Initiative: Hyper-Personalized Customer Experience

To track the success of creating a more personalized customer journey, you need to understand the relationship between customer behaviors, interactions, and outcomes like satisfaction and increase purchasing.

\* Demographics (age, gender, location)

\* Interaction data (number of interactions, feedback scores)

\* Behavioral data (products viewed, time spent on website)

\* Outcome data (satisfaction scores, and critically, retention status).

\* How to use it for Strategic Tracking:

\* Model Drivers of Churn and Satisfaction: Before deploying changes to your own systems, your data science team can use this dataset to build and test models that predict what factors most influence customer retention and satisfaction. This de-risks your "Project Empathy" initiative.

\* Develop "Effort to Resolution" Proxies: While this dataset doesn't have a direct "effort" score, your team can use fields like "number of interactions" as a proxy to model how increased effort impacts satisfaction, validating the importance of your "Frictionless Commerce" initiative.

\* Benchmark Personalization Impact: This dataset can provide a baseline for the expected uplift in satisfaction or retention from basic segmentation before you invest in more complex hyper-personalization technology.

**Phase 1.**

This phase sets up the foundation of your data platform. Its goal is to take a simulated stream of raw user events, like clicks and purchases, and land them reliably in your data lake for future processing.

## 1. Setup & Prerequisites ⚙️

Before you write any code, you need to have your environment ready. The easiest way to run Kafka locally is with Docker.

\* Get the Dataset: Download the 2019-Oct.csv file from the eCommerce behavior dataset on Kaggle and place it in your project directory.

\* Run Kafka with Docker: Create a docker-compose.yml file. This file defines the services needed to run Kafka: Zookeeper (which Kafka uses for coordination) and the Kafka broker itself.

# docker-compose.yml

version: '3'

services:

zookeeper:

image: confluentinc/cp-zookeeper:latest

environment:

ZOOKEEPER\_CLIENT\_PORT: 2181

ZOOKEEPER\_TICK\_TIME: 2000

kafka:

image: confluentinc/cp-kafka:latest

depends\_on:

- zookeeper

ports:

- "9092:9092"

environment:

KAFKA\_BROKER\_ID: 1

KAFKA\_ZOOKEEPER\_CONNECT: zookeeper:2181

KAFKA\_ADVERTISED\_LISTENERS: PLAINTEXT://kafka:29092,PLAINTEXT\_HOST://localhost:9092

KAFKA\_LISTENER\_SECURITY\_PROTOCOL\_MAP: PLAINTEXT:PLAINTEXT,PLAINTEXT\_HOST:PLAINTEXT

KAFKA\_INTER\_BROKER\_LISTENER\_NAME: PLAINTEXT

KAFKA\_OFFSETS\_TOPIC\_REPLICATION\_FACTOR: 1

\* Start Kafka: Open your terminal in the same directory and run:

docker-compose up -d

\* Create the Kafka Topic: Once the containers are running, execute a command inside the Kafka container to create the user\_events topic.

docker-compose exec kafka kafka-topics --create --topic user\_events --bootstrap-server localhost:9092 --partitions 1 --replication-factor 1

## 2. The Python Producer Script 🚚

This script reads the massive CSV file one row at a time (to avoid loading it all into memory) and sends each row as a message to your Kafka topic.

\* Install Library: You'll need the Python client for Kafka.

pip install kafka-python

\* Create the Script: Save this code as producer.py. It reads the CSV, converts each row to a JSON object, and sends it to Kafka, pausing briefly between messages to simulate a real-world event stream.

# producer.py

import csv

import json

import time

from kafka import KafkaProducer

# Configuration

KAFKA\_TOPIC = 'user\_events'

KAFKA\_SERVER = 'localhost:9092'

CSV\_FILE\_PATH = '2019-Oct.csv' # Make sure this path is correct

# Initialize Kafka Producer

# value\_serializer converts Python objects to JSON bytes

producer = KafkaProducer(

bootstrap\_servers=KAFKA\_SERVER,

value\_serializer=lambda v: json.dumps(v).encode('utf-8')

)

print("Starting producer...")

# Open the massive CSV file

with open(CSV\_FILE\_PATH, mode='r') as csv\_file:

csv\_reader = csv.DictReader(csv\_file)

# Read file row-by-row

for row in csv\_reader:

try:

# Send the row (as a dictionary) to the Kafka topic

producer.send(KAFKA\_TOPIC, value=row)

print(f"Sent: {row['event\_type']} for user {row['user\_id']}")

# Control the stream speed (e.g., 10 messages per second)

time.sleep(0.1)

except Exception as e:

print(f"Error sending message: {e}")

# Ensure all messages are sent before exiting

producer.flush()

print("Producer finished.")

## 3. The Spark Structured Streaming Job 💾

This is the core of the data pipeline. The Spark job will connect to Kafka, process the incoming data in micro-batches, and write it to a data lake location.

\* Run the Producer: Start the stream of data by running your producer script from the terminal:

python producer.py

\* Create the Spark Consumer Script: Save this code as stream\_consumer.py. This script defines the schema, connects to Kafka, performs a simple validation, and writes the stream to a destination.

# stream\_consumer.py

from pyspark.sql import SparkSession

from pyspark.sql.types import StructType, StructField, StringType, DoubleType, LongType, TimestampType

from pyspark.sql.functions import from\_json, col, window

# --- Configuration ---

KAFKA\_TOPIC = "user\_events"

KAFKA\_SERVER = "localhost:9092"

# Replace with your actual S3/ADLS/GCS path

DATA\_LAKE\_PATH = "s3a://your-bucket-name/raw/user\_events"

CHECKPOINT\_PATH = "s3a://your-bucket-name/checkpoints/user\_events"

# --- Initialize Spark Session ---

# The packages option includes the necessary Kafka connector

spark = SparkSession.builder \

.appName("UserEventStreamConsumer") \

.config("spark.jars.packages", "org.apache.spark:spark-sql-kafka-0-10\_2.12:3.5.0") \

.getOrCreate()

# --- Define Schema for Incoming JSON Data ---

# This enforces structure on your raw data

schema = StructType([

StructField("event\_time", StringType(), True),

StructField("event\_type", StringType(), True),

StructField("product\_id", StringType(), True),

StructField("category\_id", StringType(), True),

StructField("category\_code", StringType(), True),

StructField("brand", StringType(), True),

StructField("price", StringType(), True), # Read as string, cast later

StructField("user\_id", StringType(), True),

StructField("user\_session", StringType(), True),

])

# --- Read from Kafka Source ---

kafka\_df = spark.readStream \

.format("kafka") \

.option("kafka.bootstrap.servers", KAFKA\_SERVER) \

.option("subscribe", KAFKA\_TOPIC) \

.option("startingOffsets", "latest") \

.load()

# --- Transform the Data ---

# Kafka messages are key-value pairs. The 'value' column contains our JSON data.

# 1. Cast the binary 'value' to a string.

# 2. Parse the JSON string using the defined schema.

# 3. Select the fields out of the parsed struct.

# 4. Perform type casting and simple validation.

transformed\_df = kafka\_df.select(from\_json(col("value").cast("string"), schema).alias("data")) \

.select("data.\*") \

.withColumn("price", col("price").cast(DoubleType())) \

.withColumn("event\_timestamp", col("event\_time").cast(TimestampType())) \

.filter(col("user\_id").isNotNull()) # Simple validation

# --- Write Stream to Data Lake ---

# This is where the magic happens.

query = transformed\_df.writeStream \

.format("parquet") \

.outputMode("append") \

.partitionBy("event\_type") \

.option("path", DATA\_LAKE\_PATH) \

.option("checkpointLocation", CHECKPOINT\_PATH) \

.trigger(processingTime='1 minute') \

.start()

# Wait for the stream to terminate (e.g., by manual intervention)

query.awaitTermination()

\* Submit the Spark Job: Run this consumer script using spark-submit. You need to ensure Spark can connect to your cloud storage (e.g., by setting up AWS keys for S3).

spark-submit --packages org.apache.spark:spark-sql-kafka-0-10\_2.12:3.5.0 stream\_consumer.py

As this pipeline runs, you'll see Parquet files appearing in your S3 bucket, organized into folders by event\_type (e.g., raw/user\_events/event\_type=view/, raw/user\_events/event\_type=cart/). This partitioned data is now ready for the next phase of complex batch processing.

**Phase 2**

This phase is the heart of your data platform. You'll transform the raw, chaotic event data sitting in your data lake into a clean, structured, and insightful asset in a powerful data warehouse. We'll use Airflow to schedule the job and Spark to perform the heavy lifting. For the destination, we'll use Snowflake.

Part 1: The Big Picture - Architecture & Concepts

Before we dive into code, let's understand why these tools are used together.

\* Why Airflow? Airflow is your Orchestrator. It's like the general contractor for your data project. It doesn't move the bricks itself, but it reads the blueprint (your DAG file) and tells the right worker (Spark) when to start working, when to stop, and what to do if a job fails. It's responsible for scheduling, dependency management, and monitoring.

\* Why a Daily Batch Job? The streaming job in Phase 1 was designed for one purpose: to capture raw data quickly. This daily batch job is for deep, complex, and resource-intensive analysis. It's far more efficient to process an entire day's worth of data at once, often during off-peak hours, to build a comprehensive analytical model.

\* Why Spark? Spark is your Processing Engine. It's the heavy-duty machinery. It's designed to read terabytes of data from your data lake (S3), distribute the processing work across many nodes in a cluster, and perform complex transformations and aggregations in memory, making it incredibly fast.

\* Why Snowflake? Snowflake is your Cloud Data Warehouse. It's the clean, organized, and publicly accessible showroom for your finished data product. It's built for incredibly fast analytical queries, allowing business intelligence (BI) tools, analysts, and data scientists to easily explore the structured data you've prepared.

Part 2: Setup & Prerequisites

This part involves setting up Airflow and Snowflake.

Step 1: Get Airflow Running with Docker

The official Airflow documentation provides the easiest way to get started using Docker Compose.

\* Create a project folder for your Airflow setup, e.g., airflow-project.

\* Download the official docker-compose.yml file. Open a terminal in your new folder and run this command:

curl -LfO "https://airflow.apache.org/docs/apache-airflow/2.9.2/docker-compose.yaml"

\* Create necessary folders:

mkdir -p ./dags ./logs ./plugins ./config

\* Initialize the Environment: Run the following command. It will create a new user with the username airflow and password airflow.

docker-compose up airflow-init

\* Start Airflow:

docker-compose up -d

Once complete, you can access the Airflow UI by navigating to http://localhost:8080 in your web browser. Log in with airflow / airflow.

Step 2: Set Up Your Snowflake Environment

\* Create a Free Trial Account: If you don't have one, sign up for a 30-day free Snowflake trial.

\* Create Objects: Once logged into your Snowflake account, open a new "Worksheet" and run the following SQL commands to create the necessary database, schema, table, and warehouse.

-- Use a role that has permission to create objects, like ACCOUNTADMIN for this setup

USE ROLE ACCOUNTADMIN;

-- Create a virtual warehouse for computing power

CREATE WAREHOUSE IF NOT EXISTS COMPUTE\_WH WITH WAREHOUSE\_SIZE = 'X-SMALL';

-- Create a database and a schema to hold our tables

CREATE DATABASE IF NOT EXISTS ECOMMERCE\_DB;

CREATE SCHEMA IF NOT EXISTS ECOMMERCE\_DB.ANALYTICS;

-- Create the final destination table for our transformed data

CREATE OR REPLACE TABLE ECOMMERCE\_DB.ANALYTICS.CUSTOMER\_360\_PROFILE (

USER\_ID BIGINT,

FAVORITE\_BRAND VARCHAR,

MOST\_VIEWED\_CATEGORY VARCHAR,

TOTAL\_PURCHASES BIGINT,

TOTAL\_SPEND DOUBLE,

LAST\_SEEN\_DATE TIMESTAMP\_NTZ,

PRIMARY KEY (USER\_ID)

);

Step 3: Configure Connections in Airflow

Airflow needs to know how to connect to AWS (for Spark to access S3) and Snowflake.

\* In the Airflow UI (http://localhost:8080), go to Admin -> Connections.

\* Create the AWS Connection:

\* Click the + button to add a new connection.

\* Connection ID: aws\_default (This is a standard name Spark looks for).

\* Connection Type: Amazon Web Services.

\* Login: Your AWS Access Key ID.

\* Password: Your AWS Secret Access Key.

\* Click Save.

\* Create the Snowflake Connection:

\* Click the + button again.

\* Connection ID: snowflake\_default.

\* Connection Type: Snowflake.

\* Host: Your Snowflake account URL (e.g., youraccount.us-east-1.snowflakecomputing.com).

\* Schema: ANALYTICS

\* Login: Your Snowflake username.

\* Password: Your Snowflake password.

\* Account: Your account identifier (e.g., youraccount).

\* Warehouse: COMPUTE\_WH

\* Database: ECOMMERCE\_DB

\* Role: The role you want to use (e.g., ACCOUNTADMIN for this setup).

\* Click Save.

Part 3: The Spark Batch Job Script

This Python script contains the logic for our ETL job. It will read the raw Parquet data, clean it, aggregate it to create user profiles, and load it into Snowflake.

Save this script as process\_daily\_events.py inside the dags/spark\_scripts folder of your Airflow project.

# dags/spark\_scripts/process\_daily\_events.py

from pyspark.sql import SparkSession

from pyspark.sql.functions import col, count, sum, max, split, mode

# --- CONFIGURATION (These will be passed by Airflow) ---

# Example paths for local testing

S3\_INPUT\_PATH = "s3a://your-bucket-name/raw/user\_events"

SNOWFLAKE\_TABLE = "ECOMMERCE\_DB.ANALYTICS.CUSTOMER\_360\_PROFILE"

# Note: Snowflake connection details are handled by the connector options

def main():

"""

Main ETL script.

"""

spark = SparkSession.builder \

.appName("EcommerceDailyETL") \

.getOrCreate()

# 1. READ (Extract)

# Read the partitioned Parquet data from the data lake

print("Reading raw data from S3...")

raw\_df = spark.read.parquet(S3\_INPUT\_PATH)

# 2. TRANSFORM

# Perform cleaning and feature engineering

print("Transforming data...")

# Basic Cleaning

cleaned\_df = raw\_df.filter(col("user\_id").isNotNull()) \

.filter(col("brand").isNotNull()) \

.withColumn("price", col("price").cast("double")) \

.na.drop(subset=["user\_id", "brand", "price"])

# Feature Engineering: Create user-level aggregates

# For a real project, this would be much more complex!

# Separate dataframes for different event types

views\_df = cleaned\_df.filter(col("event\_type") == "view")

purchases\_df = cleaned\_df.filter(col("event\_type") == "purchase")

# Aggregate favorite brand from purchases

favorite\_brand\_df = purchases\_df.groupBy("user\_id").agg(

mode("brand").alias("FAVORITE\_BRAND")

)

# Aggregate most viewed category from views

most\_viewed\_category\_df = views\_df.groupBy("user\_id").agg(

mode("category\_code").alias("MOST\_VIEWED\_CATEGORY")

)

# Aggregate purchase metrics

purchase\_metrics\_df = purchases\_df.groupBy("user\_id").agg(

count("\*").alias("TOTAL\_PURCHASES"),

sum("price").alias("TOTAL\_SPEND")

)

# Aggregate last seen date from all events

last\_seen\_df = cleaned\_df.groupBy("user\_id").agg(

max("event\_time").alias("LAST\_SEEN\_DATE")

)

# Join all aggregates together to build the final profile

customer\_360\_df = last\_seen\_df \

.join(favorite\_brand\_df, "user\_id", "left") \

.join(most\_viewed\_category\_df, "user\_id", "left") \

.join(purchase\_metrics\_df, "user\_id", "left") \

.select("USER\_ID", "FAVORITE\_BRAND", "MOST\_VIEWED\_CATEGORY", "TOTAL\_PURCHASES", "TOTAL\_SPEND", "LAST\_SEEN\_DATE")

# 3. LOAD

print("Loading data into Snowflake...")

# Use the Spark-Snowflake connector to write the DataFrame

# Connection options are read from the environment/Airflow connection

customer\_360\_df.write \

.format("net.snowflake.spark.snowflake") \

.options(\*\*sf\_options) \

.option("dbtable", SNOWFLAKE\_TABLE) \

.mode("overwrite") \

.save()

print("ETL job completed successfully.")

if \_\_name\_\_ == '\_\_main\_\_':

# This block allows you to get arguments if needed, but for simplicity

# we'll rely on the DAG to configure the environment.

# Note: sf\_options would be constructed from Airflow variables/connections

# For now, this is a placeholder. The DAG will configure it.

sf\_options = {

"sfURL": "your\_account.snowflakecomputing.com",

"sfUser": "your\_user",

"sfPassword": "your\_password",

"sfDatabase": "ECOMMERCE\_DB",

"sfSchema": "ANALYTICS",

"sfWarehouse": "COMPUTE\_WH",

"sfRole": "ACCOUNTADMIN"

}

main()

Part 4: The Airflow DAG

This Python script defines the Airflow workflow. It has one task: to submit the Spark job we just wrote.

Save this script as ecommerce\_daily\_etl.py inside the dags/ folder of your Airflow project. Airflow will automatically discover it.

# dags/ecommerce\_daily\_etl.py

from \_\_future\_\_ import annotations

import pendulum

from airflow.models.dag import DAG

from airflow.providers.apache.spark.operators.spark\_submit import SparkSubmitOperator

with DAG(

dag\_id="ecommerce\_daily\_etl",

start\_date=pendulum.datetime(2025, 6, 16, tz="America/Halifax"),

schedule\_interval="0 1 \* \* \*", # Run daily at 1:00 AM

catchup=False,

tags=["ecommerce", "spark", "etl"],

) as dag:

process\_events = SparkSubmitOperator(

task\_id="process\_and\_load\_to\_snowflake",

application="/opt/airflow/dags/spark\_scripts/process\_daily\_events.py", # Path inside the Airflow container

conn\_id="aws\_default", # Uses AWS connection for S3 access

packages="net.snowflake:snowflake-jdbc:3.13.22,net.snowflake:spark-snowflake\_2.12:2.11.0-spark\_3.3",

# Pass Snowflake credentials securely from our Airflow Connection to the Spark job

application\_args=[

"--snowflake\_url", "{{ conn.snowflake\_default.host }}",

"--snowflake\_user", "{{ conn.snowflake\_default.login }}",

"--snowflake\_password", "{{ conn.snowflake\_default.password }}",

"--snowflake\_db", "{{ conn.snowflake\_default.schema }}", # The "schema" field in Airflow holds the DB

"--snowflake\_schema", "{{ conn.snowflake\_default.extra\_dejson.schema }}",

"--snowflake\_warehouse", "{{ conn.snowflake\_default.extra\_dejson.warehouse }}",

"--snowflake\_role", "{{ conn.snowflake\_default.extra\_dejson.role }}",

]

)

Note: The application\_args part is an advanced but powerful way to pass credentials securely. It requires modifying the Spark script to parse these arguments, which adds complexity. For simplicity in this guide, we've omitted the argument parsing from the Spark script, but this DAG shows the best practice.

Part 5: Running the Pipeline

\* Place the files: Make sure ecommerce\_daily\_etl.py is in your dags/ folder and process\_daily\_events.py is in dags/spark\_scripts/.

\* Un-pause the DAG: In the Airflow UI, you will see a new DAG called ecommerce\_daily\_etl. Toggle the switch to un-pause it.

\* Trigger a Run: Since the schedule is for 1 AM, you can trigger a manual run by clicking the "Play" button on the right side of the DAG's row.

\* Monitor the Run: Click on the DAG name to see the Graph View. You can watch your process\_and\_load\_to\_snowflake task go from queued -> running -> success (or failed). You can click on the task and view its logs to see the output from Spark.

\* Verify in Snowflake: Once the task is successful, go to your Snowflake worksheet and run SELECT \* FROM ECOMMERCE\_DB.ANALYTICS.CUSTOMER\_360\_PROFILE;. You should see your newly processed data, ready for analysis.

**Phase 3**

The goal of this phase is to take the clean, aggregated data from your Snowflake warehouse and make it available to two very different audiences: the live e-commerce website which needs millisecond-speed recommendations, and business analysts who need to run complex queries for operational dashboards.

This phase is broken into three distinct parts.

Part 1: Generate Product Recommendations in Snowflake

First, we need to create the recommendations themselves. We will use our existing data in Snowflake to generate a simple but effective recommendation model: "Top 5 Most Purchased Products in a User's Favorite Category." This is an "offline" model because we run it as a batch process.

Application Steps (Snowflake UI)

\* Log in to your Snowflake Worksheet. Ensure you are using a role that has access to ECOMMERCE\_DB.

\* Run the Recommendation Logic: Copy and run the following SQL query. This query joins the user profiles with the raw event data to find the most popular products within each user's most-viewed category and saves the results into a new table called USER\_PRODUCT\_RECOMMENDATIONS.

-- Use the correct database and schema

USE ECOMMERCE\_DB.ANALYTICS;

-- Create a new table to store the final recommendations

CREATE OR REPLACE TABLE USER\_PRODUCT\_RECOMMENDATIONS AS

WITH

-- Step 1: Get the raw purchase data, cleaning it up a bit

ProductPurchases AS (

SELECT

product\_id,

category\_code

FROM ECOMMERCE\_DB.RAW\_DATA.USER\_EVENTS -- Assuming your raw data is here

WHERE event\_type = 'purchase'

AND category\_code IS NOT NULL

AND product\_id IS NOT NULL

),

-- Step 2: Calculate the top 5 most purchased products in each category

TopProductsPerCategory AS (

SELECT

category\_code,

product\_id,

COUNT(\*) as purchase\_count,

-- Rank products by purchase count within each category

ROW\_NUMBER() OVER (PARTITION BY category\_code ORDER BY purchase\_count DESC) as rnk

FROM ProductPurchases

GROUP BY 1, 2

QUALIFY rnk <= 5 -- Keep only the top 5

),

-- Step 3: Aggregate the top 5 product IDs into a single array for each category

CategoryRecommendations AS (

SELECT

category\_code,

-- Group the product IDs into a list

ARRAY\_AGG(product\_id) WITHIN GROUP (ORDER BY rnk) as recommended\_products

FROM TopProductsPerCategory

GROUP BY 1

)

-- Step 4: Join the recommendations back to our Customer 360 profiles

-- This assigns a list of recommended products to each user based on their favorite category

SELECT

prof.USER\_ID,

rec.recommended\_products

FROM CUSTOMER\_360\_PROFILE prof

JOIN CategoryRecommendations rec

ON prof.MOST\_VIEWED\_CATEGORY = rec.category\_code;

-- Verify the results

SELECT \* FROM USER\_PRODUCT\_RECOMMENDATIONS LIMIT 10;

You now have a table in Snowflake where each user\_id is mapped to a list of recommended product\_ids.

Part 2: Serve Recommendations via a Low-Latency NoSQL Database

Snowflake is not designed for fast, single-row lookups from a website. For that, we need a NoSQL database like AWS DynamoDB. We will export our recommendations from Snowflake to DynamoDB.

Application Steps (AWS Console)

\* Navigate to DynamoDB: In the AWS Console, search for and go to the DynamoDB service.

\* Create Table:

\* Click "Create table".

\* Table name: product\_recommendations

\* Partition key (Primary Key): user\_id

\* Data type: Number

\* Leave all other settings as default and click "Create table".

Code: The Export Script

This Python script connects to Snowflake, reads the recommendations, and writes them to DynamoDB.

\* Install Libraries:

pip install "snowflake-connector-python[pandas]" boto3

\* Create the script (export\_to\_dynamodb.py):

import os

import json

import boto3

import snowflake.connector

from decimal import Decimal

# --- Configuration ---

# Assumes AWS credentials are set as environment variables

# Snowflake credentials should be securely stored, e.g., in environment variables

SNOWFLAKE\_USER = os.getenv("SNOWFLAKE\_USER")

SNOWFLAKE\_PASSWORD = os.getenv("SNOWFLAKE\_PASSWORD")

SNOWFLAKE\_ACCOUNT = os.getenv("SNOWFLAKE\_ACCOUNT")

SNOWFLAKE\_WAREHOUSE = "COMPUTE\_WH"

SNOWFLAKE\_DATABASE = "ECOMMERCE\_DB"

SNOWFLAKE\_SCHEMA = "ANALYTICS"

DYNAMODB\_TABLE\_NAME = 'product\_recommendations'

def fetch\_snowflake\_data():

"""Fetches product recommendations from Snowflake."""

print("Connecting to Snowflake...")

with snowflake.connector.connect(

user=SNOWFLAKE\_USER,

password=SNOWFLAKE\_PASSWORD,

account=SNOWFLAKE\_ACCOUNT,

warehouse=SNOWFLAKE\_WAREHOUSE,

database=SNOWFLAKE\_DATABASE,

schema=SNOWFLAKE\_SCHEMA

) as conn:

print("Successfully connected. Fetching data...")

cursor = conn.cursor()

cursor.execute("SELECT USER\_ID, RECOMMENDED\_PRODUCTS FROM USER\_PRODUCT\_RECOMMENDATIONS")

# Fetch all results

results = cursor.fetchall()

print(f"Fetched {len(results)} rows from Snowflake.")

return results

def write\_to\_dynamodb(data):

"""Writes data in batches to DynamoDB."""

print("Connecting to DynamoDB...")

dynamodb = boto3.resource('dynamodb')

table = dynamodb.Table(DYNAMODB\_TABLE\_NAME)

print(f"Starting batch write to {DYNAMODB\_TABLE\_NAME}...")

with table.batch\_writer() as batch:

for row in data:

user\_id, recommended\_products = row

# The recommended\_products from Snowflake is a string representation of a list

# We need to parse it. It may look like '[\n 123,\n 456\n]'

product\_list = json.loads(recommended\_products)

batch.put\_item(

Item={

'user\_id': int(user\_id),

'recommended\_products': [int(p) for p in product\_list]

}

)

print("Batch write to DynamoDB complete.")

if \_\_name\_\_ == "\_\_main\_\_":

snowflake\_data = fetch\_snowflake\_data()

if snowflake\_data:

write\_to\_dynamodb(snowflake\_data)

Now, when your e-commerce website needs recommendations for a user, it can make a simple, super-fast query to this DynamoDB table using the user\_id.

Part 3: Populate an Operational Dashboard Database

For business analysts, we need to provide the aggregated CUSTOMER\_360\_PROFILE data in a traditional relational database like PostgreSQL where they can connect BI tools.

Application Steps (Docker)

The easiest way to run PostgreSQL locally is with Docker.

\* Run PostgreSQL in Docker: Open a terminal and run this command. It will start a PostgreSQL container, set a password, and expose the port.

docker run --name postgres-dashboard -e POSTGRES\_PASSWORD=mysecretpassword -p 5432:5432 -d postgres

Code: The Export Script

This script reads the CUSTOMER\_360\_PROFILE table from Snowflake and writes it to PostgreSQL.

\* Install Libraries:

pip install "snowflake-connector-python[pandas]" psycopg2-binary sqlalchemy pandas

\* Create the script (export\_to\_postgres.py):

import os

import snowflake.connector

import pandas as pd

from sqlalchemy import create\_engine

# --- Configuration ---

SNOWFLAKE\_USER = os.getenv("SNOWFLAKE\_USER")

SNOWFLAKE\_PASSWORD = os.getenv("SNOWFLAKE\_PASSWORD")

SNOWFLAKE\_ACCOUNT = os.getenv("SNOWFLAKE\_ACCOUNT")

# PostgreSQL connection string

POSTGRES\_CONN\_STRING = "postgresql://postgres:mysecretpassword@localhost:5432/postgres"

def fetch\_customer\_profiles():

"""Fetches the customer 360 profiles from Snowflake."""

print("Connecting to Snowflake...")

with snowflake.connector.connect(

user=SNOWFLAKE\_USER,

password=SNOWFLAKE\_PASSWORD,

account=SNOWFLAKE\_ACCOUNT,

warehouse="COMPUTE\_WH",

database="ECOMMERCE\_DB",

schema="ANALYTICS"

) as conn:

print("Fetching customer profiles...")

query = "SELECT \* FROM CUSTOMER\_360\_PROFILE"

df = pd.read\_sql(query, conn)

print(f"Fetched {len(df)} profiles.")

# Snowflake column names are often uppercase, convert to lowercase for PostgreSQL

df.columns = [x.lower() for x in df.columns]

return df

def write\_to\_postgres(df):

"""Writes the DataFrame to a PostgreSQL table."""

print("Connecting to PostgreSQL...")

engine = create\_engine(POSTGRES\_CONN\_STRING)

print("Writing data to customer\_profiles table...")

# Use pandas to\_sql to write the dataframe. 'replace' will drop the table if it exists.

df.to\_sql('customer\_profiles', engine, if\_exists='replace', index=False)

print("Write to PostgreSQL complete.")

if \_\_name\_\_ == "\_\_main\_\_":

customer\_df = fetch\_customer\_profiles()

if not customer\_df.empty:

write\_to\_postgres(customer\_df)

After running this script, business analysts can connect any standard BI tool to your PostgreSQL database and start building dashboards from the customer\_profiles table.

Phase 4: Infrastructure & MLOps

The entire platform is managed as a professional software project.

\* Infrastructure as Code: Terraform is used to define and provision the entire cloud stack: the Kubernetes cluster, the Kafka service, the Spark environment on Databricks, and the Snowflake warehouse.

\* Containerization: All applications (the streaming jobs, the Airflow workers, the API serving the recommendations) are packaged into Docker containers.

\* Deployment & Orchestration: Kubernetes is used to deploy and manage the containerized applications, ensuring they are scalable and resilient. A CI/CD pipeline managed in Git automates the building, testing, and deployment of any code changes.

<!-- end list -->

\* Skills Used: Terraform, Docker, Kubernetes, CI/CD, Git, Linux, Agile (methodology for managing the project).

**Proposed changes**

Executive Summary: Key Recommendations

| Area | Current Approach | Recommended Enhancement | Benefit |

|---|---|---|---|

| Processing Paradigm | ETL: Extract from S3, Transform in Spark, Load into Snowflake. | ELT: Extract from S3, Load raw data into Snowflake, then Transform using Snowflake's compute. | Simplicity, cost-effectiveness, and leveraging the power of the data warehouse engine. |

| Data Modeling in DWH | Single, wide CUSTOMER\_360\_PROFILE table. | Star Schema: Create fact (fct\_events) and dimension (dim\_user, dim\_product, dim\_date) tables. The C360 table becomes a final data mart view. | Increased flexibility for analytics, reduced data redundancy, and improved query performance for BI tools. |

| Real-Time Ingestion | JSON strings sent to Kafka. Schema is defined only in the Spark consumer. | Use Avro for data serialization and a central Schema Registry with Kafka. | Improved data quality, schema evolution management, reduced network bandwidth, and storage costs. |

| Batch Job Efficiency | Reads all raw data daily and overwrites the destination table. | Implement incremental processing by partitioning the data lake by date and using Snowflake's MERGE command. | Massively improved job performance, reduced compute costs, and atomic, safer data loads. |

| Data Serving | Custom Python scripts to move data to PostgreSQL for BI and DynamoDB for recommendations. | Connect BI tools directly to Snowflake. Use managed Reverse ETL tools or orchestrated Airflow jobs to sync data to operational systems like DynamoDB. | Reduced complexity, lower maintenance, eliminates a redundant database (PostgreSQL), and ensures fresher data in serving layers. |

Detailed Recommendations

1. Architectural Paradigm: Shift from ETL to ELT

The current model uses Spark as the primary transformation engine before loading data into Snowflake. While this works, a more modern and often more efficient pattern is ELT (Extract, Load, Transform).

\* Proposed Change:

\* Modify the Spark Structured Streaming job (Phase 1) or a new micro-batch job to load the raw, partitioned Parquet data from the S3 data lake directly into a RAW\_EVENTS table in Snowflake. Use tools like Snowflake's Snowpipe for automated ingestion from S3 for even lower latency.

\* Perform all the complex transformations and aggregations—currently done in the daily Spark batch job—using Snowflake SQL or Snowpark. The Airflow DAG would orchestrate these SQL scripts within Snowflake instead of submitting a Spark job.

\* Why this is better:

\* Simplicity & Cost: It reduces dependency on a separate, potentially expensive Spark cluster for transformations. You consolidate your compute costs and leverage Snowflake's highly optimized engine, which you are already paying for.

\* Performance: Snowflake's query optimizer is purpose-built for large-scale SQL transformations on data it already holds.

\* Maintenance: Managing complex data logic in SQL within the warehouse is often more straightforward for data analysts to understand and contribute to than a separate PySpark application.

2. Data Modeling: Adopt a Layered, Star Schema Approach

The final output is a single, denormalized CUSTOMER\_360\_PROFILE table. This is an excellent "data mart" table but should be the final product of a more robust underlying model, not the only model.

\* Proposed Change:

\* Staging Layer: A schema in Snowflake (RAW or STAGING) that holds the raw event data, loaded directly via the ELT process mentioned above.

\* Core Layer (Star Schema): In your ANALYTICS schema, create a classic star schema.

\* Fact Table: FCT\_EVENTS containing foreign keys to dimensions and metrics (e.g., price).

\* Dimension Tables: DIM\_USER, DIM\_PRODUCT, DIM\_CATEGORY, DIM\_DATE. These tables would hold descriptive attributes.

\* Mart Layer: The CUSTOMER\_360\_PROFILE table would be built from the core layer. It can be a table or, even better, a materialized view that is refreshed periodically.

\* Why this is better:

\* Flexibility: A star schema can answer a much wider range of business questions than a single, pre-aggregated table. Analysts can join dimensions in any way they need without being constrained.

\* Reusability: The dimension tables can be reused across multiple fact tables and data marts as the platform grows.

\* Clarity: It provides a "single source of truth" for core business entities like customers and products.

3. Real-Time Path: Enhance Data Quality and Efficiency

The ingestion pipeline relies on a Python script sending JSON messages to Kafka. This is fragile and inefficient at scale.

\* Proposed Change:

\* Introduce a Schema Registry (e.g., Confluent Schema Registry). Define the schema for your user\_events topic centrally.

\* Change the Data Format from JSON to Avro. Avro is a binary format that is more compact and integrates seamlessly with a schema registry.

\* Implement a Dead Letter Queue (DLQ). The Spark streaming job should be configured to route any messages that fail parsing (e.g., do not conform to the schema) to a separate Kafka topic for later analysis, preventing the entire stream from failing due to one bad message.

\* Why this is better:

\* Data Integrity: Schema enforcement happens at the producer level, preventing bad data from entering your pipeline in the first place.

\* Efficiency: Avro messages are significantly smaller than JSON, reducing Kafka broker load, network traffic, and S3 storage costs.

\* Resilience: A DLQ makes your real-time pipeline robust against data corruption or unexpected format changes.

4. Operational Systems: Streamline Data Serving

The architecture proposes moving data from Snowflake to both DynamoDB and PostgreSQL. This introduces unnecessary complexity.

\* Proposed Change:

\* Eliminate the PostgreSQL Export: Business intelligence and dashboarding tools (like Tableau, Looker, Power BI) are designed to connect directly and efficiently to Snowflake. Exporting the CUSTOMER\_360\_PROFILE to a separate PostgreSQL database adds a point of failure, data latency, and an extra system to maintain for no significant benefit.

\* Automate Reverse ETL: The export of recommendations to DynamoDB is a classic "Reverse ETL" pattern. Instead of a standalone Python script, this process should be orchestrated. Add a task to your Airflow DAG that runs after the main Snowflake transformations are complete to ensure data freshness. For more complex needs, consider a dedicated Reverse ETL tool.

\* Why this is better:

\* Reduced Architectural Complexity: Removing the PostgreSQL hop simplifies the entire right-hand side of the architecture diagram.

\* Single Source of Truth for BI: Analysts will always query the freshest, most complete data directly from the warehouse.

\* Operational Reliability: Orchestrating the sync to DynamoDB ensures that your website's recommendations are never stale.