NO.

Title

Enjoy the most efficient handwriting experience! Taking notes on the go, whether for inspiration, ideas, knowledge learning, business insights, or even sketches...

NOTE IN

Creative notes

By reading we enrich the mind;
by writing we polish it.

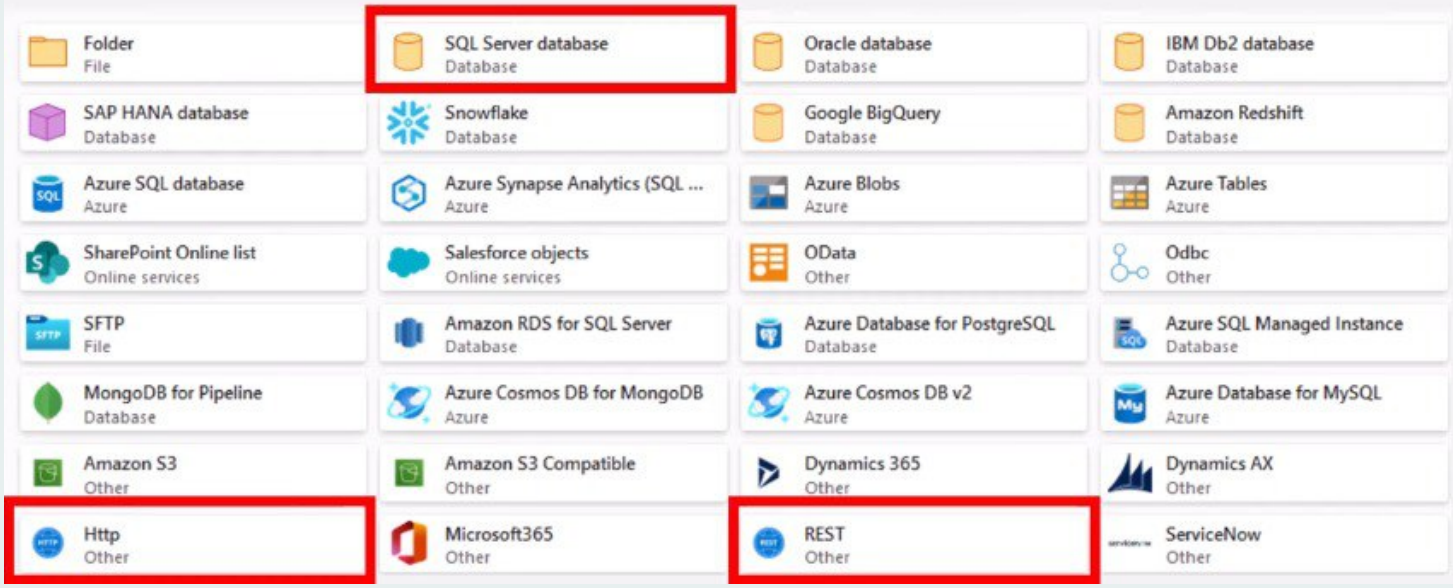
Data Ingestion for the Copy Data Activity

The Copy Data activity is the **primary mechanism to ingest data into Fabric (using a Data Pipeline)**:

- On-premise databases (via the on-premise data gateway)
- Most Azure services, plus data from most cloud data storage providers →
- REST APIs (including basic pagination)
- HTTP (website data, or openly accessible data)

Authentication methods vary between different connections

SOURCES:
(not all available sources shown)



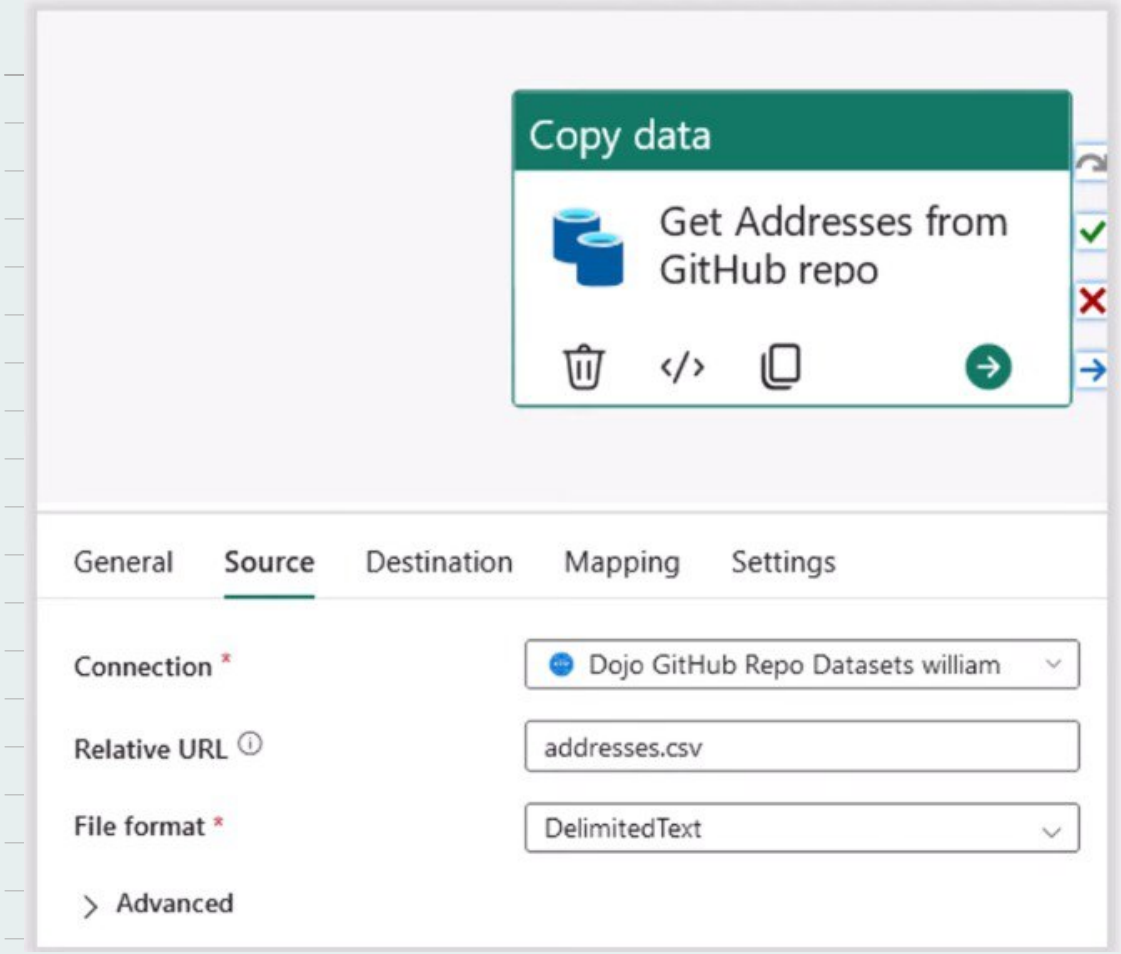
DESTINATIONS:

File Data? Lakehouse Files area

Table Data? Any Fabric data store

PLUS: many external data stores.

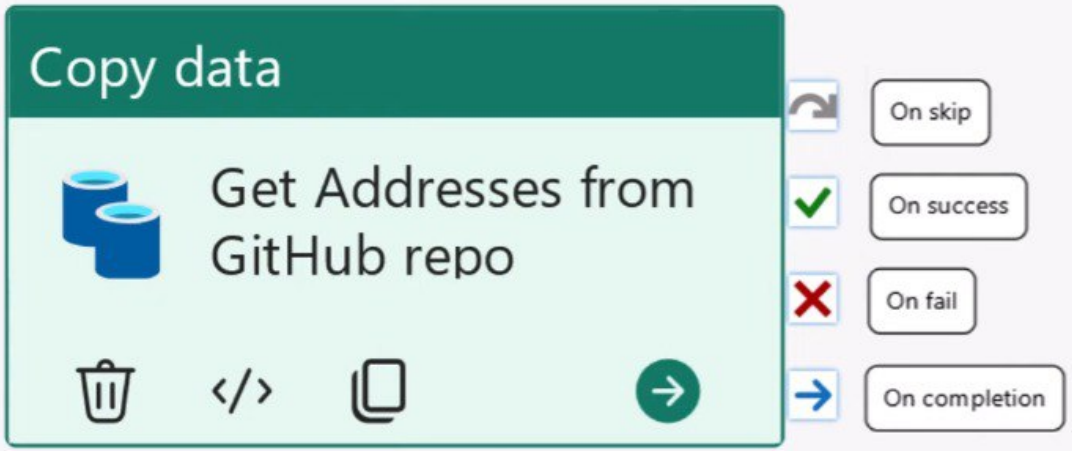
- Some connections can be only used as source and not as destination for copy data activity.



This simple pipeline gets some data from a GitHub repository, using a Copy Data activity, HTTP connection, Delimited Text File Format.

The Connection and relative URL is hard-coded (we'll look at more dynamic pipelines a bit later).

Creating dependencies between activities (do this, then this)

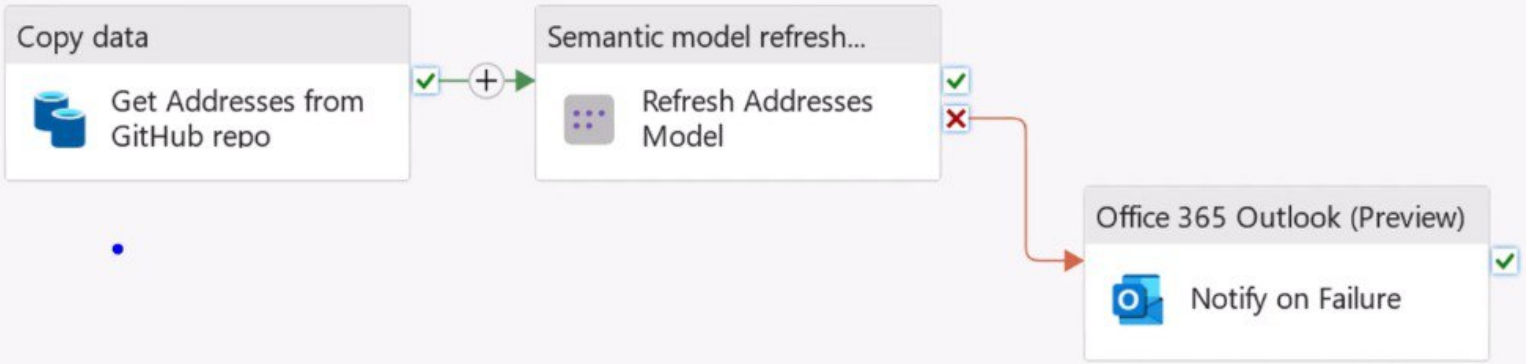


On Skip – next activity will run, only if previous activity is skipped

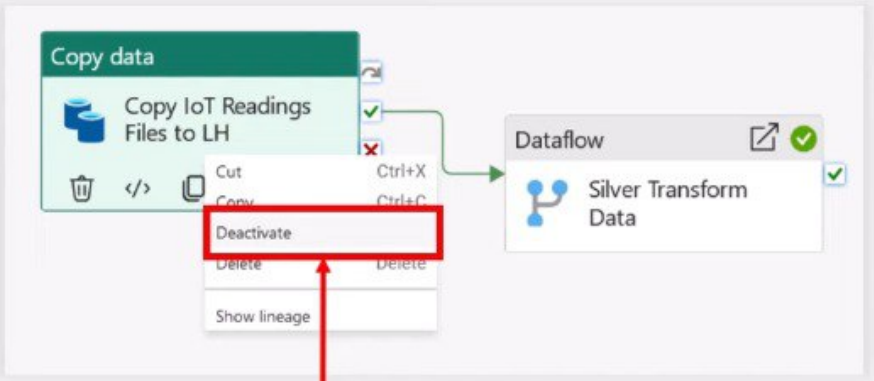
On Success – next activity will run, only if previous activity ran successfully

On Fail – next activity will run, only if previous activity failed (useful for notifications)

On Completion – next activity will run, when the previous activity completes (pass or fail)



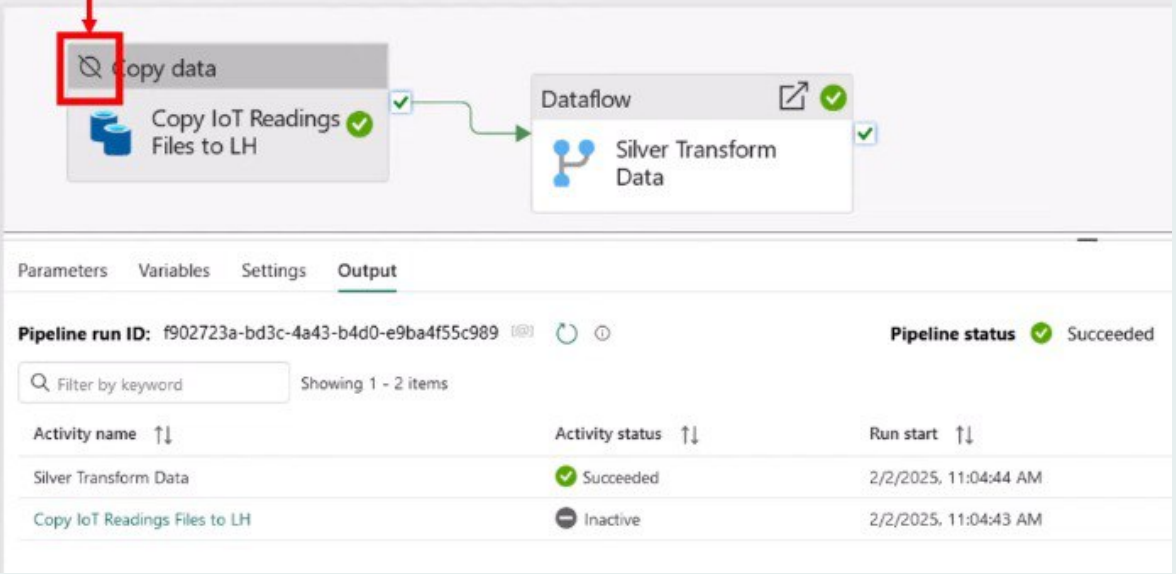
Active vs Inactive activities



Activities can be active, or they can be deactivated, by right clicking on it, and clicking Deactivate.

- Deactivaty activity is a successful activity

This icon signifies that this activity is deactivated

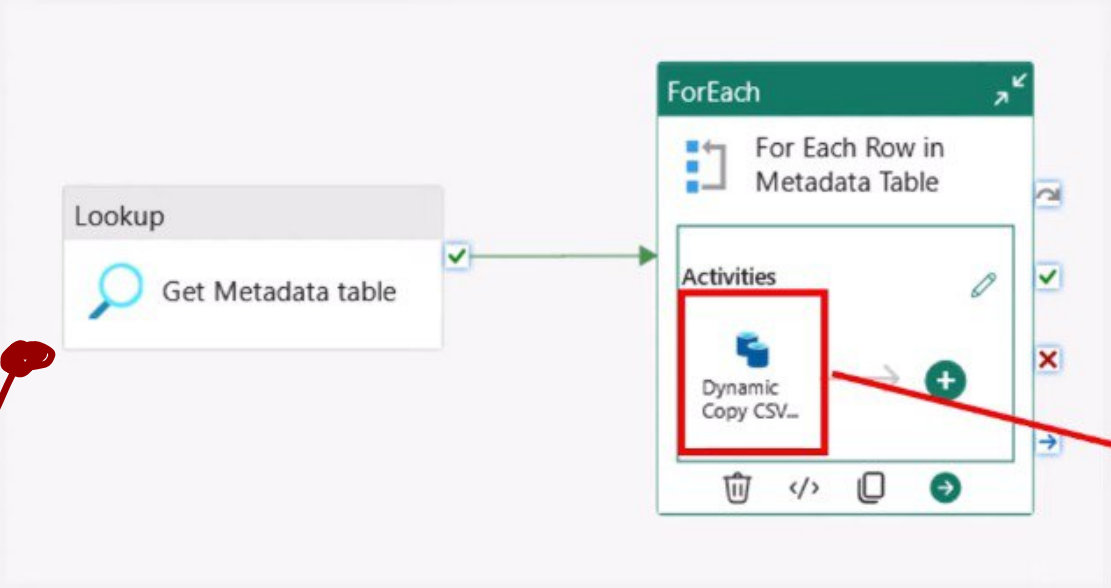


Architecture

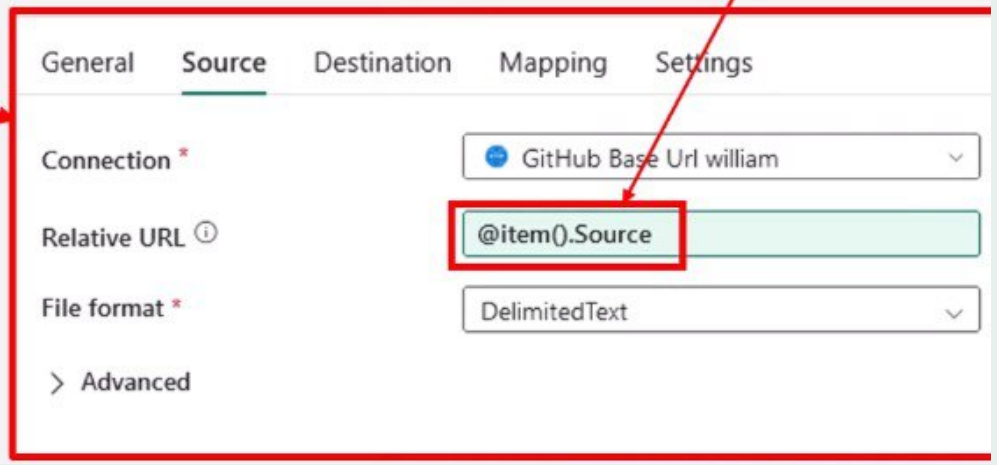
Building dynamic data pipelines (metadata-driven)

Some example use cases:

1) "I want to ingest 25 different tables from an Azure SQL Database, and load them into 25 tables in a Fabric Data Warehouse"
2) "I want to query 10 different REST API endpoints, to get all my data from a SaaS product, and save all the raw JSON in a Lakehouse Files Area"



Making the relative URL dynamic (rather than hardcoded). This means we can perform the Copy Data activity for all rows in the Metadata table.



Benefit: once the pipeline is setup, you shouldn't need to edit it. The metadata table is used to vary the datasets being processed.

1

- Store the connection details (table names, schema names, ...) in a database / files let's call it metadata Table/db in some data store (data warehouse, lakehouse, ...) in your fabric environment

[2] and Then use a lookup script/ lookup activity to get these data in a table format and bring it up to our pipeline

[3] loop on each data of a table in a for each activity

[4] @ Item() . Source

↙

we are now iterating over each data table (Item) and get the current value/data through property source

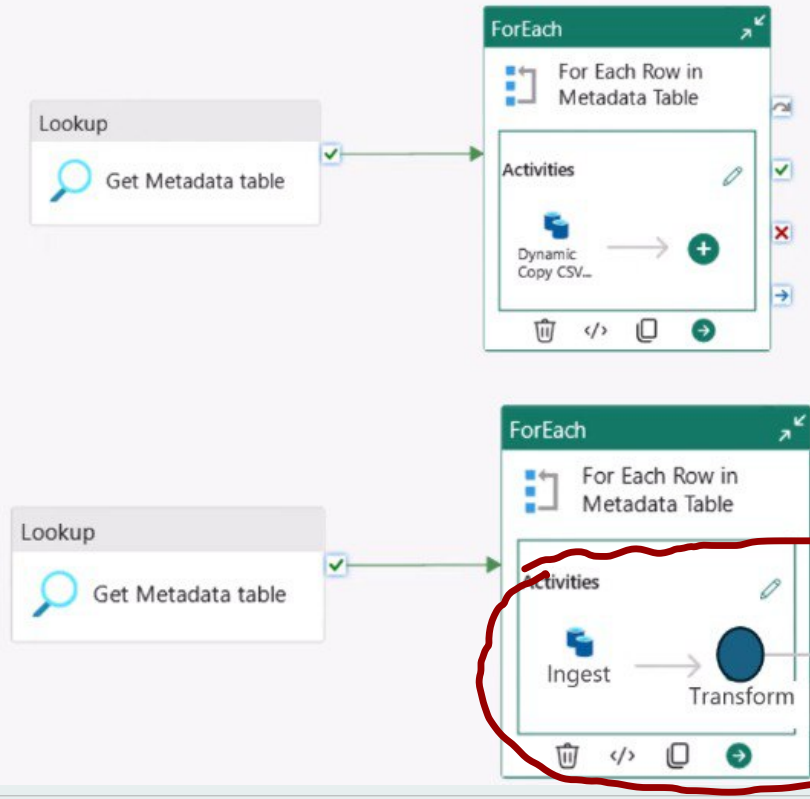
[5] So, If we need to add 2 more tables to copy, we don't need to change our pipeline, just add the 2 rows of connection strings to the metadata table

What if we want to do more than Copying?

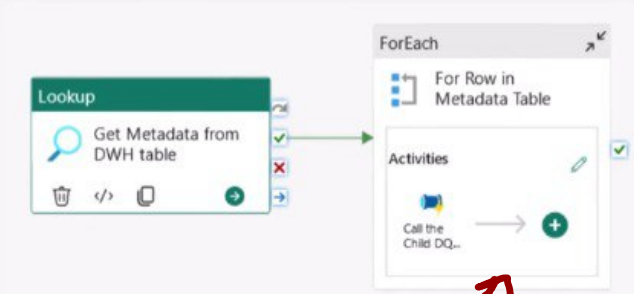
→ we want for each item to transform, load, do some script on it, ...

Parent / Child pipelines

In the last slide, we saw an example like this:



What we can do instead is move all those activities into a 'Child' pipeline, and call the Invoke Pipeline activity, like this:



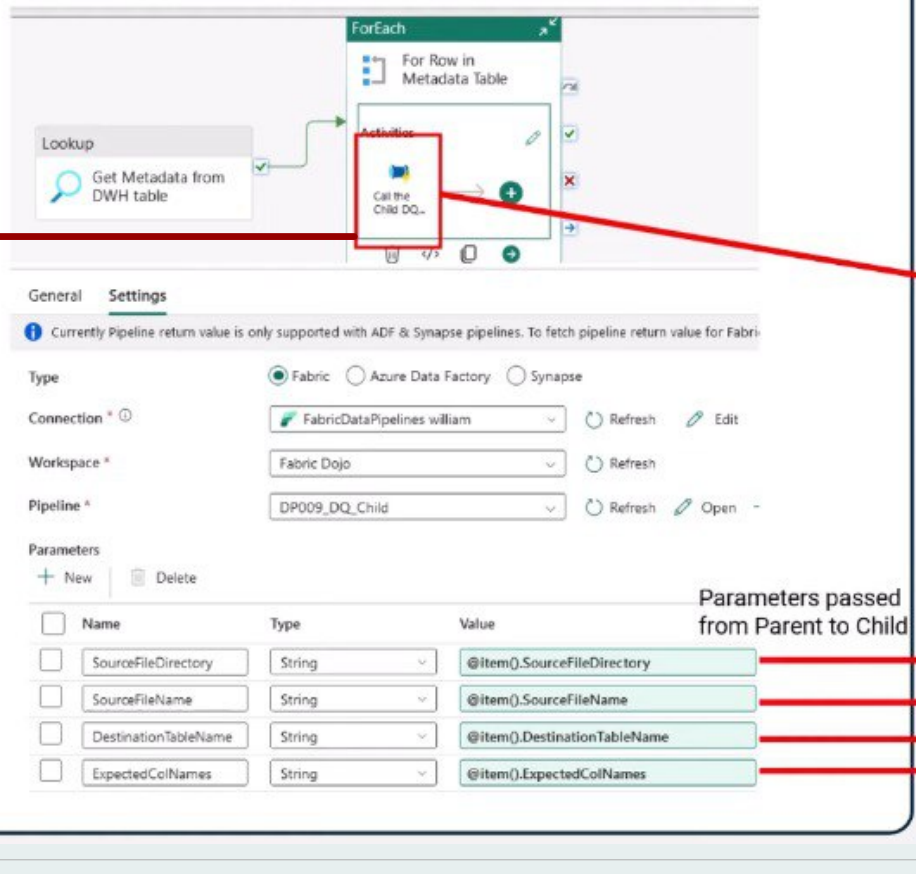
- we can just construct this in a pipeline and invoke the pipeline for each row in a metadata table

How to do that ?

Key features to enable the Parent / Child architecture

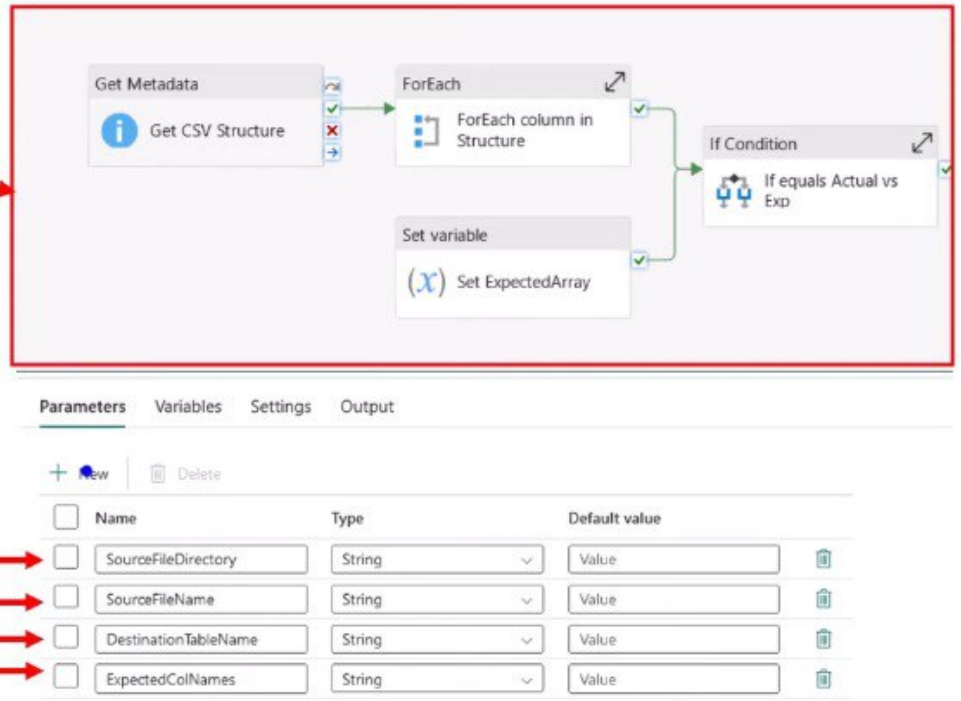
Parent Pipeline:

- must contain an Invoke Pipeline - invoking the Child pipeline
- normally the Parent will pass parameters to the Invoke Pipeline activity (to be used in the Child), in this case metadata from a Metadata table.



Child Pipeline:

- create Pipeline Parameters to accept dynamic Parameters from the Parent.
- then, you can use the Parameters dynamically in your child, using: `@pipeline().parameters.SourceFileName`



recommendation: use legacy invoke pipeline activity. especially if you want to pass pipeline return values up from child back to parent

Orchestration using Notebooks

Notebook Orchestration

From within a Notebook, you can call the execution of other notebooks, using the **notebookutils** package.

There are two main options:

Option 1: Executing notebooks in parallel:
When we pass a list of Notebook names into nb.runMultiple(), it will execute them in parallel.

```
1  from notebookutils import notebook as nb
2
3  nb.runMultiple(["NotebookName1", "NotebookName2"])
```

✓ - Command executed in 17 sec 115 ms by Will Needham on 12:41:41 PM, 12/08/24

Activity name	Snapshot	Status	Progress
0	NotebookName1	Succeeded	100%
1	NotebookName2	Succeeded	100%

```
{'0': {'exitVal': '', 'exception': None},
 '1': {'exitVal': '', 'exception': None}}
```

→ Run Notebooks in any order

Notebook Orchestration (using DAG)

Option 2: Executing notebooks in a predefined order (using the DAG):
Alternatively, we can define a DAG object (directed-acyclic graph) - this is a Python object that follows a pre-defined structure. Here's an example:

```
1  DAG = {
2      "activities": [
3          {
4              "name": "NotebookName1",
5              "path": "NotebookName1",
6              "timeoutPerCellInSeconds": 90,
7
8          },
```

Key features of the DAG:

- the "**activities**" list: a list of objects. Each object represents a Notebook that you want to orchestrate.

→ order is important

→ we can use function validateDAG() to make

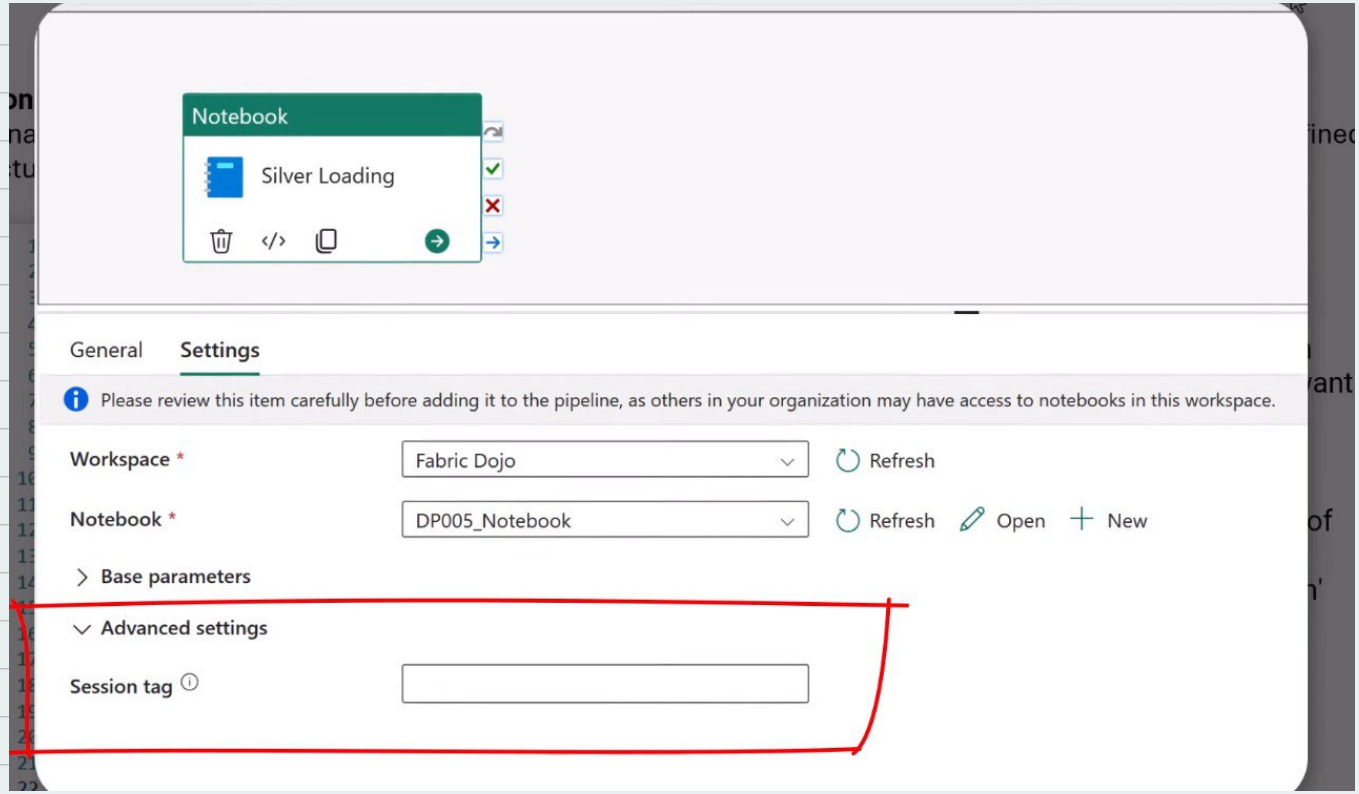
```
        {
            "name": "NotebookName2",
            "path": "NotebookName2",
            "timeoutPerCellInSeconds": 120,
            "retry": 1,
            "retryIntervalInSeconds": 10,
            "dependencies": ["NotebookName1"]
        }
    ],
    "timeoutInSeconds": 43200,
}

nb.runMultiple(DAG, {"displayDAGViaGraphviz": True})
```

- **"dependencies"** - the "dependencies" property dictates the order of execution of your DAG. In the example above, the execution of Notebook2 'is dependent on' the completion of Notebook1, so Notebook2 will run after Notebook1.

Sure of our dag

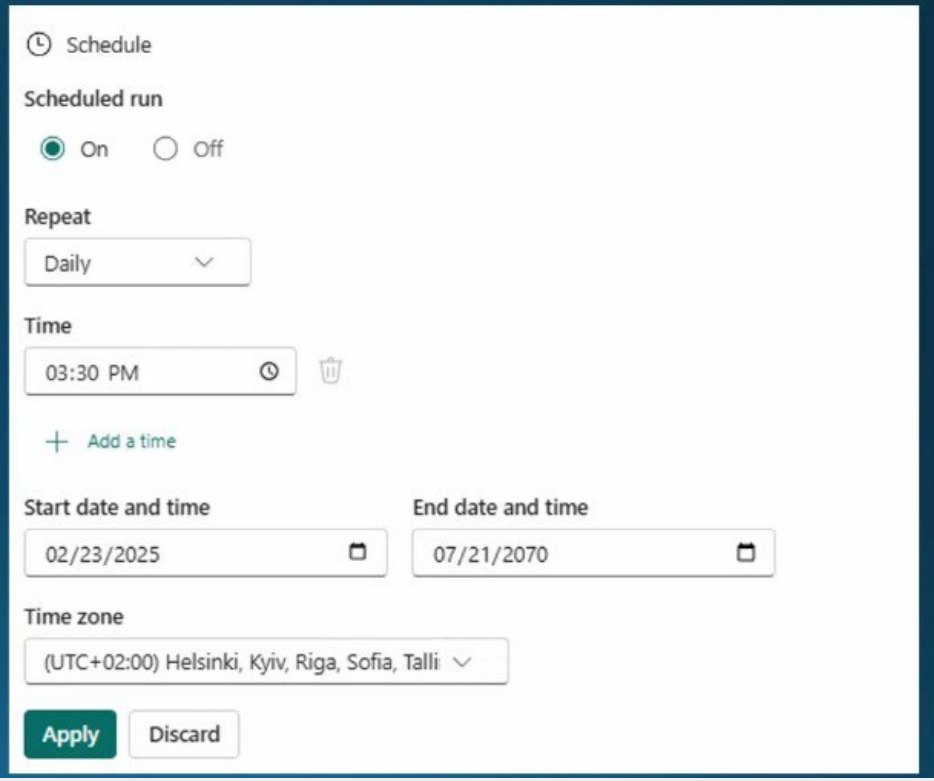
↳ we can run multiple (spark) notebooks at same time in same spark session Thanks to session tag feature

Where you creates what is called session tag That means That multiple note book executed in a pipeline can all run on The same session



Scheduling

- Runs pipeline on a set time interval
  - Can be turned on or off
- Different time intervals supported
  - Can be configured from minute level to month level
- Start and end date and time
  - Control the period during which the pipeline runs
- Time zone
  - Supports different time zones and adjusts automatically to summer and winter times



Triggers

Recommended

- schedule trigger execution of: Data pipelines, Notebooks, Dataflow Gen2

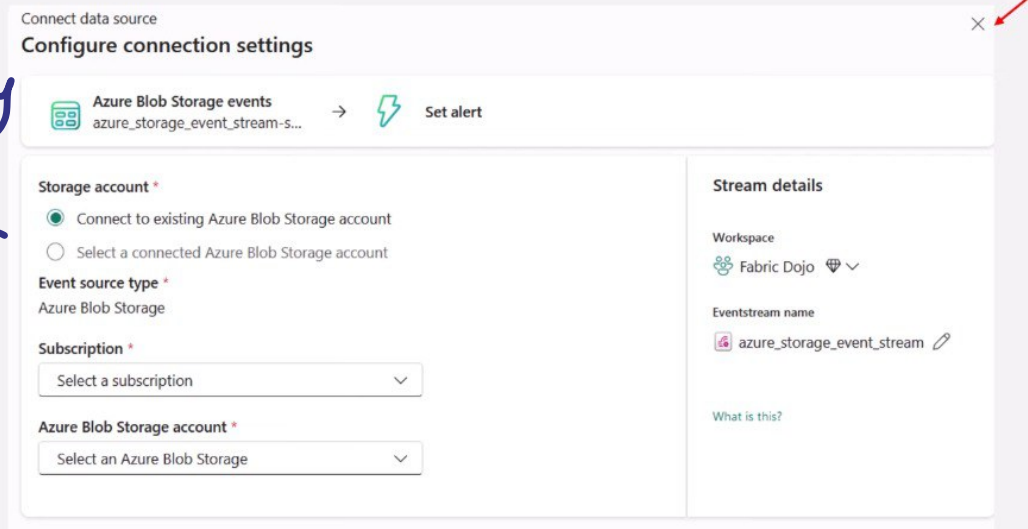
Where it will be executed on regular schedule

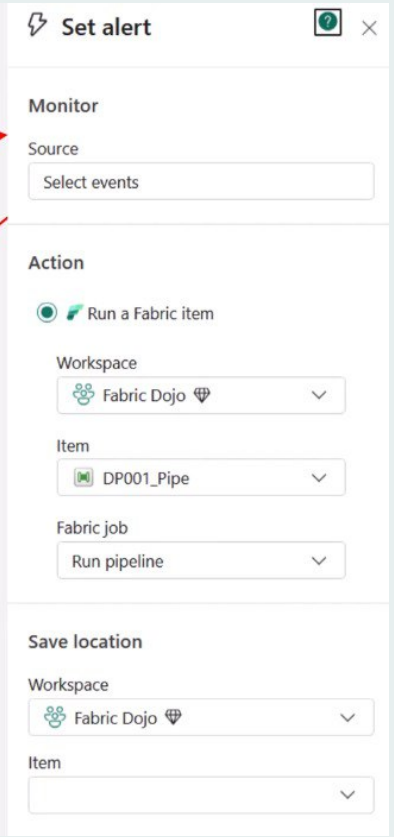
This is useful for people migrating from synapse (ex) and need this functionality

They just want to trigger fabric items rather than ADF

Event-based trigger for Data Pipeline (Preview)

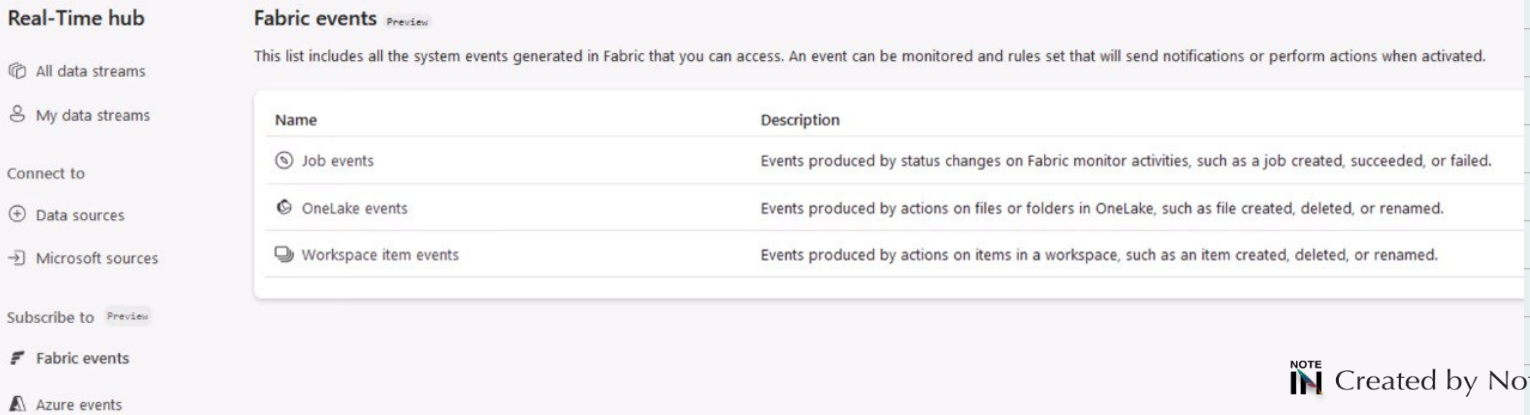
Data Pipelines can now be triggered, based on Events in an Azure Blob storage account (e.g. new file uploaded), click this button: Add trigger (preview)



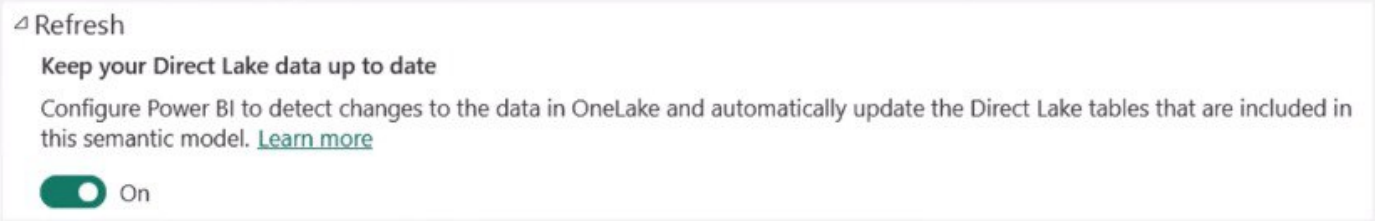


Event-based triggers in the Real-time Hub

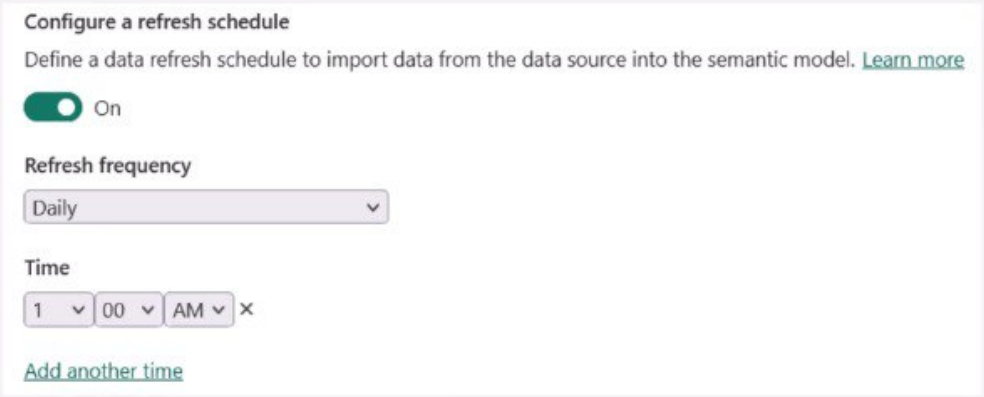
In the Real-Time Hub in Fabric, these are new options to trigger an Alert or Eventstream, based on some interested new events: Job events, OneLake events and Workspace item events.



Options for triggering a Semantic Model Refresh

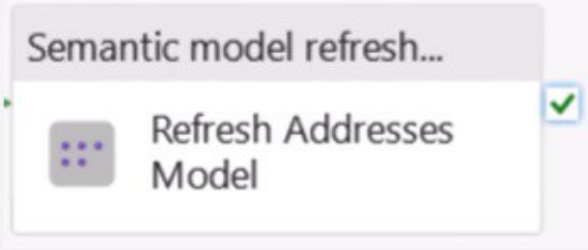


Option 1 (in the Semantic Model Settings): Auto-refresh if using DirectLake



Option 2 (in the Semantic Model Settings): Scheduled refresh

Option 3:
using the Semantic Model Refresh activity in a Data Pipeline



Option 4:
Using Semantic Link (sempy) from a Notebook:

refresh_dataset

Refresh data associated with the given dataset.

For detailed documentation on the implementation see Enhanced refresh with the Power BI REST API.

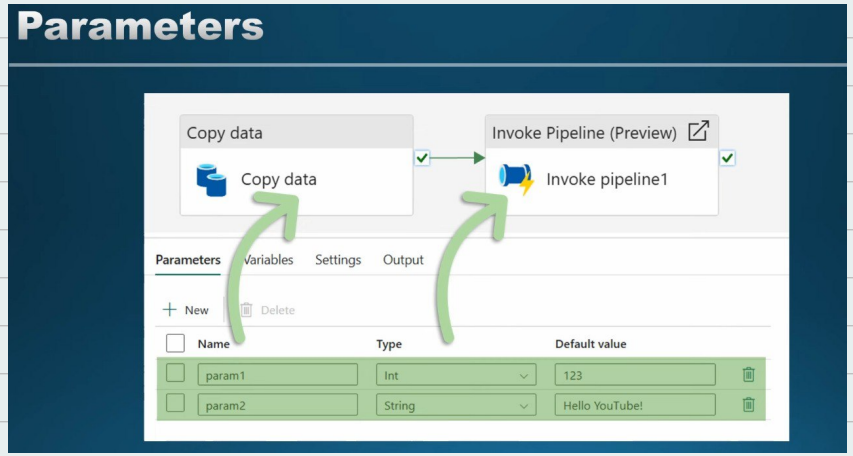
Python Copy

```
refresh_dataset(dataset: str | UUID, workspace: str | UUID | None = None,
refresh_type: str = 'automatic', max_parallelism: int = 10, commit_mode:
str = 'transactional', retry_count: int = 0, objects: List | None = None,
apply_refresh_policy: bool = True, effective_date: date =
datetime.date(2025, 1, 30), verbose: int = 0) -> str
```

pipeline parameters & Variables in Azure factory

Parameters

- defined at pipeline level
- Can not be modified during a pipeline run
- Can Control The behaviour of a pipeline & it's activities, such as by passing in The Connection details for a dataset or The path of a file to be processed.
- you Can access it using `@pipeline().parameters.<parameter name>` expression in a pipeline activity



Variables

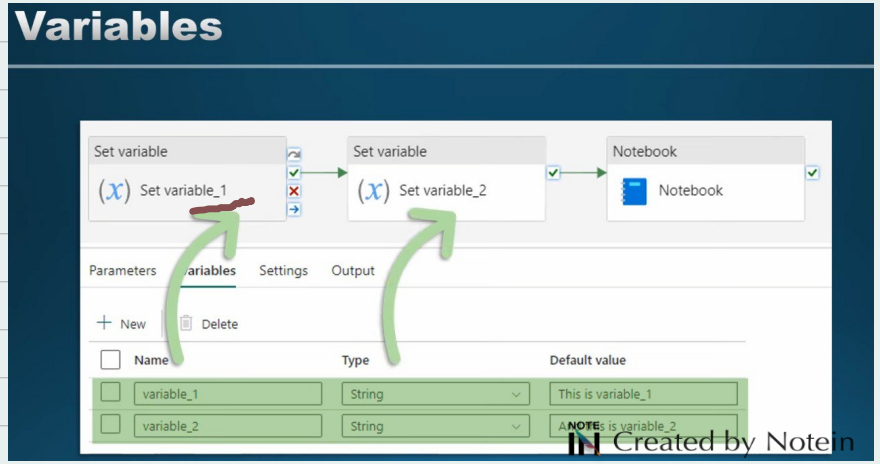
- Values That can be set & modified during pipeline run using a **set Variable** activity.

- Can be used to store & manipulate data during a pipeline run, such as by storing results of a computation or current state of a process.

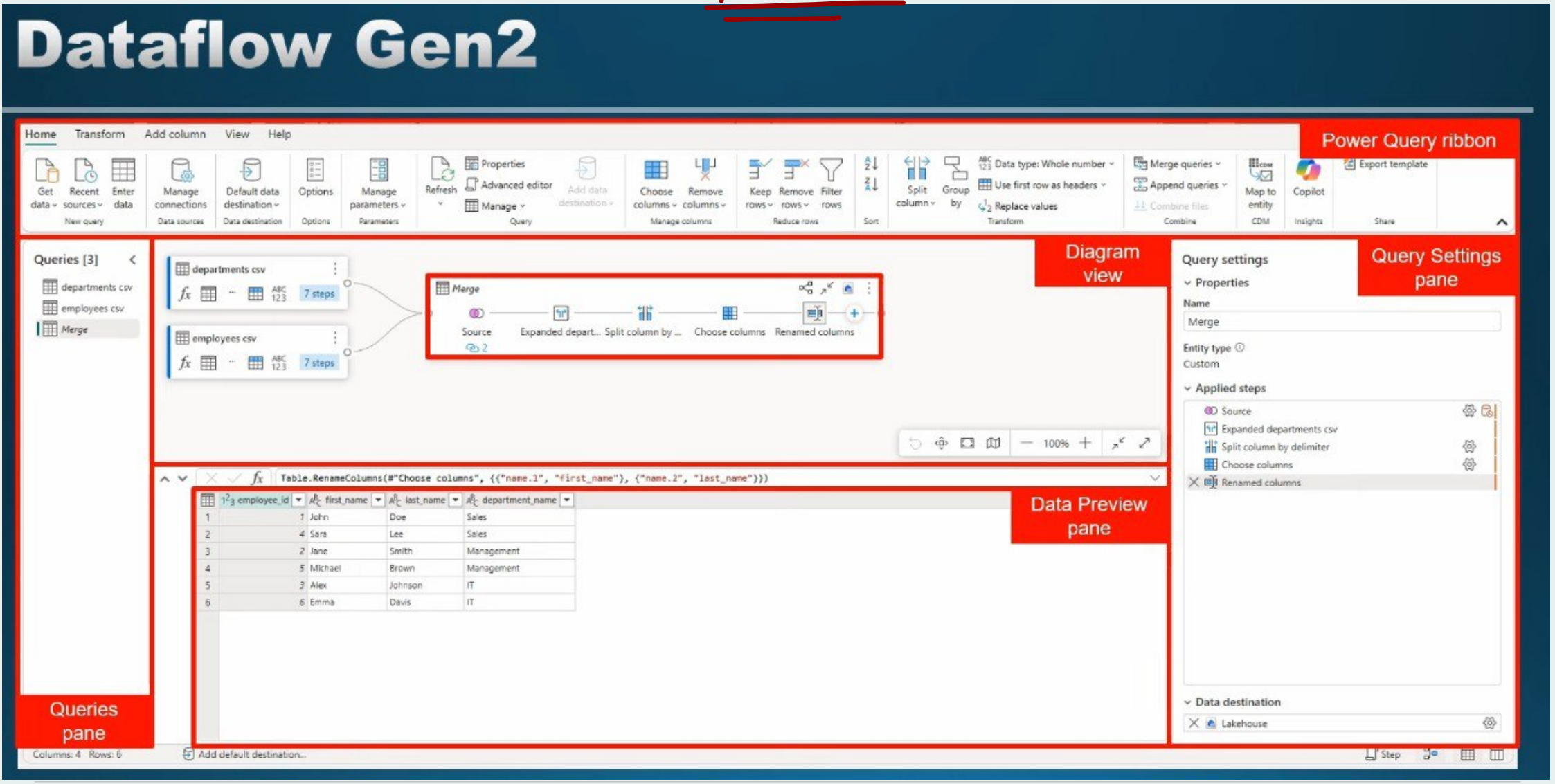
- are scoped at the pipeline level currently → are not safe thread and can cause unexpected & undesirable behaviour

- you can access a pipeline variable by using:

```
@variables('<variable name>')
```

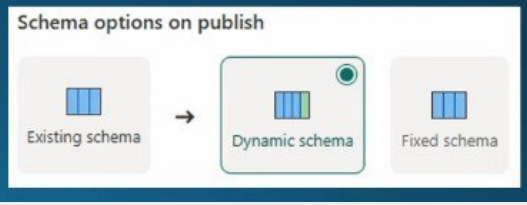


Dataflow Gen 2



- supported sources → many sources

- Supported destinations
  - Fabric Lakehouse
  - Fabric Warehouse
  - Fabric SQL database
  - Azure SQL database
  - Azure Data Explorer
- Support for both append and replace update methods
  - Azure Data Explorer don't support replace
- Some destination support dynamic schema changes for data
- Creating new tables when writing is supported with some limitations



M Query

```
let
  Source = Lakehouse.Contents(null){[workspaceId = "ccf8c8e3-e8c4-4e4d-b59f-b60d31dbbaff"]}[Data]{[lakehouseId =
    "d00dd570-352f-42d0-949a-c877d4b0b0d3"]}[Data],
  #"Navigation 1" = Source{[Id = "Files", ItemKind = "Folder"]}[Data],
  #"Navigation 2" = #"Navigation 1"{[Name = "dp700_e008"]}[Content],
  #"Navigation 3" = #"Navigation 2"{[Name = "departments.csv"]}[Content],
  #"Imported CSV" = Csv.Document(#"Navigation 3", [Delimiter = ",", Columns = 4, QuoteStyle = QuoteStyle.None]),
  #"Promoted headers" = Table.PromoteHeaders(#"Imported CSV", [PromoteAllScalars = true]),
  #"Changed column type" = Table.TransformColumnTypes(#"Promoted headers", {{"department_id", Int64.Type}, {"department_name",
    type text}, {"manager", type text}, {"location", type text}})
in
  #"Changed column type"
```

- M Query is the functional programming language used in Power Query
  - Also called M Language
- Dataflow generates M Query when creating queries

- Dataflows can be triggered manually
- Scheduling supported from the minute level to month level
  - Can be then monitored from the monitor hub
- Can be added to a Data Pipeline
  - Easy way to have a Dataflow as part of a bigger data solution