data lab

Fabric Notebook Orchestration

Comparison between different methods

Thank you sponsors!



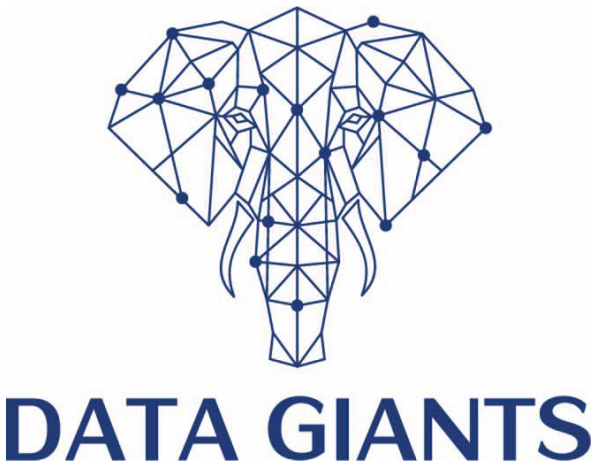
























Who am I

Ásgeir Gunnarsson

Data Platform MVP

Partner, Data Platform Architect @ data lab

20+ years in BI
Co-organizer of Power BI Next Step, Data Platform Next Step
Co-author of Pro Microsoft Power BI Administration

E-mail: asgeir@thesmalldatalab.com
Bluesky: @bidgeir
LinkedIn: linkedin.com/in/asgeirgun



We're The Small Data Lab – a modern Danish data consultancy helping companies access and create real value from data across the entire organisation.

We work with businesses that want more than just dashboards. Our focus is on delivering insight, clear governance, and data platforms that evolve with both people and technology – on site or remotely.

Agenda

- Intro & orchestration fundamentals
- Orchestration with pipelines
- Orchestration with notebooks
- Orchestration with Materialized lake views
- Orchestration with third party tools
- Monitoring
- What to choose
- Q&A

Format

- This session is based on my experience
- You might have different experiences
  - I would love to hear about it
- We have good time
  - Not enough for hands-on exercises
  - But enough for a good discussions
- Let’s keep the discussions going

Why this talk exists

- “Should we orchestrate with Pipelines or Notebooks?”
- “Can we keep using our favourite tool?”
- “What is the best practice?”
- Reality - there is no single best answer

What do we mean by orchestration

- Scheduling workloads
- Dependency management
- Retry and failure handling
- Parameter passing
- Monitoring & observability

The Fabric orchestration landscape

- Fabric Pipelines (native control plane)
- Notebook orchestration (code-driven control plane)
- Materialized lake views (code-driven control plane)
- Third-party orchestration (external control plane)

ORCHESTRATION FUNDAMENTALS

- Spark notebooks → data plane
- Orchestration → control plane
- Mixing the two creates trade-offs
- Separation of concerns vs simplicity

What makes orchestration “good”

- Observability & monitoring
- Failure handling & retries
- Dependency management
- CI/CD & environments
- Governance & security
- Developer productivity

ORCHESTRATION WITH FABRIC PIPELINES

Using the control plane

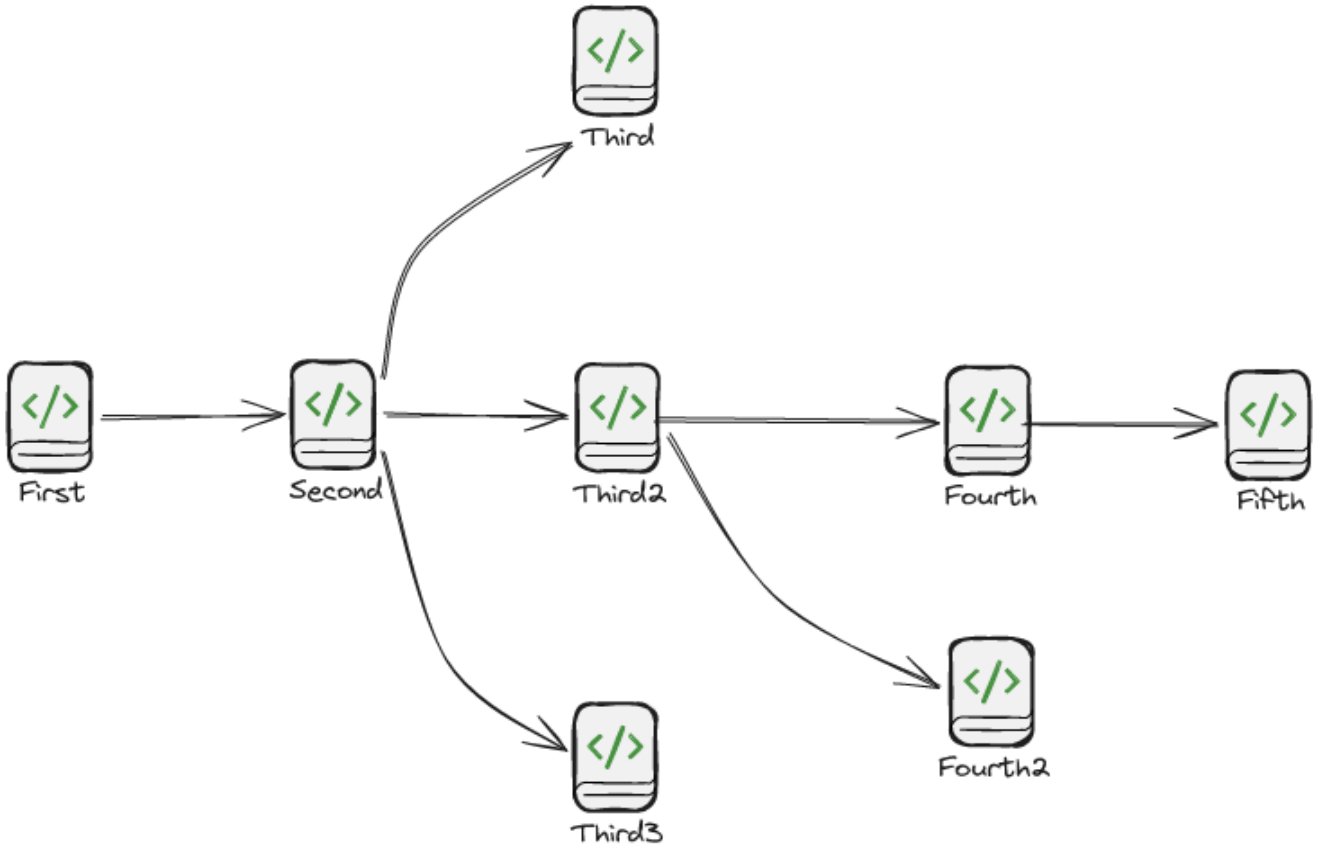
Fabric Pipelines overview

- Successor to Azure Data Factory pipelines
- Visual workflow orchestration
- Native integration with Fabric items
- Low-code control plane

How pipelines orchestrate notebooks

- Notebook activity
- Parameter passing
- Sequential & parallel execution (like this)
- Triggers & schedules
- Monitoring UI

So, this will be difficult/complex



How pipelines orchestrate notebooks

- Notebook activity
- Parameter passing
- Sequential & parallel execution
- Triggers & schedules
- Monitoring UI

Sequential or parallel

- By default, each notebook will run in parallel
  - One cluster per notebook
  - Can consume a lot of CUs if the number of notebook is large
  - Pipelines are really good at closing clusters after run – so it's not all bad
- If you want them to run in parallel (and not consume as much CUs), you need to say so
  - This means you need to
    - turn on high concurrency
    - Tell the activity to use high concurrency

High concurrency in pipelines

- Turn on in the Workspace settings
- Session tag added to activity
- Will be run by the same user (the person/identity who owns the pipeline)
- Have to:
  - Have the same default lakehouse. Notebooks without a default lakehouse can share sessions with other notebooks that don't have a default lakehouse.
  - Have the same Spark compute configurations.
  - Have the same library packages. You can have different inline library installations as part of notebook cells and still share the session with notebooks having different library dependencies.

https://learn.microsoft.com/en-us/fabric/data-engineering/high-concurrency-overview

High concurrency in pipelines

- Session sharing applies when these conditions are met:
  - Sessions are within a single-user boundary.
  - Sessions use the same default Lakehouse configuration.
  - Sessions use the same Spark compute settings.
- **If any requirement differs, Fabric starts a separate Spark session**
- **Need to be aware of how much compute you have**

High concurrency in pipelines

- By default, new sessions (cluster) automatically created for each 5 notebook
- Pipelines have default of 20 and max 50 parallel activities in a for each loop
- Max 50 notebooks per high concurrency session
  - Default 5
  - Need to set anything above 5 in an Environment or in the notebook activity*

https://blog.fabric.microsoft.com/en-us/blog/fabric-march-2026-feature-summary?ft=All#post-34196-_Toc224559611

High concurrency in pipelines

- To change the maximum notebooks to run in a high concurrency session
  - Workspace → Environments
  - If the environment is not in the workspace default environment –pool > means longer startup time
- Select the Environment attached to your notebook or pipeline
- Open Spark Properties
- Add the High Concurrency configuration
- Set spark.highConcurrency.max to a value between 2 – 50

Note: This update does not change the default limit of five

Pipeline demo

Pipeline strengths

- Clear separation of control vs compute
- Built-in monitoring UI
- Scheduling & triggers out-of-the-box
- Retry & failure handling built-in
- Low barrier for mixed skill teams

Pipeline weaknesses

- Limited dynamic logic
- Can become quite complex
- Harder to version & test than code
  - Getting much easier with variable libraries
- Large solutions → pipeline sprawl
- Lower concurrency threshold than notebooks

When pipelines are ideal

- Enterprise governed environments
- Many stakeholders
- Clear operational ownership
- Batch workloads
- Traditional medallion pipelines
- Similar notebooks throughout

ORCHESTRATION WITH FABRIC NOTEBOOKS

Using the data plane

Notebook orchestration overview

- One notebook calling other notebooks
- Pure code orchestration
- Often used by data engineering teams

How it works technically

- notebook.run() /notebookutils.notebook.run/ notebookutils.notebook.runMultiple
- Parameter passing
- Loops & dynamic logic
- **Shared Spark context**
- **Alternatively run via API**
  - Technically harder than runMutliple

runMultiple

- Build a DAG (Directed Acyclic Graph) to control:
  - Which notebooks run
    - Name needs to be unique
  - What dependencies are there
  - Parameters to pass to the notebook
  - Timeouts, retries and other settings

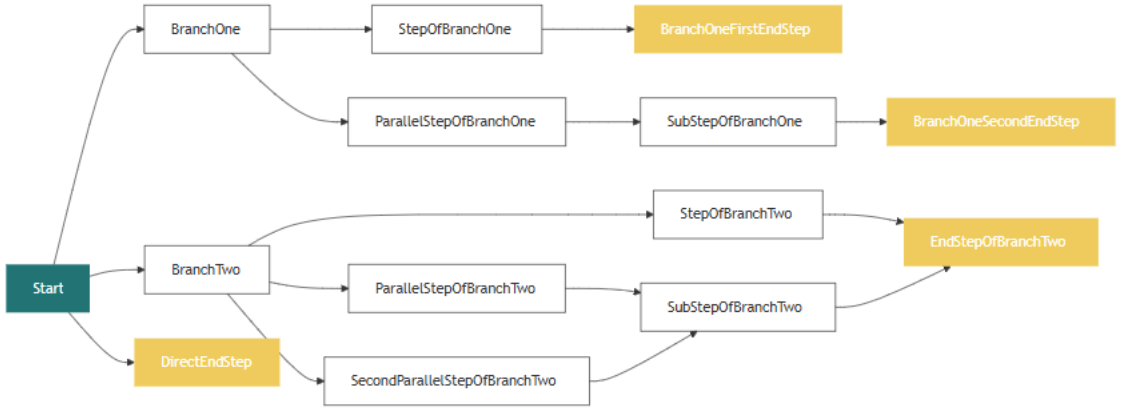


Image from Štěpán Rešl - Orchestration Fabric Notebooks · Data Meerkat

runMultiple

```
# run multiple notebooks with parameters
DAG = {
    "activities": [
        {
            "name": "Process_1", # activity name, must be unique
            "path": "NotebookSimple", # notebook item name
            "timeoutPerCellInSeconds": 90, # max timeout for each cell, default to 90 seconds
            "args": {"p1": "changed value", "p2": 100}, # notebook parameters
            "workspace":"WorkspaceName" # both name and id are supported
        },
        {
            "name": "Process_2",
            "path": "NotebookSimple2",
            "timeoutPerCellInSeconds": 120,
            "args": {"p1": "changed value 2", "p2": 200},
            "workspace":"id" # both name and id are supported
        },
        {
            "name": "Process_1.1",
            "path": "NotebookSimple2",
            "timeoutPerCellInSeconds": 120,
            "args": {"p1": "changed value 3", "p2": 300},
            "retry": 1,
            "retryIntervalInSeconds": 10,
            "dependencies": ["Process_1"] # list of activity names that this activity depends on
        }
    ],
    "timeoutInSeconds": 43200, # max timeout for the entire DAG, default to 12 hours
    "concurrency": 50 # max number of notebooks to run concurrently, default to 50
}
notebookutils.notebook.runMultiple(DAG, {"displayDAGViaGraphviz": False})
```

runMultiple

```
    notebook_name = table['notebookName']
    target_table = table['sourceTable']
    file_name = table['tableName']
    environment = table['environment']
    table_key_list = table['primaryCompositeKey']
    database_name = table['databaseName']
    source = table['SourceName']
    load_type = table['loadType']
    watermark_column = table['watermarkAttribute']
    watermark_value = table['watermarkValue']
    workspace_id = table['targetWorkspace']
    lakehouse_id = table['targetLakehouse']

    args = {
        'tableName': target_table,
        'fileName': file_name,
        'primaryKey': table_key_list,
        'environment': environment,
        'databaseName': database_name,
        'system': source,
        'loadType': load_type,
        'watermarkColumn': watermark_column,
        'watermarkValue': watermark_value,
        'DW_ID_Audit_Insert': AuditId,
        'targetWorkspace': workspace_id,
        'targetLakehouse': lakehouse_id,
        'useRootDefaultLakehouse': True
    }

activity = {
    'name': target_table,
    'path': notebook_name,
    'args': args,
    'timeoutPerCellInSeconds': 43200,
    'retry': 0,
    'retryIntervalInSeconds': 10,
    'dependencies': []
}
activities.append(activity)
```

runMultiple

```
DAG = {
    'activities': activities,
    'timeoutInSeconds': 43200,
    "timeoutPerCellInSeconds": 43200,
    'concurrency': 50
}
```

- Look out for timeoutPerCellInSeconds which tells how much each cell in the children's notebooks can take before timeout
- Concurrency tells you how many notebooks should run in parallel
  - It can be any number but 50 seems to be the sweet spot
  - Your cluster will still decide how many notebooks can actually run in parallel

runMultiple

- The parallelism degree of the multiple notebook run is restricted to the total available compute resource of a Spark session.
- The default number of concurrent notebooks is 50 for Spark notebook, while it's default to 25 for Python Notebook.
- The default time-out for entire DAG is 12 hours, and the default time-out for each **cell in child notebook is 90 seconds**.
- exit() to exit gracefully
- The parent and child notebooks need to have the same default lakehouse (better not have any)
- Need to write your own error handling

runMultiple – error handling

```
failed_notebooks = [] # Create an array
failed_details = {}  # optional: keep the exception objects/text for later use

try:
    notebookutils.notebook.runMultiple(DAG)
```

runMultiple – error handling

```
except Exception as e:
      print(f"RunMultipleFailedException: {str(e)}")

      results = getattr(e, "result", None)

      if isinstance(results, dict):
          for notebook_name, r in results.items():
              exc = None
              if isinstance(r, dict):
                  exc = r.get("exception")
              if exc is not None:
                  failed_notebooks.append(notebook_name)
                  failed_details[notebook_name] = exc
```

runMultiple – error handling

```
if failed_notebooks:
            print("\nThe following notebooks failed:")
            for nb in failed_notebooks:
                print(f"- {nb}")
            for nb, exc in failed_details.items():
                message_id = send_email_via_acs(
                    recipient=environments.failureEmailAddress,
                    subject="Failed notebooks",
                    plain_text=f"The notebook {nb} failed with the error
{str(exc)}",
                    )
        else:
            print("\nAll notebooks executed successfully.")
```

ITEM API

- Call an API endpoint to run an item (notebook)
- Run On Demand Job Item endpoint
- The orchestrator can be a Python notebook
- Call another API endpoint to monitor the outcome of the run
- Need to manage the parallelization yourself
- Need to manage the failure and retries yourself
- Need to handle authentication and token management yourself

Run an Item via the API

```
import requests
Notebook = ”XXXXXXXX-YYYYY-XXXX-YYYY-XXXXXXXXXXXX”
WorkspaceId = ”XXXXXXXX-YYYYY-XXXX-YYYY-XXXXXXXXXXXX"

token = notebookutils.credentials.getToken('pbi')
header = {'Content-Type':'application/json', 'Authorization':f'Bearer {token}'}


uri =
f"https://api.fabric.microsoft.com/v1/workspaces/{workspaceId}/items/{Notebook}/jobs/i
nstances?jobType=RunNotebook"


post_ItemRun = requests.post(url=uri, headers=header)
```

Run an Item via the API

- Need to catch the return header
  - Where you call the status of your request
  - Where you get a time when you should call for the status
    - Sometimes it will return nothing, but it does not mean that you should call it immediately as that might cause a throttling situation
    - I typically wait for at least 20 seconds before calling to ask for status – but if you have 200 notebooks to call that might not be the right way to do it

```
location = post_ItemRun.headers.get("Location")
retryAfter = post_ItemRun.headers.get("Retry-After")
```

Run an Item via the API

- You need to keep an eye out for timeouts
- The token will expire after certain amount of time (not certain how much time)
- If it happens you will get an error back when calling for the status
- Renew the token and you can continue to monitor the original request (with the original URL)

```
if http_status>=400:
        token = notebookutils.credentials.getToken('pbi')
```

- Otherwise wait for completion

Notebook runmultiple() demo

Notebook ITEM API demo

Notebook orchestration strengths

- Full Python control & flexibility
- Dynamic workload generation
- Easy reuse of code patterns
- Git friendly & testable

Notebook orchestration weaknesses

- More complex monitoring UX
- Harder for ops teams
- Scheduling still needed
- Risk of mixing control + compute
- Can become “DIY orchestration platform”

When notebook orchestration shines

- Complex metadata-driven frameworks
- Highly dynamic pipelines
- Advanced data engineering teams
- Different “types” of notebook environments/compute needs
- Heavy Git/DevOps workflows
- Framework/platform teams

ORCHESTRATION WITH FABRIC MATERILIZED LAKE VIEWS

Using the data plane

Materialized lake views

- Spark notebooks with benefits
- Precomputed, stored results of SQL queries that can be refreshed on demand or on a schedule
- dbt/databricks lakeflow competition (my words)
- Takes care of the order of execution
- Has built in testing syntax
- Needs a schema enabled lakehouse
- Can be created from tables, files or other materialized views (data must be in Fabric)

https://learn.microsoft.com/en-us/fabric/data-engineering/materialized-lake-views/overview-materialized-lake-view

Why use materialized lake views

- SQL logic all the way (for the moment)
  - PySpark is in preview
- Built-in dependency tracker
- Nice overview / linage
- Easy to schedule from the outside with simple APIs

Why use materialized lake views (Microsoft words)

- **Performance**: Instead of running expensive queries repeatedly, results are precomputed and stored
- **Consistency**: Everyone accesses the same transformed data, reducing discrepancies
- **Efficiency**: Only refresh when source data actually changes, saving compute resources
- **Simplicity**: Define transformations once using familiar SQL syntax

Materialized lake view demo

Materialized lake view strengths

- SQL coding
- Dynamic workload generation
- Nice UI for monitoring
- Automatic orchestration linage
- Automatic optimization of incremental load
- Built-in testing
- Git friendly & testable

Materialized lake view weaknesses

- Can become very cluttered with lots of tables
- Need to use lakehouses with schema (not Spark native)
- Can be harder for ops teams
- Scheduling still needed

When Materialized lake view orchestration shines

- Complex linage to maintain
- Complex incremental data loading
- Teams with good SQL knowledge
  - But perhaps less PySpark knowledge

ORCHESTRATION WITH THIRD-PARTY TOOLS

Using the external plane

External orchestration overview

- Airflow
- Prefect
- Dagster
- Existing ADF environments
- Exmon

Why teams consider this

- Existing enterprise orchestrator
- Cross-platform workloads
- DevOps-driven culture
- Single orchestration platform strategy
- Extra features needed/wanted

Exmon Orchestration and Data quality

- Tool for orchestration and data quality originally from Iceland
  - Now owned by TimeXtender form Denmark
- Only external tool I have experience with hence why I’m showing it to you
- Not endorsing them in any particular way

Third party tool demo

Strengths of external tools

- Cross-system orchestration
- Strong CI/CD patterns
- Centralized enterprise control plane
- Often other enterprise features

Weaknesses of external tools

- Extra complexity & integration work
- Less native Fabric visibility
- Higher operational overhead
- Requires platform maturity

When third-party is best

- Large enterprise environments
- Multi-cloud / multi-platform orchestration
- Existing investment
- Strong platform engineering teams

Monitoring Hub

When things go wrong

Fabric Monitoring Hub

- One place for monitoring regardless of which orchestrator used
- Everything from superficial information to in-depth logging
- Some parts are very approachable but others very complex
- New version just released – you can still use the older if you prefer

WHAT TO USE

Decision framework

The orchestration spectrum

- Beginner → Pipelines
- Intermediate → Pipelines + notebooks
- Advanced → External control plane

Comparison table

Capability	Pipelines	Notebook Orchestration	Third-party Tools
Ease of use	⭐⭐⭐⭐	⭐⭐	⭐⭐⭐
Flexibility	⭐⭐	⭐⭐⭐⭐	⭐⭐⭐⭐
Monitoring UX	⭐⭐⭐⭐	⭐⭐	⭐⭐⭐
DevOps integration	⭐⭐⭐	⭐⭐⭐⭐	⭐⭐⭐⭐
Governance	⭐⭐⭐⭐	⭐⭐	⭐⭐⭐⭐
Enterprise scale	⭐⭐⭐	⭐⭐⭐⭐	⭐⭐⭐⭐
Setup complexity	⭐⭐⭐⭐	⭐⭐⭐	⭐⭐

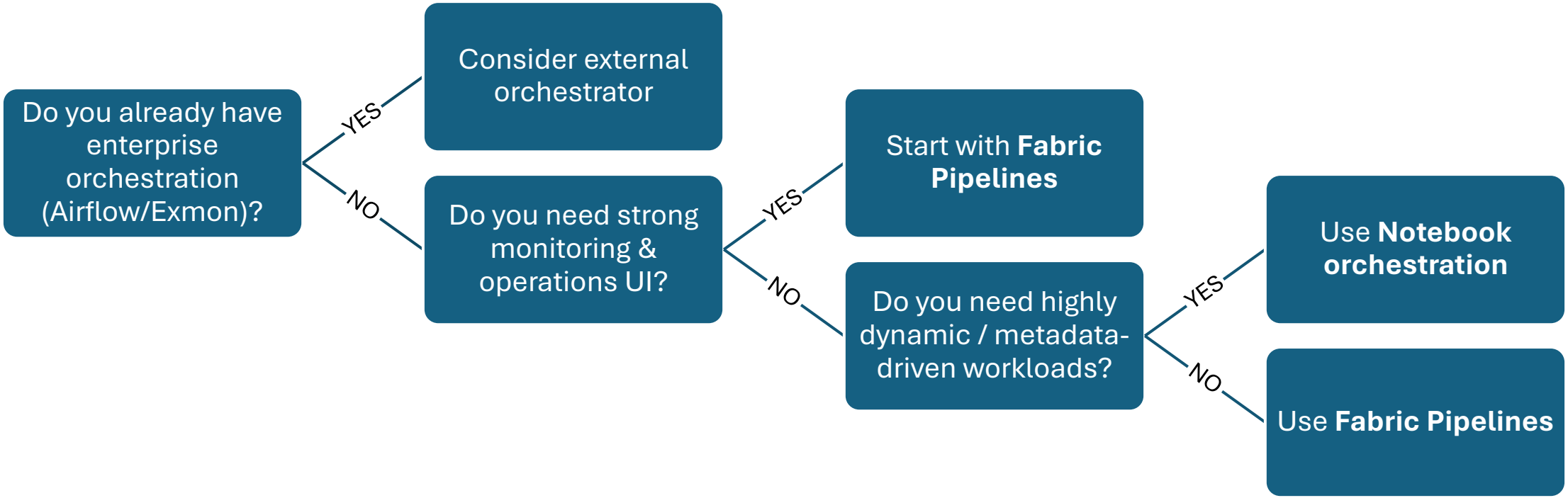
Real-world pattern we see

- Many successful Fabric platforms use **hybrid orchestration**
- Typical pattern:
  - Pipelines for scheduling & monitoring
  - Notebooks for dynamic workloads
  - Or
  - Pipelines for ingestion
  - Notebooks for the rest
  - Or
  - Pipelines/Notebooks for item orchestration
  - External tool for overall orchestration

Key takeaways

- There is no universal “best” orchestrator
- Choose based on team maturity
- Separate control plane vs data plane consciously
- Hybrid orchestration is normal
- Start simple → evolve intentionally

How should you orchestrate



Q&A

Rate This session

From Chaos to Control: Orchestrating Lakehouse Workloads in Microsoft Fabric

