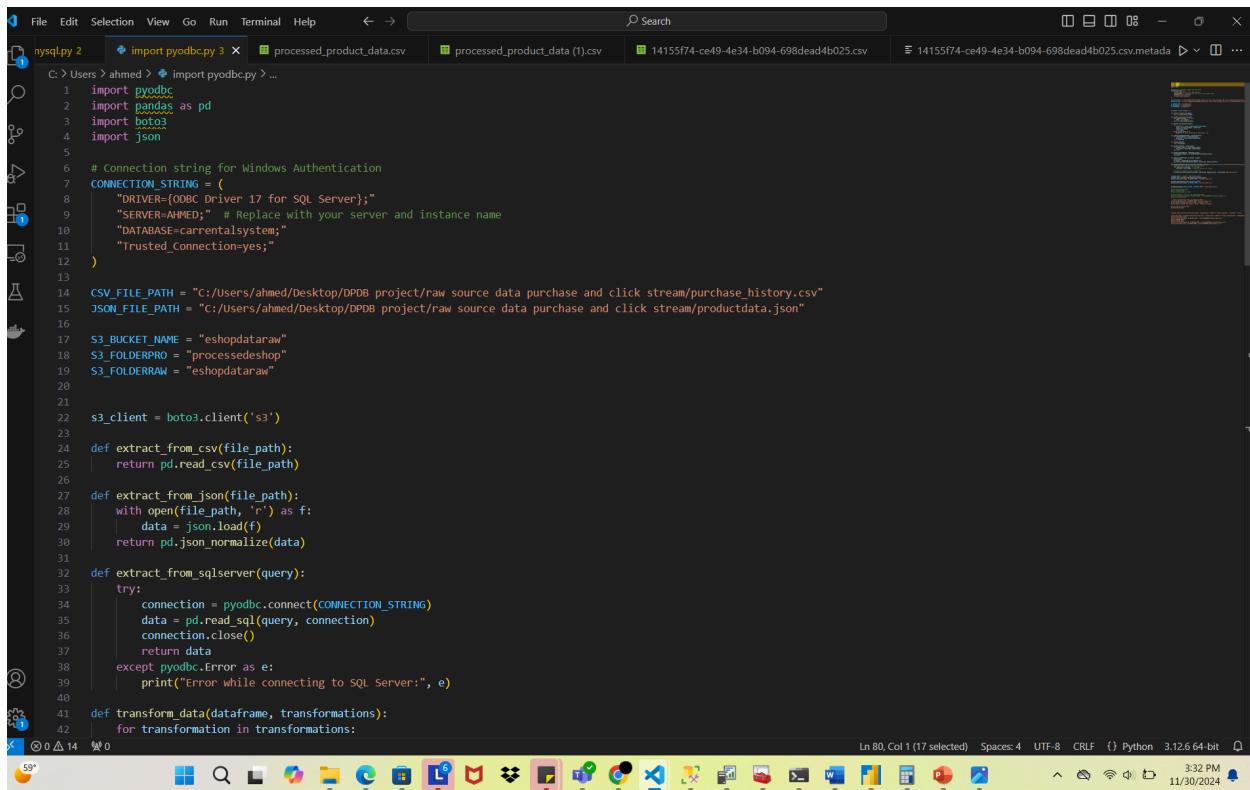
The content of the JSON file is then uploaded to the specified S3 bucket with the `s3_client.put_object()` method. The `ContentType='application/json'` is specified to ensure that the file is recognized as JSON. The `s3_key` specifies the path where the JSON file will be stored in the S3 bucket.

Raw Data S3 Bucket after loading data:

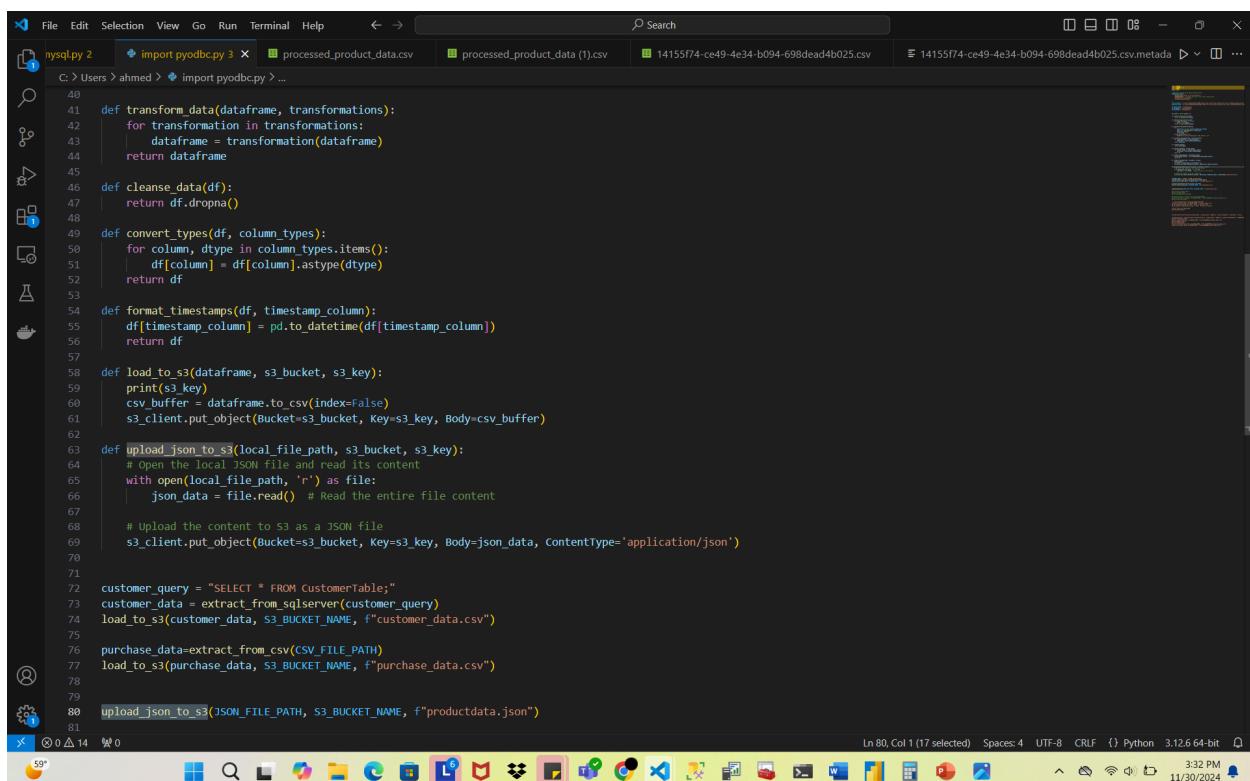
The screenshot shows the AWS S3 console interface. The left sidebar shows navigation options like Buckets, Storage Lens, and Feature spotlight. The main area displays the 'Objects' tab for the 'eshopdataraw' bucket. There are four objects listed:

Name	Type	Last modified	Size	Storage class
Athenaqueryresults/	Folder	-	-	-
customer_data.csv	csv	November 30, 2024, 01:56:02 (UTC-06:00)	603.0 B	Standard
productdata.json	json	November 30, 2024, 01:56:02 (UTC-06:00)	2.8 KB	Standard
purchase_data.csv	csv	November 30, 2024, 01:56:02 (UTC-06:00)	2.2 KB	Standard

## Python script to upload data from sources to S3 bucket.



```
mysql.py 2  import pyodbc.py 3  processed_product_data.csv  processed_product_data (1).csv  14155f74-ce49-4e34-b094-698dead4b025.csv  14155f74-ce49-4e34-b094-698dead4b025.csv.metadata
C:\> Users > ahmed >  import pyodbc.py > ...
1  import pyodbc
2  import pandas as pd
3  import boto3
4  import json
5
6  # Connection string for Windows Authentication
7  CONNECTION_STRING = (
8      "DRIVER={ODBC Driver 17 for SQL Server};"
9      "SERVER=AHMED;" # Replace with your server and instance name
10     "DATABASE=carrentalsystem;"
11     "Trusted_Connection=yes;"
12 )
13
14 CSV_FILE_PATH = "C:/Users/ahmed/Desktop/DPOB project/raw source data/purchase_and_click_stream/purchase_history.csv"
15 JSON_FILE_PATH = "C:/Users/ahmed/Desktop/DPOB project/raw source data/purchase_and_click_stream/productdata.json"
16
17 S3_BUCKET_NAME = "eshopdataraw"
18 S3_FOLDERPRO = "processedeshop"
19 S3_FOLDERRAW = "eshopdataraw"
20
21
22 s3_client = boto3.client('s3')
23
24 def extract_from_csv(file_path):
25     return pd.read_csv(file_path)
26
27 def extract_from_json(file_path):
28     with open(file_path, 'r') as f:
29         data = json.load(f)
30     return pd.json_normalize(data)
31
32 def extract_from_sqlserver(query):
33     try:
34         connection = pyodbc.connect(CONNECTION_STRING)
35         data = pd.read_sql(query, connection)
36         connection.close()
37         return data
38     except pyodbc.Error as e:
39         print("Error while connecting to SQL Server:", e)
40
41 def transform_data(dataframe, transformations):
42     for transformation in transformations:
43         dataframe = transformation(dataframe)
44
45
46 def cleanse_data(df):
47     return df.dropna()
48
49 def convert_types(df, column_types):
50     for column, dtype in column_types.items():
51         df[column] = df[column].astype(dtype)
52     return df
53
54 def format_timestamps(df, timestamp_column):
55     df[timestamp_column] = pd.to_datetime(df[timestamp_column])
56     return df
57
58 def load_to_s3(dataframe, s3_bucket, s3_key):
59     print(s3_key)
60     csv_buffer = dataframe.to_csv(index=False)
61     s3_client.put_object(Bucket=s3_bucket, Key=s3_key, Body=csv_buffer)
62
63 def upload_json_to_s3(local_file_path, s3_bucket, s3_key):
64     # Open the local JSON file and read its content
65     with open(local_file_path, 'r') as file:
66         json_data = file.read() # Read the entire file content
67
68     # Upload the content to S3 as a JSON file
69     s3_client.put_object(Bucket=s3_bucket, Key=s3_key, Body=json_data, ContentType='application/json')
70
71
72 customer_query = "SELECT * FROM CustomerTable;"
73 customer_data = extract_from_sqlserver(customer_query)
74 load_to_s3(customer_data, S3_BUCKET_NAME, f"customer_data.csv")
75
76 purchase_data=extract_from_csv(CSV_FILE_PATH)
77 load_to_s3(purchase_data, S3_BUCKET_NAME, f"purchase_data.csv")
78
79
80 upload_json_to_s3(JSON_FILE_PATH, S3_BUCKET_NAME, f"productdata.json")
81
```



```
mysql.py 2  import pyodbc.py 3  processed_product_data.csv  processed_product_data (1).csv  14155f74-ce49-4e34-b094-698dead4b025.csv  14155f74-ce49-4e34-b094-698dead4b025.csv.metadata
C:\> Users > ahmed >  import pyodbc.py > ...
40
41 def transform_data(dataframe, transformations):
42     for transformation in transformations:
43         dataframe = transformation(dataframe)
44     return dataframe
45
46 def cleanse_data(df):
47     return df.dropna()
48
49 def convert_types(df, column_types):
50     for column, dtype in column_types.items():
51         df[column] = df[column].astype(dtype)
52     return df
53
54 def format_timestamps(df, timestamp_column):
55     df[timestamp_column] = pd.to_datetime(df[timestamp_column])
56     return df
57
58 def load_to_s3(dataframe, s3_bucket, s3_key):
59     print(s3_key)
60     csv_buffer = dataframe.to_csv(index=False)
61     s3_client.put_object(Bucket=s3_bucket, Key=s3_key, Body=csv_buffer)
62
63 def upload_json_to_s3(local_file_path, s3_bucket, s3_key):
64     # Open the local JSON file and read its content
65     with open(local_file_path, 'r') as file:
66         json_data = file.read() # Read the entire file content
67
68     # Upload the content to S3 as a JSON file
69     s3_client.put_object(Bucket=s3_bucket, Key=s3_key, Body=json_data, ContentType='application/json')
70
71
72 customer_query = "SELECT * FROM CustomerTable;"
73 customer_data = extract_from_sqlserver(customer_query)
74 load_to_s3(customer_data, S3_BUCKET_NAME, f"customer_data.csv")
75
76 purchase_data=extract_from_csv(CSV_FILE_PATH)
77 load_to_s3(purchase_data, S3_BUCKET_NAME, f"purchase_data.csv")
78
79
80 upload_json_to_s3(JSON_FILE_PATH, S3_BUCKET_NAME, f"productdata.json")
81
```

## **Data Transformation and Loading:**

For both data transformation and data loading, we have used AWS Lambda Function. In the AWS lambda function, we have added layers to include libraries such as Pandas to process and clean the data.

### **Data Transformation with AWS Lambda:**

AWS Lambda was set up to automatically trigger whenever a file was uploaded to an S3 bucket, either a raw CSV or JSON file. Lambda would then process the data without the need for external servers or compute resources.

To transform the data, Python libraries, especially **Pandas**, were added as Lambda layers. This allowed the Lambda function to perform tasks like:

- **Data Cleansing:** Removing any rows with missing or null values.
- **Type Conversions:** Changing the data types of columns as needed (e.g., converting strings into integers for numerical columns).
- **Timestamp Formatting:** Standardizing date columns to a consistent timestamp format using the `pandas.to_datetime()` function.

### **Data Loading:**

Once the data was transformed inside Lambda, it was uploaded to a designated S3 bucket for processed data. The transformed files were stored in separate folders, making them ready for analysis. The raw data was placed in a different folder in the S3 bucket, keeping the original files for reference or reprocessing if needed.

### **Automation and Monitoring:**

The entire process was automated using **S3 event notifications**. Whenever a file was added to the raw data bucket, it would trigger the Lambda function to process the file and move it to the appropriate location in S3.

**Logging and monitoring** were set up with **Amazon CloudWatch**. This helped track errors or issues in the pipeline and allowed real-time visibility.

## Event Notification in rawdata se bucket for lambda function:

The screenshot shows the AWS CloudTrail data events configuration page for the 'eshopdata' bucket. The left sidebar lists various AWS services like S3 Buckets, Storage Lens, and Marketplace. The main panel has sections for 'Access' (no data events), 'Event notifications' (one entry for 'All object create events' to a 'Lambda function' named 'eshopdata-lambda'), 'Amazon EventBridge' (disabled), 'Transfer acceleration' (disabled), and 'Object Lock' (disabled). The bottom status bar shows the date as 11/30/2024.

Lambda function for data transformation and loading into s3 bucket processed was written in Python 3.13 runtime:

The screenshot shows the AWS Lambda function editor for the 'eshopdata-lambda' function. The code is written in Python 3.13 and processes S3 events to load JSON data into a CSV file. The code uses the boto3 library to interact with S3 and json to parse the JSON data. A tooltip for 'Amazon Q tip' suggests using JSON data as a list of records. The interface includes tabs for 'DEPLOY' (Ctrl+Shift+F1) and 'TEST' (Ctrl+Shift+F4).

```
import json
s3Client = boto3.client('s3')
def lambda_handler(event, context):
    bucket = event['Records'][0]['s3']['bucket']['name']
    key = event['Records'][0]['s3']['object']['key']
    print(f"Processing file: {key} from bucket: {bucket}")
    # Retrieve the object from S3
    response = s3Client.get_object(Bucket=bucket, Key=key)
    data = response['Body'].read().decode('utf-8')
    # Determine the new file name based on the trigger file
    if key == "processed_product_data.csv":
        new_key = "processed_product_data.csv"
        # Process JSON data (assuming JSON data is a list of records)
        json_data = json.loads(data)
        # Convert JSON to CSV
        output = io.StringIO()
        writer = csv.writer(output)
        # Write header (keys of the first item as column names)
        if len(json_data) > 0:
            header = json_data[0].keys()
            writer.writerow(header)
        # Write data rows
        for item in json_data:
            writer.writerow(item.values())
        # Write the CSV file
        with open(new_key, 'w') as f:
            f.write(output.getvalue())
    else:
        print(f"Skipping file: {key} from bucket: {bucket}")

```

Successfully updated the function eshopdata-lambda.

```
def lambda_handler(event, context):
    if key == "productdata.json":
        # Write header (keys of the first item as column names)
        if len(json_data) > 0:
            header = json_data[0].keys()
            writer.writerow(header)

        # Write data rows
        for item in json_data:
            writer.writerow(item.values())

        output.seek(0) # Reset pointer to start
        content = output.getvalue()

    elif key == "purchase_data.csv":
        new_key = "processed_purchase_data.csv"

        # Read CSV content
        csv_data = io.StringIO(data)
        reader = csv.reader(csv_data)

        # Write processed CSV content
        output = io.StringIO()
        writer = csv.writer(output)

        for row in reader:
            writer.writerow(row)

        output.seek(0) # Reset pointer to start
        content = output.getvalue()

    else:
        return {
            "statusCode": 400,
            "body": "Unsupported file type"
        }

    # Upload processed file to the new key (S3 bucket)
    s3Client.put_object(Bucket="eshopdataadpd", Key=new_key, Body=content)

print(f"Processed file uploaded to {bucket}/{new_key}")
```

CloudShell Feedback © 2024, Amazon Web Services, Inc. or its affiliates. Privacy Terms Cookie preferences 3:50 PM 11/30/2024

Successfully updated the function eshopdata-lambda.

```
def lambda_handler(event, context):
    elif key == "purchase_data.csv":
        content = output.getvalue()

    elif key == "customer_data.csv":
        new_key = "processed_customer_data.csv"

        # Read CSV content
        csv_data = io.StringIO(data)
        reader = csv.reader(csv_data)

        # Write processed CSV content
        output = io.StringIO()
        writer = csv.writer(output)

        for row in reader:
            writer.writerow(row)

        output.seek(0) # Reset pointer to start
        content = output.getvalue()

    else:
        return {
            "statusCode": 400,
            "body": "Unsupported file type"
        }

    # Upload processed file to the new key (S3 bucket)
    s3Client.put_object(Bucket="eshopdataadpd", Key=new_key, Body=content)

print(f"Processed file uploaded to {bucket}/{new_key}")
```

CloudShell Feedback © 2024, Amazon Web Services, Inc. or its affiliates. Privacy Terms Cookie preferences 3:50 PM 11/30/2024

Successfully updated the function eshopdata-lambda.

```
def lambda_handler(event, context):
    elif key == "customer_data.csv":
        writer = csv.writer(output)

        for row in reader:
            writer.writerow(row)

        output.seek(0) # Reset pointer to start
        content = output.getvalue()

    else:
        return {
            "statusCode": 400,
            "body": "Unsupported file type"
        }

    # Upload processed file to the new key (S3 bucket)
    s3Client.put_object(Bucket="eshopdataadpd", Key=new_key, Body=content)

print(f"Processed file uploaded to {bucket}/{new_key}")
```

CloudShell Feedback © 2024, Amazon Web Services, Inc. or its affiliates. Privacy Terms Cookie preferences 3:51 PM 11/30/2024

## Configuration and Runtime Settings for Lambda function:

The screenshot shows the 'Edit runtime settings' page for the 'eshopdata-lambda' function. It includes sections for Runtime (Python 3.13), Handler (lambda\_function.lambda\_handler), Architecture (x86\_64), and Connected layers (Pandasact). A table lists the connected layer details. At the bottom are 'Cancel' and 'Save' buttons.

Merge order	Name	Layer version	Compatible runtimes	Compatible architectures
1	Pandasact	1	python3.13	x86_64

The screenshot shows the 'Configure' tab of the 'eshopdata-lambda' function configuration. It displays triggers for S3 and Lambda. A red box highlights the trigger details for the S3 trigger, which is configured to respond to object creation events in the 'eshopdataraw' bucket.

**Triggers (1) Info**

- Trigger: S3: eshopdataraw
- Bucket arn: arn:aws:s3:::eshopdataraw
- Event type: s3:ObjectCreated:New
- No. notifications: 77 (144)
- Service principal: s3.amazonaws.com
- Source account: 296062583
- Statement ID: lambda-e93
- \_event\_per

The screenshot shows the AWS Lambda console with the URL <https://us-east-1.console.aws.amazon.com/lambda/home?region=us-east-1#/functions/eshopdata-lambda?subtab=permissions&tab=configure>. The left sidebar lists various AWS services like Environment variables, Tags, VPC, RDS databases, etc. The main area shows two resource-based policy statements:

Statement ID	Principal	Actions
lambda-c951a060-91...	s3.amazonaws.com	StringEquals,ArnLike lambda:InvokeFunction
296062582756_event...	s3.amazonaws.com	StringEquals,ArnLike lambda:InvokeFunction

Below the policy statements, there's an "Auditing and compliance" section mentioning AWS CloudTrail.

## Logs for AWS Lambda in Amazon Cloudwatch:

The screenshot shows the AWS CloudWatch Logs console with the URL [https://us-east-1.console.aws.amazon.com/cloudwatch/home?region=us-east-1#logsV2/log-groups/log-group:\\$252Faws\\$252Flambda\\$252Fshopdata-lambda\\$log-events/2024\\$252F11\\$252F30...](https://us-east-1.console.aws.amazon.com/cloudwatch/home?region=us-east-1#logsV2/log-groups/log-group:$252Faws$252Flambda$252Fshopdata-lambda$log-events/2024$252F11$252F30...). The left sidebar shows Log groups, Metrics, X-Ray traces, Events, Application Signals, Network Monitoring, and Insights. The main area displays log events for the eshopdata-lambda function, with a timestamp column and a message column showing log entries like INIT, START, and REPORT requests.

## Data Loaded onto Datalake:

The screenshot shows the AWS S3 console with the path `us-east-1.console.aws.amazon.com/s3/buckets/eshopdataadb?region=us-east-1&bucketType=general&prefix=products/&showversions=false`. The left sidebar shows the navigation menu for Amazon S3, including Buckets, Storage Lens, and Marketplace. The main content area displays the 'products/' folder with one object: `processed_product_data.csv`, which is a CSV file.

Name	Type	Last modified	Size	Storage class
<code>processed_product_data.csv</code>	csv	November 10, 2024, 02:50:14 (UTC-06:00)	1.1 KB	Standard

The screenshot shows the AWS S3 console with the path `us-east-1.console.aws.amazon.com/s3/buckets/eshopdataadb?region=us-east-1&bucketType=general&prefix=purchase/&showversions=false`. The left sidebar shows the navigation menu for Amazon S3. The main content area displays the 'purchase/' folder with one object: `processed_purchase_data.csv`, which is a CSV file.

Name	Type	Last modified	Size	Storage class
<code>processed_purchase_data.csv</code>	csv	November 10, 2024, 02:50:30 (UTC-06:00)	2.2 KB	Standard

The screenshot shows the AWS S3 console with the path `us-east-1.console.aws.amazon.com/s3/buckets/eshopdataadb?region=us-east-1&tab=objects`. The left sidebar shows the navigation menu for Amazon S3. The main content area displays the root bucket `eshopdataadb` with three objects/folders: `customer/`, `products/`, and `purchase/`.

Name	Type	Last modified	Size	Storage class
<code>customer/</code>	Folder	-	-	-
<code>products/</code>	Folder	-	-	-
<code>purchase/</code>	Folder	-	-	-

# Data Querying and Analysis

For data querying performing SQL-like queries, we have used Amazon Athena. We have used AWS Glue to create tables.

**AWS Glue Crawler Creation:** A Glue Crawler was created to scan the data in the processed data S3 bucket. The crawler was configured to automatically detect the structure of the data and create corresponding table definitions in the Glue Data Catalog.

**Crawler Execution:** Once the crawler was executed, it created metadata tables in the Glue Data Catalog, which included details like column names, data types, and partitioning based on the data format and structure.

**Athena Queries:** With the metadata stored in the Glue Data Catalog, queries were executed in Amazon Athena. The tables created by Glue were used to run SQL queries for data analysis.

**Automation:** The Glue crawler was scheduled to run periodically, enabling automatic detection of new data in the S3 bucket. This ensured that the Athena tables remained up-to-date without manual intervention.

Databases created in AWS Glue:

The screenshot shows the AWS Glue Data Catalog Databases page. The left sidebar has sections for AWS Glue (Getting started, ETL jobs, Visual ETL, Notebooks, Job run monitoring, Data Catalog tables, Data connections, Workflows (orchestration)), Data Catalog (Databases, Tables, Stream schema registries, Schemas, Connections, Crawlers, Classifiers, Catalog settings), Data Integration and ETL, and Legacy pages. The main content area is titled "Databases (3)". It contains a table with three rows:

Name	Description	Location URI	Created on (UTC)
ahmeddbprocessed	-	-	November 29, 2024 at 23:16:04
default	-	-	November 30, 2024 at 00:20:07
productsdb	-	-	November 30, 2024 at 08:02:21

Tables consisting of processed Data:

All the table columns and properties were created by AWS Glue. The folder path to the S3 directory was given as input. In the excluded fields, all other directories were added. Difficulty was encountered when specific files were given as input to the Data Catalog. Upon debugging, we found that it is ideal to give the folder path of an s3 file rather than a file name consisting of the data.

The screenshot shows the AWS Glue Data Catalog interface. On the left, there's a navigation sidebar with links like 'Getting started', 'ETL jobs', 'Virtual ETL', 'Notebooks', 'Job run monitoring', 'Data Catalog tables', 'Data connections', 'Workflows (orchestration)', 'Data Catalog', 'Databases', 'Tables', 'Stream schema registries', 'Schemas', 'Connections', 'Crawlers', 'Classifiers', 'Catalog settings', 'Data Integration and ETL', and 'Legacy pages'. A message at the top says 'Announcing new optimization features for Apache Iceberg tables'.

The main area is titled 'productsdb' and shows 'Database properties' with a table for 'Tables (3)'. The table has columns for Name, Database, Location, Classification, Deprecated, View data, Data quality, and Column statistics. The three tables listed are:

Name	Database	Location	Classification	Deprecated	View data	Data quality	Column statistics
customer	productsdb	s3://eshopdataadb/c	CSV	-	Table data	View data quality	View statistics
products	productsdb	s3://eshopdataadb/p	CSV	-	Table data	View data quality	View statistics
purchase	productsdb	s3://eshopdataadb/p	CSV	-	Table data	View data quality	View statistics

At the bottom of the page, there are links for 'What's New', 'Documentation', 'AWS Marketplace', and checkboxes for 'Enable compact mode' and 'Enable new navigation'.

## AWS Glue Crawler properties and configuration:

Frequency of running crawler is set to onDemand.

The screenshot shows the AWS Glue Crawler configuration wizard. The left sidebar is identical to the one in the previous screenshot. The main area is titled 'Review and update' and contains five steps:

- Step 1: Set crawler properties**: Shows a table for 'Set crawler properties' with columns for Name, Description, and Tags. One entry is 'productable'.
- Step 2: Choose data sources and classifiers**: Shows a table for 'Data sources' with a single entry: Type 'S3' and Data source 's3://eshopdataadb/'.
- Step 3: Configure security settings**: Shows 'Configure security settings' with IAM role 'AWSGlueServiceRole-ahmed' and Security configuration 'Lake Formation configuration'.
- Step 4: Set output and scheduling**: Shows 'Set output and scheduling' with Database 'productsdb', Table prefix - optional, Maximum table threshold - optional, and Schedule 'On demand'.

At the bottom right, there are 'Cancel', 'Previous', and 'Update' buttons.

The screenshot shows the AWS Glue Crawler properties page. On the left, there's a sidebar with navigation links like Getting started, ETL jobs, Visual ETL, Notebooks, Job run monitoring, Data Catalog tables, Data connections, Workflows (orchestration), Data Catalog, Databases, Tables, Stream schema registries, Schemas, Connections, Crawlers, Catalog settings, Data Integration and ETL, Legacy pages, What's New, Documentation, and AWS Marketplace. Under Crawlers, 'productstable' is selected. The main panel shows 'Crawler properties' with tabs for IAM role (AWSGlueServiceRole-ahmed), Database (productsdb), State (READY), and Table prefix. Below is a 'Maximum table threshold' section and an 'Advanced settings' tab. The 'Crawler runs' tab is active, displaying a table of runs from November 30, 2024. The table includes columns for Start time (UTC), End time (UTC), Current/last duration, Status, DPU hours, and Table changes. Most runs show 'Completed' status with 0 DPU hours and 3 table changes. A 'View CloudWatch logs' button is also present.

## AWS Cloudwatch logs when Crawler was running:

The screenshot shows the AWS CloudWatch Log Groups page. The left sidebar includes CloudWatch, Favorites and recent dashboards, Alarms, Logs, Log groups, Log Anomalies, Live Tail, Logs Insights, Contributor Insights, Metrics, X-Ray traces, Events, Application Signals, Network Monitoring, and Insights. The 'Logs' section is expanded, showing 'Log groups' with 'aws-glue/crawlers' selected. The main area displays log entries for the 'productstable' crawler. The logs show various INFO messages indicating the crawler's progress, such as 'Classification complete, writing results to database productsdb', 'Created table processed.purchase\_data\_csv in database productsdb', and 'Finished writing to Catalog'. There are approximately 20 log entries listed, all timestamped between November 30, 2024, and December 1, 2024.

## Sample Table created after Crawler run was successful:

**Table details**

**Associated columns (5)**

Name	Type
customerid	bigint
firstname	string
lastname	string
email	string
signupdate	string

## Sample query result

**Data source**: AwsDataCatalog

**Tables and views**: Tables (1) customer

**Query results**

#	customerid	firstname	lastname	email	signupdate
1	1	Ah	med	ahmed@gmail.com	2024-07-25
2	2	Jane	Smith	jane.smith@example.com	2025-02-10
3	3	Emily	Johnson	emily.johnson@example.com	2025-03-05
4	4	Michael	Brown	michael.brown@example.com	2024-07-11

## Queries:

### Total Sales by Product

```

Query 19 : X | ❌ Query 20 : X | ✅ Query 21 : X | ✅ Query 22 : X | Query 23 : X | Query 24 : X | ✅ Query 25 : X
1  SELECT p.item, SUM(pr.amount * pu.quantity) AS total_sales
2  FROM "productsdb"."purchase" pu
3  JOIN "productsdb"."products" pr ON pu.item = pr.item
4  GROUP BY p.item
5  ORDER BY total_sales DESC;
    
```

### Top Selling Products

Query 19 : X | ✖ Query 20 : X | ✔ Query 21 : X | ✔ Query 22 : X | Query 23 : X | Query 24 : X | ✔ Query 25 : X |

```

1  SELECT p.item, SUM(pu.quantity) AS total_quantity_sold
2  FROM "productsdb"."purchase" pu
3  JOIN "productsdb"."products" p ON pu.item = p.item
4  GROUP BY p.item
5  ORDER BY total_quantity_sold DESC
6  LIMIT 10;

```

## Average Purchase Amount by Payment Method

< | Query 19 : X | ✖ Query 20 : X | ✔ Query 21 : X | ✔ Query 22 : X | ✔ Query 23 : X | Query 24 : X | ✔ Query 25

```

1  SELECT paymentmethod, AVG(amount) AS avg_purchase_amount
2  FROM "productsdb"."purchase"
3  GROUP BY paymentmethod
4  ORDER BY avg_purchase_amount DESC;

```

## Sales by category

< | ✔ Query 21 : X | ✔ Query 22 : X | ✔ Query 23 : X | ✔ Query 24 : X | ✔ Query 25 : X | ✔ Query 26 : X | ✔ Query 27 : X | >

```

1  SELECT pr.category, SUM(pu.amount * pu.quantity) AS total_sales
2  FROM "productsdb"."purchase" pu
3  JOIN "productsdb"."products" pr ON pu.item = pr.item
4  GROUP BY pr.category
5  ORDER BY total_sales DESC;

```

## Query to segregate customers based on purchase data. Low value vs high value

< | ✔ Query 29 : X | ✖ Query 20 : X | ✔ Query 21 : X | ✔ Query 22 : X | ✔ Query 23 : X | ✔ Query 24 : X | ✔ Que

```

1  WITH CustomerPurchaseSummary AS (
2      SELECT
3          c.customerid,
4          c.firstname,
5          c.lastname,
6          c.email,
7          SUM(p.amount * p.quantity) AS total_spent
8      FROM "productsdb"."purchase" p
9      JOIN "productsdb"."customer" c ON p.customerid = c.customerid
10     GROUP BY c.customerid, c.firstname, c.lastname, c.email
11   )
12   SELECT
13       cps.customerid,
14       cps.firstname,
15       cps.lastname,
16       cps.email,
17       cps.total_spent,
18       CASE
19           WHEN cps.total_spent >= 1000 THEN 'High-Value'
20           WHEN cps.total_spent >= 500 AND cps.total_spent < 1000 THEN 'Medium-Value'
21           ELSE 'Low-Value'
22       END AS customer_segment
23   FROM CustomerPurchaseSummary cps
24   ORDER BY cps.total_spent DESC;

```

< | ✔ Query 29 : X | ✖ Query 20 : X | ✔ Query 21 : X | ✔ Query 22 : X | ✔ Query 23 : X | ✔ Query 24 : X | ✔ Que

```

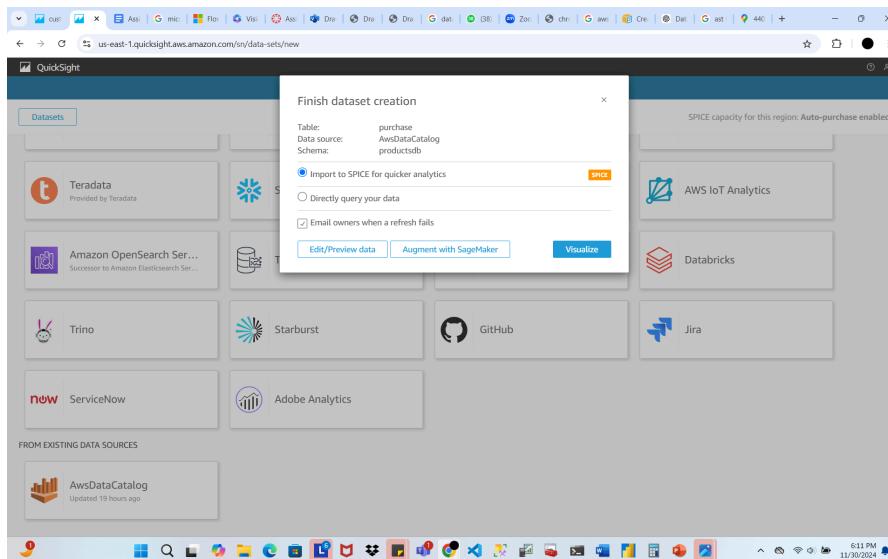
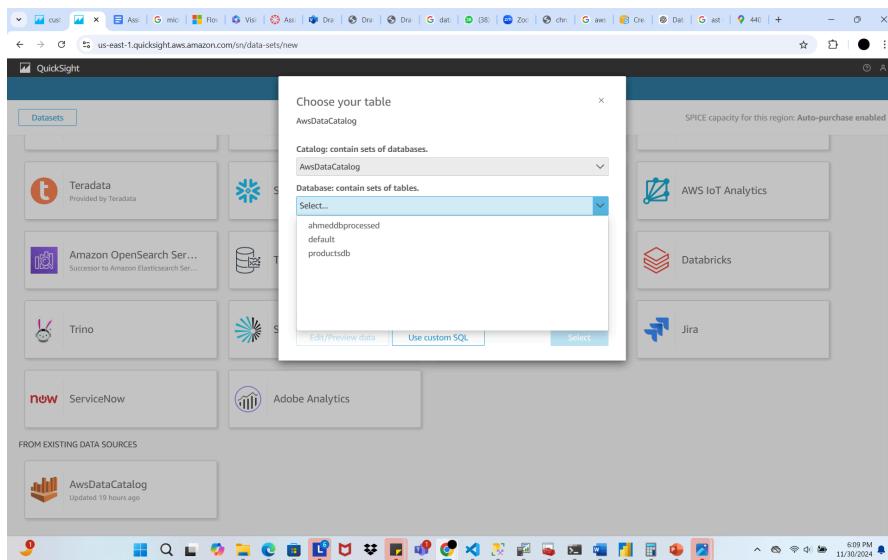
11  )
12   SELECT
13       cps.customerid,
14       cps.firstname,
15       cps.lastname,
16       cps.email,
17       cps.total_spent,
18       CASE
19           WHEN cps.total_spent >= 1000 THEN 'High-Value'
20           WHEN cps.total_spent >= 500 AND cps.total_spent < 1000 THEN 'Medium-Value'
21           ELSE 'Low-Value'
22       END AS customer_segment
23   FROM CustomerPurchaseSummary cps
24   ORDER BY cps.total_spent DESC;

```

## Data Visualization:

We have used Amazon Quicksight for visualization of datasets in the datalake in s3 bucket. First, data is made in table format using AWS Glue and Crawlers. Then, it is loaded onto Amazon Athena from AWS DataCatalog. Quicksight also loads data from AWA DataCatalog.

Creating datasets in Amazon Quicksight using AWS DataCatalog



# Visualization:

## Joining Tables in Amazon QuickSight:

The screenshot shows the Amazon QuickSight Data Editor interface. On the left, the 'Fields' pane lists fields from the 'purchase' dataset: transactionid, customerid, purchasedate, item, amount, quantity, paymentmethod, productid, itemproducts, price, model, and category. The 'Focus' dropdown is set to 'All fields'. The 'Parameters' pane shows no filters applied. The 'Community' section indicates 'No fields excluded'.

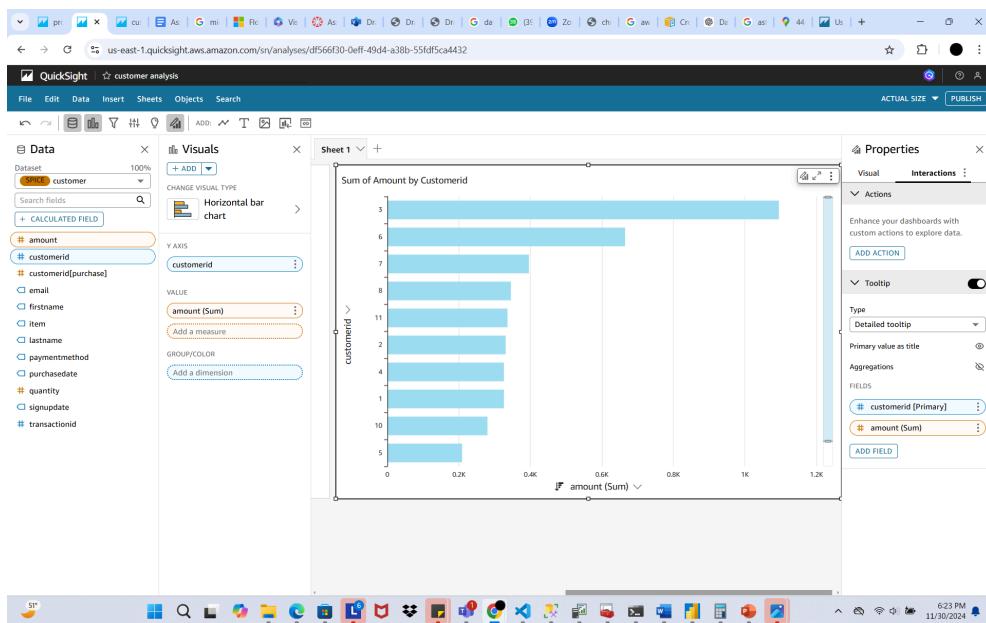
The main workspace displays two datasets: 'purchase' and 'products'. A join line connects the 'item' field in the 'purchase' dataset to the 'item' field in the 'products' dataset. Below the datasets, a preview table shows 8 rows of data from the joined dataset. The columns are: transactionid, customerid, purchasedate, item, amount, quantity, paymentmethod, productid, itemproducts, price, model, and category. The data includes items like a 'Wireless Mouse' at \$25.99 and a 'Monitor Stand' at \$39.99.

At the bottom, the 'Query mode' is set to 'SPICE'. The status bar shows the time as 6:19 PM and the date as 11/30/2024.

A second screenshot below shows the 'Join configuration' dialog box. It displays the join clause 'purchase.item = products.item' and four join type options: Inner, Left, Right, and Full. The 'Left' join type is selected. The status bar at the bottom right shows 5:39 PM and 11/30/2024.

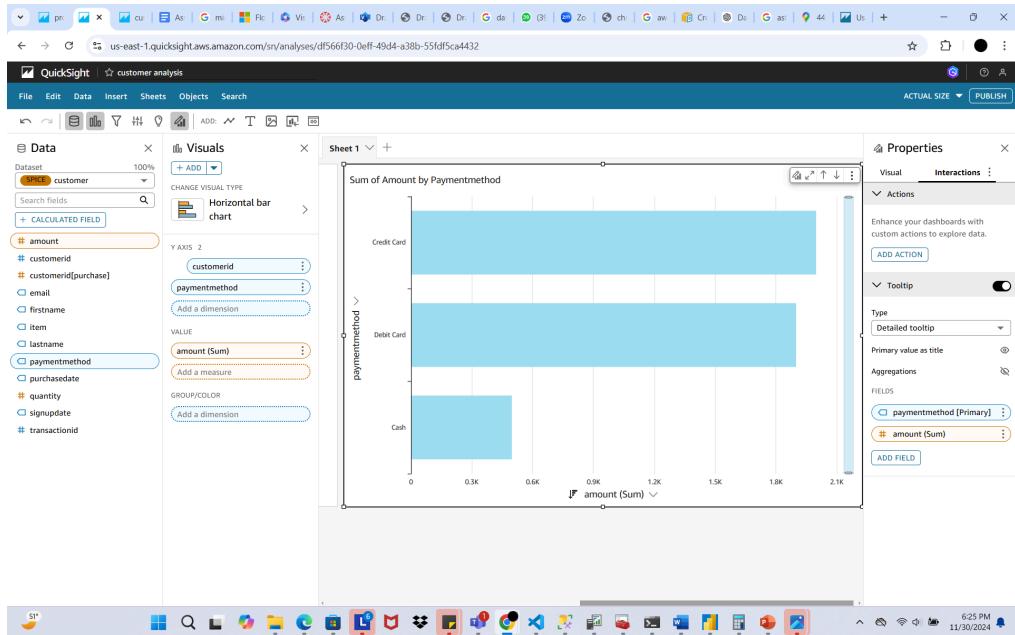
## Ranking customers in accordance with their purchase amounts

This horizontal bar graph shows the customer spending habits, depicting the total of purchase amounts against each customer ID. This will help E-Shop identify its most lucrative customers, targeting those whose contribution to revenue is very high. Insights from this might drive the development of loyalty programs, personalized marketing, and VIP offers in order to retain these top customers. Additionally, customers with lower spending could be allowed promotional offers or discounts that would help in repeat purchases. By understanding the trend in spending, E-Shop can segment the customer base and optimize marketing budgets to draw up strategies to maximize customer lifetime value and overall revenue growth.



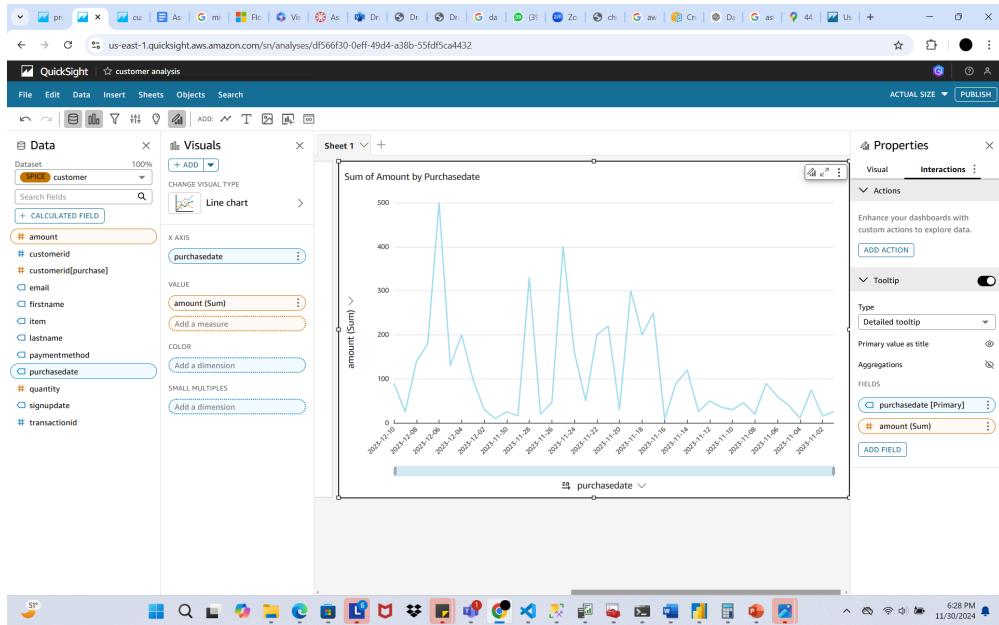
## Grouping purchases as a function of payment method

This bar chart shows the total amount of transactions by different payment methods: Credit Card, Debit Card, and Cash. The chart provides important information about the preferences of customers regarding payments, showing that combined debit and credit card transactions comprise a far larger proportion than cash purchases. E-Shop shall use these findings to improve its payment infrastructure, ensuring seamless digital payment experiences. It also helps in understanding the lower cash reliance to make informed decisions that can further promote digital payment options and reduce operational costs related to cash handling. Strategic alliances with card providers will also allow for incentives on more card-based purchases, promoting convenience for customers and increasing overall revenue.

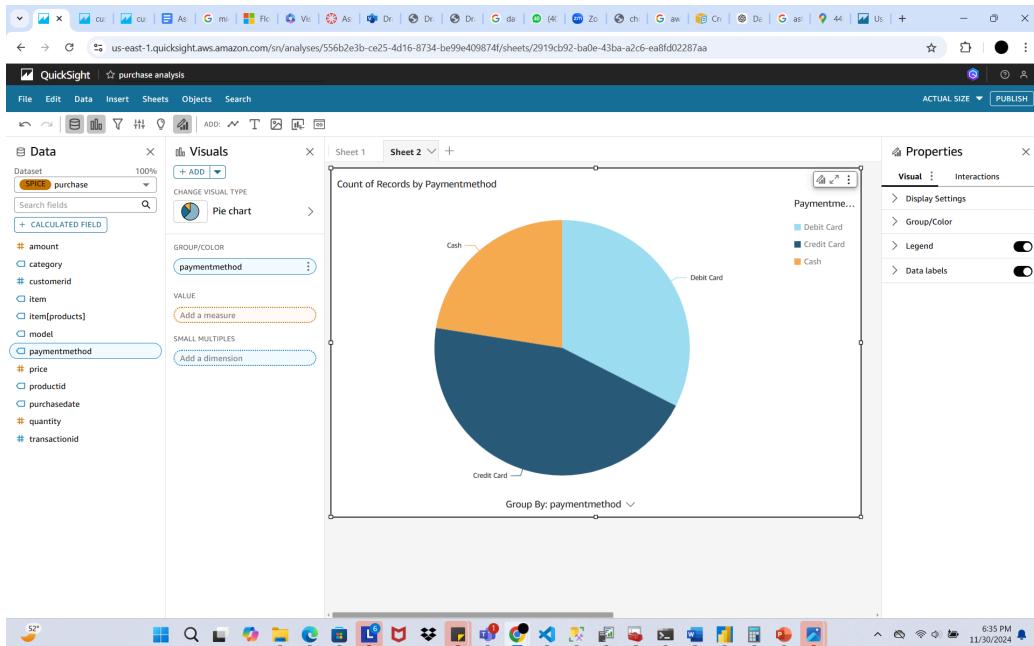


## Line graph with purchase date on x axis and amount on y axis

This line chart shows the trend of purchase amounts over time, giving insight into sales performance for different dates. E-Shop can analyze these to identify high-revenue periods, such as during promotions, holidays, or product launches. This information will help optimize marketing efforts by focusing on dates with potential for higher sales. Similarly, low-revenue periods provide opportunities for targeted campaigns to drive customer engagement. Also, consistent analysis of sales trends can aid in inventory planning, ensuring stock availability during peak demand and reducing overstocking during slow periods.



The chart shows what proportion of transactions were completed using debit cards, credit cards, and cash. It helps E-Shop understand customer payment habits for better alignment with their preferred ways of paying. In case the share of debit and credit card settlements is higher, E-Shop can focus on enhancing its digital payment systems. In the case of high cash usage, it suggests improvement in cash handling procedures at physical stores or cash-on-delivery support for online orders.



### Graph depicting transaction count with users(email address):

Through the monitoring of the transactions that are associated with particular email addresses, E-Shop can determine its top customers. This helps the business in identifying the most valuable customers and work on strategies to retain them such as loyalty programs, discounts, and exclusive offers. Using the active customer profiles (through email) E-Shop can enhance the effectiveness of its marketing strategies by creating targeted campaigns, sending out emails to encourage customers to make another purchase, or to promote specific products based on their interests or purchase history as well as to re-engage inactive customers with exclusive offers.

