



SQL Saturday Atlanta 2025 - AI & BI (#1102)

08 March 2025

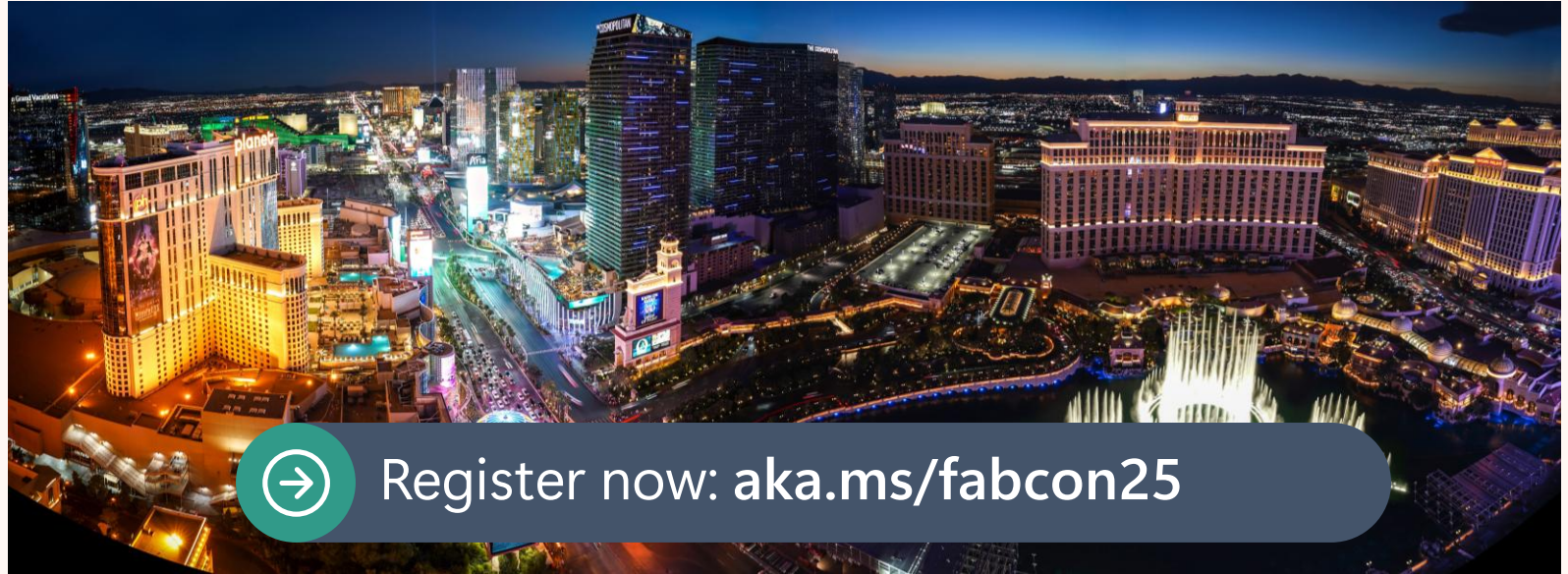
Fabric Pipelines
Metadata-driven ETL Patterns!
Mike Diehl Improving Winnipeg

Join us at the Fabric Community Conference

MGM GRAND, Las Vegas, NV

**March 31 -
April 2, 2025**

Workshops: March 29, 30, and
April 3



Register now: aka.ms/fabcon25

Join us at the second annual Microsoft Fabric Community Conference and get up close with the latest data, analytics, and AI developments—plus network with community leaders and other technical experts from around the world.

~~Use code FABINSIDER for a \$400 discount*~~

Use code DIEHL for a \$200 discount.

*Discount must be redeemed by March 18

Thanks to our sponsors

Please visit our sponsors!



Closing Ceremony

Please join us in the in the **auditorium** for the closing ceremony **right after the last session** of the day.

This is where sponsors will give **raffle prizes**.



Fabric Pipelines

Metadata-driven ETL Patterns

Mike Diehl
Director of Data Engineering and
Business Intelligence

Mike.Diehl@improving.com



/mikediehlsqlbi



1997

imagninet



improving*

18
OFFICES

Why use metadata for Fabric Pipelines?

- Reduces ETL development, increases velocity
- Compared to....SSIS
- Lakehouse tables - schema evolution (vs SQL database)
- Code and slides:

<https://github.com/xhead/SqlSaturdayATL-2025>

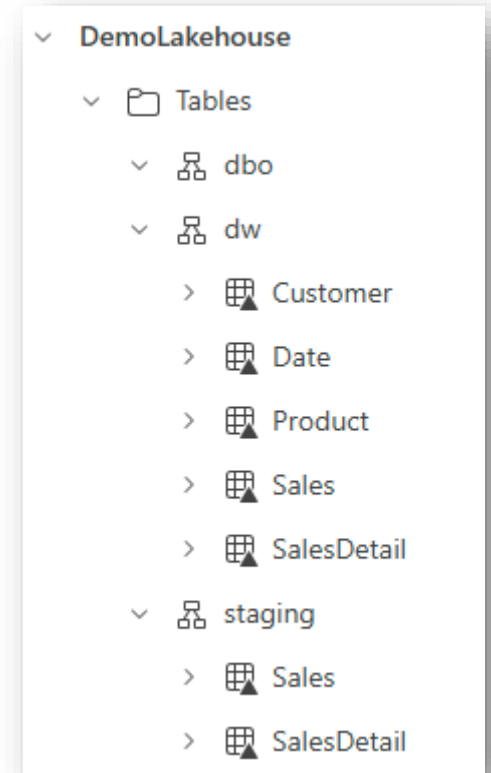
Survey:

- SQL Server Integration Services
- Azure Data Factory
- Fabric Pipelines

Scenario: Metadata-driven Approach

Target: Lakehouse in OneLake

- Set of tasks:
 - Get data from source
 - Load all data, Overwrite all data
 - Load some data in staging area, merge it into target (incremental)
- Dependencies
 - Load dimensions first, then facts that depend on dimensions
- Develop Pipelines at the Data Source type level
 - SQL server, Oracle server, plus authentication type
 - File source (File System, SharePoint, Azure) and type (XML, JSON, CSV)



Data Warehouses, Data Lakes, and Data Lakehouses



Data Warehouse

- Structured data
- Great for BI
- Quickly becomes expensive at larger data sets



Data Lake

- Unstructured data
- Great for BI and ML/AI
- Better economics for large data sets



Lakehouse

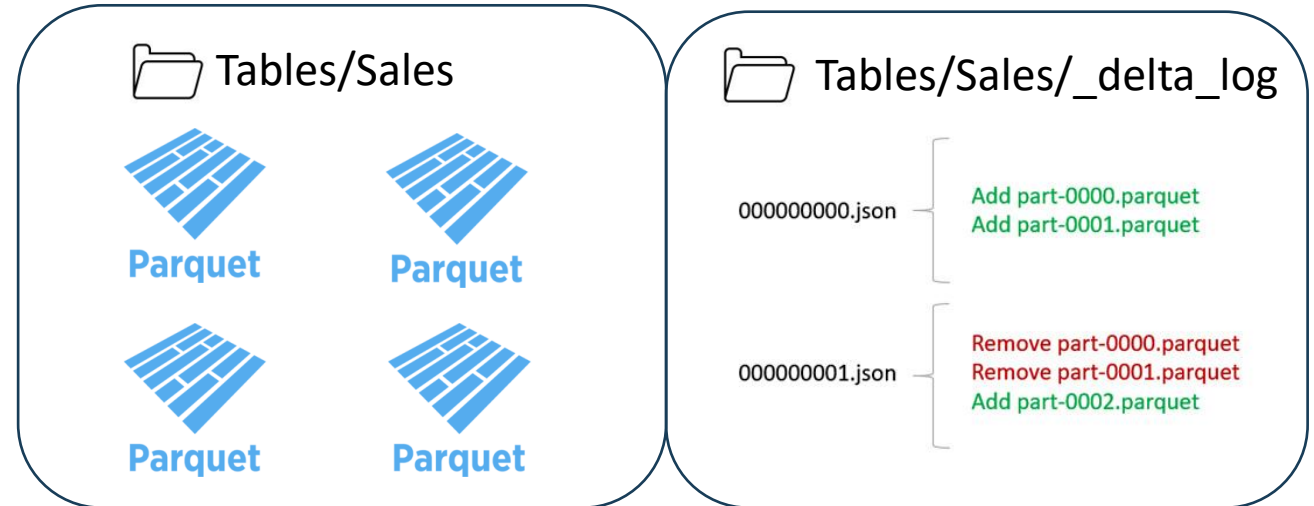
- Both structured and unstructured data in one place
- Blazing-fast
- Great for BI and ML/AI
- Great economics

Delta tables

OneLake Lakehouse



- Parquet files
 - Compressed, column-store data
 - Fast queries (like Vertipaq)
- Transaction log (json)
 - Time travel
- Easy schema evolution
- Optimize/vacuum
 - Rewrites/removes parquet files
- Easier/faster to drop/overwrite than to update
 - Unlike RDBMS: use change detection to minimize changes from staging to dw



Viewer Discretion Advised

These slides may contain screenshots that are disturbing.

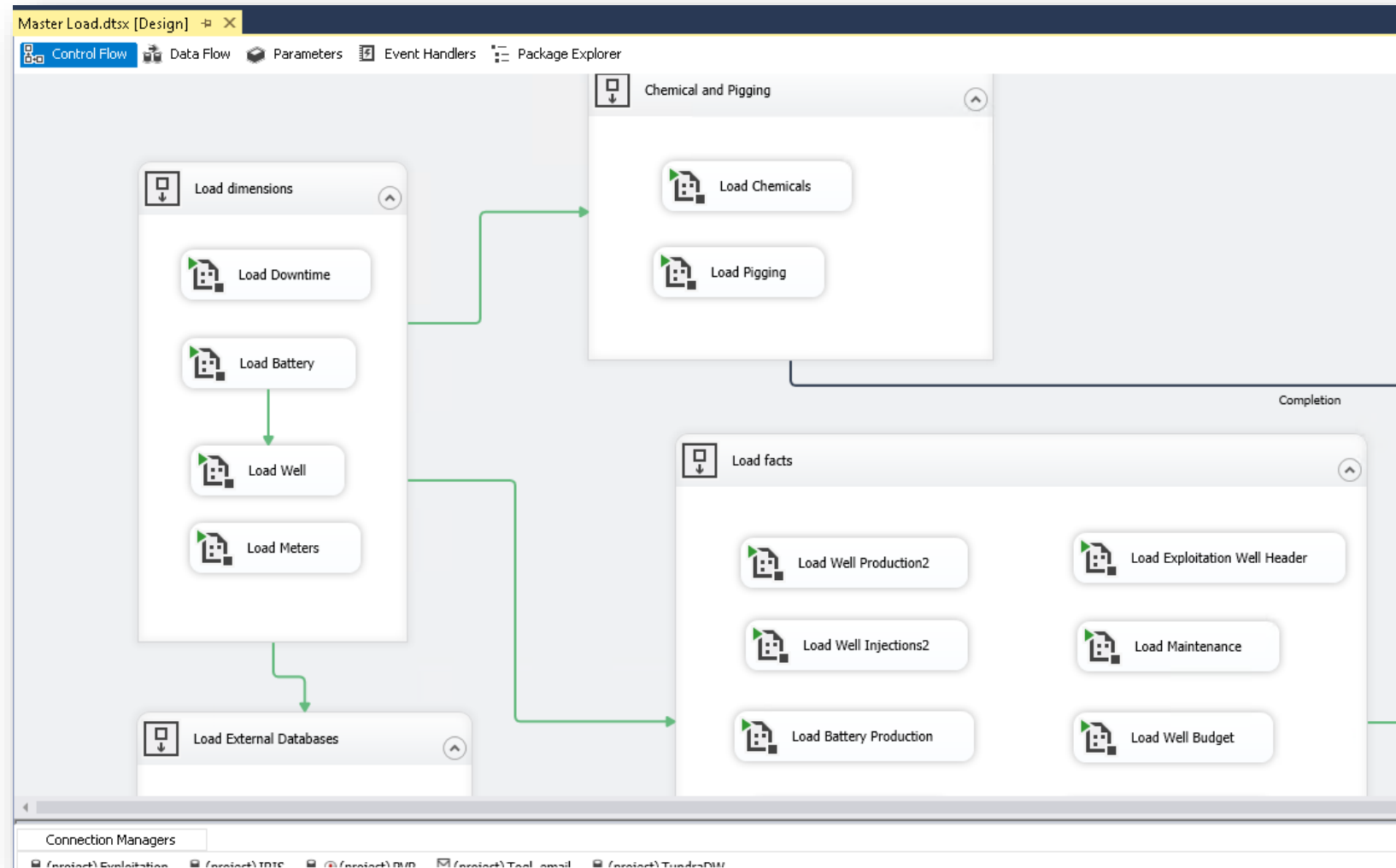
Those that have bad memories of SSIS development may be triggered.



SSIS Approach

Develop at the COLUMN Level

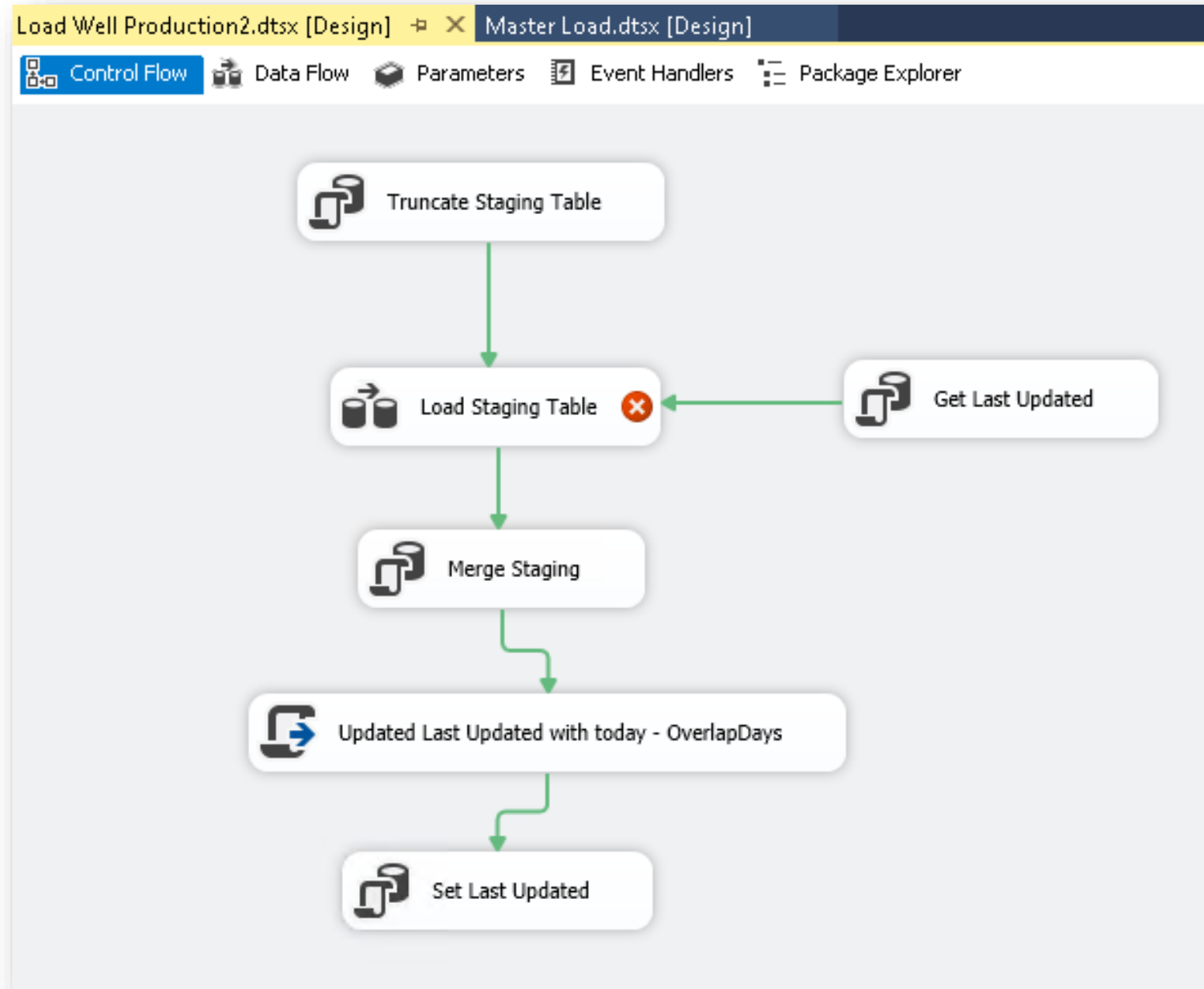
- Column changes require updating of mappings in Data flow tasks
- Redeploy updated SSIS packages



SSIS Approach

Develop at the COLUMN Level

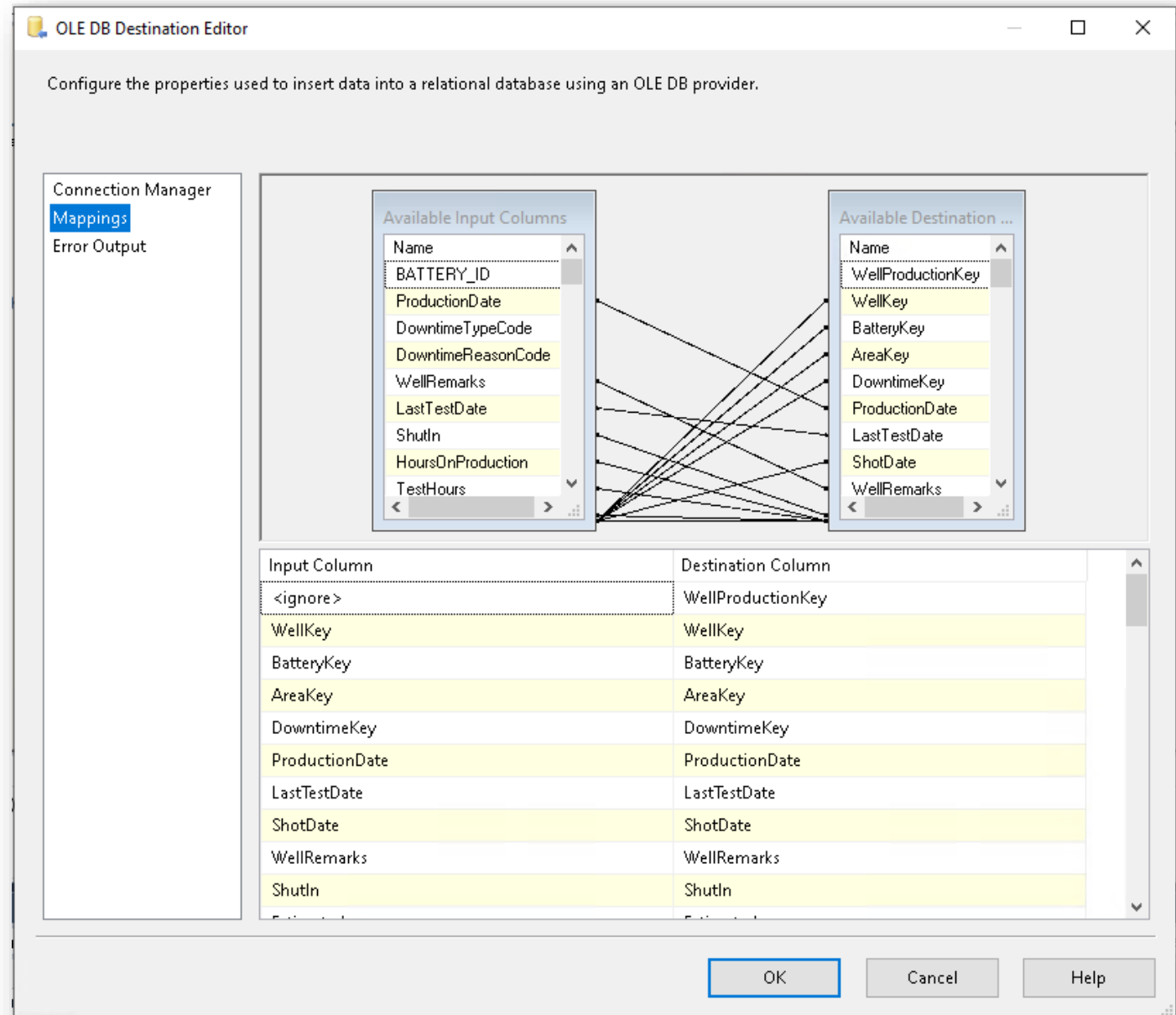
- Column changes require updating of mappings in Data flow tasks
- Redeploy updated SSIS packages



SSIS Approach

Develop at the COLUMN Level

- Column changes require updating of mappings in Data flow tasks
- Redeploy updated SSIS packages



Advantages of Metadata-driven Approach

ETL Change	SSIS	Deploy SSIS?	Fabric
New data from new source	New data connection, new package or dataflow,	Yes	New service connection, new pipeline, new metadata files
New data from existing source	New package or dataflow	Yes	Add task metadata files
Add/change/drop column in existing source	Refresh column metadata in source & target	Yes	Update task metadata files
Change dependency	Update package	Yes	Update task metadata files, update pipeline (sometimes)
Stop loading existing data	Remove or disable package	Yes	Update task metadata files

Fabric Pipeline features

Parameters


Parameters







Variables

Settings

Output

+ New

 Delete

<input type="checkbox"/>	Name	Type	Default value	
<input type="checkbox"/>	<input type="text" value="TaskFolder"/>	<input type="text" value="String"/> 	<input type="text" value="PipelineTasks"/>	
<input type="checkbox"/>	<input type="text" value="TaskFile"/>	<input type="text" value="String"/> 	<input type="text" value="SampleDBDaily.json"/>	
<input type="checkbox"/>	<input type="text" value="SourceQueryFilter"/>	<input type="text" value="String"/> 	<input type="text" value="***ALL***"/>	

Fabric Pipeline features

Dynamic Content

Add dynamic content [Alt+Shift+D]

`@pipeline().parameters.TaskFolder`

Pipeline expression builder

Add dynamic content below using any combination of [expressions](#), [functions](#) and [system variables](#).

`@pipeline().parameters.TaskFolder`

Clear contents

Parameters

System variables

Trigger parameters

Functions

Variables

Search

SourceQueryFilter

TaskFile

TaskFolder

OK

Cancel

Azure Data Factory → Fabric Pipelines

No more Datasets!



Azure Data Factory

The screenshot displays the Azure Data Factory portal interface. On the left, the 'Factory Resources' pane shows a tree view with 'Pipelines' (2 items), 'Datasets' (4 items), 'Data flows' (0 items), and 'Power Query' (0 items). The 'CsvDepartment' dataset is selected. The main pane shows the 'Sink dataset' configuration for a 'DelimitedText' source. The 'Sink dataset' is named 'GarysWineDW'. The 'Dataset properties' section shows 'TableName' as '@item().TargetTable' and 'SchemaName' as '@item().TargetSchema'. The 'Write behavior' is set to 'Insert'. The 'Bulk insert table lock' is set to 'No'. The 'Table option' is set to 'Use existing'. The 'Pre-copy script' is '@{item().TruncateStatement}'.

Factory Resources

- Pipelines (2)
 - Old Style (2...)
 - Load Department
 - Load Project
 - Change data capture (preview) (0)
- Datasets (4)
 - Old Style (4...)
 - CsvDepartment
 - CsvProject
 - StagingDepartment
 - StagingProject
- Data flows (0)
- Power Query (0)

Sink dataset *

Connection: **GarysWineDW** [Open](#) [New](#) [Learn more](#)

Dataset properties

Name	Value
TableName	@item().TargetTable
SchemaName	@item().TargetSchema

Write behavior

☒ Insert ☐ Upsert ☐ Stored procedure

Bulk insert table lock

☐ Yes ☒ No

Table option

☒ Use existing ☐ Auto create table

Pre-copy script

@{item().TruncateStatement}

Azure Data Factory → Fabric Pipelines

No more Datasets!



Azure Data Factory



Fabric Data Pipelines

General Source **Sink** Mapping Settings User properties

Sink dataset *

GarysWineDW Open New [Learn more](#)

Dataset properties ⓘ

Name	Value
TableName	<input type="text" value="@item().TargetTable"/>
Schema	<input type="text" value="ema"/>

General Source **Destination** Mapping Settings

Connection *

DemoLakehouse Refresh Open

Root folder

☒ Tables ☐ Files

Table *

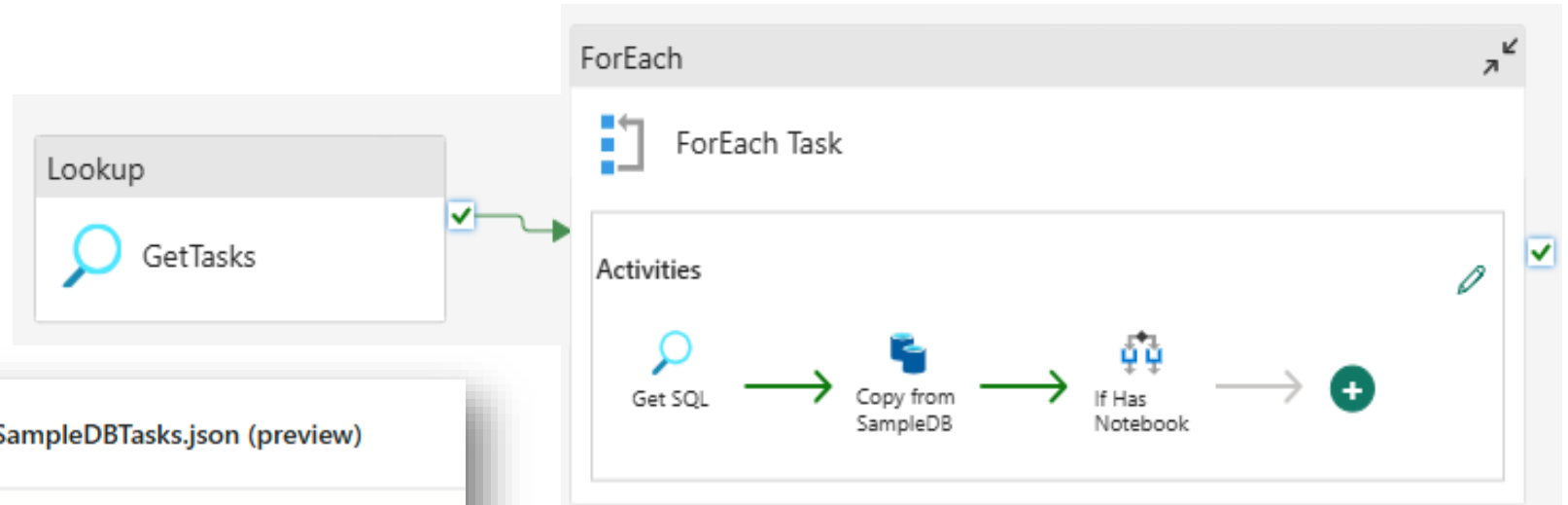
. Preview data New

☒ Enter manually

Typical Metadata-driven Pipeline Pattern



Fabric Pipeline



Lakehouse

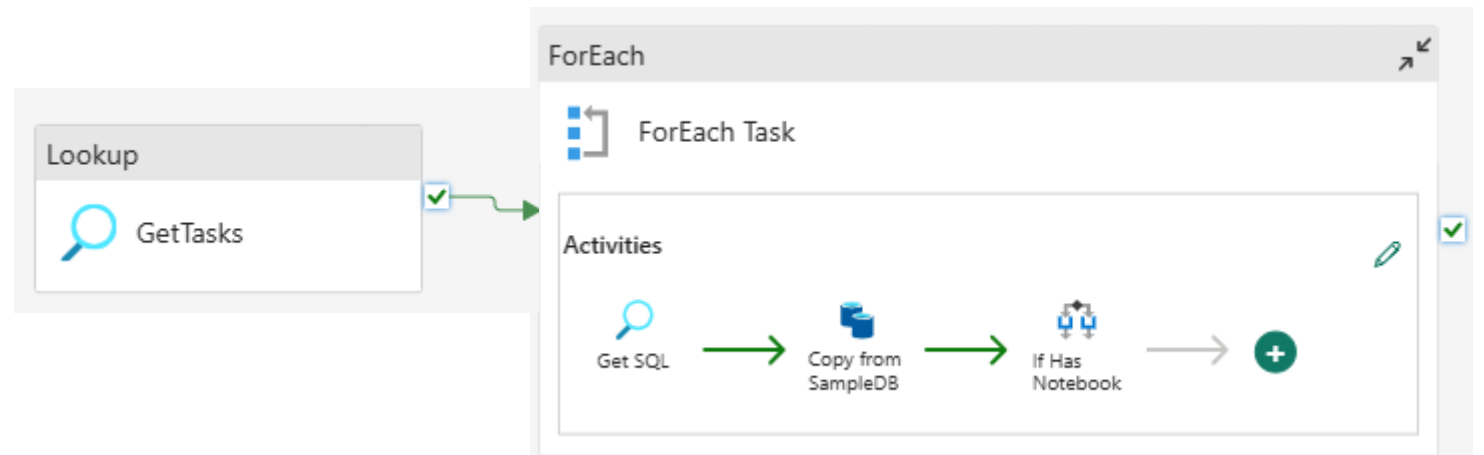
Files > PipelineTasks > SampleDBTasks.json (preview)

```
1  [  
2    {  
3      "source_query": "Products.sql",  
4      "target_schema": "dw",  
5      "target_table": "Product",  
6      "notebook": ""  
7    },  
8    {  
9      "source_query": "Customers.sql",  
10     "target_schema": "dw",  
11     "target_table": "Customer",  
12     "notebook": ""  
13   }  
14 ]
```

Fabric: Typical Pipeline

(per data source type)

- Get Tasks (Lookup)
 - Gets a json file from Lakehouse files
 - (filter for debug purposes)
- For Each Task
 - Get SQL query from Lakehouse file (Lookup)
 - Copy Data Task
 - Execute Notebook



Get Tasks (Lookup)

Gets a JSON file from Lakehouse

- Task folder and file name parameters
- Contains JSON array

General

Settings

Connection *

DemoLakehouse

Refresh

Open

Root folder

Tables

Files

File path type

File path

Wildcard file path

List of files ⓘ

File path

@pipeline().parameters.TaskFolder

/

@pipeline().parameters.TaskFile

Recursively ⓘ

☐

File format *

JSON


Settings

First row only

☐

> Advanced

Output

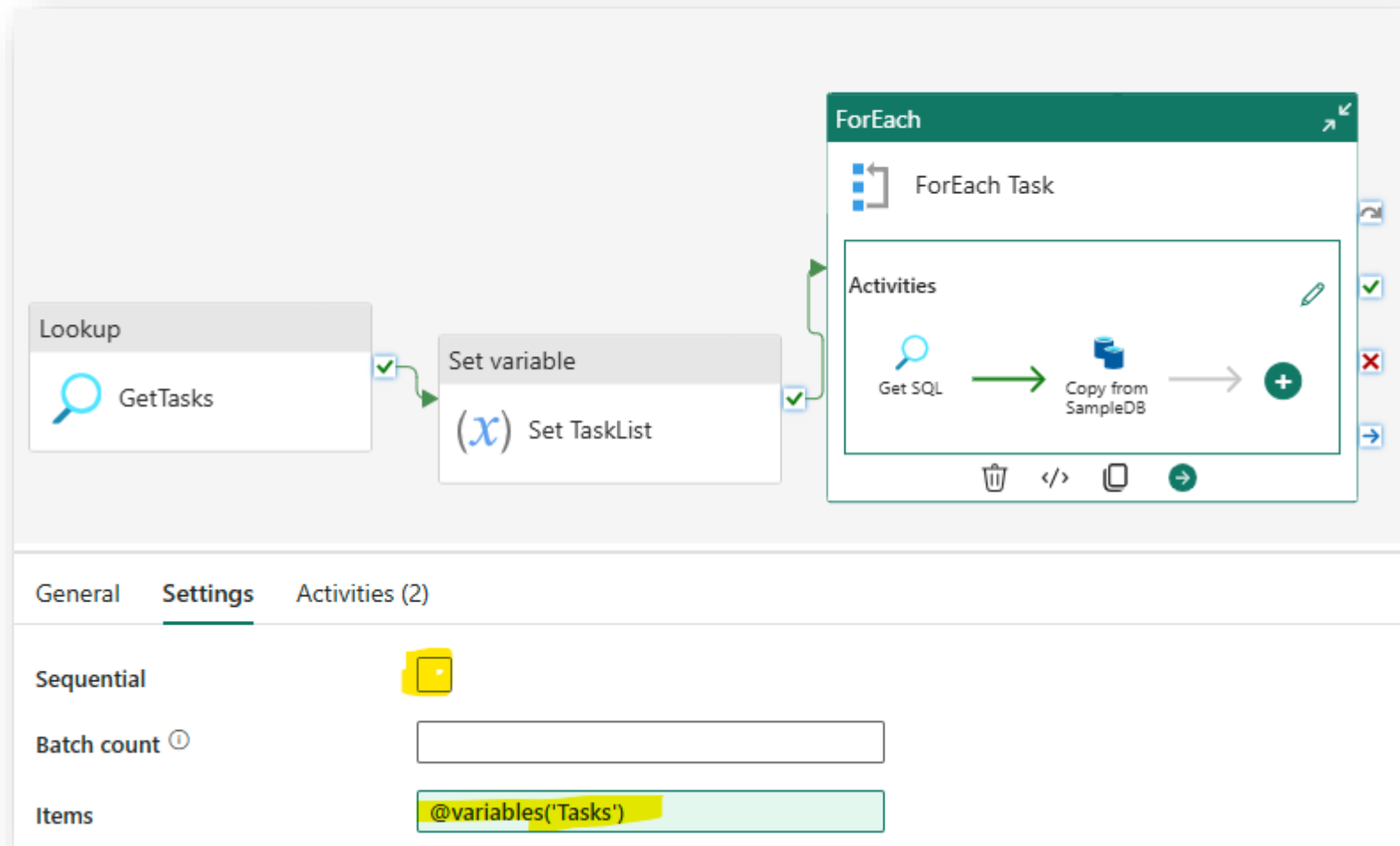
 Copy to clipboard

```
{
  "count": 5,
  "value": [
    {
      "source_query": "Products.sql",
      "target_schema": "dw",
      "target_table": "Product",
      "notebook": ""
    },
    {
      "source_query": "Customers.sql",
      "target_schema": "dw",
      "target_table": "Customer",
      "notebook": ""
    },
    {
      "source_query": "Date.sql",
      "target_schema": "dw",
      "target_table": "Date"
    },
    {
      "source_query": "CurrentYearSales.sql",
      "target_schema": "staging",
      "target_table": "Sales",
      "notebook": "MergeSales"
    },
    {
      "source_query": "CurrentYearSalesDetails.sql",
      "target_schema": "staging",
      "target_table": "SalesDetail",
      "notebook": "MergeSalesDetail"
    }
  ]
}
```

For Each Task

Iterates through item array

- Output of GetTasks
- Item() object
- Parallel or serial execution



For Each – Read SQL

Lookup task to get query text

- Read file as DelimitedText
- Set delimiters so only one row, one column is found
- No Header

The screenshot displays the configuration for a 'For Each' task in Azure Data Factory. The task is named 'For Each Task' and contains a sequence of activities: 'Get SQL', 'Copy from SampleDB', 'If Has Notebook', and a plus sign indicating further activities. The 'Settings' tab is active, showing the following configuration:

- Connection:** DemoLakehouse
- Root folder:** Files
- File path type:** File path
- File path:** @pipeline().parameters.TaskFolder / @item().source_query
- Recursively:** checked
- File format:** DelimitedText


A yellow highlight is present on the 'Settings' button at the bottom right of the settings pane.

For Each – Read SQL

Lookup task to get query text

- Read file as DelimitedText
- Set delimiters so only one row, one column is found
- No Header

File format settings

 Detect format

Compression type	No compression
Column delimiter ⓘ	#\$#\$#\$#\$#\$
Row delimiter ⓘ	##%##%##%##%
Encoding ⓘ	Default(UTF-8)
Quote character ⓘ	Double quote (")
Escape character ⓘ	Backslash (\)
First row as header ⓘ	<input type="checkbox"/>
Null value ⓘ	

For Each – Copy Data Task

Source

- Dynamic property for Source Query
- `@activity('Get SQL').output.firstRow.prop_0`

The screenshot displays the 'For Each' task configuration in Azure Data Factory. The top section shows a workflow diagram within the 'Activities' pane, consisting of four steps: 'Get SQL' (represented by a magnifying glass icon), 'Copy from SampleDB' (represented by a database icon), 'If Has Notebook' (represented by a document icon), and a final connector (represented by a plus sign icon). Below the diagram is a toolbar with icons for deleting, editing code, saving, and running the task.

The bottom section shows the configuration tabs: 'General', 'Source' (which is selected), 'Destination', 'Mapping', and 'Settings'. Under the 'Source' tab, the following settings are visible:

- Connection ***: A dropdown menu showing 'SampleDB'.
- Use query**: Three radio buttons labeled 'Table', 'Query' (which is selected), and 'Stored procedure'.
- Query**: A text box containing the dynamic property `@activity('Get SQL').output.firstRow....`.
- > Advanced**: A link to expand advanced settings.

For Each – Copy Data Task

Destination

- Dynamic properties for Target Schema & Table
- Overwrite
- Target table may not exist
 - But schema needs to exist

The screenshot displays the 'For Each' task configuration interface. The top section shows a workflow diagram with four activities: 'Get SQL', 'Copy from SampleDB', 'If Has Notebook', and a final step represented by a plus sign. Below this, the 'Destination' tab is selected, showing configuration options for the data destination.

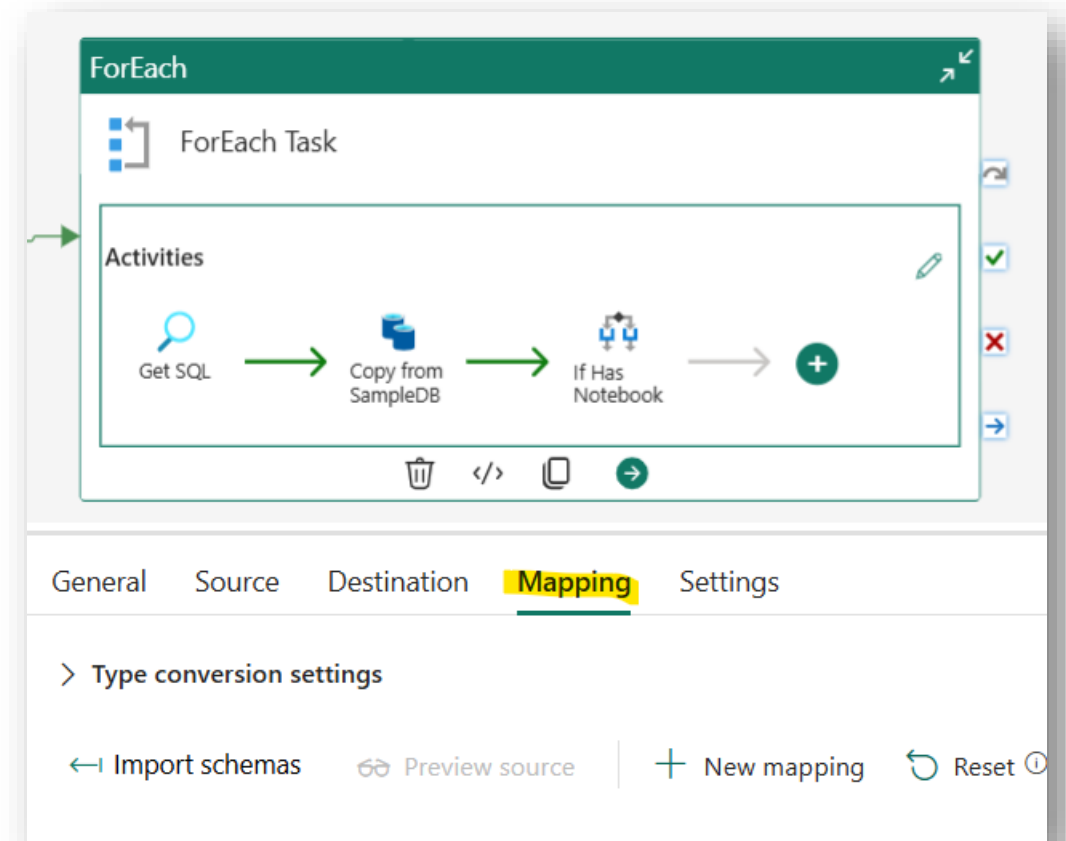
Destination Tab Configuration:

- Connection:** DemoLakehouse
- Root folder:** ☒ Tables ☐ Files
- Table:** `@item().target_schema` . `@item().target_table`
- ☒ Enter manually
- Table action:** ☐ Append ☒ Overwrite

For Each – Copy Data Task

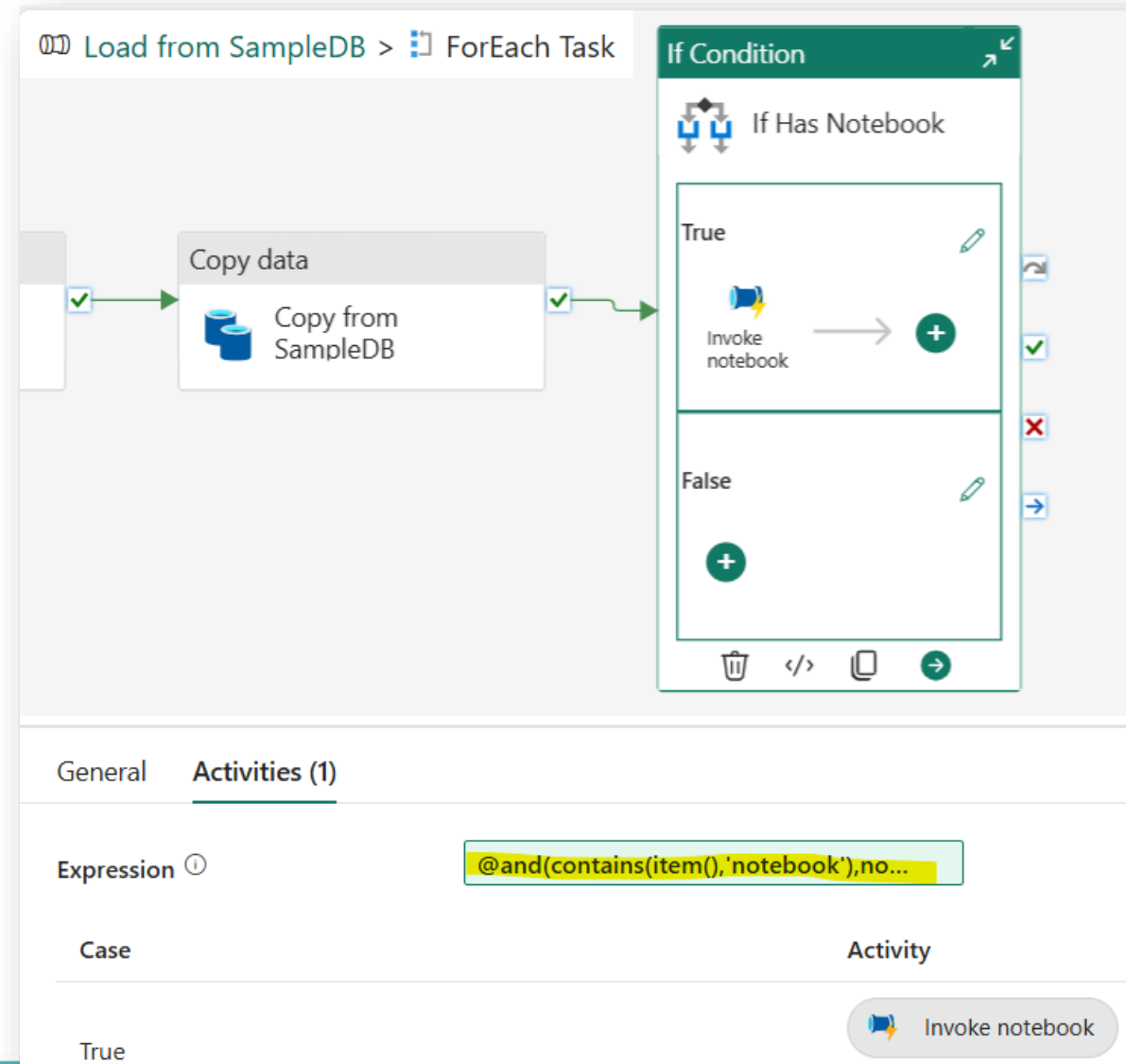
Mapping

- **Zero** column mapping (very different from SSIS)



For Each – Execute notebook

- Execute notebook to process staging data
- `@and(contains(item(),'notebook'), not(empty(item().notebook)))`



For Each – Execute notebook

- Calls another pipeline to execute the notebook

General **Settings**

i Currently Pipeline return value is only supported with ADF & Synapse pipelines. To fetch pipeline return value for Fabric pipelines, p

Type ☒ Fabric ☐ Azure Data Factory ☐ Synapse

Connection * ⓘ

FabricDataPipelines Miked

 Refresh Edit

Workspace *

MetaPipelines

 Refresh

Pipeline *

ExecuteNotebook

 Refresh Open + New

Parameters

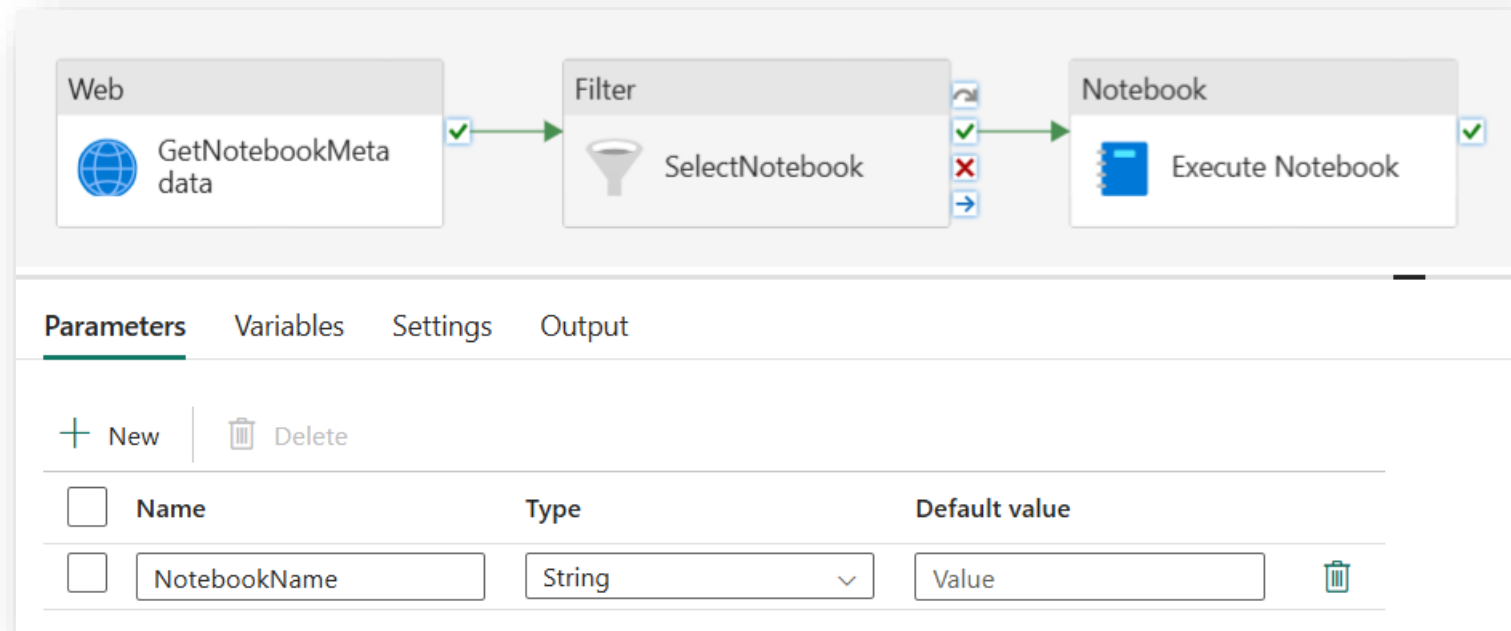
+ New | Delete

<input type="checkbox"/>	Name	Type	Value
<input type="checkbox"/>	NotebookName	String	@item().notebook

For Each – Execute notebook

Invoke Notebook pipeline

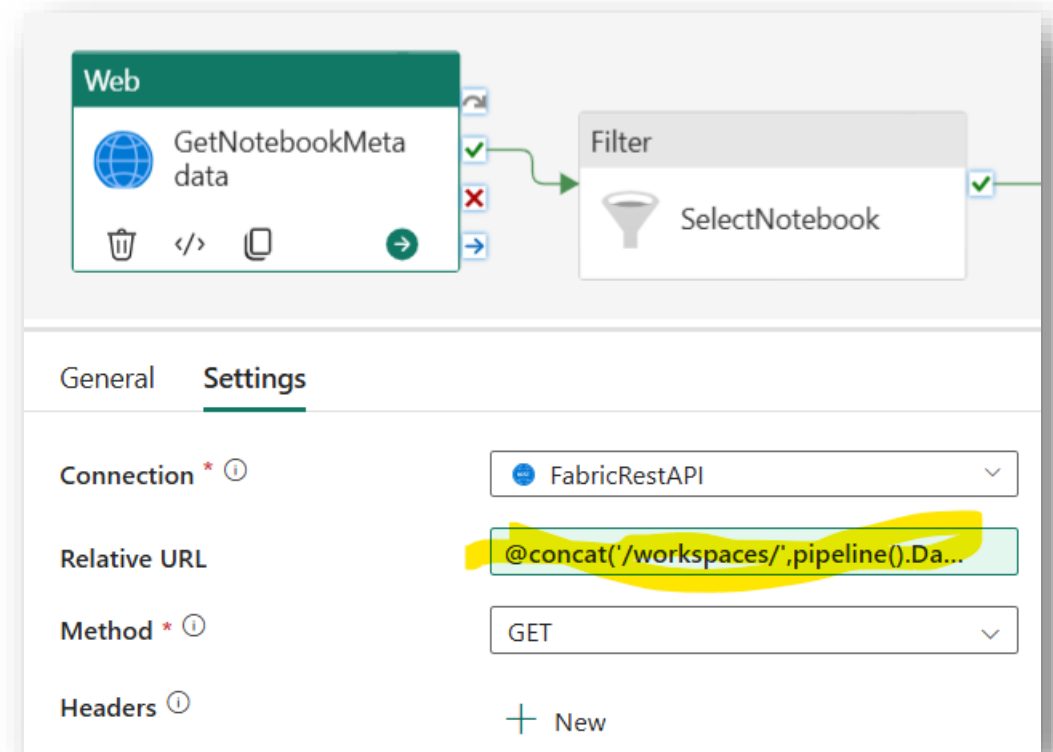
- Need the resource ID of the notebook, not the name
- REST API gets all notebook metadata
- Filter selects by name
- Dynamic properties for notebook execution task



For Each – Execute notebook

Invoke Notebook pipeline

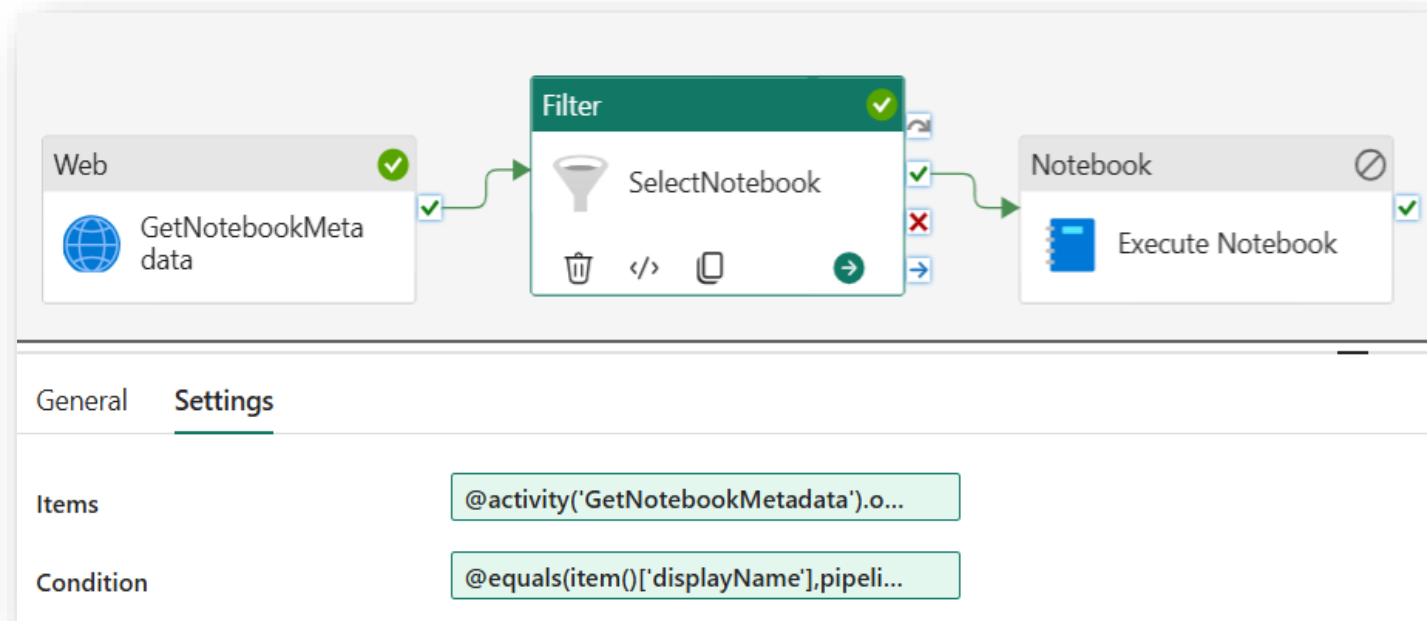
- Need the **resource ID** of the notebook, **not the name**
- REST API gets all notebook metadata
- Base URL:
<https://api.fabric.microsoft.com/v1/>
- Relative URL:
`@concat('/workspaces/', pipeline().DataFactory, '/items?type=Notebook')`
 - (all notebooks in this workspace)



For Each – Execute notebook

Invoke Notebook pipeline

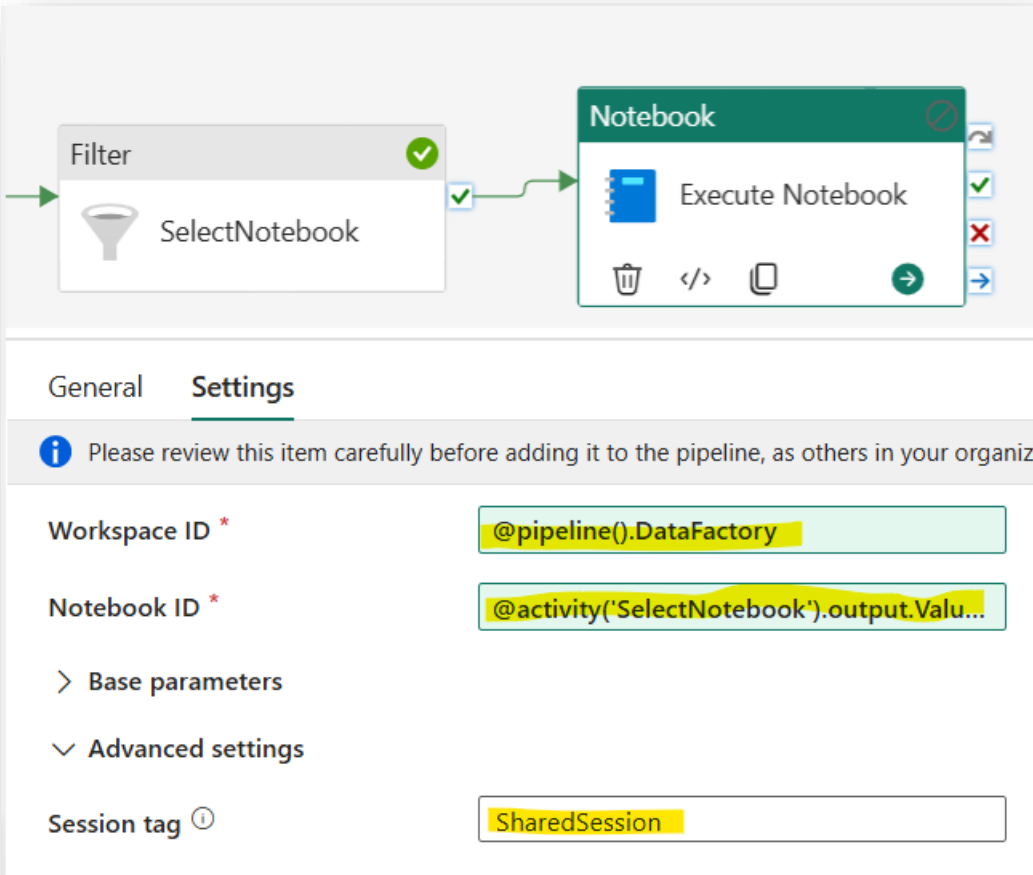
- Filter selects by name
- Items: `@activity('GetNotebookMetadata').output.value`
- Condition: `@equals(item()['displayName'], pipeline().NotebookName)`



For Each – Execute notebook

Invoke Notebook pipeline

- Workspace ID:
`@pipeline().DataFactory`
("this workspace")
- Notebook ID:
`@activity('SelectNotebook')`
`.output.Value[0].id`
- Session tag



The screenshot displays the Azure Data Factory pipeline editor. At the top, a pipeline diagram shows a 'Filter' activity (labeled 'SelectNotebook') connected to an 'Execute Notebook' activity. Below the diagram, the 'Settings' tab is active, showing the following configuration:

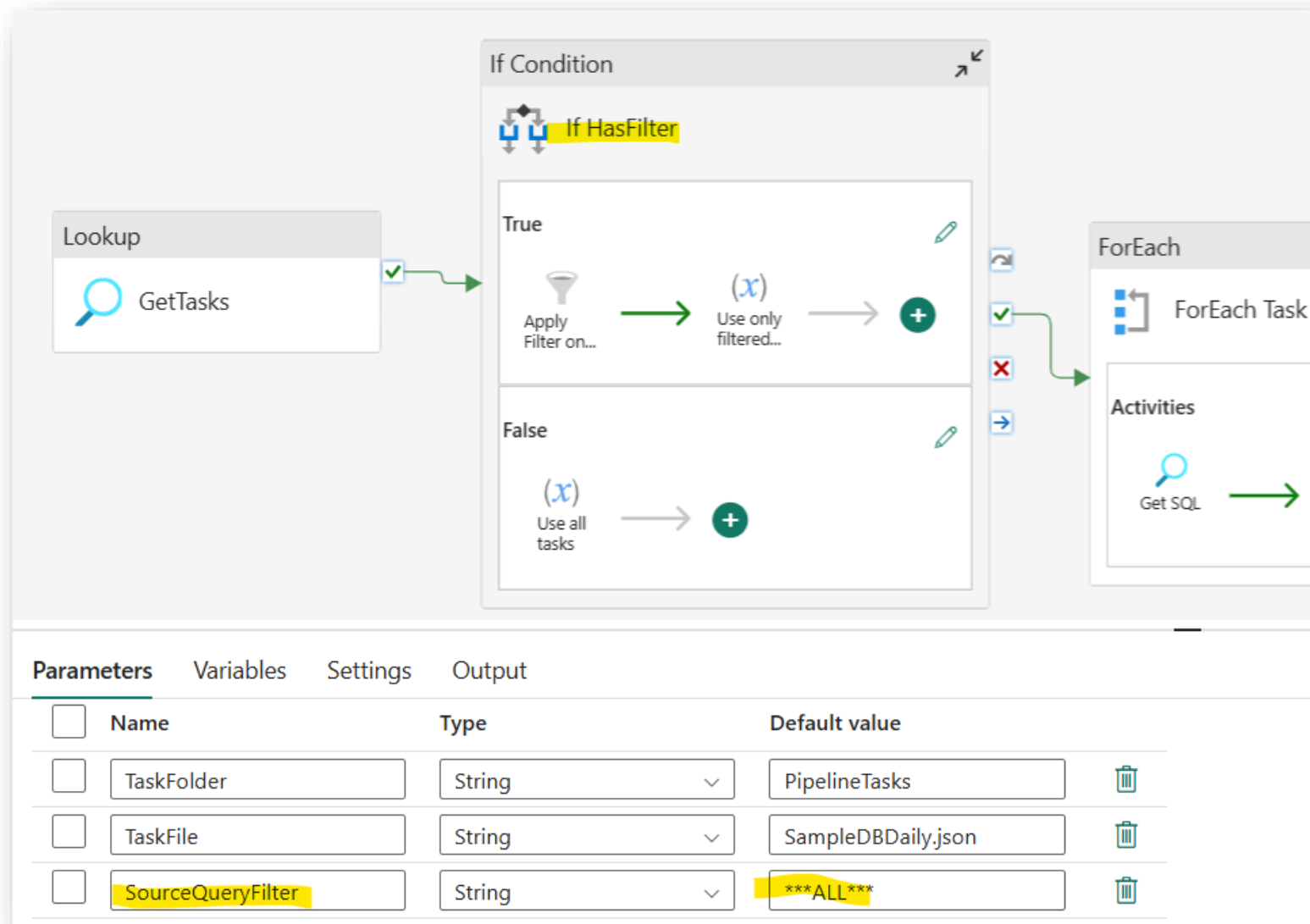
- Workspace ID ***: `@pipeline().DataFactory`
- Notebook ID ***: `@activity('SelectNotebook').output.Value[0].id`
- Base parameters**: (collapsed)
- Advanced settings**: (collapsed)
- Session tag ①**: `SharedSession`

Debugging tasks

Filter by task name

- If HasFilter:

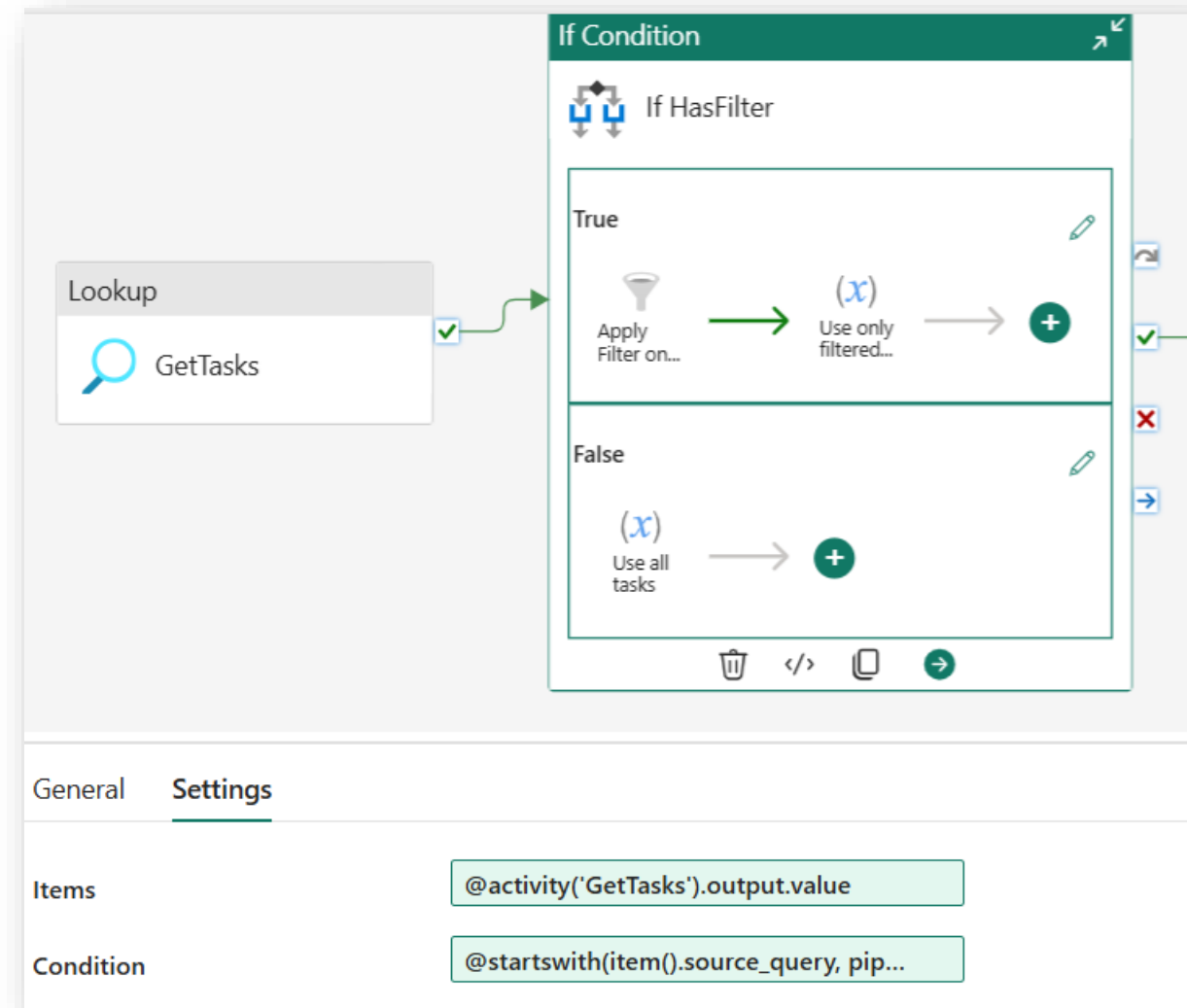
```
@not(equals(pipeline()  
  .parameters.SourceQueryFilter  
  , '***ALL***'))
```



Debugging tasks

Filter by task name

- Filter Condition:
`@startswith(item().source_query, pipeline().parameters.SourceQueryFilter)`
- Set tasks variable



How Many Pipelines?

- Usually One per Data Source, plus one Main pipeline
- Most pipelines use the Lookup/ForEach/Copy Data/Exec notebook pattern

Main Pipeline

- Calls Task-driven pipelines with parameter for task file
- Main pipeline handles dependencies
 - “Level 1” pipelines – loading dimensions
 - “Level 2” pipelines – loading facts
 - Additional levels as needed, based on dependency tree
- Advanced: Main pipeline can use the Lookup/ForEach pattern to get a list of pipelines (and their parameters) to execute
- Could arrange pipelines in medallion order

Bronze



Raw

Silver



Validated

Gold



Enriched

Notebooks

Incremental loading

- Staging table contains complete data for a range of dates
 - CurrentYearSales.sql (from Jan 1 of this year)
 - PastYearSales.sql (before Jan 1 of this year)
- Notebook: MergeSales

```
%run CommonLakehouseMerge  
MergeByDateRange("staging.Sales", "dw.Sales", "OrderDate")
```

```
1 def MergeByDateRange(staging_table, target_table, date_range_replace_column):
```

Notebooks

Incremental loading

- Staging table contains data identified by natural key
 - CurrentYearSalesDetails.sql (no Order Date)
- Notebook: MergeSalesDetails

```
%run CommonLakehouseMerge  
MergeUpsert("staging.SalesDetail", "dw.SaleDetail",  
"SalesOrderID")
```

```
def MergeUpsert(staging_table, target_table, key_column):
```


CommonLakehouseMerge

MergeByDateRange

- Check that target table exists
- Gets Min and Max values
- Builds ReplaceWhere clause
- Merge Schema

```
1 def MergeByDateRange(staging_table, target_table, date_range_replace_column):
2
3     target_df = spark.sql(f"select * from {target_table}")
4     source_df = spark.sql(f"select * from {staging_table}")
5
6     #check if table has any columns or has already columns but they are empty
7     if len(target_df.columns)==0 or target_df.isEmpty():
8         # first time insert when nothing to delete
9         (source_df
10          .write
11          .format("delta")
12          .mode("overwrite")
13          .option("mergeSchema", "true")
14          .saveAsTable(target_table)
15         )
16     else:
17         # insert with replacement by 'date_range_replace_column'
18         min_date = source_df.agg(min(col(date_range_replace_column))).collect()[0][0]
19         max_date = source_df.agg(max(col(date_range_replace_column))).collect()[0][0]
20         replace_where = f"`{date_range_replace_column}` between '{min_date}' and '{max_date}'"
21         (source_df
22          .write
23          .format("delta")
24          .mode("overwrite")
25          .option("replaceWhere", replace_where)
26          .option("mergeSchema", "true")
27          .saveAsTable(target_table)
28         )
```

CommonLakehouseMerge

MergeUpsert

- Opens native DeltaTable from Lakehouse path
- Merge with match clause
- Match – update
- No match – insert
- Schema evolution

```
1  def MergeUpsert(staging_table, target_table, key_column):
2
3      #read from delta tables to spark dataframes
4      target_df = spark.sql(f"select * from {target_table}")
5      source_df = spark.sql(f"select * from {staging_table}")
6
7      #check if table has any columns or has already columns but they are empty
8      if len(target_df.columns)==0 or target_df.isEmpty():
9
10         )
11     else:
12         #read DeltaTable from Lakehouse path
13         lakehouse_table_path = f"{get_lakehouse_path()}/Tables/{target_table.replace('.', '/')}"
14         target_dt = DeltaTable.forPath(spark, lakehouse_table_path)
15
16         # Alias the columns to avoid name conflicts
17         target_dt = target_dt.alias("target")
18         source_df = source_df.alias("source")
19
20         (target_dt
21             .merge(source_df, f"source.`{key_column}` = target.`{key_column}`")
22             .whenMatchedUpdateAll()
23             .whenNotMatchedInsertAll()
24             # .whenNotMatchedBySourceDelete()
25             .withSchemaEvolution()
26             .execute()
27         )
28
29 )
```

Demo

Changing ETL Requirements

- New table in target from same source?

~~Add Staging and dw tables,~~

Add an item to task file

Add query file

No pipeline changes required

Merge Notebook (maybe)

- Change in target table? (Add column, remove column, change column)

~~Change Staging and dw tables,~~

Update query file

Merge Notebook (maybe)

- Incorrect logic in SourceQuery?

Update query file

- Target table no longer needed?

Remove item from task file

Advantages of Metadata-driven Approach

ETL Change	SSIS	Deploy SSIS?	Fabric
New data from new source	New data connection, new package or dataflow,	Yes	New service connection, new pipeline, new metadata files
New data from existing source	New package or dataflow	Yes	Add task metadata files
Add/change/drop column in existing source	Refresh column metadata in source & target	Yes	Update task metadata files
Change dependency	Update package	Yes	Update task metadata files, update pipeline (sometimes)
Stop loading existing data	Remove or disable package	Yes	Update task metadata files

Questions?

Session Evaluation



Code and slides:

<https://github.com/xhead/SqlSaturdayATL-2025>

Mike Diehl, Mike.Diehl@improving.com
Director of Data Engineering and Business
Intelligence



/mikediehlsqldb