ADVANCING
ANALYTICS

Dynamic Reading & Writing

Automating Dataframes & Notebooks

Introduction to Metadata

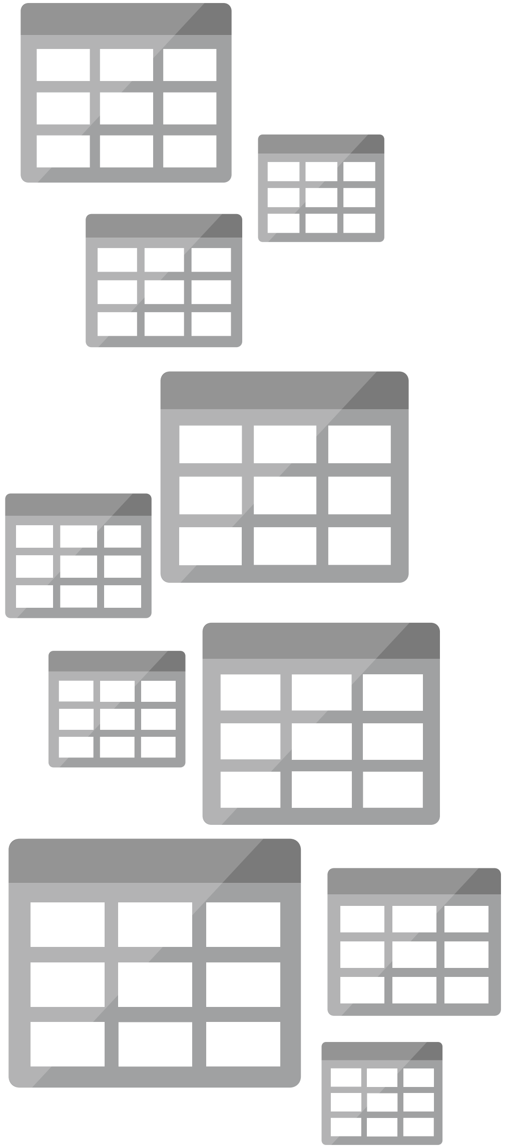
What is Metadata?

- Metadata is **data about data**.

- Delta Tables **store both** data and metadata
  - Rows = data etc
  - Metadata = schema, source, timestamps etc

- Metadata **gives insight** into your data

- Metadata can **drive ETL** processes



History of ETL Automation

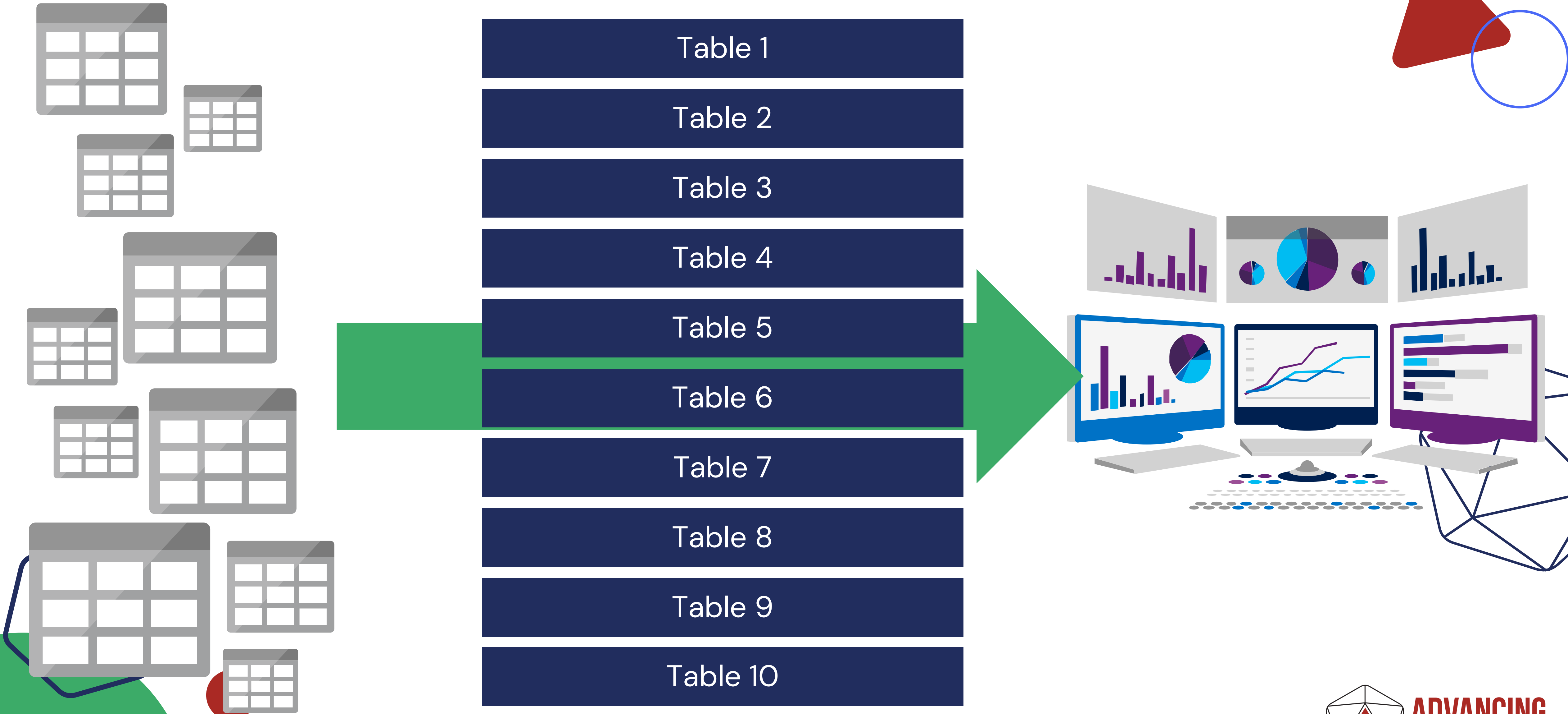


Table 1

Table 2

Table 3

Table 4

Table 5

Table 6

Table 7

Table 8

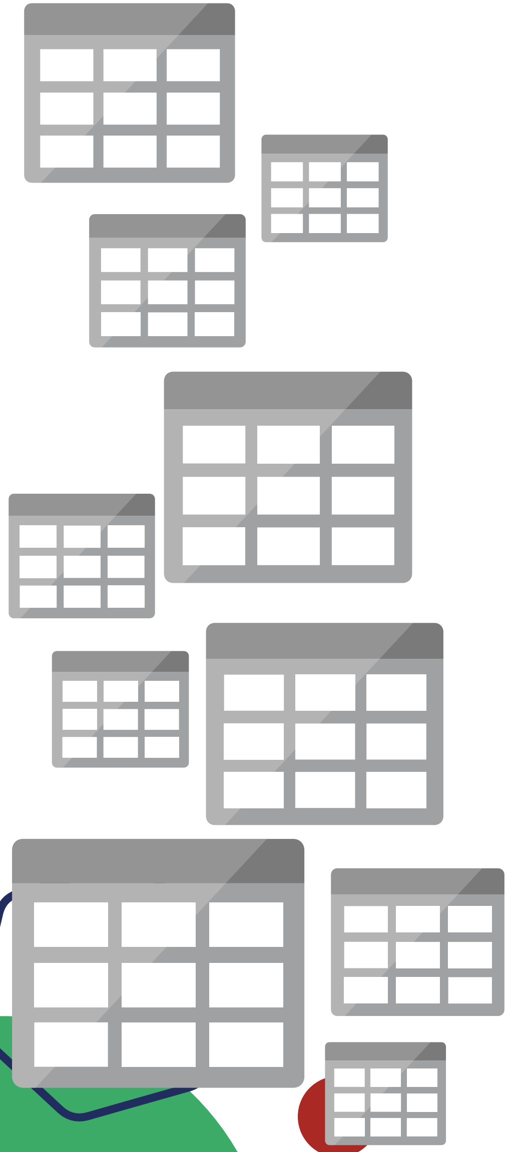
Table 9

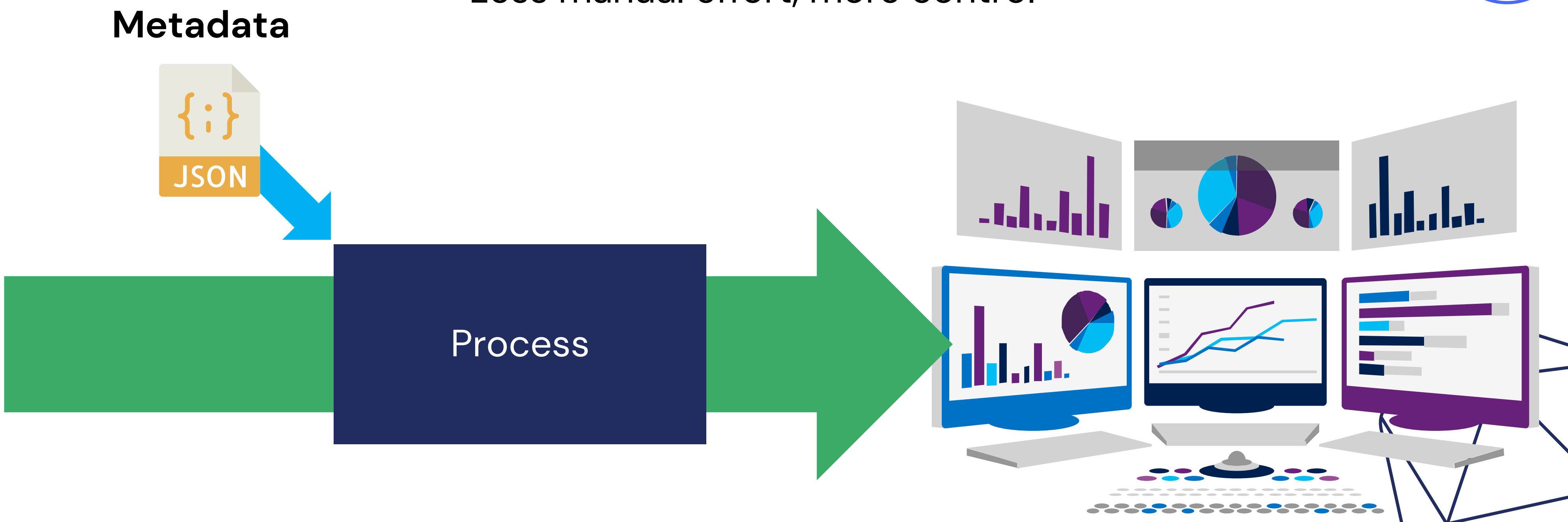
Table 10

ADVANCING ANALYTICS

Code Automation

- Reusable code for any dataset
- Faster and more consistent ETL
- Less manual effort, more control



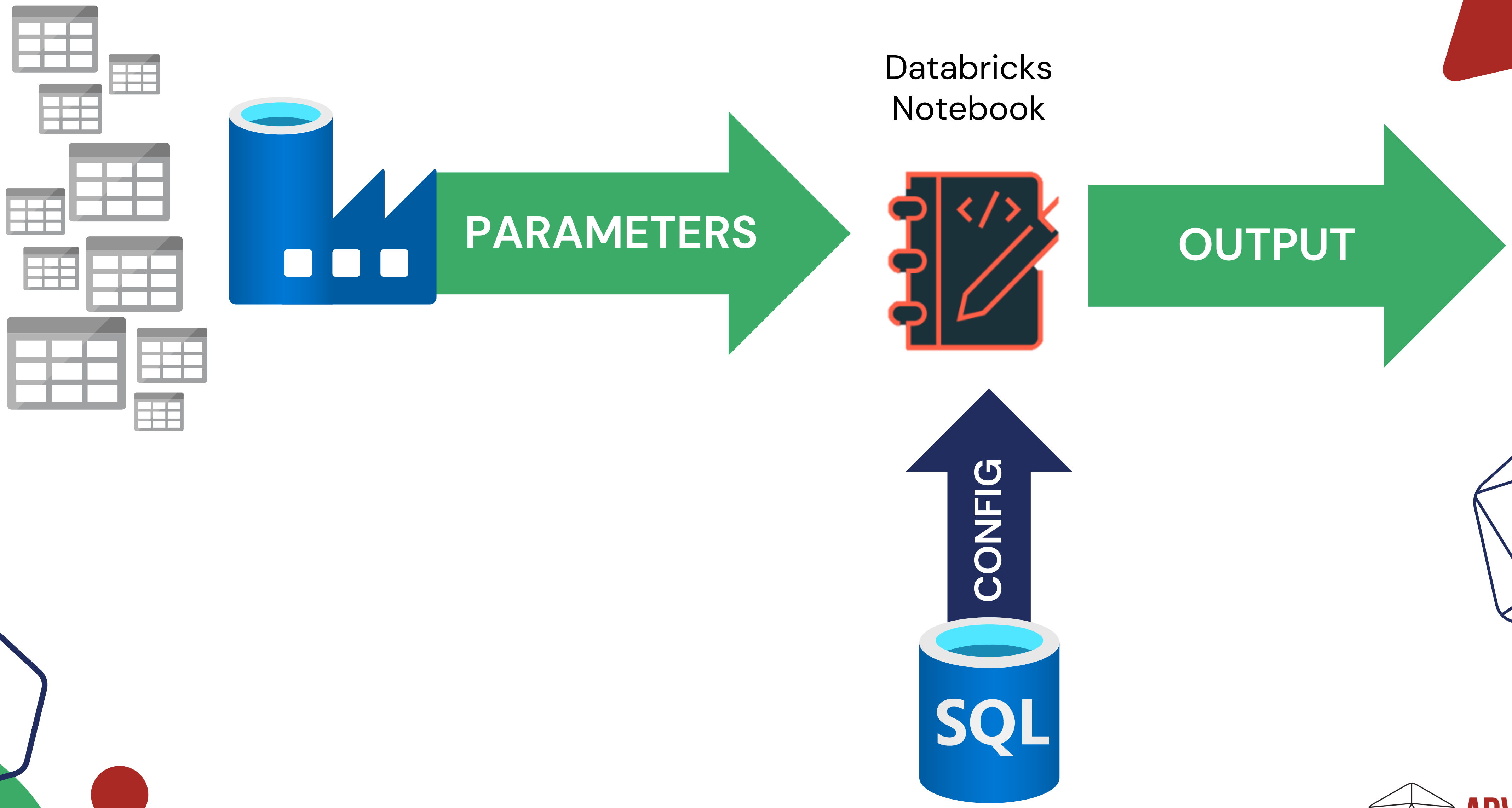


Considerations:

- Code Complexity
- Metadata Management
- Shift in mindset

ADVANCING ANALYTICS

In Our Scenario



Databricks Notebook

PARAMETERS

OUTPUT

CONFIG

SQL

ADVANCING ANALYTICS

Acquiring Metadata

- Query a SQL Database
- Use Dataframe Results as Variables

Acquiring Metadata

We can do a lot using variable, but we don’t want to have to fetch configuration for each variable individually.

We commonly need to bring back dictionary objects or even query other sources such as a SQL Database.

However, if we query dataframes, the individual values are not accessible to the surface python layer… So how do we do this?

Using Widgets

We can implement widgets, which are parameters set at the notebook level. These allow us to pass parameters into the notebook from external sources.

The following Widget Types are available:

- Text
- Dropdown
- Combobox
- Multiselect

```
# Create a new Text Widget
dbutils.widgets.text([objectname],[default],[label text])

# Allocate the current widget value to a variable
myString = dbutils.widgets.get([objectname])
```

DEMO: Notebook metadata

- Creating reuseable code

Database Lookups

```
# Create a dataframe from a database source
df = (spark.read
     .format("jdbc")
     .option("driver", "com.microsoft.sqlserver.jdbc.SQLServerDriver")
     .option("url",
             "jdbc:sqlserver://[server].database.windows.net;database=[database]")
     .option("user", [username])
     .option("password", [password])
     .option("query", [query])
     .load()
    )
```

We can create a dataframe over a SQL database directly in the same way we would read data from the lake. This returns a dataframe with the rows returned from the database.

As with all other sources, this will query the database each time an action is triggered, unless we have cached the dataframe first

Encapsulating Complexity

- With all the techniques we've looked at – we're starting to build a fairly complex script just to load a dataframe!

- Once we reach this point, we can split out some common functions to simplify our code.

- The easiest first step is by running other notebooks!
  - **dbutils.notebook.run** – this utility runs a Databricks notebook in a separate thread. We can drill down on the other notebook but objects created are not visible to the parent notebook
  - **%Run** – this magic command runs the notebook in the same session context as our current notebook, so any variables, functions etc created are available for us to use!

- Both can take a map input of parameters, in case the child notebooks have widgets!

ADVANCING ANALYTICS

Running Child Notebooks

```
# Run an inline notebook in the same folder context in the workspace
%run ‘./MyChildNotebook’

# Run an inline notebook in a different folder under the same parent folder
%run ‘../OtherExamples/MyCousinNotebook’


# Run a notebook as a separate job
dbutils.notebook.run(“./MyChildNotebook”,{“widget1”:”True”})
```

We can return data from a separate child notebook using the following command:

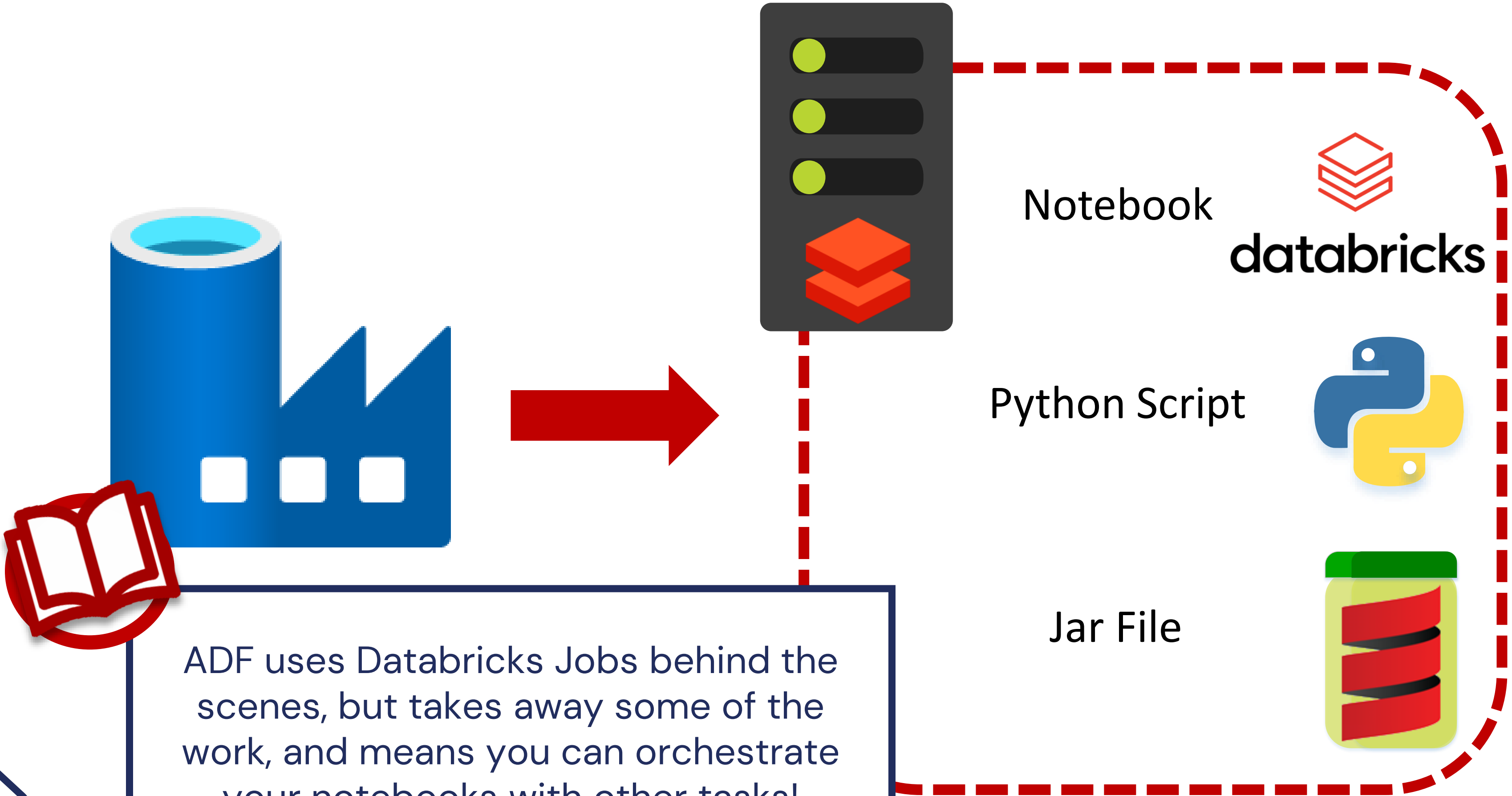
```
dbutils.notebook.exit(“myReturnString”)
```

We use this same method to pass results back to a calling job, whether through the REST API or via tools such as Data Factory

ADVANCING ANALYTICS

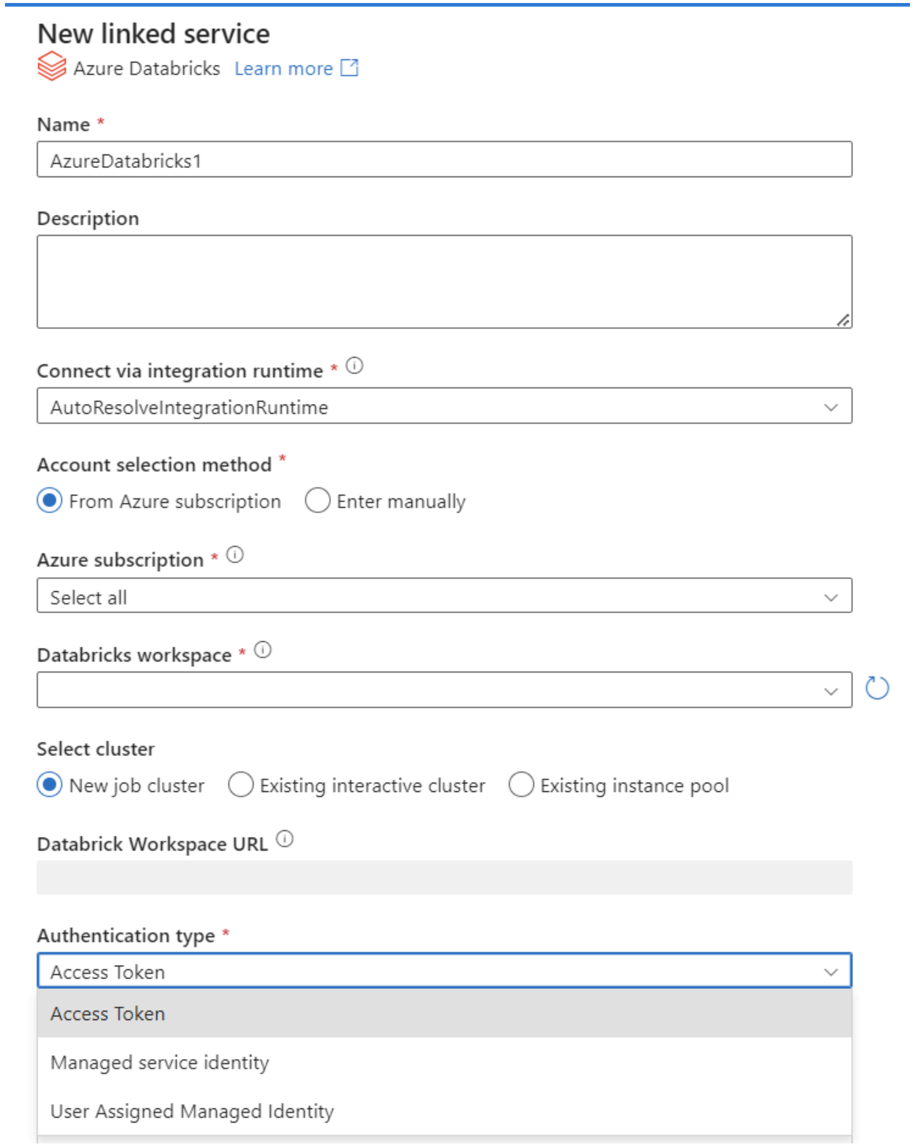
Passing Parameters

Azure Data Factory



ADF uses Databricks Jobs behind the scenes, but takes away some of the work, and means you can orchestrate your notebooks with other tasks!

Azure Data Factory Linked Services



New linked service

Azure Databricks Learn more

Name *

AzureDatabricks1

Description

Connect via integration runtime *

AutoResolveIntegrationRuntime

Account selection method *

● From Azure subscription ○ Enter manually

Azure subscription *

Select all

Databricks workspace *

Select cluster

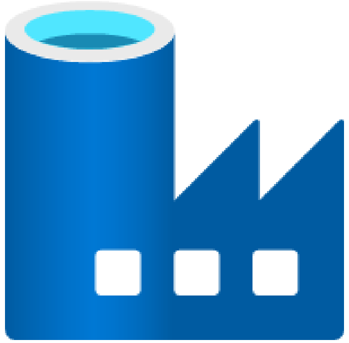
● New job cluster ○ Existing interactive cluster ○ Existing instance pool

Databrick Workspace URL

Authentication type *

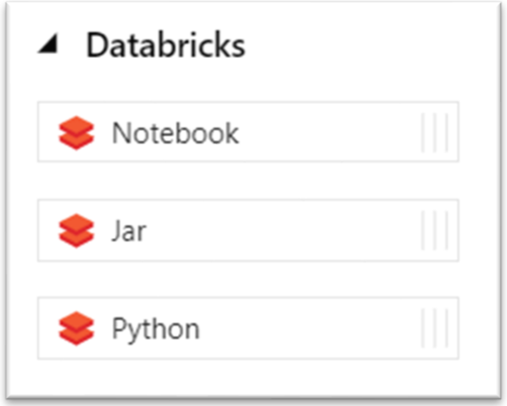
Access Token

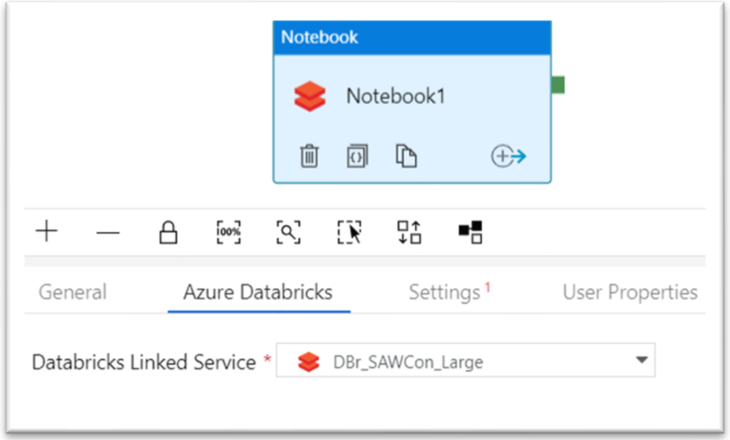
- Access Token
- Managed service identity
- User Assigned Managed Identity



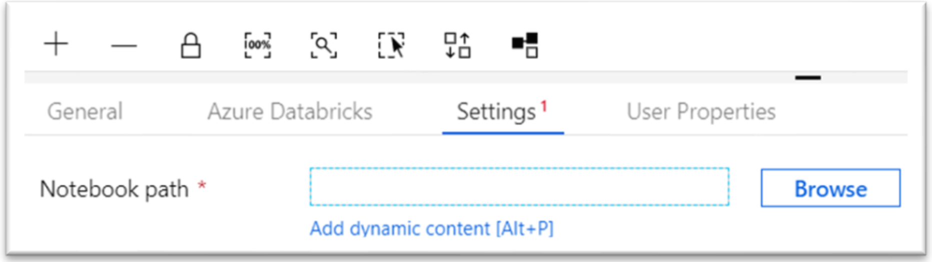
ADVANCING ANALYTICS

Azure Data Factory Pipelines

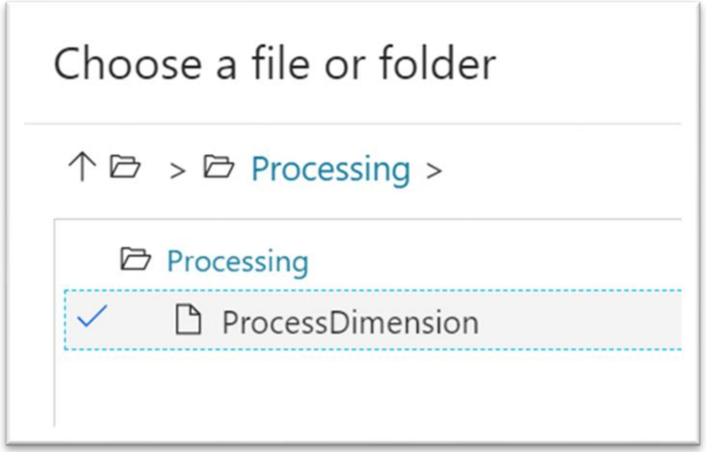




Databricks Linked Service * DBr_SAWCon_Large

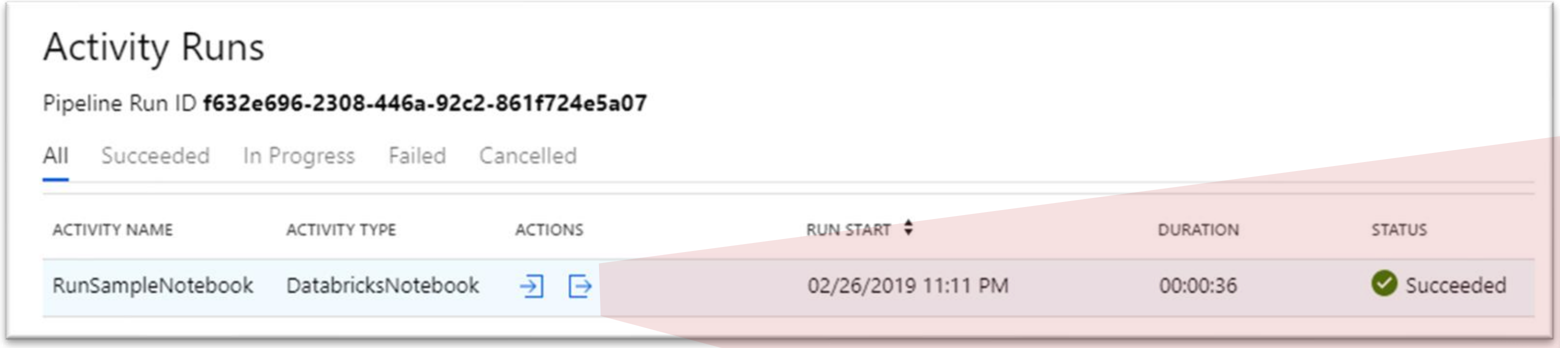


Notebook path *



ADVANCING ANALYTICS

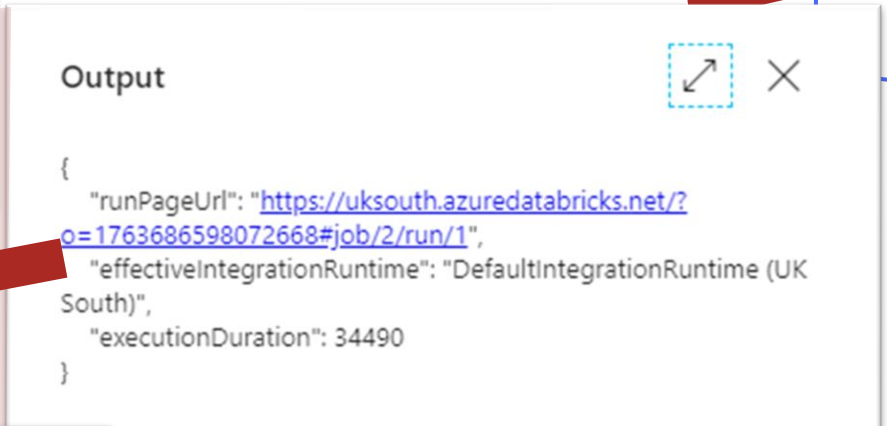
Execution Results



Activity Runs

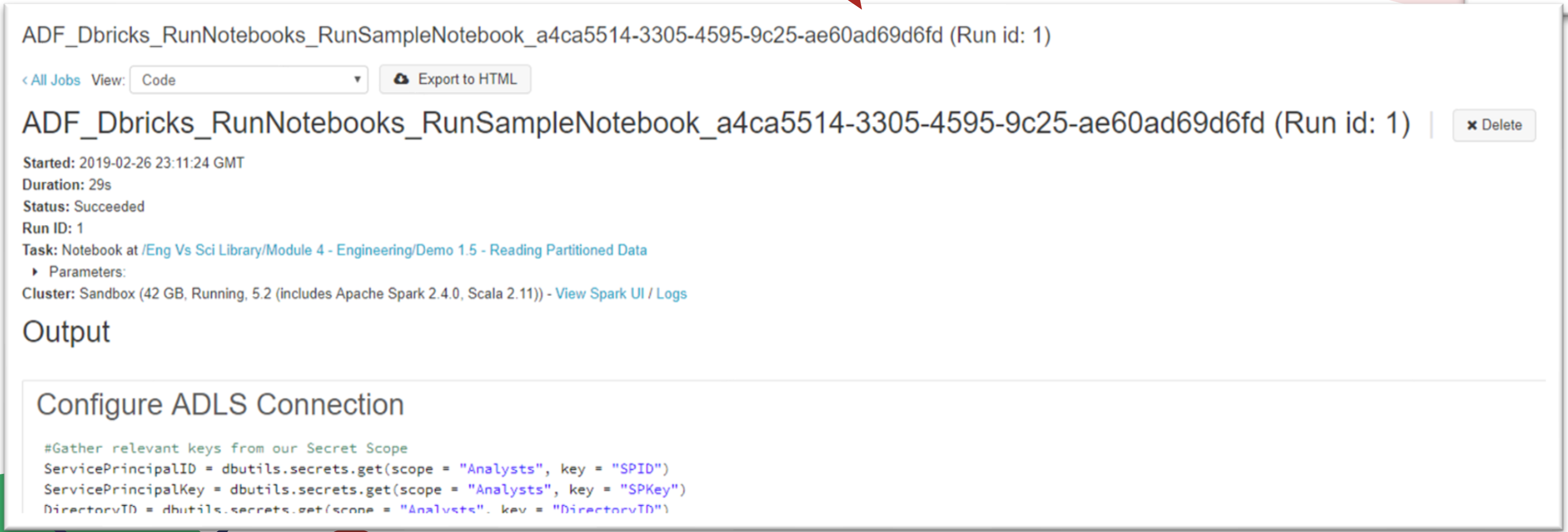
Pipeline Run ID **f632e696-2308-446a-92c2-861f724e5a07**

ACTIVITY NAME	ACTIVITY TYPE	ACTIONS	RUN START	DURATION	STATUS
RunSampleNotebook	DatabricksNotebook		02/26/2019 11:11 PM	00:00:36	✓ Succeeded



Output

```
{
    "runPageUrl": "https://uksouth.azuredatabricks.net/?o=1763686598072668#job/2/run/1",
    "effectiveIntegrationRuntime": "DefaultIntegrationRuntime (UK South)",
    "executionDuration": 34490
}
```



ADF_Dbricks_RunNotebooks_RunSampleNotebook_a4ca5514-3305-4595-9c25-ae60ad69d6fd (Run id: 1)

ADF_Dbricks_RunNotebooks_RunSampleNotebook_a4ca5514-3305-4595-9c25-ae60ad69d6fd (Run id: 1)

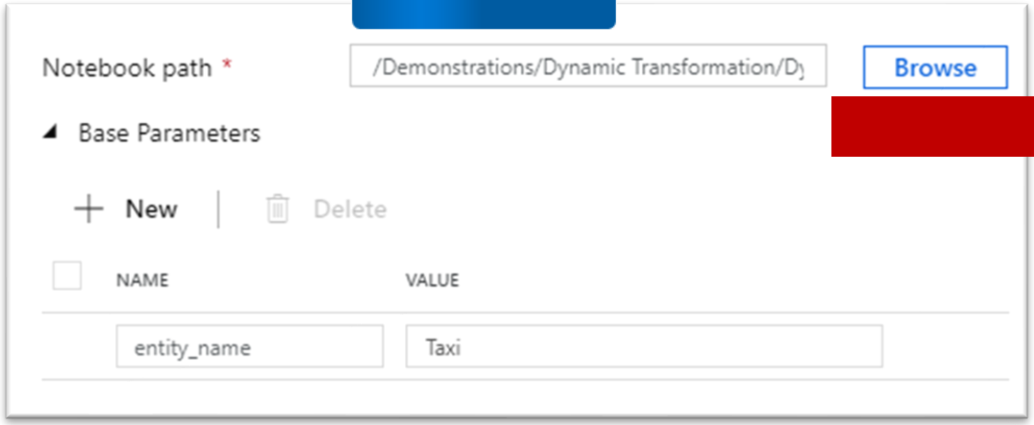
Started: 2019-02-26 23:11:24 GMT
Duration: 29s
Status: Succeeded
Run ID: 1
Task: Notebook at /Eng Vs Sci Library/Module 4 - Engineering/Demo 1.5 - Reading Partitioned Data
▸ Parameters:
Cluster: Sandbox (42 GB, Running, 5.2 (includes Apache Spark 2.4.0, Scala 2.11)) - View Spark UI / Logs

Output

Configure ADLS Connection

```
#Gather relevant keys from our Secret Scope
ServicePrincipalID = dbutils.secrets.get(scope = "Analysts", key = "SPID")
ServicePrincipalKey = dbutils.secrets.get(scope = "Analysts", key = "SPKey")
DirectoryID = dbutils.secrets.get(scope = "Analysts", key = "DirectoryID")
```

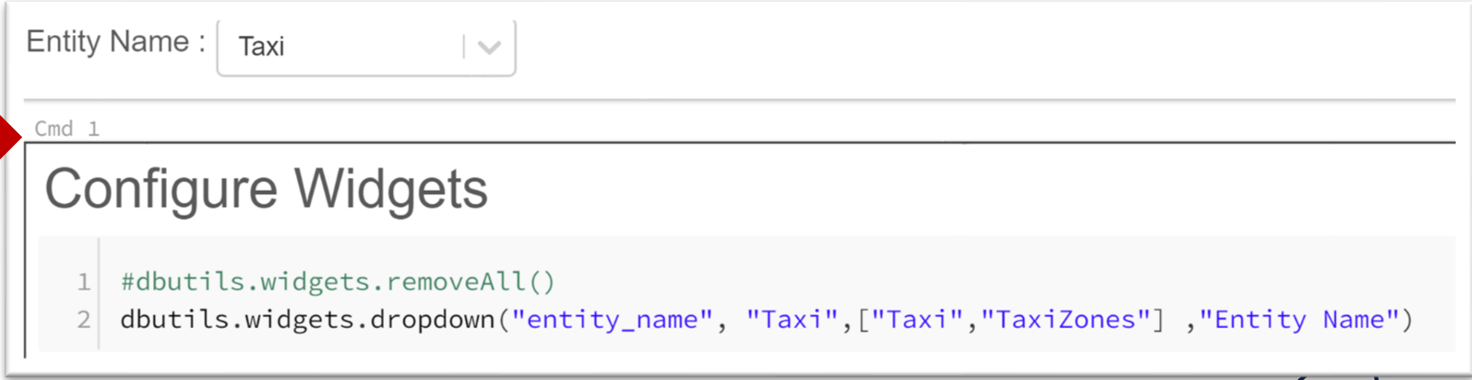
Azure Data Factory



Notebook path * /Demonstrations/Dynamic Transformation/Dy

◢ Base Parameters

□	NAME	VALUE
	entity_name	Taxi

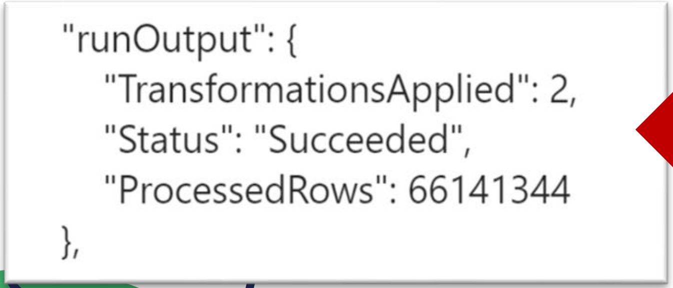


Entity Name : Taxi

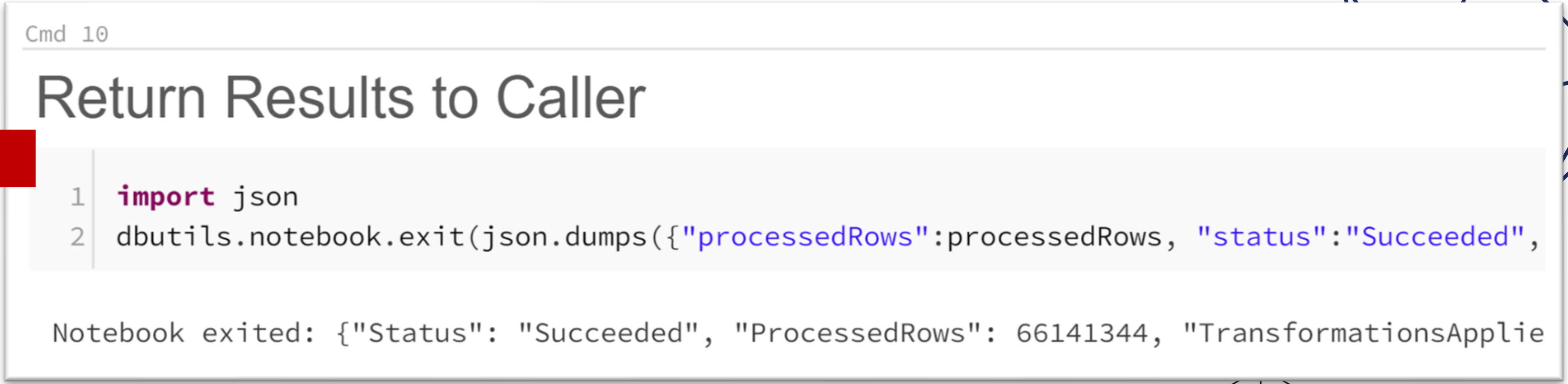
Cmd 1

Configure Widgets

```
#dbutils.widgets.removeAll()
dbutils.widgets.dropdown("entity_name", "Taxi",["Taxi","TaxiZones"] ,"Entity Name")
```



```
"runOutput": {
    "TransformationsApplied": 2,
    "Status": "Succeeded",
    "ProcessedRows": 66141344
},
```



Cmd 10

Return Results to Caller

```
import json
dbutils.notebook.exit(json.dumps({"processedRows":processedRows, "status":"Succeeded",
```

```
Notebook exited: {"Status": "Succeeded", "ProcessedRows": 66141344, "TransformationsApplie
```

Parameterising Dataframes

Structure of a Dataframe

```
df = (
       spark
       .read
       .option(“header”,True)
       .option(“sep”,”|”)
       .format(“csv”)
        .load(“abfss://container@storage.dfs.core.windows.net/data/customer”)
)
```

Parameterise Strings

```
df = (
        spark
        .read
        .option(“header”,headers)
        .option(“sep”,separator)
        .format(format)
        .load(path)
)
```

```
headers   = True
separator = “|”
format    = “csv”
path      = “abfss://file@path...”
```

What if there are different options?

ARGS & KWARGS

In Python, you can pass multiple arguments at once in two ways:

- ***args** – A single Asterix can be used to pass in a list of arguments, all for the same parameter. We might use this for passing in a list of columns to partition by, for example

- ****kwargs** – A dict of parameter names & associated values, this is very powerful for providing several settings at once

Options Argument & KWARGS

```
df = (
        spark
        .read
        .options(**config)
        .format(format)
        .load(path)
)
```

```
config = {“headers”:”True”, “sep”:“|”}
format = “csv”
path   = “abfss://file@path...”
```

Here we are using a different parameter called “options” which expects a dictionary/array of various config settings.

We are using the ** operator to unpack our config variable into those settings

This means we can dynamically supply different sets of options!

ADVANCING ANALYTICS

F Strings

Python has a few methods for string formatting, the most modern approach being a technique called **f-strings.** These allow for variables to be injected into a string at runtime.

```
# Create a variable
myString = “World”

# Inject the variable into a new string and print it
print(f“This is the usual Hello {myString} example”)

Out[]: This is the usual Hello World example
```

This technique is incredibly useful for deriving lake paths, writing dynamic SQL commands, building out expression transformations and more. Use it!

You may find some older code using the .format() syntax. This does the same thing but is harder to read:

```
“hello {}”.format(myString)
```

ADVANCING ANALYTICS

DEMO: Parameterised Dataframes

- Creating reuseable code

LAB01: Writing a parameter driven notebook

Python Iterators

We often find ourselves with lists of transformations we want to apply – for example, we might want to make all string columns uppercase to standardise a dataset. In SQL, applying each of these transformations in serial would be incredibly inefficient.

Because of the lazy evaluating nature of spark – we can apply dataframe transformations quickly and efficiently in ways we wouldn’t in SQL!

```
# Create a variable with an array of column names
Columns = [“FirstName”,”LastName”,”Title”]

# Loop through the list
for colName in Columns:
      print(colName)
```

ADVANCING ANALYTICS

Iterators & Transformations

Certain transformations have an “expression” mode, where we can supply a Spark SQL string instead of python commands. This is incredibly useful when combined with iterators & variables!

```
# Create a variable with a list of column names
Columns = [“FirstName”,”LastName”,”Title”]

# Loop through the list, applying a dataframe transformation for each
for colName in Columns:
      SQL = f“UPPER({colName})”
      df = df.withColumn(colName, expr(SQL))
```

ADVANCING ANALYTICS

Writing Out

```
(
    df
    .write
    .options(**writeConfig)
    .partitionBy(*partitionCols)
    .mode(writeMode)
    .format(fileFormat)
    .save(writePath)
)
```

ADVANCING ANALYTICS

LABO2: Metadata driven mini pipeline for two datasets

Recap

Recap

- Parameterise all the things
- We can’t inspect dataframe data without a collect()/take()/first()
- Practice with F-Strings, For Loops & IF Statements
- Store Metadata Outside of Databricks
- Build out function libraries over time

ADVANCING ANALYTICS

ADVANCING
ANALYTICS