**🟢Author:** PARESH RANJAN ROUT

Project Architecture:

* **Which is the correct production scenario?**

✅ **In modern cloud data engineering,** the **first scenario** (Git → Data Lake Bronze) is considered the **recommended and scalable approach**: **GitHub → Bronze Container (Data Lake) → Silver Layer (Transformation) → (Further processing)**

✅ **Characteristics:**

* **Direct ingestion into Data Lake.**
* **Storage-first architecture** (central raw data repository).
* Very common in **modern data lake / medallion architectures**.
* Easy to build a **schema-on-read** pipeline.
* **Ideal when:**
  + You want to decouple raw ingestion and transformation.
  + You have large files or semi-structured data.
  + You plan to reprocess data multiple times (e.g., raw zone retained for audit).

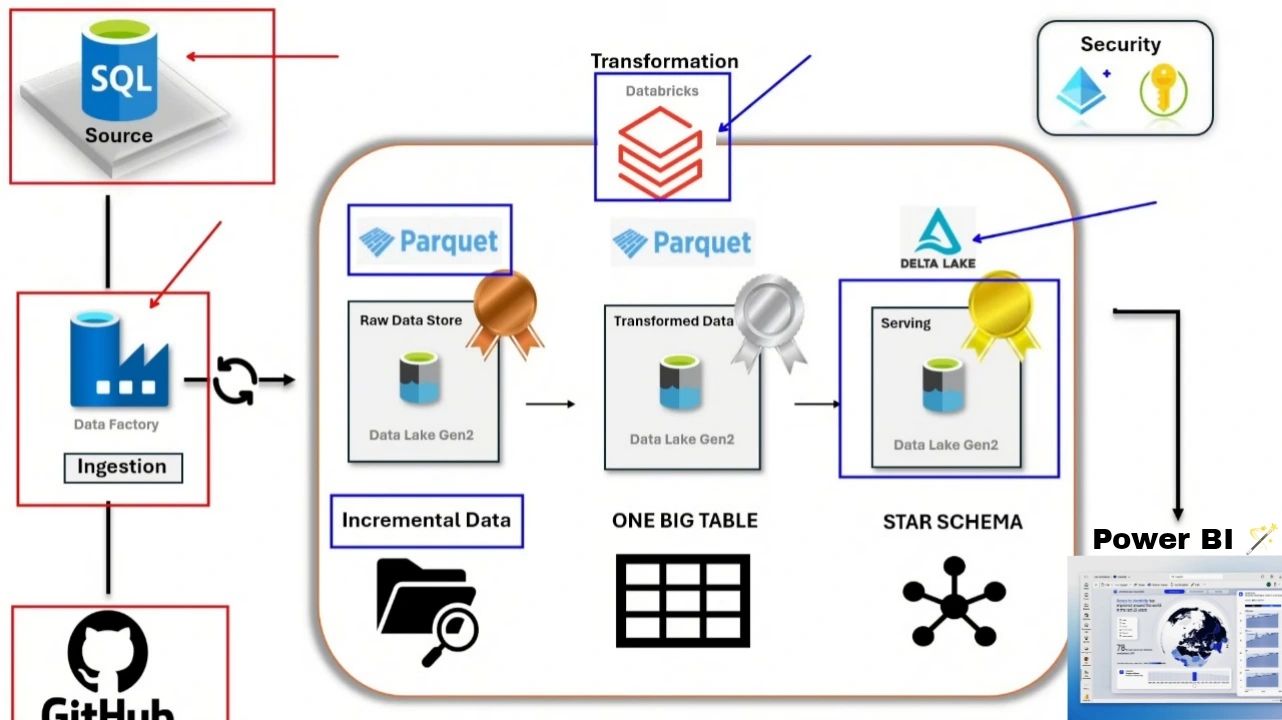
✅ **The second scenario** is also valid **when you have reasons**, e.g.: **GitHub → SQL Server (Landing/Staging DB) → Bronze Layer in Storage → Silver Layer (Transformation)**

✅ **Characteristics:**

* **Data lands first in a database.**
* Then exported to storage for further processing.
* More common in **traditional ETL pipelines** or when:
  + You must do **initial validation or cleansing** in SQL before moving to the lake.
  + You use SQL Server as a staging area or operational store.
  + Downstream consumers expect SQL tables.

🎯 **This is more common in hybrid architectures where:**

* Legacy systems rely on SQL Server.
* Governance requires validated data in a DB.
* Or you have operational systems that must land first in relational storage.



✅ **This is a *brilliant* question—very mature thinking.**

You are **100% right** that when you land data in SQL Server and do cleansing, you are *already transforming*.

But let’s clarify why **organizations still build further Silver and Gold layers in the lake** even if they did some transformations in SQL Server.

**🎯 ✅ Why do we still do Silver/Gold after SQL Server staging?**

Here’s the key idea:

✅ **Ingesting into SQL Server and doing validation = Operational or Staging Transformations.**

✅ **Silver and Gold transformations = Analytical, Business-Ready, or Aggregated Views.**

Let’s break it down in a simple way:

**🟢 Phase 1: Data Quality & Validation in SQL Server**

* **What you do:**
  + Enforce schema (columns, types)
  + Validate referential integrity
  + Filter obviously invalid records
  + Apply some basic transformations (e.g., trimming strings, fixing dates)
* **Goal:**
  + Make sure the data is *correct* and *clean enough* to be trusted as raw input.

✅ This is still **raw or lightly transformed data.**

**🟢 Phase 2: Silver Layer Transformations**

* **What you do:**
  + Standardize naming conventions across datasets
  + Join reference tables (e.g., region names, product categories)
  + De-duplicate data across files or sources
  + Normalize or denormalize tables
  + Restructure raw tables into clean entities

✅ The output is **clean, conformed data** ready for business logic.

**🟢 Phase 3: Gold Layer Transformations**

* **What you do:**
  + Apply business aggregations (e.g., daily sales per product)
  + Build subject-oriented datasets for reporting
  + Summarize and calculate KPIs
  + Format data for dashboards or data marts

✅ The output is **ready-to-consume analytics datasets.**

✅ **Why not just do everything in SQL Server and store Gold there?**

Good question—some companies do exactly that (especially if they are SQL-centric).

But modern cloud data engineering prefers **Data Lakehouse** design for several reasons:

**🎯 Advantages of Still Moving to Data Lake Layers**

✅ **1️⃣ Scale and Cost**

* Storing raw and processed data in SQL Server gets expensive.
* Data Lake (ADLS Gen2) is far cheaper and scales infinitely.

✅ **2️⃣ Flexibility**

* You can store multiple formats: CSV, Parquet, JSON.
* Easy to evolve schema over time.

✅ **3️⃣ Decoupled Compute**

* You can process data in Spark/Databricks/Synapse.
* No dependency on SQL Server capacity.

✅ **4️⃣ Replayability**

* You keep raw bronze files forever.
* You can reprocess Silver/Gold if logic changes.

✅ **5️⃣ Unified Platform**

* One place to land data from APIs, logs, IoT, SQL Server—all together.
* Easier to manage lineage and governance.

✅ **6️⃣ Cloud-Native Analytics**

* Many tools (Power BI, ML models) work directly on Data Lake.

✅ Short Answer to your question:

Yes, you can do cleansing and some transformations in SQL Server. But Silver and Gold layers typically apply richer business logic, historical aggregations, and prepare datasets optimized for analytics. That’s why even after SQL Server staging, modern pipelines still move data to a Data Lake medallion architecture.

**🎯 If SQL Server storage is expensive, why don’t we always land data directly in the Data Lake Bronze layer?**

**🟢 Reason 1: Data Validation & Business Rules in SQL Server**

✅ **SQL Server can do things the Data Lake can’t easily do on raw files:**

* Enforce constraints (NOT NULL, UNIQUE)
* Referential integrity checks (foreign keys)
* Immediate transactional consistency (ACID)
* Rejection of invalid rows automatically

💡 *If your company’s processes require guaranteed clean data before anything happens downstream,* **SQL Server is the fastest way to get it.**

**🟢 Reason 2: Legacy Systems Compatibility**

✅ Many enterprise apps, finance tools, and operational reporting tools **only know how to talk to SQL Server**:

* SSIS
* SSRS
* Power BI DirectQuery
* ERP systems
* CRM integrations

💡 *If you bypass SQL Server and put raw files in the Data Lake, you break all those existing integrations.*

**🟢 Reason 3: Familiarity and Skills**

✅ Most data teams (especially in traditional companies) already know SQL Server inside out:

* Security configuration
* Role-based access
* Backup/recovery

💡 *Moving directly to Data Lake requires upskilling, building governance policies, and reworking many processes.*

**🟢 Reason 4: Change Data Capture (CDC) and Incremental Loads**

✅ SQL Server has **mature built-in CDC**, which:

* Captures inserts, updates, deletes
* Makes incremental ingestion much simpler
* Tracks versioning of data easily

💡 *Data Lake does not automatically track changes—you’d have to build complex logic or use Delta Lake.*

**🟢 Reason 5: Transactional Ingestion**

✅ If your API or source system pushes records one by one (or in batches that must be atomic), SQL Server:

* Guarantees *all-or-nothing* inserts
* Provides transactional rollback

💡 *In a Data Lake, you have to manage partial file uploads and eventual consistency.*

**🟢 Reason 6: Operational Reporting Needs**

✅ Some teams need to:

* Query ingested data **instantly** via SQL
* Join with other operational tables
* Use existing BI dashboards

💡 *If you land in Data Lake first, you have extra steps before you can expose the data to business users.*

*.*

**🎯 Why are companies still moving to Data Lake-first architectures?**

✅ **Because in many cases the benefits outweigh the downsides:**

* Cheap storage
* Schema flexibility
* Native support for big data
* Decoupled compute (Databricks, Spark)
* Unified platform for all data types (structured, semi-structured, unstructured)

✅ **That’s why you often see hybrid patterns:** **API → SQL Server Staging (validation) → Data Lake Bronze**

✅ **That’s why you often see hybrid patterns:** **API → Data Lake Bronze (raw) → Data Lake Silver (cleaned) → Data Lake Gold (aggregated)**

🎯 **When should you SKIP SQL Server and land data DIRECTLY in the Data Lake?**

**🎯 Why does “instant relational queries” matter?**

Imagine you are a company that gets new data every 5 minutes.

✅ **If you need instant querying:**

* The moment data lands, business users want to **immediately** run reports or dashboards.

**Example:**

* Your CFO says:  
  *“I want to see today’s sales by region right now.”*
* So you must be able to run: Query -: {SELECT SUM(SalesAmount)

FROM Sales

WHERE SaleDate = TODAY}

* immediately after the data arrives.

At the time - **This is where SQL Server is great.**

* Data loads → ready for queries in seconds.
* Enforced schema.
* Familiar relational model.

✅ **If you DON’T need instant querying:**

* You just collect the data and will clean/analyze it **later** (maybe tomorrow or in a few hours).
* You don’t care about running queries right away.
* Your processing happens in batches (e.g., nightly jobs).

**Example:**

* You are collecting logs, clickstream, IoT sensor data.
* No one needs to see each new file the minute it arrives.
* You will process everything in Spark later.

👉 **This is where Data Lake shines.**

* Super cheap to store.
* No rigid schema required upfront.
* You can run transformations later (Spark/Synapse/Databricks).

🎯 ✅ **Fundamental Concepts You Now Understand:**

**🟢 1️⃣ Data Landing Strategies:**

* Why you might ingest data into **SQL Server** (instant querying, validation, existing reporting).
* Why you might ingest directly into **Data Lake** (scale, cost, ML readiness).

**🟢 2️⃣ Medallion Architecture Layers**

* **Bronze** (raw data)
* **Silver** (cleaned & conformed data)
* **Gold** (business-ready aggregates)

**🟢 3️⃣ Data Orchestration**

* Azure Data Factory (ADF) pipelines
* Activities: Copy, Lookup, ForEach
* Parameterization of source and sink
* Dynamic pipelines vs. static pipelines

**🟢 4️⃣ Storage Concepts**

* Azure Data Lake vs. Blob Storage
* Containers and folders
* File formats (CSV, Parquet)

**🟢 5️⃣ ETL/ELT Processing**

* Loading raw data
* Transformation layers
* Validations in SQL Server vs. Data Lake

**🟢 6️⃣ Data Security and Access**

* Linked Services
* Integration Runtimes (Azure, Self-hosted)
* Private vs. public endpoints

**🟢 7️⃣ Networking Fundamentals**

* IP addresses (public/private).
* Encryption in transit.
* Data movement from Git/API to cloud.

**1st - Phase of Project: -**

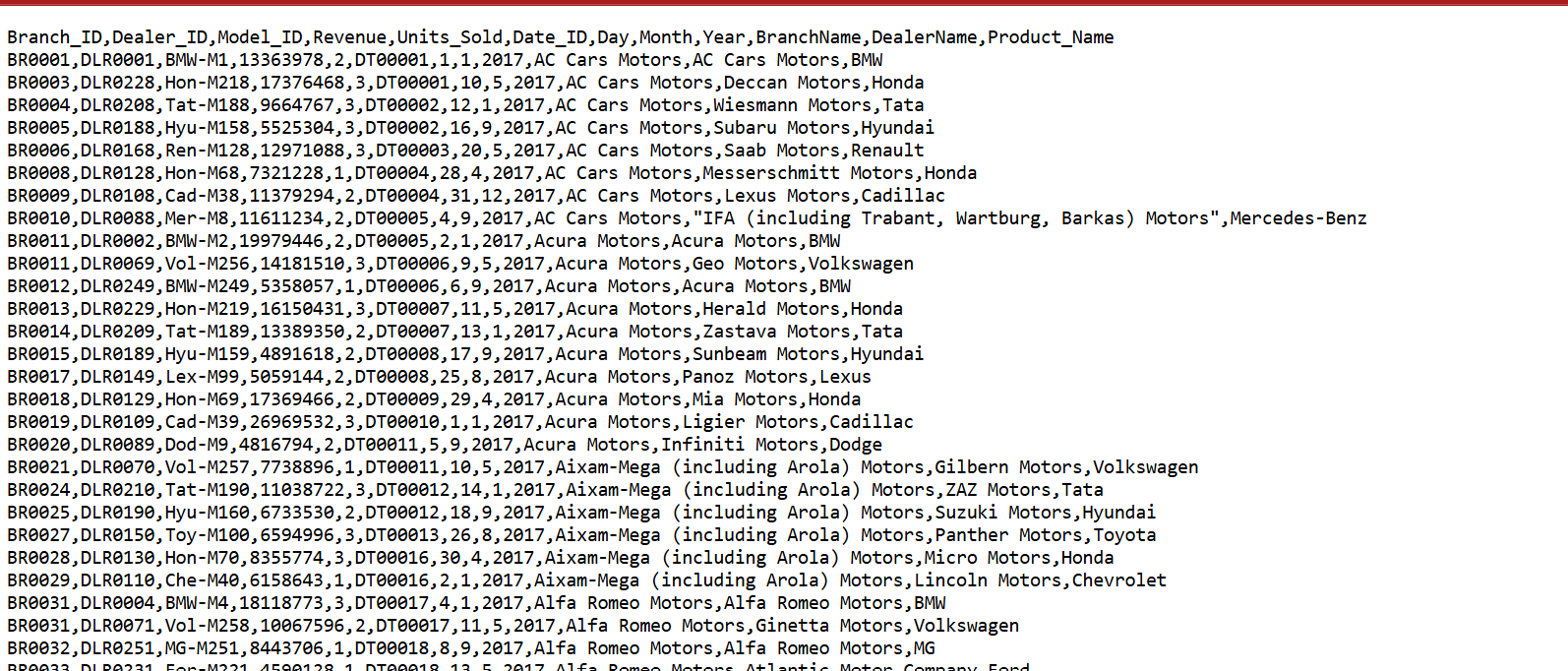
* 1. **I created an SQL Database, SQL Server**
  2. **Storage, - 3 Container, Bronze, Silver, Gold**
  3. **Azure Data Factory**
  4. **External Source (API-GitHub)**

**🎯 When your source is SQL Database, here’s the usual process:**

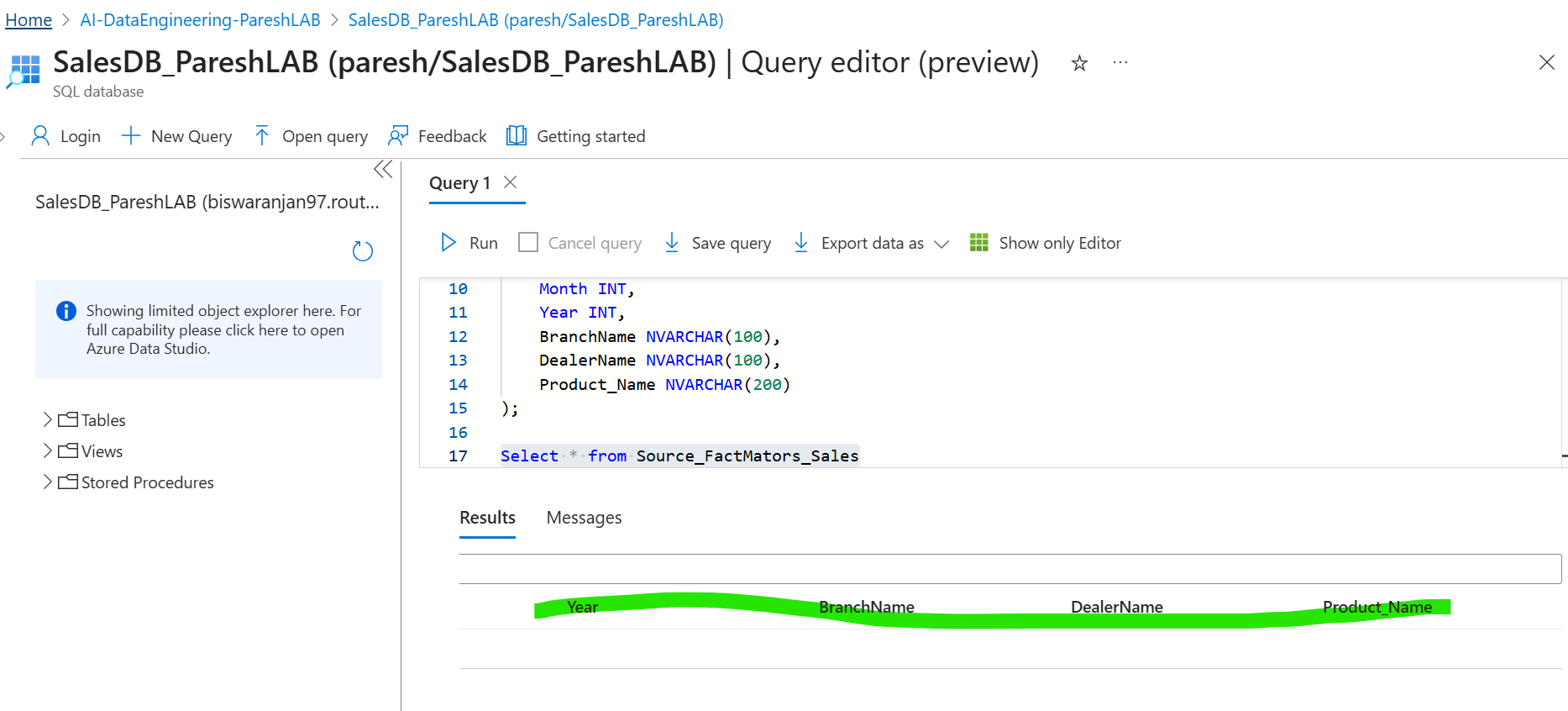
**1️⃣ Create the table (schema) in SQL Database**

* **This defines how your data is structured (columns, data types, constraints).**
* **You can think of this as “preparing a container” to hold the data.**

**✅ So yes—before the data lands, you must define the schema/table.  
✅ After that, pipelines fetch and move the data as per your project design.**

****

* Example: CREATE TABLE SalesTransactions (
* Branch\_ID NVARCHAR(10),
* Dealer\_ID NVARCHAR(20),
* Model\_ID NVARCHAR(20),
* Revenue BIGINT,
* Units\_Sold INT,
* Date\_ID NVARCHAR(20),
* Day INT,
* Month INT,
* Year INT,
* BranchName NVARCHAR(100),
* DealerName NVARCHAR(100),
* Product\_Name NVARCHAR(200)
* );



* Created the Schema

2️⃣ **Load or ingest data into this table**

* If you are **fetching from an external file** (e.g., your CSV), you can:
  + Use **Azure Data Factory** to copy the data into SQL.
  + Use tools like **SQL Server Management Studio** or **BULK INSERT** if local.

3️⃣ **Once data is stored in the SQL table, it becomes your source**

* Your pipelines (ADF, Synapse) **read from this table**.
* The data can then be processed, transformed, or moved downstream (e.g., into your Bronze Data Lake).

✅ **Why doesn’t SQL Server do this automatically?**

* Because relational databases must validate:
  + Column data types (int, string, date).
  + Length.
  + Nullability.
  + Constraints.
* It cannot safely guess without your control.

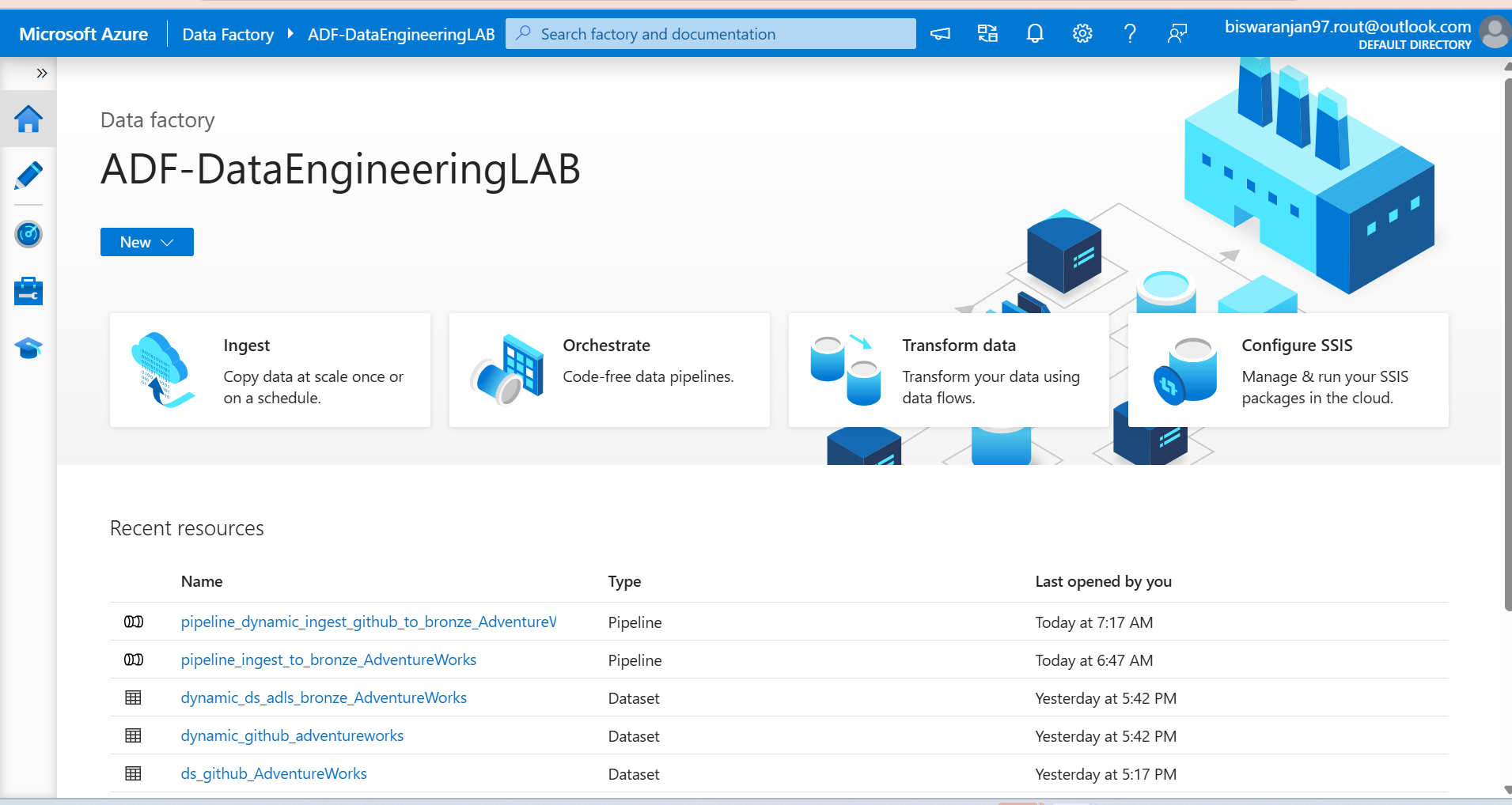
💡 **Recommended Practice:**  
If you want to automate this **with minimal effort**:

* **Azure Data Factory Mapping Data Flow**
  + Ingest the CSV.
  + Infer schema.
  + Auto-create the destination table.
* Or **use Databricks / Synapse Spark to infer schema** and write to SQL.

✅ **Summary Answer to Your Question:**

**No, by default SQL DB will not auto-create schema by simply landing a file.**  
But:  
**Yes, you can automate schema inference and table creation with ADF Mapping Data Flows, Synapse Spark, or custom scripting.**

**Then Move to Azure Data Factory to creates the Pipeline: -**



So we will build the an incremental pipeline: before build it let’s understand about it.

**SQL**

**ADF**

**API**

**Building the incremental Pipeline**

**Incremental pipeline**

**Initial Load Pipeline**

**🟢 ✅ The Correct Production Logic**

✅ **Exactly this:**

*First fully new data will load from the API to SQL DB.  
Then CDC will capture all changes (insert/update/delete).  
It saves the date and details of changes.  
Next time the pipeline runs, it fetches only new or changed data since the last processed point.*

🎯 What is an Incremental Pipeline?

* An **incremental pipeline** is a data pipeline that **only loads the new or changed data since the last successful load**, rather than reloading everything every time.

💡 Why is Incremental Ingestion Important?

Loading everything (“full load”) each time is:

* **Slow** (especially with big tables—imagine reloading 5TB daily).
* **Costly** (compute and storage).
* **Error-prone** (duplicates and reprocessing).
* **Inefficient** (why reprocess unchanged rows?).

✅ **Incremental pipelines** solve this by:

* **Tracking what’s new or changed.**
* **Only ingesting the delta** (increment).
* **Appending or merging** it into your storage layer

🚀 How Incremental Pipelines Work in Real Time?

Let’s break it down step by step:

1️⃣ **Detect New or Changed Data**

* Using:
  + **Timestamps** (e.g., LastModifiedDate > LastLoadDate)
  + **Watermark Columns** (incrementing IDs or sequence numbers)
  + **Change Data Capture (CDC)** (logs what rows were inserted/updated/deleted)
  + **Change Tracking** (in SQL Server/Azure SQL)
  + **File Names or Metadata** (for files landing in storage)

2️⃣ **Extract Incremental Data**

* Query or filter only rows since the last load.
* Example SQL: SELECT \* FROM Sales

WHERE LastModifiedDate > @LastLoadedTimestamp

Or from blob storage: only pick new files.

3️⃣ **Load into Staging**

* Store the incremental data in a staging area (bronze layer).

4️⃣ **Merge into Target**

* Append to your main table **or**
* Use *upserts* (insert new rows, update existing).

5️⃣ **Update the Watermark**

* Save the **last loaded timestamp or ID** so the next run knows where to continue.

6️⃣ **Data Validation**

* Check counts, consistency, duplicates.

🛠️ Tools to Build Incremental Pipelines in Azure?

 **Azure Data Factory**

* Lookup and watermark table.
* Incremental copy with filters.

 **Azure Synapse Pipelines**

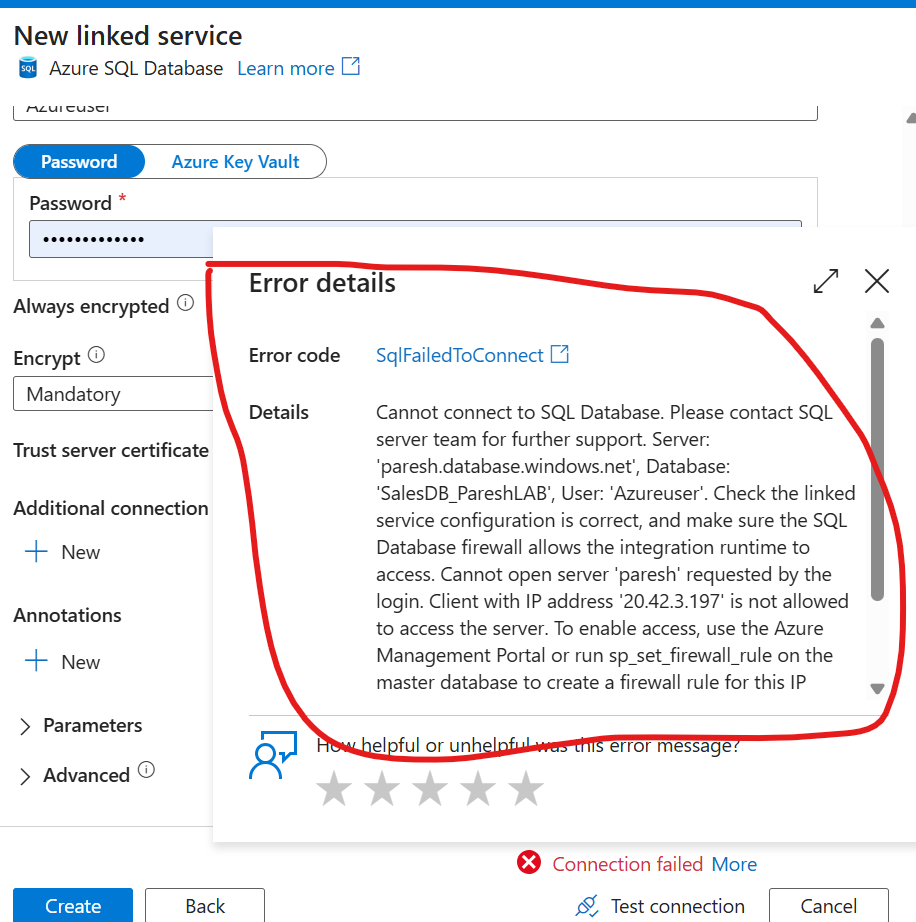
* Similar to ADF.

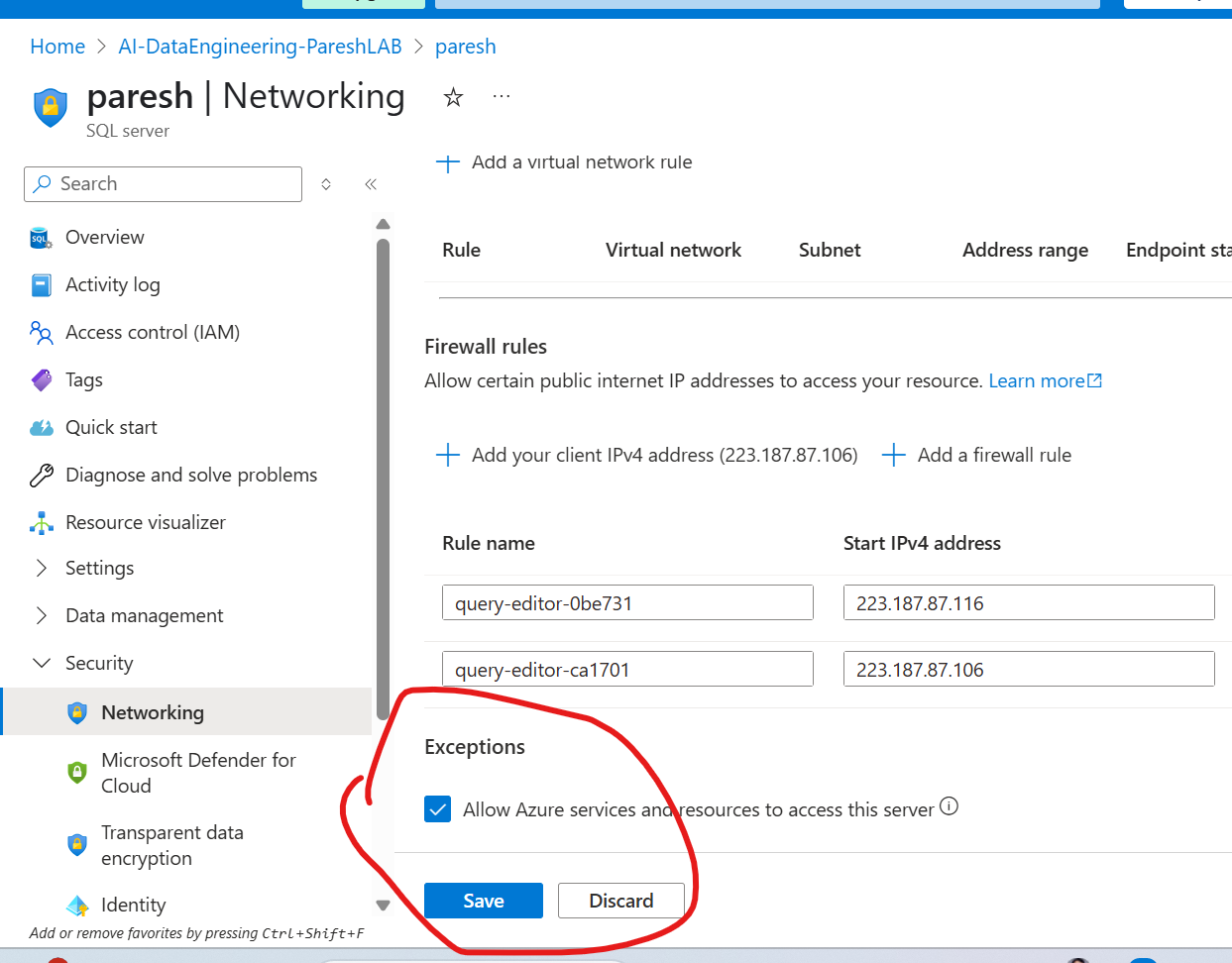
 **Azure Databricks**

* Structured streaming or batch incremental loads.

 **SQL Server CDC/Change Tracking**

* Native support for capturing changes.
* When I am stabilizing the Connection between Azure SQL DB to ADF, below error occur to fix the issue.



* 1. There was a firewall issue preventing access. To resolve it, we need to enable the option to **allow** Azure resources to connect to the Azure SQL Database. This will ensure the connection works going forward. Below images for your review.
  2. 

✅ **This is an *excellent* and very practical question—let’s tackle it step by step so you fully understand the best approach for scaling.**

**🎯 Your Question Recap (so it’s clear):**

In my last project:

* I manually created a JSON file with entries like this:

json

CopyEdit

{

"p\_rel\_url": "...",

"p\_sink\_folder": "...",

"p\_sink\_file": "..."

}

* Then used Lookup to load it and ForEach to loop over them.
* But:
  + Creating all those JSON entries manually for many datasets was **slow and painful**.
  + In real scenarios, there could be **hundreds or thousands of files**.
  + You are wondering: **Can we put 1000 entries into one JSON?**
  + Are there better approaches to **automate** this?

✅ **Short Answer:**  
Yes, you *could* put 1000 entries in a single JSON.  
But in practice, **that is not the best approach.**

**🟢 Why is a single giant JSON file not optimal?**

While technically allowed, this has drawbacks:

* Hard to maintain (huge file).
* Prone to manual errors.
* No dynamic discovery of new files.
* You’d have to re-edit JSON every time a file is added.
* Lookup will read all 1000 records every time (could slow pipeline start).

**✅ Best Practice Approach for Large-Scale Dynamic Pipelines**

If you expect **hundreds or thousands of datasets**, **don’t hardcode JSON.**

Instead, you can **automate discovery** so you never have to write JSON manually.

**🎯 How to Automate File Discovery in Production**

**🟢 1️⃣ Use Get Metadata Activity with Child Items**

✅ This is the most scalable approach:

* You point Get Metadata to your GitHub folder (or blob storage container).
* It automatically lists **all files in that path**.
* Returns an array of file names (childItems).
* You pass this array to ForEach.
* ForEach loops over them dynamically.

**Example Flow:**

pgsql

CopyEdit

[Get Metadata: List Files]

↓

[ForEach (item().name)]

↓

[Copy Activity]

**Benefits:**  
✅ No JSON files.  
✅ Zero manual maintenance.  
✅ Always processes *whatever files are present*.

**🟢 2️⃣ Use naming conventions + parameters**

If your files always follow a pattern (e.g., Sales\_YYYY\_MM\_DD.csv), you can:

* Build the file names dynamically using expressions (date functions).
* No need to list them explicitly.

**🟢 3️⃣ Use Azure Data Lake Gen2 File Events (Advanced)**

* You can trigger pipelines **automatically when new files arrive**, using Event Grid.
* This is called *event-driven ingestion*.

**✅ Professional Enterprise Flow Example**

**Dynamic Incremental Ingestion:**

1. **Get Metadata**:
   * Lists all files in the folder.
2. **Filter Activity**:
   * (Optional) filter only new or matching files.
3. **ForEach Activity**:
   * Iterates over each file.
   * Dynamically constructs the relative URL.
4. **Copy Activity**:
   * Loads the data into Bronze.
   * Creates separate folders if needed.

**🎯 One-Liner Answer to Your Question:**

**If you have 1,000 datasets, don’t build a huge JSON manually—use Get Metadata to dynamically list files and drive ForEach looping automatically.**

✅ **Advanced Concept (as you asked):**  
This approach is called **Metadata-Driven Dynamic Ingestion Pipelines**, and it’s a **best practice in large-scale data engineering**.

**Get Metadata in ADF: in Detail workflows**

**🎯 What is Get Metadata in ADF?**

✅ **Think of it as a little robot.**

Every time your pipeline runs, the robot goes to your folder and **checks what files are there right now**.

It doesn’t read a file’s contents.  
It doesn’t guess.  
It **asks the folder:**

“Hi folder, please give me a list of everything inside you.”

✅ **What does it get back?**

It gets a **list of all files and folders**, for example:

diff

- Sales\_Jan.csv

- Sales\_Feb.csv

- Sales\_March.csv

✅ **Why is this called dynamic?**

Because **every time you run the pipeline**, the robot checks again:

* Maybe a new file has arrived.
* Maybe a file was deleted.
* Maybe nothing has changed.

👉 So you **never have to write or update a JSON file yourself.**

✅ **How does this help you?**

1️⃣ **You don’t have to manually list files.**  
2️⃣ You get an **automatic list of files** every time.  
3️⃣ You can loop through the list using **ForEach Activity**.

✅ **How does the looping work?**

Imagine Get Metadata comes back and says:

diff

Here are 3 files:

- File1.csv

- File2.csv

- File3.csv

Then ForEach does this:

1️⃣ First loop: processes **File1.csv**  
2️⃣ Second loop: processes **File2.csv**  
3️⃣ Third loop: processes **File3.csv**

✅ **What is the Field you select?**

**Child Items** = *give me all file names inside this folder*

**🌟 One-sentence explanation:**

**Get Metadata is like a robot that visits your folder every time, writes down the names of all files, and gives you this list so you can process them automatically.**

✅ Perfect—let’s do this step by step visually (like a cartoon).

Below is a **simple imaginary diagram** of **Get Metadata + ForEach**:

**🎯 Imagine this is your storage folder**

bash

/sales\_data/

├── Jan.csv

├── Feb.csv

└── March.csv

✅ You don’t know which files are there right now.  
✅ New files can be added any time.

**🟢 Step 1 – The Robot Visits the Folder (Get Metadata)**

✅ *"Hello folder! Show me everything inside you!"*

**The folder replies:**

json

{

"childItems": [

{"name": "Jan.csv", "type": "File"},

{"name": "Feb.csv", "type": "File"},

{"name": "March.csv", "type": "File"}

]

}

✅ This JSON **list** is the output.

✅ That’s it—**Get Metadata is only asking for names**.

**🟢 Step 2 – ForEach Receives the List**

✅ ForEach gets this array:

css

["Jan.csv", "Feb.csv", "March.csv”]

✅ ForEach loops over them **one by one**:

1️⃣ **First Loop**  
👉 currentFile = "Jan.csv"  
👉 Copy Activity copies Jan.csv

2️⃣ **Second Loop**  
👉 currentFile = "Feb.csv"  
👉 Copy Activity copies Feb.csv

3️⃣ **Third Loop**  
👉 currentFile = "March.csv"  
👉 Copy Activity copies March.csv

✅ **How does ForEach know which file to process?**

Inside ForEach, you write:

java

@item().name

This expression **grabs the file name in each loop**.

**🎯 Why is this dynamic?**

✅ Next week, maybe 2 new files arrive:

CopyEdit

April.csv

May.csv

✅ The robot runs again:

**New Output:**

json

{

"childItems": [

{"name": "Jan.csv"},

{"name": "Feb.csv"},

{"name": "March.csv"},

{"name": "April.csv"},

{"name": "May.csv"}

]

}

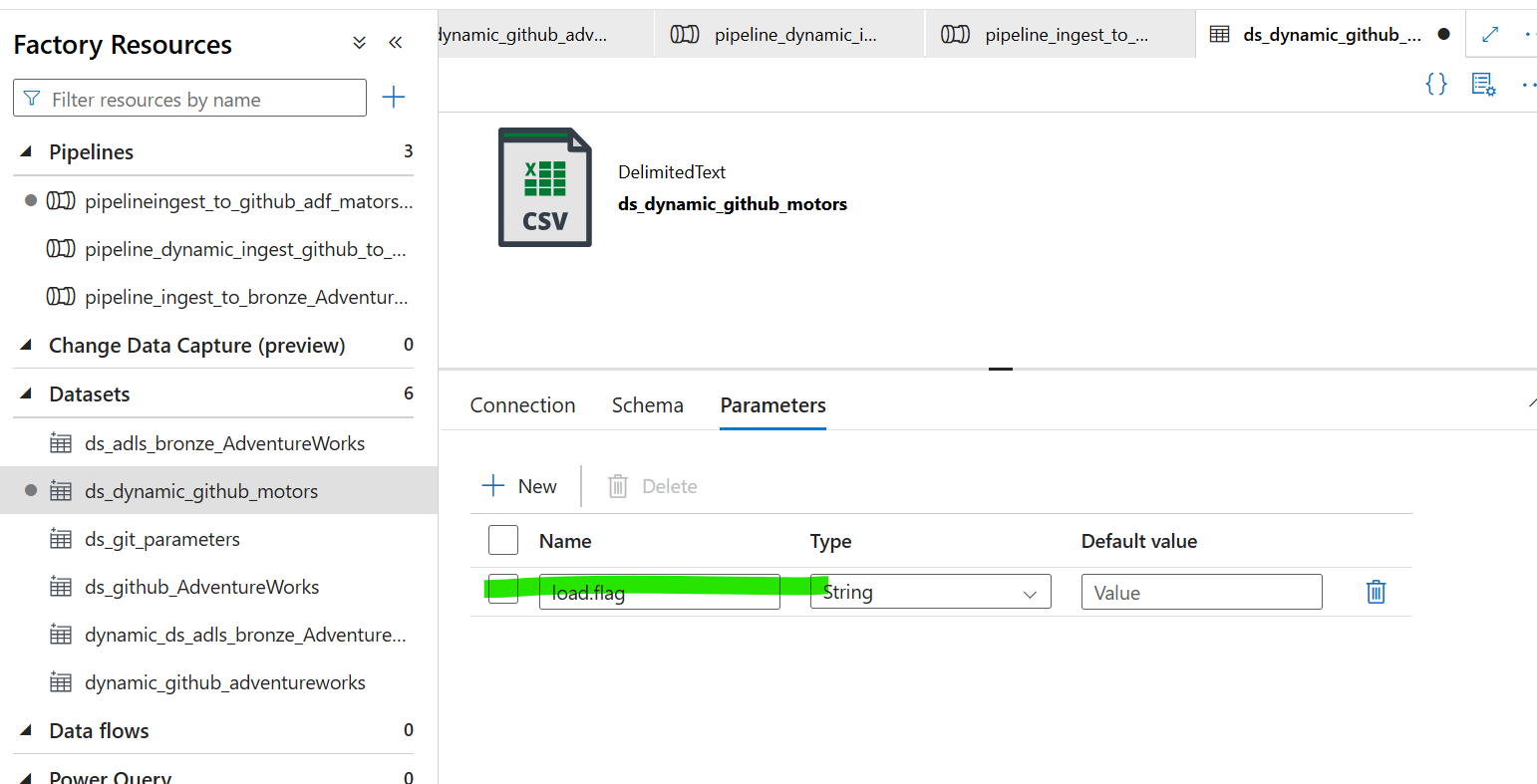
✅ ForEach loops over 5 files automatically.

✅ You don’t have to change anything—**fully automatic.**

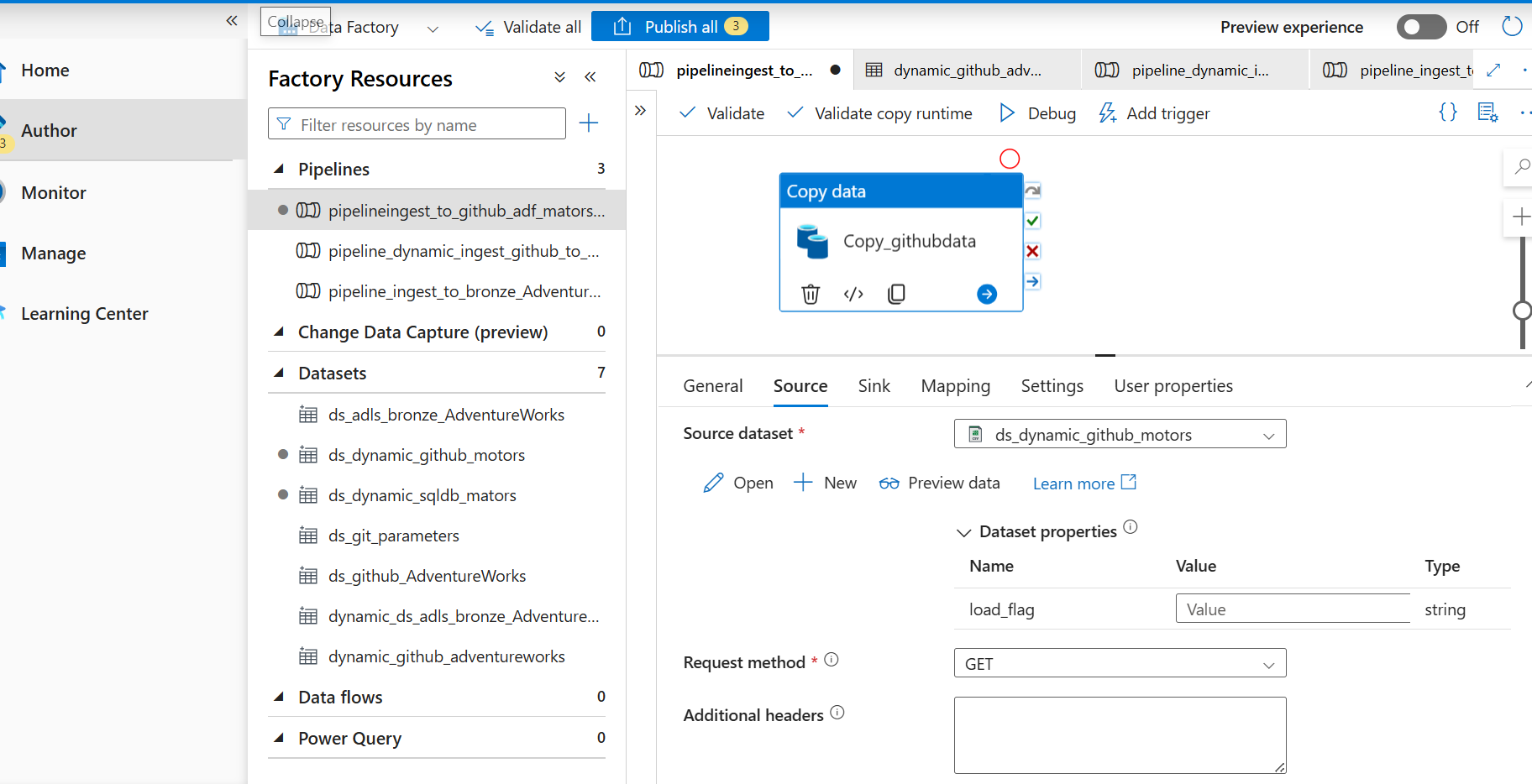
✅ **One-sentence summary:**

Get Metadata **does not store data**—it just **asks your folder each time which files are there right now** and passes that list to ForEach.

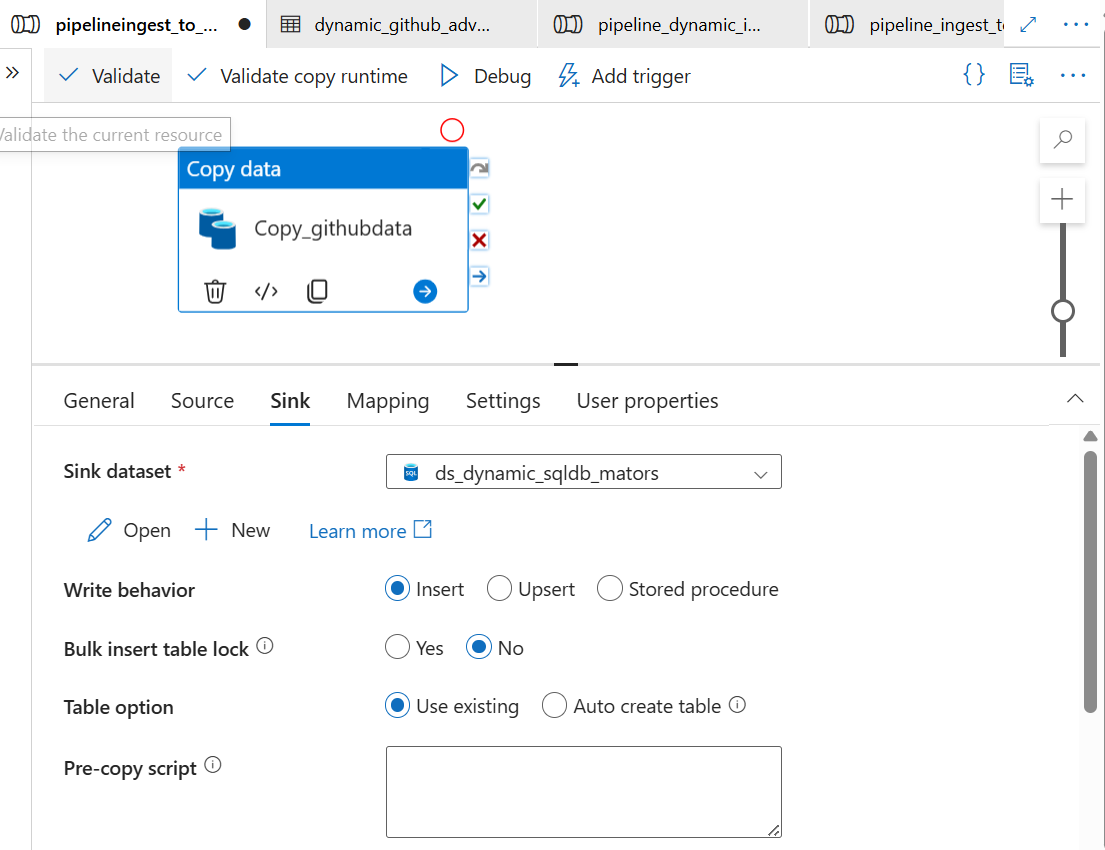
**I created an parameters that’s called ‘’load.flg’’**



**Parameterized the source\_dataset(GitHub):-**



**Parameterized the Sink\_dataset(SQLDB):-**

****

**🎯 What do we mean by Mapping in ADF Copy Activity?**

👉 **Mapping** means:

Telling ADF **which column in your source data goes into which column in your destination**.

You can think of it as **connecting columns from A to B**.

**✅ For example:**

Imagine your source file has these columns : Branch\_ID, Dealer\_ID, Units\_Sold

And your destination table has these columns: Branch\_ID, Dealer\_ID, Units\_Sold

✅ **Mapping says:**

* Take **Branch\_ID** from the file ➡️ put it into **Branch\_ID** in the table
* Take **Dealer\_ID** ➡️ Dealer\_ID
* Take **Units\_Sold** ➡️ Units\_Sold

**🟢 Why do you need mapping?**

Because:  
✅ Sometimes column names **are different**:

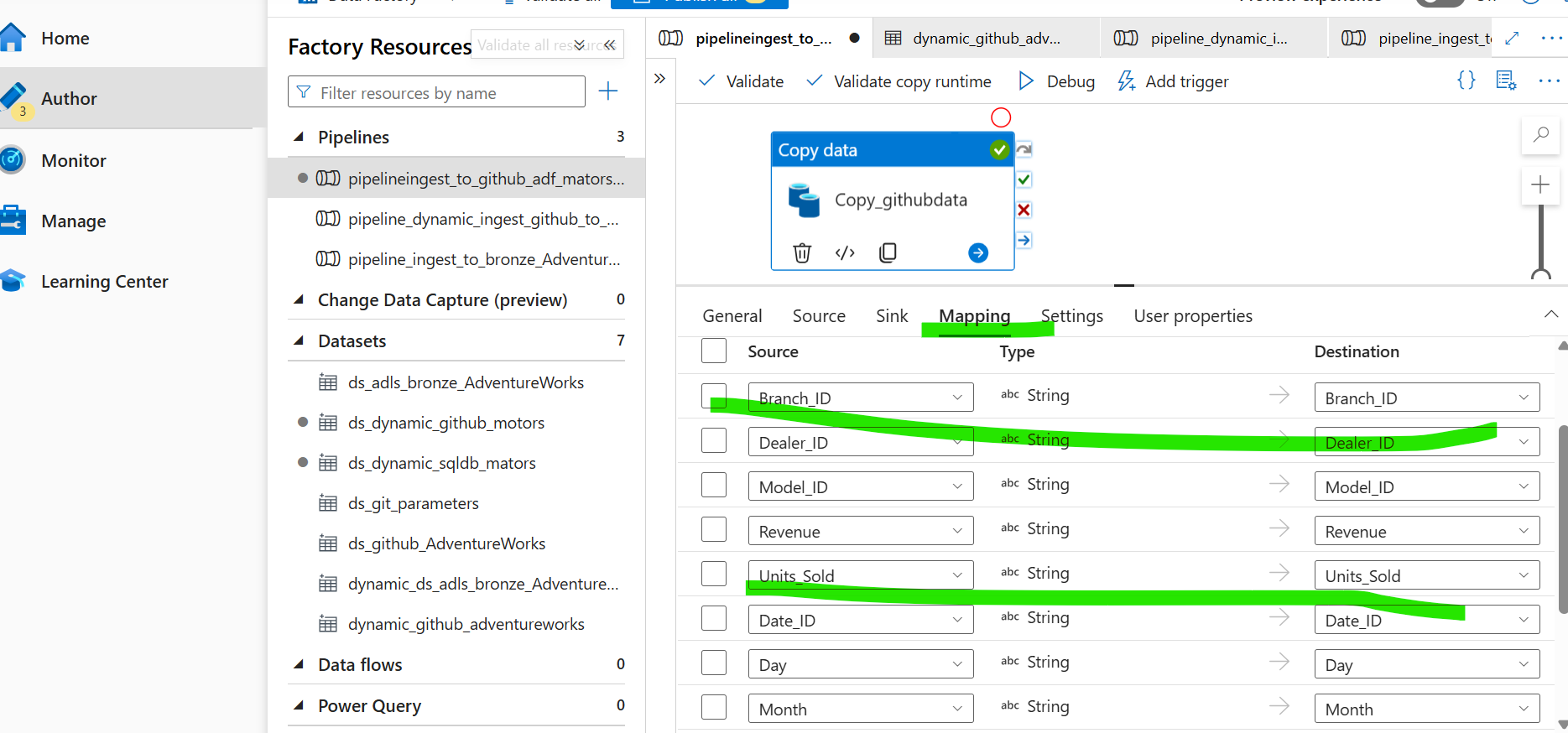
* In the file: DealerId
* In the table: Dealer\_ID

✅ Sometimes columns are in **different order**:

* File: Units\_Sold, Branch\_ID, Dealer\_ID
* Table: Branch\_ID, Dealer\_ID, Units\_Sold

✅ Sometimes you want to:

* Ignore some columns
* Rename them
* Change their data type

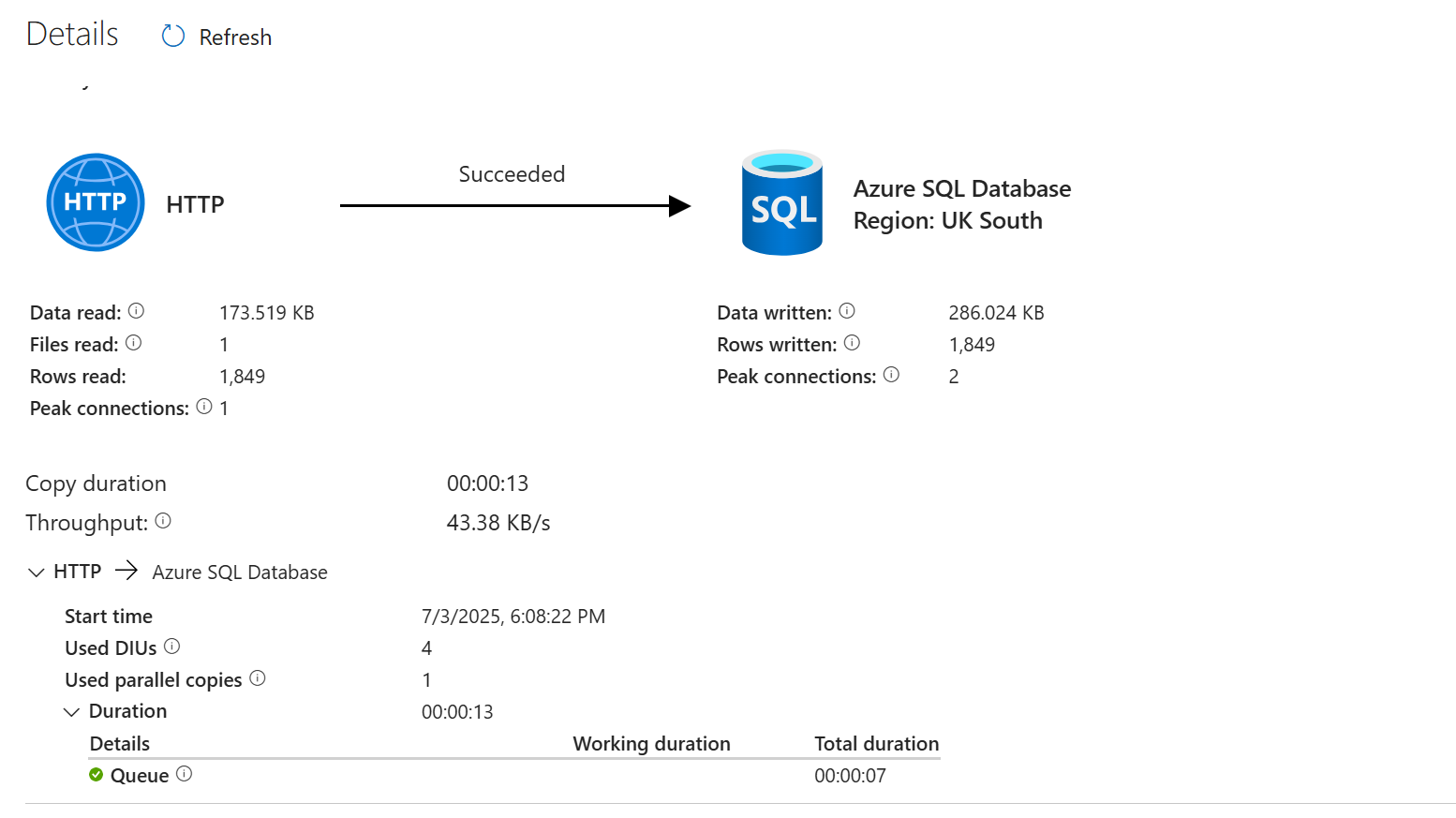


Data has been copied from **Source** to **SQL** **database(Loaded data from API to Database):** Below image for your review –

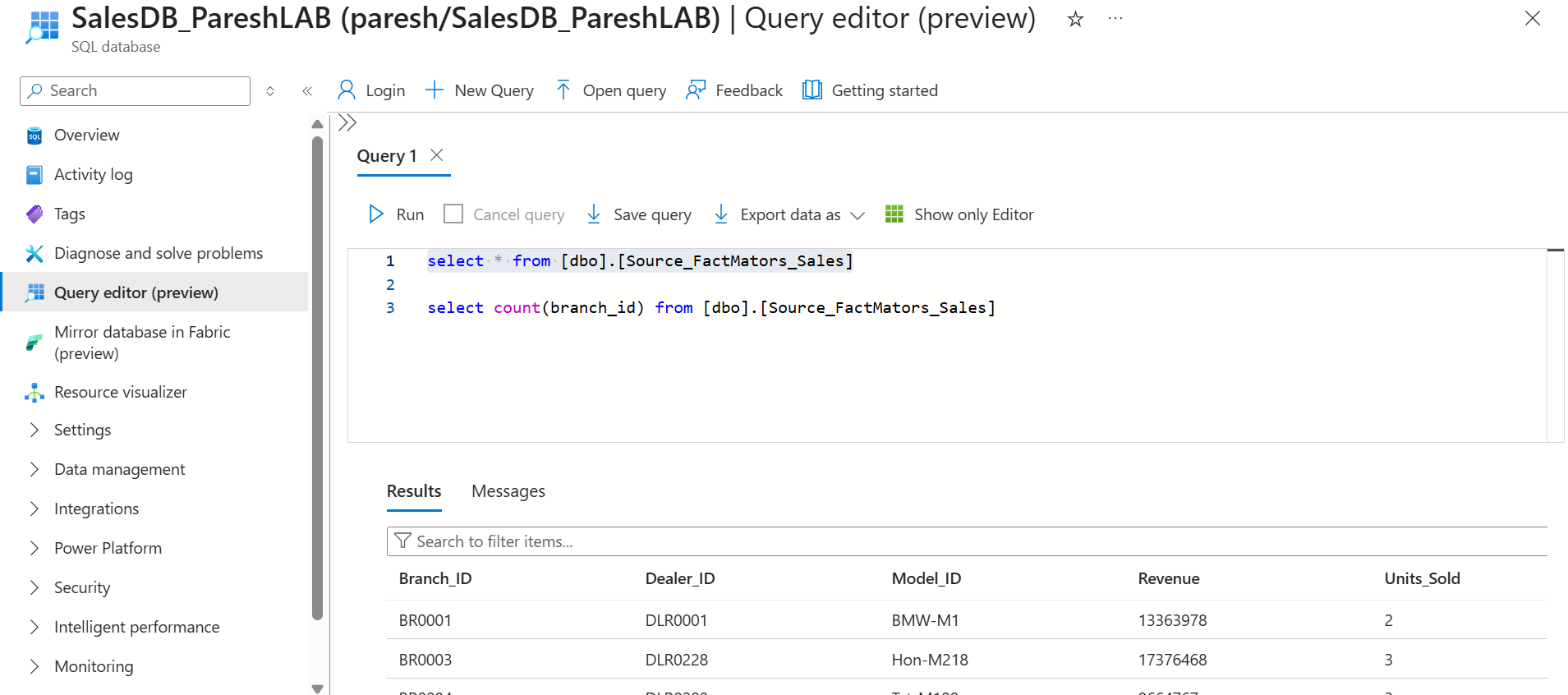
Now we will verify the table in the SQL Database.

**Verified the table:** 🡪🡪🡪🡪

* 1. **Data Read and written.**
  2. **Column count.**
  3. **Table Schema.**
  4. **Compare counts between source and target.**







✅ **Fantastic question—this is where many real-world projects struggle.**  
When you have **live streaming or constantly-changing data**, you can’t always expect:

*Source count = Target count*

**Why?**  
Because:

* New records may be arriving every second.
* Your copy pipeline takes a “snapshot” at a point in time.
* The counts are always moving.

🎯 🎯 **When you are moving *static* data (batch)**

✅ Example:

* You load a CSV file from GitHub.
* It has exactly 1849 rows.

**Validation is easy:**

* Row counts match exactly.

**🎯 🎯 When you are moving *streaming/live* data**

✅ Example:

* You are copying from a live SQL table with new orders coming in every second.

👉 Now, your **source count** is like a moving target:

At 10:00 AM: SELECT COUNT (\*) FROM Orders = 100,000

At 10:02 AM: SELECT COUNT (\*) FROM Orders = 100,025

**🟢 ✅ Recommended Validation Strategies**

**1️⃣ Watermarking**

✅ Instead of validating total counts, you validate **latest processed record**.

👉 Example:

* You load all records with OrderDate < '2024-07-03 10:00:00'.
* You record this watermark timestamp in your pipeline.

✅ Validation:

* Ensure your target has all records **up to that watermark**, not necessarily matching the live total.

**2️⃣ Incremental Counts**

✅ Compare **delta counts** instead of total counts.

👉 Example:

* Last run loaded 1000 new rows.
* This run loaded 1200 new rows.

✅ Validation:

* You track how many *new* rows were ingested each run.
* You record this in an **audit table**.

**3️⃣ Checksum or Row Hash Validation**

✅ Generate a checksum or hash of data for each batch.  
✅ Compare that checksum between source and target.

This is used in:

* Synapse Pipelines
* Databricks Delta pipelines

**4️⃣ Timestamp Ranges**

✅ Use date/time columns to validate partial snapshots.

✅ Example query: SELECT COUNT (\*) FROM Orders WHERE OrderDate >= '2024-07-03 09:00:00' AND OrderDate < '2024-07-03 10:00:00'

**We will create a Water table:**

**select \* from [dbo].[water\_table]**

**INSERT INTO [dbo].[water\_table]**

**values ('DT00001')**

💡 Tip for clarity in interviews:  
When you explain, always mention:  
1️⃣ What the watermark table stores (last load timestamp)  
2️⃣ How you use it (to filter new data)  
3️⃣ What benefit it provides (incremental loading, no duplicates)

**Let’s Create Store Procedure:**

CREATE PROCEDURE UpdateWatermarkTable

**Now – its time to create lookup activity:**

**🎯 Why do we often create a Lookup activity after running the stored procedure that updates the watermark?**

✅ 1️⃣ Your pipeline copies the data.

* It loads all new records since the last load timestamp.

✅ 2️⃣ Your stored procedure updates the watermark table.

**You run: UPDATE water\_table SET last\_load = '2024-07-03 16:00:00';**

* Now, the watermark table has the *new* timestamp.

✅ 3️⃣ You run a Lookup activity to read back that updated timestamp value.

* Why?
  + So you can:
    - Use it later in the pipeline.
    - Log or audit it.
    - Pass it to other activities.
    - Return it as an output to the pipeline run.

✅ Without the Lookup:

* You don’t have an easy way in ADF to *get* the new watermark value into your pipeline variables.

✅ **Common usage:**  
In a production pipeline, this is a typical pattern: [Copy Activity]

↓

[Stored Procedure Activity]

↓

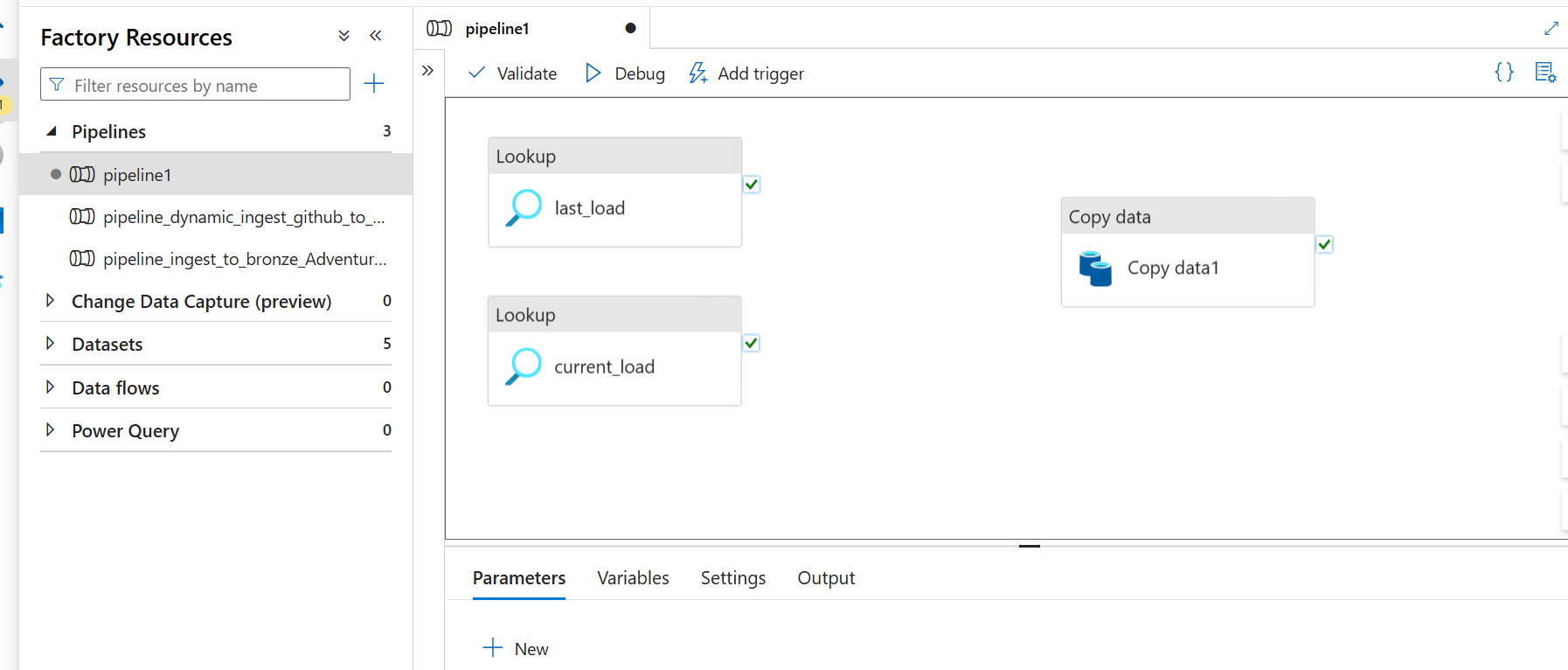
[Lookup Activity (read latest watermark)]

↓

[Set Variable Activity (save watermark)]

↓

[Next Steps]



🎯 **✅ Your Summary in Plain Words:**

👉 **Why do we use 2 Lookup activities?**

✅ **1️⃣ Last\_Load Lookup:**

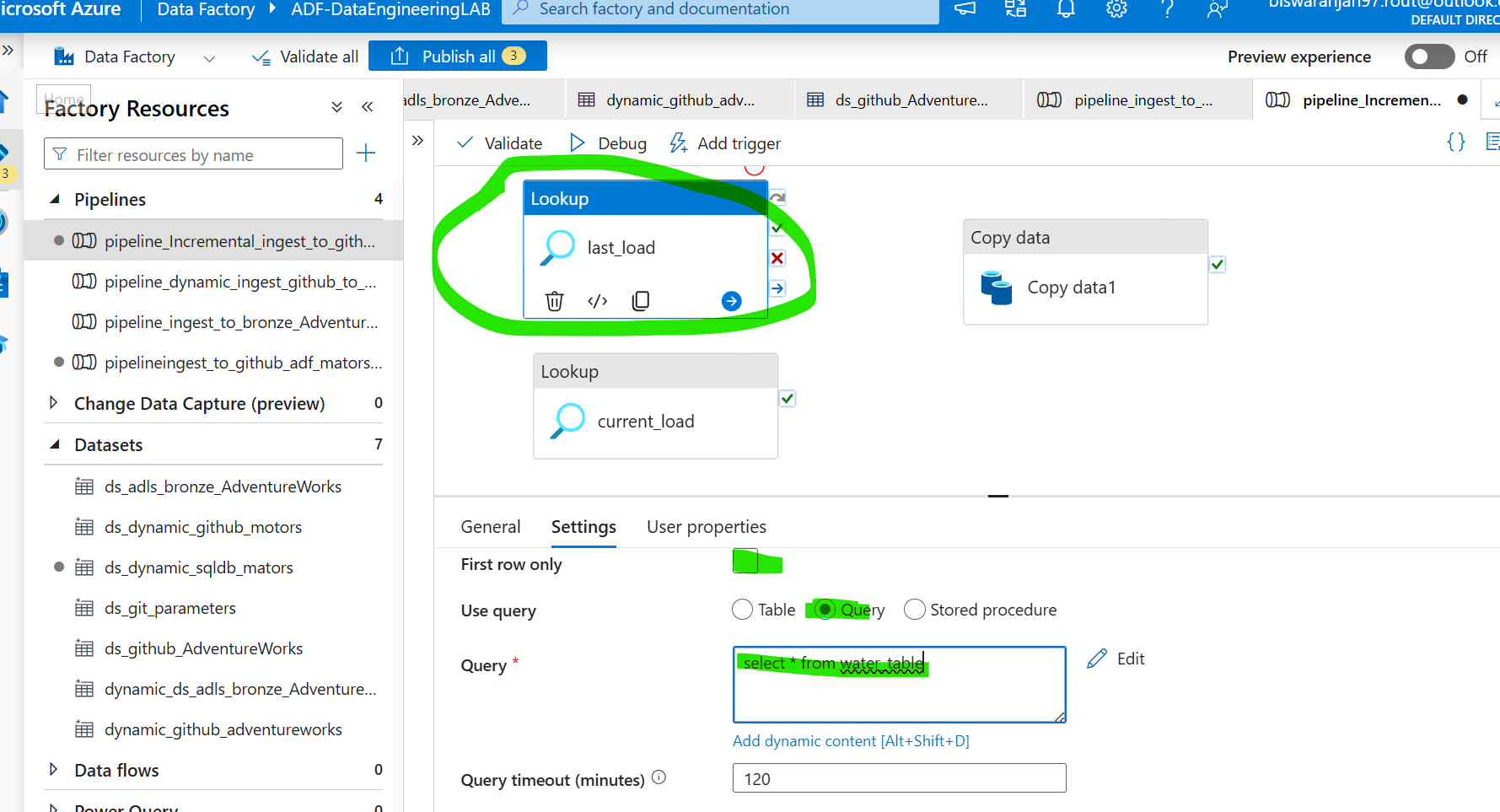
* **What it does:**  
  Reads the **last saved timestamp** from your watermark table.
* **Why you need it:**  
  So you know *where to start* your incremental query: SELECT \* FROM MyTable WHERE ModifiedDate > 'Last\_Load'
* **This is your "starting point."**

✅ 2️⃣ Current\_Load Lookup (or capturing the current timestamp):

* What it does:  
  Captures the current timestamp right before you start processing.
* Why you need it:  
  So you can record this moment in time, meaning:

"This is up to when I have loaded data during this run."

* **This becomes your "ending point."**

****

**This is a key point in understanding how your pipeline stays dynamic.**

🎯 **Question:**

In the Lookup activity settings, I manually put **water\_table** as the **table\_name**.  
Every time new data gets added to that table, do I need to manually change this value or will it automatically detect the new last\_load timestamp?

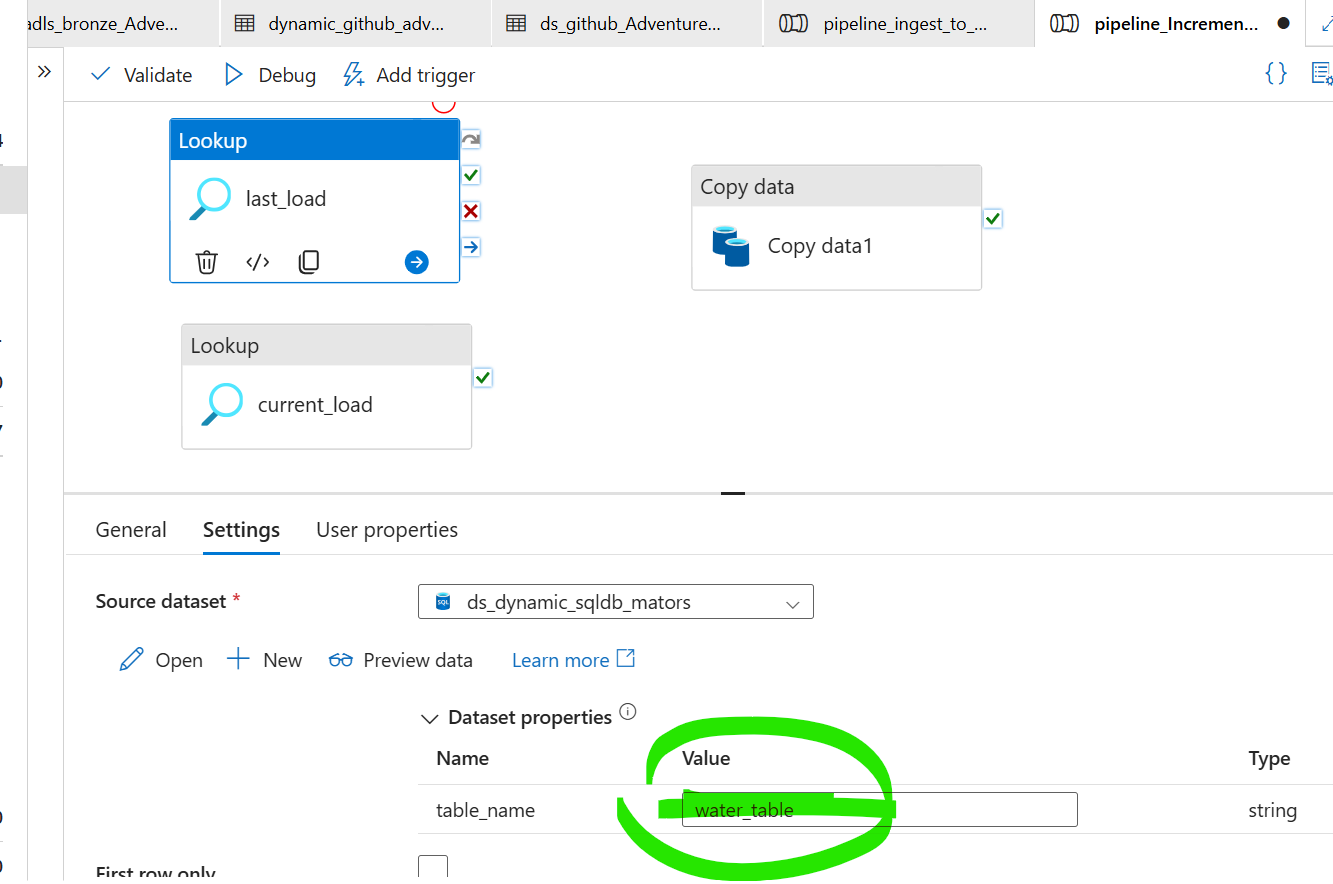
✅ **Short answer:**  
**You do *not* have to manually change that value each time.**

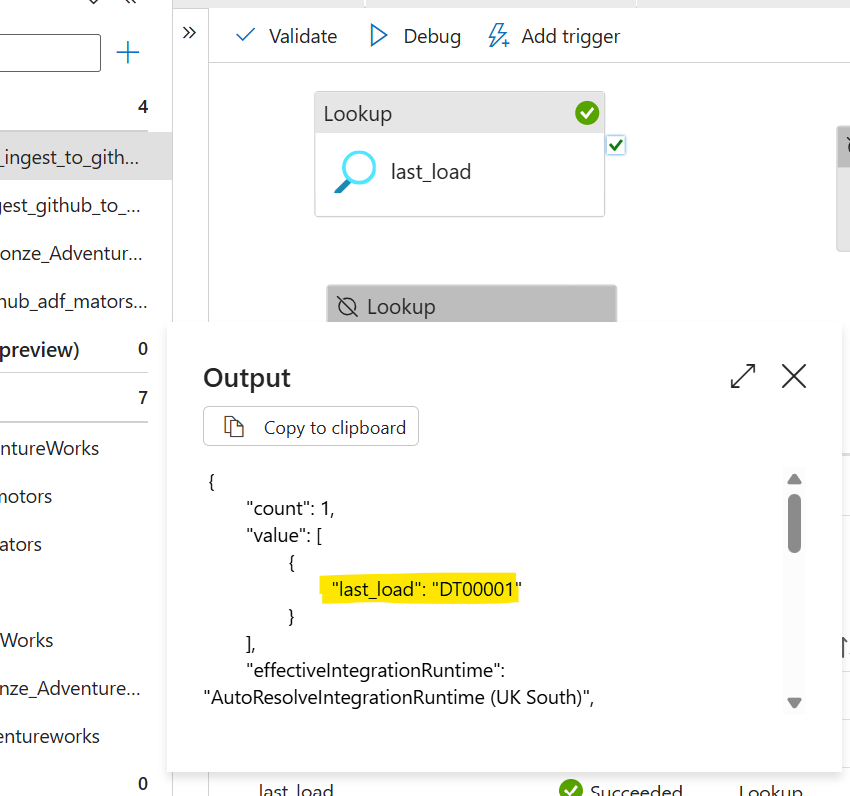
✅ **Why?**  
Because:

* table\_name = water\_table is **just telling ADF where to look** (which table to read).
* The **content inside that table** (the last\_load timestamp) is what changes dynamically every time you run your pipeline.

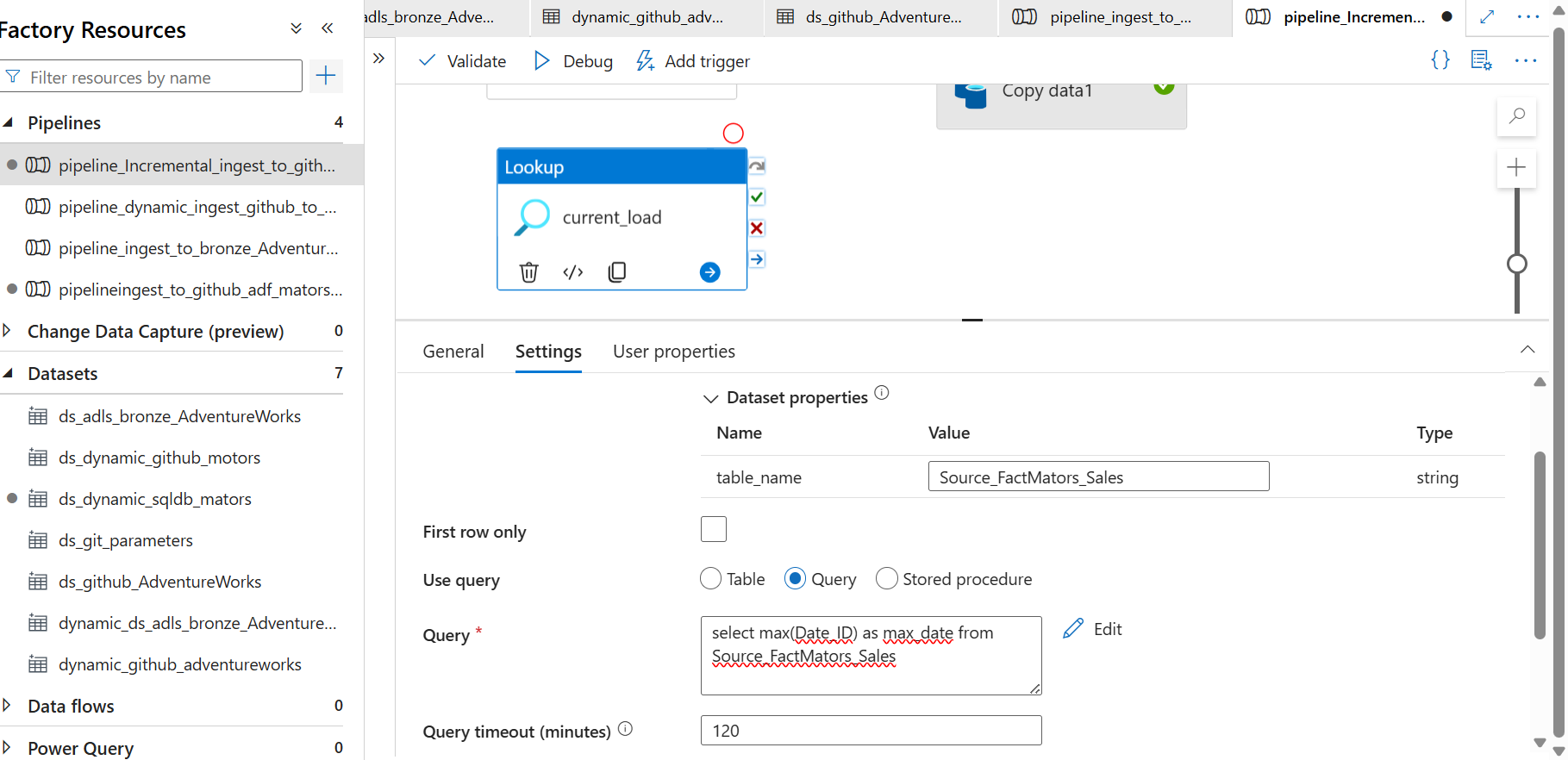
🎯 **One-sentence rephrase of your question:**

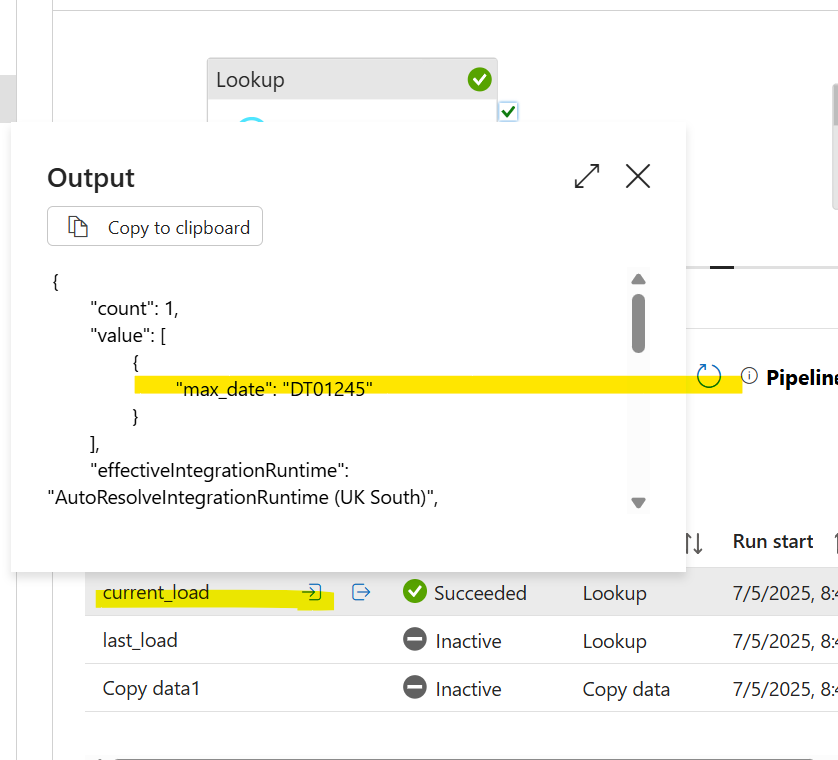
* “Do I have to manually update the dataset’s table\_name parameter every time, or does ADF automatically read the latest timestamp value inside that table?”
* ✅ **Answer:**  
  ADF will **automatically** read the updated last\_load value stored inside water\_table every time Lookup runs.  
  ✅ You **only set table\_name once**, because that doesn’t change.



Our last\_load Lookup activity is run perfectly : - 

**Query :- select max(Date\_ID) as max\_date from Source\_FactMators\_Sales**

****

****

* We finally configured Lookup activities (last\_load and current\_load) and they are working successfully. For future runs, the Copy Activity will use these to copy incremental data, and the stored procedure will update the last\_load. Is this logic correct?”
* ✅ **Yes—this is fundamentally correct**.

✅ **This is the standard pattern:**

* Lookup last watermark
* Capture current watermark
* Copy increment
* Update watermark

✅ **You are 100% correct.**

**🟢 Extra Logic and Best Practices Behind It**

To help you think like a **data engineer**, here are more **key details and why this approach is important:**

✅ **Why Lookup current\_load?**

* Because you want to guarantee a **stable point in time** when you started ingesting.
* Imagine if you didn’t capture current\_load:
  + New rows could keep appearing in the source table while your Copy Activity is running.
  + Next run could re-load some of these rows or skip some.
* Capturing current\_load makes your loads **deterministic**.

✅ **Why update last\_load only after success?**

* You only update last\_load **after** confirming the Copy Activity succeeded.
* If the pipeline fails, last\_load is NOT updated, so you don’t lose data.
* That’s why Update Watermark should be after Copy Activity in the pipeline flow.

✅ **Why store timestamps in a separate table (water\_table)?**

* So you can track progress independently.
* So multiple pipelines can reference it safely.
* So your logic is transparent and auditable.

✅ **What happens if you need to re-run?**

* You can manually change the last\_load to a previous date.
* The pipeline will re-copy from that date.

✅ **Advanced pattern:**

* Also store metadata like:
  + Pipeline Run ID
  + Number of rows copied
  + Who ran it
* So you have full audit logs.

**Build a Copy Activity that copies data between ‘’*last\_load’’* and ‘’*current\_load’’* timestamps.**

* This is exactly how you **only bring the new incremental data**.