Project-1 Customer 360° Analytics

Description: Developed a Customer 360° Analytics solution to track customer behavior and purchasing patterns by integrating data from RDS MySQL (transactional data), DynamoDB (customer interactions), and S3 (historical data). Leveraged AWS Glue and Athena for data transformation and reporting.

Key Achievements:

- Created an ETL pipeline that consolidated customer data from multiple sources.
- Enabled customer insights by integrating transactional and engagement data.
- Developed Athena queries & QuickSight dashboards for customer segmentation and churn prediction.

Tech Stack: AWS S3, AWS Glue, AWS Athena, RDS MySQL, DynamoDB, QuickSight, SNS , SES , Bedrock

Data Sources (Microservices)

Retail customer data is typically scattered across different microservices, such as:

- **E-commerce Service:** Orders, cart activity, browsing history
- **CRM System:** Customer profiles, preferences, support interactions
- **Marketing Platform:** Email campaigns, social media engagement
- **Loyalty Program:** Points, rewards, redemptions

- **Support & Feedback:** Call center logs, chatbot interactions, reviews
- **In-store Systems:** POS transactions, RFID data, IoT sensors

All these services generate structured and unstructured data.

Here are the core microservices typically found in a retail system:

Microservice	Function	Key Data Generated
User Service	Manages customer profiles, authentication, preferences	Customer details, login activity, account settings
Order Service	Handles purchases, payments, order tracking	Order history, payment status, refund requests
Marketing Service	Manages promotions, email campaigns, ads	Customer engagement, email clicks, campaign ROI
Customer Service	Handles support tickets, chatbot interactions, feedback	Support logs, issue resolution times, sentiment analysis
Product Service	Manages product catalog, inventory, pricing	Product details, stock levels, discounts
Recommendation Service	Suggests products based on customer behavior	Personalization data, clickstream insights
Loyalty Program Service	Tracks reward points, redemptions, customer tiers	Loyalty scores, customer engagement levels
Review & Feedback Service	Collects and analyzes product reviews, social media feedback	Review ratings, complaints, sentiment analysis
Analytics Service	Processes and analyzes data for insights	Customer segmentation, churn prediction

Architecture Overview

- **Data Sources**:
  - **Amazon RDS MySQL** → Stores transactional sales data.
  - **Amazon DynamoDB** → Stores customer interactions.

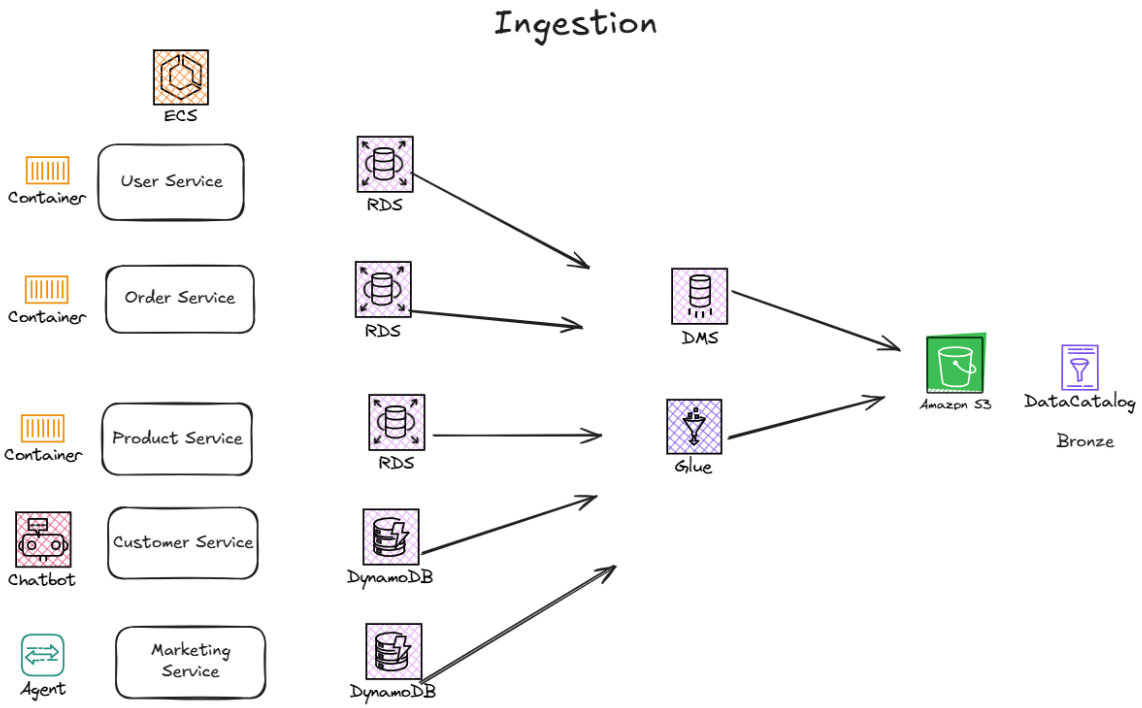
- **Amazon S3** → Stores historical customer data (e.g., old sales records, logs).

Ingestion data S3

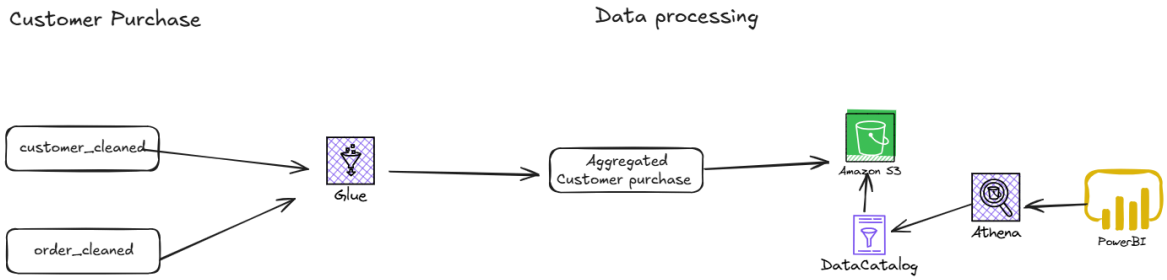
- **ETL & Data Processing**:
  - **AWS Glue** → Extracts, transforms, and loads (ETL) data from different sources into a structured format.
  - **AWS Glue Catalog** → Serves as the metadata store.
- **Query & Analytics**:
  - **Amazon Athena** → Enables ad hoc SQL queries.
  - **Amazon QuickSight** → (Optional) For visualization and reporting.

- **User Service** (MySQL/PostgreSQL) → Customer profiles, login history
- **Order Service** (MySQL/PostgreSQL) → Purchases, payments, refunds
- **Product Service** (MySQL) → Product details, inventory

- **Marketing Service** (DynamoDB/API) → Campaign clicks, emails, promotions
- **Customer Service** (DynamoDB/API) → Support tickets, chatbot interactions



Scenario	Suggested Frequency	Why
Trend Analysis	Daily or Weekly	Daily gives near-real-time insight
Fraud Detection Query	Daily or Hourly	Should be more frequent for responsiveness
Omni-Channel Engagement Analysis	Daily	Track cross-channel activity promptly
Churn Prediction Query	Weekly	Enough if based on rolling engagement
Customer Purchase Behavior	Daily	Useful for fast-moving product or marketing alignment



1. Customer Purchase Behavior Analysis

Goal: Identify high-value customers and their buying patterns.

Optimizations Used:

✅ **Partition by order_date** for efficient filtering

✅ **Use window functions** for ranking

✅ **Limit the dataset to recent orders**

```
WITH customer_spending AS (
    SELECT
        customer_id,
        SUM(order_total) AS total_spent,
        COUNT(order_id) AS total_orders,
        MAX(order_date) AS last_purchase_date
    FROM orders
    WHERE order_date >= date_add('day', -365, current_date)  -- Last 1 year
    GROUP BY customer_id
),
customer_ranking AS (
    SELECT
        customer_id,
        total_spent,
        total_orders,
        last_purchase_date,
        RANK() OVER (ORDER BY total_spent DESC) AS spending_rank
    FROM customer_spending
```

```
)
SELECT *
FROM customer_ranking
WHERE spending_rank <= 100;  -- Get top 100 customers
```

💡 **Why this works?**

- Using **RANK()** avoids a costly `ORDER BY` on the full dataset.
- **Pre-aggregating with `customer_spending` CTE** reduces query complexity.

2. Churn Prediction Query

Goal: Identify customers at risk of churn based on inactivity and order decline.

Optimizations Used:

✅ **Partition by customer_id for faster lookups**

✅ **Use LAG() function** to calculate order gaps

```
WITH customer_activity AS (
    SELECT
        customer_id,
        order_id,
        order_date,
        LAG(order_date) OVER (PARTITION BY customer_id ORDER BY order_dat
e) AS prev_order_date
    FROM orders
),
churn_risk AS (
    SELECT
        customer_id,
        COUNT(order_id) AS total_orders,
        MAX(order_date) AS last_order_date,
        DATEDIFF('day', MAX(order_date), current_date) AS days_since_last_purc
hase,
```

```
        AVG(DATEDIFF('day', order_date, prev_order_date)) AS avg_order_gap
    FROM customer_activity
    GROUP BY customer_id
)
SELECT *
FROM churn_risk
WHERE days_since_last_purchase > (avg_order_gap * 2)  -- Customers inactiv
e for double their average gap
ORDER BY days_since_last_purchase DESC;
```

💡 **Why this works?**

- **Using** `LAG()` eliminates the need for costly self-joins.
- **Pre-calculating gaps** in `customer_activity` speeds up filtering.

---

3. Omni-Channel Engagement Analysis

Goal: Measure engagement levels by combining email, ads, and purchase data.

Optimizations Used:

✅ **Pre-aggregated views** for each data source

✅ **Minimize joins with coalesced IDs**

```
WITH email_engagement AS (
    SELECT
        customer_id,
        COUNT(email_open) AS emails_opened,
        COUNT(email_click) AS emails_clicked
    FROM marketing_events
    WHERE event_date >= date_add('day', -90, current_date)  -- Last 90 days
    GROUP BY customer_id
),
ad_engagement AS (
    SELECT
        customer_id,
```

```
        COUNT(ad_click) AS ad_clicks,
        COUNT(ad_view) AS ad_views
    FROM ad_events
    WHERE event_date >= date_add('day', -90, current_date)
    GROUP BY customer_id
),
purchase_activity AS (
    SELECT
        customer_id,
        COUNT(order_id) AS purchases,
        SUM(order_total) AS total_spent
    FROM orders
    WHERE order_date >= date_add('day', -90, current_date)
    GROUP BY customer_id
)
SELECT
    COALESCE(e.customer_id, a.customer_id, p.customer_id) AS customer_id,
    COALESCE(emails_opened, 0) AS emails_opened,
    COALESCE(emails_clicked, 0) AS emails_clicked,
    COALESCE(ad_clicks, 0) AS ad_clicks,
    COALESCE(ad_views, 0) AS ad_views,
    COALESCE(purchases, 0) AS purchases,
    COALESCE(total_spent, 0) AS total_spent
FROM email_engagement e
FULL OUTER JOIN ad_engagement a ON e.customer_id = a.customer_id
FULL OUTER JOIN purchase_activity p ON COALESCE(e.customer_id, a.custo
mer_id) = p.customer_id
ORDER BY total_spent DESC;
```

💡 **Why this works?**

- `COALESCE()` **ensures all customers are included** even if they have data in only one source.
- `FULL OUTER JOIN` **merges engagement data** without excessive JOIN operations.

---

4. Fraud Detection Query

Goal: Detect anomalies in login and order behavior.

Optimizations Used:

✅ **Partitioning by login_date**

✅ **JOIN only when necessary**

```
WITH suspicious_logins AS (
    SELECT
        customer_id,
        COUNT(DISTINCT ip_address) AS unique_ips,
        COUNT(login_attempt) AS total_attempts
    FROM user_logins
    WHERE login_date >= date_add('day', -30, current_date)
    GROUP BY customer_id
    HAVING unique_ips > 3 OR total_attempts > 10  -- Multiple IPs or too many l
ogins
),
high_risk_orders AS (
    SELECT
        customer_id,
        COUNT(order_id) AS order_count,
        SUM(order_total) AS total_spent
    FROM orders
    WHERE order_date >= date_add('day', -30, current_date)
    GROUP BY customer_id
    HAVING total_spent > 5000 OR order_count > 5  -- High-value transactions
)
SELECT DISTINCT s.customer_id
FROM suspicious_logins s
JOIN high_risk_orders h ON s.customer_id = h.customer_id;
```

💡 **Why this works?**

- **Pre-filtering risky logins & orders separately** makes joins faster.

- **Avoids scanning entire datasets** by focusing only on the last 30 days.

---

5. Trend Analysis

we need to
combine historical and real-time customer data from multiple sources

- **Historical data from Amazon S3**
- **Current transactional data from Amazon RDS (MySQL/PostgreSQL)**
- Seasonal Sales Patterns- Identify peak months or quarters
- Product Performance- Top-selling products over time
- High-value customers and their trends

Identify
Peak Sales Months

```
SELECT
  date_format(order_date, '%Y-%m') AS year_month,
  SUM(sales_amount) AS total_sales
FROM sales_table
GROUP BY year_month
ORDER BY total_sales DESC
LIMIT 12;
 This shows the 12 months with the highest sales.
```

Top-Selling Products Over Time

```
SELECT
  product_id,
  date_format(order_date, '%Y-%m') AS year_month,
  SUM(sales_amount) AS monthly_product_sales
FROM sales_table
GROUP BY product_id, year_month
ORDER BY product_id, year_month;
```

Monthly Trend for Top N Customers:

```
WITH top_customers AS (
  SELECT customer_id
  FROM sales_table
  GROUP BY customer_id
  ORDER BY SUM(sales_amount) DESC
  LIMIT 5
)
SELECT
  s.customer_id,
  date_format(s.order_date, '%Y-%m') AS year_month,
  SUM(s.sales_amount) AS monthly_sales
FROM sales_table s
JOIN top_customers t ON s.customer_id = t.customer_id
GROUP BY s.customer_id, year_month
ORDER BY s.customer_id, year_month;
```

Power BI with AWS Athena

Integrating Power BI with AWS Athena requires setting up an ODBC connection and configuring Power BI to connect to Athena. Follow these steps:

Step 1: Install the Amazon Athena ODBC Driver

1. Download the **ODBC Driver for Amazon Athena** from the AWS ODBC driver page.
2. Install the driver on your local machine (Windows or macOS).

Step 2: Configure the ODBC Data Source

1. Open **ODBC Data Source Administrator** (Windows: Search for "ODBC" in the Start menu).
2. Go to the **System DSN** tab and click **Add**.
3. Select **Simba Athena ODBC Driver** and click **Finish**.
4. Configure the connection:
   - **Data Source Name (DSN)**: `Athena_PowerBI`
   - **Description**: Any description (optional)
   - **AWS Region**: Select the region where your Athena database is located.
   - **S3 Output Location**: Provide an S3 bucket where Athena query results will be stored (e.g., `s3://your-bucket-name/athena-results/`).
   - **Authentication Options**:
     - If using **IAM credentials**, enter the AWS **Access Key** and **Secret Key**.
     - If using **IAM roles (SSO or STS)**, configure the **Profile Name**.
   - **Catalog**: `AwsDataCatalog` (default Glue catalog)
   - **Database**: Choose the Athena database you want to connect to.
5. Click **Test** to verify the connection.
6. Click **OK** to save.

Step 3: Connect Power BI to Amazon Athena

1. Open **Power BI Desktop**.

2. Click **Get Data** → **ODBC** → Click **Connect**.
3. Select the **Athena_PowerBI** DSN (created in Step 2) and click **OK**.
4. In the **Navigator** window, select your Athena database and tables.
5. Click **Load** to import the data into Power BI.

Step 4: Build Your Power BI Report

1. Use Power BI's visualization tools to analyze Athena data.
2. Set up **scheduled refresh** if needed (requires a Power BI Gateway).

Step 5: Publish to Power BI Service (Optional)

1. Click **Publish** in Power BI Desktop.
2. Choose your Power BI workspace.
3. Configure an **on-premises data gateway** to refresh Athena data in the Power BI cloud.
