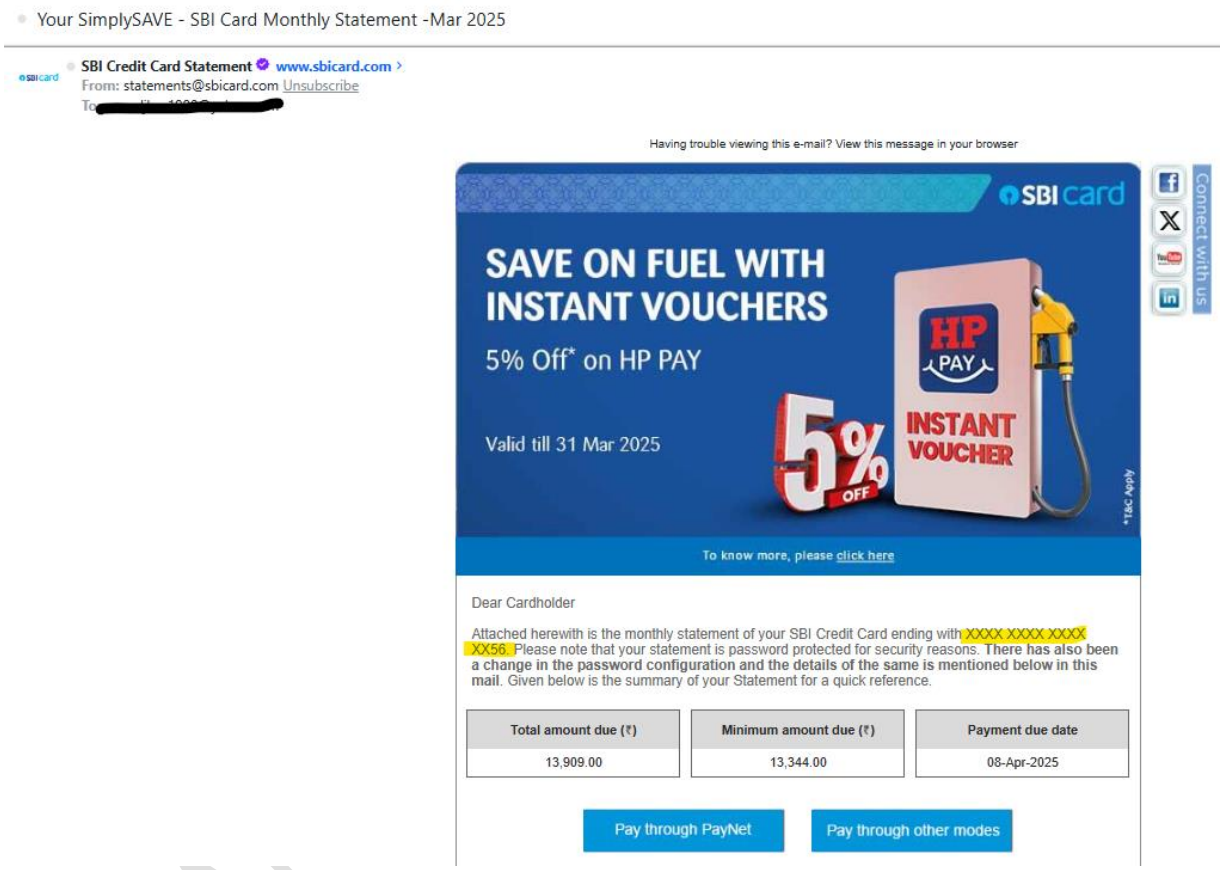
🚀 SECURING CLIENT DATA LIKE A PRO – DYNAMIC MASKING IN SNOWFLAKE & MATILLION! 🔐

Introduction & Background

Have You Ever Noticed Your Credit Card Statement?

Think about the monthly email you receive from your **credit card provider**. It contains your statement details, including **total due amount, minimum payment, and due date**. But have you ever noticed how your **credit card number is never fully visible**? Instead of showing the full number, you only see something like **XXXX-XXXX-XX56**.



Why is that? **Security.**

Financial institutions use this approach to **protect sensitive information**, ensuring that even if someone unauthorized gains access to your email, they cannot misuse your card details. This is a **real-life example of data masking**—hiding sensitive information while still making it usable for legitimate users.

Now, Imagine This in a Data Warehouse

Just like your credit card statement, companies store massive amounts of **Personally Identifiable Information (PII)**—customer names, phone numbers, email addresses, bank details, and more. If this data is exposed to unauthorized employees or external threats, it can lead to **fraud, identity theft, and regulatory violations**.

🚀 SECURING CLIENT DATA LIKE A PRO – DYNAMIC MASKING IN SNOWFLAKE & MATILLION! 🔐

This is where **Dynamic Data Masking (DDM)** comes in. Using **Snowflake and Matillion**, organizations can implement a **role-based masking strategy**, ensuring that:

✅ **Analysts see only masked data** (like XXXX-XXXX-XX56)

✅ **Authorized users (e.g., finance teams) can see full details**

✅ **Data privacy is maintained without affecting business operations**

In this document, we'll explore how **Dynamic Data Masking** helps businesses **secure PII data in real-time**, ensuring compliance while enabling smooth data processing. Let's dive in! 🚀

Why is Data Masking Critical in Today's Industry?

In the digital age, businesses collect and store vast amounts of **sensitive customer data**, including names, addresses, credit card details, and social security numbers. This **Personally Identifiable Information (PII)** is highly valuable but also a major risk if not properly protected. Data breaches, insider threats, and unauthorized access can lead to **financial losses, regulatory penalties, and loss of customer trust**.

To safeguard sensitive information, companies implement **Data Masking**, a technique that ensures that only authorized users can view or work with real data, while others see obfuscated or anonymized values. **Dynamic Data Masking (DDM)** takes this a step further by applying security rules in real-time based on user roles and access permissions.

Industry Use Case: Protecting PII in a Cloud Data Pipeline

Imagine a **global financial institution** that collects **customer transactions** and processes them in Snowflake for analytics, fraud detection, and reporting. The raw data includes:

- **Customer Names**
- **Credit Card Numbers**
- **Account Balances**
- **Phone Numbers & Email Addresses**

While data analysts and business teams need access to insights, they **do not need to see raw PII**. At the same time, compliance teams and fraud investigators may require **full access** for regulatory audits and investigations.

🚀 SECURING CLIENT DATA LIKE A PRO – DYNAMIC MASKING IN SNOWFLAKE & MATILLION! 🔐

How Dynamic Data Masking Works in Snowflake & Matillion

🔹 **Data Ingestion:** Customer transaction data is loaded into Snowflake from **Amazon S3 via Matillion**.

🔹 **Dynamic Masking Implementation:** Using **Snowflake's masking policies**, we define rules to hide or partially mask sensitive fields based on user roles.

🔹 **Role-Based Access:**

- **Analysts & Business Users** → See only masked data (e.g., XXXX-XXXX-1234 instead of a full credit card number).
- **Compliance & Fraud Teams** → See the **actual** data for investigation purposes.
  - 🔹 **Seamless Processing:** Masking happens **in real-time**, meaning **no data duplication or extra transformations** are needed.

Why This is a Must-Know Concept for Data Engineers?

✅ **Ensures Compliance** – Meets regulations like **GDPR, HIPAA, PCI-DSS**
✅ **Prevents Data Breaches** – Reduces risk of **unauthorized access**
✅ **No Performance Impact** – Happens at query execution time in Snowflake
✅ **Works Seamlessly with Matillion** – No need for extra transformations

With **Dynamic Data Masking in Snowflake & Matillion**, organizations can **protect sensitive data, maintain compliance, and empower business users with the insights they need—without compromising security**.

Step 1: Understanding Role-Based Access Control (RBAC) in Snowflake

In Snowflake, **masking policies can only be applied by the table owner or a role with sufficient privileges**. If a table is created by ACCOUNTADMIN, a **developer role cannot apply a masking policy unless explicitly granted the required privileges**.

For this demonstration, we assume:

- **ACCOUNTADMIN** creates the database, schema, and roles.
- **DEVELOPER** creates tables and applies masking policies.
- **DATA_ANALYST** queries the data, but masked values are applied.

🚀 SECURING CLIENT DATA LIKE A PRO – DYNAMIC MASKING IN SNOWFLAKE & MATILLION! 🔐

Step 2: Creating Database, Schema, and Roles

1.**Login as ACCOUNTADMIN** and create a new **database** and **schema**:

```
CREATE DATABASE MASKING_DB;

CREATE SCHEMA MASKING_DB.MASKING_SCHEMA;
```

2. **Create roles for users**:

```
CREATE ROLE DEVELOPER;

CREATE ROLE DATA_ANALYST;
```

3.**Grant necessary privileges** to DEVELOPER:

```
GRANT USAGE ON DATABASE MASKING_DB TO DEVELOPER;

GRANT USAGE ON SCHEMA MASKING_DB.MASKING_SCHEMA TO DEVELOPER;

GRANT CREATE TABLE ON SCHEMA MASKING_DB.MASKING_SCHEMA TO DEVELOPER;

GRANT APPLY MASKING POLICY ON SCHEMA MASKING_DB.MASKING_SCHEMA TO DEVELOPER;
```

- Now, the **DEVELOPER role can create tables and apply masking policies**.

4.**Grant SELECT privileges to DATA_ANALYST but without masking policy privileges**:

```
GRANT USAGE ON DATABASE MASKING_DB TO DATA_ANALYST;

GRANT USAGE ON SCHEMA MASKING_DB.MASKING_SCHEMA TO DATA_ANALYST;

GRANT SELECT ON ALL TABLES IN SCHEMA MASKING_DB.MASKING_SCHEMA TO DATA_ANALYST;
```

- **DATA_ANALYST can only query tables but will see masked values.**

🚀 SECURING CLIENT DATA LIKE A PRO – DYNAMIC MASKING IN SNOWFLAKE & MATILLION! 🔐

Step 3: Creating a Table with Sensitive PII Data

🔹 **Login as DEVELOPER and create a table with sensitive information**:

```
USE ROLE DEVELOPER;
USE DATABASE MASKING_DB;
USE SCHEMA MASKING_DB.MASKING_SCHEMA;

CREATE TABLE CUSTOMER_DATA (
   CUSTOMER_ID INT,
   FULL_NAME STRING,
   EMAIL STRING,
   PHONE_NUMBER STRING
);
```

✅ **Table CUSTOMER_DATA is now created under DEVELOPER role.**

Step 4: Creating a Masking Policy

🔹 **DEVELOPER defines a masking policy to hide PII data based on the user's role**:

```
CREATE MASKING POLICY MASK_PHONE AS (val STRING)
RETURNS STRING ->
   CASE
      WHEN CURRENT_ROLE() IN ('ACCOUNTADMIN', 'DEVELOPER') THEN val
      ELSE 'XXXX-XXX-XXXX'
   END;
```

✅ **Only ACCOUNTADMIN and DEVELOPER can see actual phone numbers.**

🚀 SECURING CLIENT DATA LIKE A PRO – DYNAMIC MASKING IN SNOWFLAKE & MATILLION! 🔐

Step 5: Applying the Masking Policy to the Table

🔹 **Apply the masking policy to the PHONE_NUMBER column**:

```
ALTER TABLE CUSTOMER_DATA

MODIFY COLUMN PHONE_NUMBER

SET MASKING POLICY MASK_PHONE;
```

🔹 If the DEVELOPER role **did not create the table**, this step will **fail** due to a lack of privileges.

🔹 **Solution:** The ACCOUNTADMIN must **grant privileges** to DEVELOPER to apply masking policies:

```
GRANT APPLY MASKING POLICY ON SCHEMA MASKING_DB.MASKING_SCHEMA TO DEVELOPER;
```

✅ Now, the **DEVELOPER role can apply masking policies.**

Step 6: Testing the Masking Policy

🔹 **Login as DEVELOPER and insert sample data**:

```
INSERT INTO CUSTOMER_DATA VALUES

(1, 'John Doe', 'john.doe@example.com', '98765-43210'),

(2, 'Jane Smith', 'jane.smith@example.com', '87654-32109');
```

🔹 **Query the data as ACCOUNTADMIN or DEVELOPER**:

```
SELECT * FROM CUSTOMER_DATA;
```

✅ Output: **Full data is visible**

CUSTOMER_ID	FULL_NAME	EMAIL	PHONE_NUMBER
1	John Doe	john.doe@example.com	98765-43210
2	Jane Smith	jane.smith@example.com	87654-32109

🚀 SECURING CLIENT DATA LIKE A PRO – DYNAMIC MASKING IN SNOWFLAKE & MATILLION! 🔐

🔹 **Query the data as DATA_ANALYST**:

SELECT * FROM CUSTOMER_DATA;

✅ Output: **Masked data is visible**

```
+-------------+-----------+--------------------+---------------+
| CUSTOMER_ID | FULL_NAME | EMAIL              | PHONE_NUMBER  |
+-------------+-----------+--------------------+---------------+
| 1           | John Doe  | john.doe@example.com | XXXX-XXX-XXXX |
| 2           | Jane Smith| jane.smith@example.com | XXXX-XXX-XXXX |
+-------------+-----------+--------------------+---------------+
```

🎯 **Success! The masking policy works based on user roles.**

Step 7: Managing Masking Policies

1.**To remove the masking policy** from a column:

ALTER TABLE CUSTOMER_DATA MODIFY COLUMN PHONE_NUMBER UNSET MASKING POLICY;

2.**To drop the masking policy entirely:**

DROP MASKING POLICY MASK_PHONE;

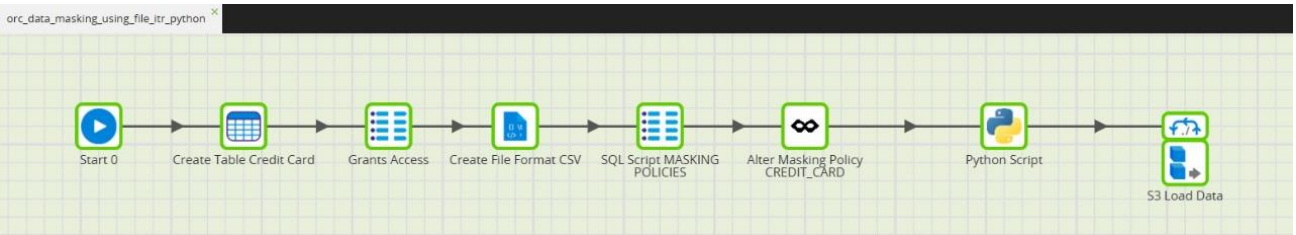
Key Takeaways for Data Engineers

✅ **Masking policies in Snowflake control access to PII dynamically**
✅ **Only the table owner or a privileged role (like ACCOUNTADMIN) can apply masking policies**
✅ **Ensure DEVELOPER has the APPLY MASKING POLICY privilege** before applying policies
✅ **Masking policies protect data in real-time** without duplicating or modifying stored values

🚀 SECURING CLIENT DATA LIKE A PRO – DYNAMIC MASKING IN SNOWFLAKE & MATILLION! 🔐

Step-by-Step Guide to Implementing Dynamic Data Masking in Matillion with Snowflake

The provided Matillion pipeline follows a structured approach to implement **Dynamic Data Masking (DDM) in Snowflake**, ensuring that **sensitive data like credit card details are protected**. Below are the **detailed steps** to accomplish this:



📌 Step 1: Create Table in Snowflake

- 🔹 **Component Used:** Create Table Credit Card (Table Creation Component)
- 🔹 **Purpose:** Create a table in Snowflake to store sensitive credit card information.

💡 SQL Script to Create the Table

```
CREATE TABLE MASKING_DB.MASKING_SCHEMA.CREDIT_CARD (
    ID INT AUTOINCREMENT,
    CUSTOMER_NAME STRING,
    CREDIT_CARD_NUMBER STRING,
    EXPIRY_DATE STRING,
    CVV STRING
);
```

✅ This table will hold credit card details that need to be masked.

📌 Step 2: Grant Necessary Access

- 🔹 **Component Used:** Grants Access (SQL Script Component)
- 🔹 **Purpose:** Ensure that **only authorized roles can apply masking policies**.

💡 SQL Script to Grant Privileges

```
GRANT USAGE ON DATABASE MASKING_DB TO DEVELOPER;
GRANT USAGE ON SCHEMA MASKING_DB.MASKING_SCHEMA TO DEVELOPER;
GRANT CREATE TABLE ON SCHEMA MASKING_DB.MASKING_SCHEMA TO DEVELOPER;
```

🚀 SECURING CLIENT DATA LIKE A PRO – DYNAMIC MASKING IN SNOWFLAKE & MATILLION! 🔐

GRANT APPLY MASKING POLICY ON SCHEMA MASKING_DB.MASKING_SCHEMA TO DEVELOPER;

GRANT SELECT ON ALL TABLES IN SCHEMA MASKING_DB.MASKING_SCHEMA TO DATA_ANALYST;

✅ This ensures that the **developer can create tables and apply masking policies** while the **data analyst can only read masked data**.

📌 Step 3: Create File Format for CSV Data Load

- 🔹 **Component Used:** Create File Format CSV (Component)
- 🔹 **Purpose:** Define a **Snowflake file format** to handle CSV files correctly.

📌 Step 4: Write SQL Script for Masking Policies

- 🔹 **Component Used:** SQL Script MASKING POLICIES (SQL Script Component)
- 🔹 **Purpose:** Define **masking policies** to control access to sensitive columns dynamically.

💡 SQL Script to Create Masking Policies

```
CREATE MASKING POLICY MASK_CREDIT_CARD AS (val STRING)
RETURNS STRING ->
  CASE
    WHEN CURRENT_ROLE() IN ('ACCOUNTADMIN', 'DEVELOPER') THEN val
    ELSE 'XXXX-XXXX-XXXX-XXXX'
  END;
```

✅ **Only authorized roles can see actual credit card numbers; others will see masked values.**

📌 Step 5: Apply the Masking Policy

- 🔹 **Component Used:** Alter Masking Policy CREDIT_CARD (Alter Masking Policy Component)
- 🔹 **Purpose:** Apply the **masking policy to the CREDIT_CARD_NUMBER column**.

✅ If the **developer does not have privileges**, this step will **fail**.
✅ **Solution:** The ACCOUNTADMIN role must grant APPLY MASKING POLICY privileges.

🚀 SECURING CLIENT DATA LIKE A PRO – DYNAMIC MASKING IN SNOWFLAKE & MATILLION! 🔐

📌 Step 6: Use Python Script to Read Latest File Based on Timestamp

- 🔹 **Component Used:** Python Script (Python Component)
- 🔹 **Purpose:** Dynamically find and process the latest CSV file.

💡 **Python Script Logic:**

- Connect to **AWS S3**.
- **Find the latest file** using timestamps.
- Prepare it for **loading into Snowflake**.

✅ This **automatically selects the most recent file** for loading.

📌 Step 7: Load Data into Snowflake from S3

- 🔹 **Component Used:** S3 Load Data (S3 Load Component)
- 🔹 **Purpose: Ingest credit card data from S3 into Snowflake**.

🔎 Testing the Implementation

1. **As ACCOUNTADMIN or DEVELOPER**, run:

```
SELECT * FROM MASKING_DB.MASKING_SCHEMA.CREDIT_CARD;
```

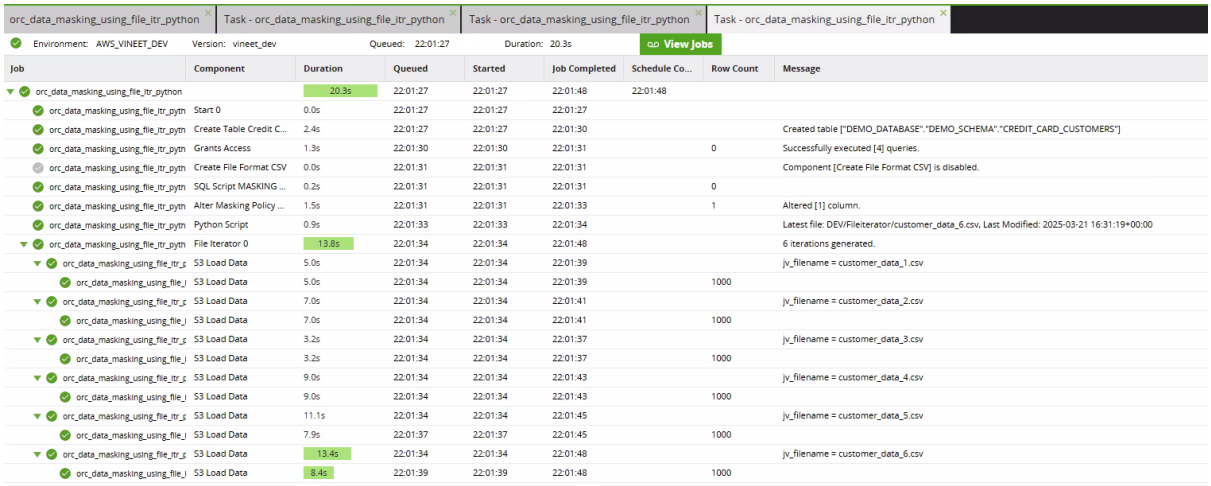
✅ Full **credit card numbers** are visible.

2. **As DATA_ANALYST**, run:

```
SELECT * FROM MASKING_DB.MASKING_SCHEMA.CREDIT_CARD;
```

✅ **Masked values (XXXX-XXXX-XXXX-XXXX)** are shown.

🚀 SECURING CLIENT DATA LIKE A PRO – DYNAMIC MASKING IN SNOWFLAKE & MATILLION! 🔒



orc_data_masking_using_file_itr_python | Task - orc_data_masking_using_file_itr_python | Task - orc_data_masking_using_file_itr_python | Task - orc_data_masking_using_file_itr_python

Environment: AWS_VINEET_DEV Version: vineet_dev Queued: 22:01:27 Duration: 20.3s View Jobs

Job	Component	Duration	Queued	Started	Job Completed	Schedule Co...	Row Count	Message
orc_data_masking_using_file_itr_python		20.3s	22:01:27	22:01:27	22:01:48	22:01:48		
orc_data_masking_using_file_itr_pyth	Start 0	0.0s	22:01:27	22:01:27	22:01:27			
orc_data_masking_using_file_itr_pyth	Create Table Credit C...	2.4s	22:01:27	22:01:27	22:01:30			Created table ["DEMO_DATABASE"."DEMO_SCHEMA"."CREDIT_CARD_CUSTOMERS"]
orc_data_masking_using_file_itr_pyth	Grants Access	1.3s	22:01:30	22:01:30	22:01:31		0	Successfully executed [4] queries.
orc_data_masking_using_file_itr_pyth	Create File Format CSV	0.0s	22:01:31	22:01:31	22:01:31			Component [Create File Format CSV] is disabled.
orc_data_masking_using_file_itr_pyth	SQL Script MASKING ...	0.2s	22:01:31	22:01:31	22:01:31		0	
orc_data_masking_using_file_itr_pyth	Alter Masking Policy ...	1.5s	22:01:31	22:01:31	22:01:33		1	Altered [1] column.
orc_data_masking_using_file_itr_pyth	Python Script	0.9s	22:01:33	22:01:33	22:01:34			Latest file: DEV/FileIterator/customer_data_6.csv, Last Modified: 2025-03-21 16:31:19+00:00
orc_data_masking_using_file_itr_pyth	File Iterator 0	13.8s	22:01:34	22:01:34	22:01:48			6 iterations generated.
orc_data_masking_using_file_itr_s	S3 Load Data	5.0s	22:01:34	22:01:34	22:01:39			jv_filename = customer_data_1.csv
orc_data_masking_using_file_l	S3 Load Data	5.0s	22:01:34	22:01:34	22:01:39		1000	
orc_data_masking_using_file_itr_s	S3 Load Data	7.0s	22:01:34	22:01:34	22:01:41			jv_filename = customer_data_2.csv
orc_data_masking_using_file_l	S3 Load Data	7.0s	22:01:34	22:01:34	22:01:41		1000	
orc_data_masking_using_file_itr_s	S3 Load Data	3.2s	22:01:34	22:01:34	22:01:37			jv_filename = customer_data_3.csv
orc_data_masking_using_file_l	S3 Load Data	3.2s	22:01:34	22:01:34	22:01:37		1000	
orc_data_masking_using_file_itr_s	S3 Load Data	9.0s	22:01:34	22:01:34	22:01:43			jv_filename = customer_data_4.csv
orc_data_masking_using_file_l	S3 Load Data	9.0s	22:01:34	22:01:34	22:01:43		1000	
orc_data_masking_using_file_itr_s	S3 Load Data	11.1s	22:01:34	22:01:34	22:01:45			jv_filename = customer_data_5.csv
orc_data_masking_using_file_l	S3 Load Data	7.9s	22:01:37	22:01:37	22:01:45		1000	
orc_data_masking_using_file_itr_s	S3 Load Data	13.4s	22:01:34	22:01:34	22:01:48			jv_filename = customer_data_6.csv
orc_data_masking_using_file_l	S3 Load Data	8.4s	22:01:39	22:01:39	22:01:48		1000	

🎯 Key Takeaways

✅ **Matillion does not have a built-in component for masking policies, so SQL scripts are used.**

✅ **Role-based access ensures that only authorized users can see unmasked data.**

✅ **Python dynamically selects the latest data file from S3 before loading into Snowflake.**

✅ **Snowflake's dynamic masking ensures real-time security without modifying stored values.**

🚀 Conclusion

This **end-to-end Matillion pipeline** integrates **Snowflake's Dynamic Data Masking (DDM)** with **S3-based data ingestion**, ensuring that **sensitive credit card data is protected at all stages** while maintaining **flexibility and automation**.