Fabric Lakehouse

for SQL Data Engineers

Kimura
data intelligence

Bas Land

- BI consultant since 2013
- Co-founder of two data companies:
  - Kimura Data Intelligence (consultancy)
  - DataChimp (SaaS analytics for accounting firms)

- Married to Anouk, we have a daxhund (☺) called Chester
- Sports: Brazilian jiu-jitsu, weight lifting, running

How do you manage all of that?

Data warehouse experience

- Since 2013 I have personally built >20 data warehouses, guided >50 in total

- Most in SQL technology (SQL Server and Azure SQL)
- And now we’ve done 4 MS Fabric implementations with 2 more on the way

- Moving from SQL to Fabric Lakehouses is easier than it seems!

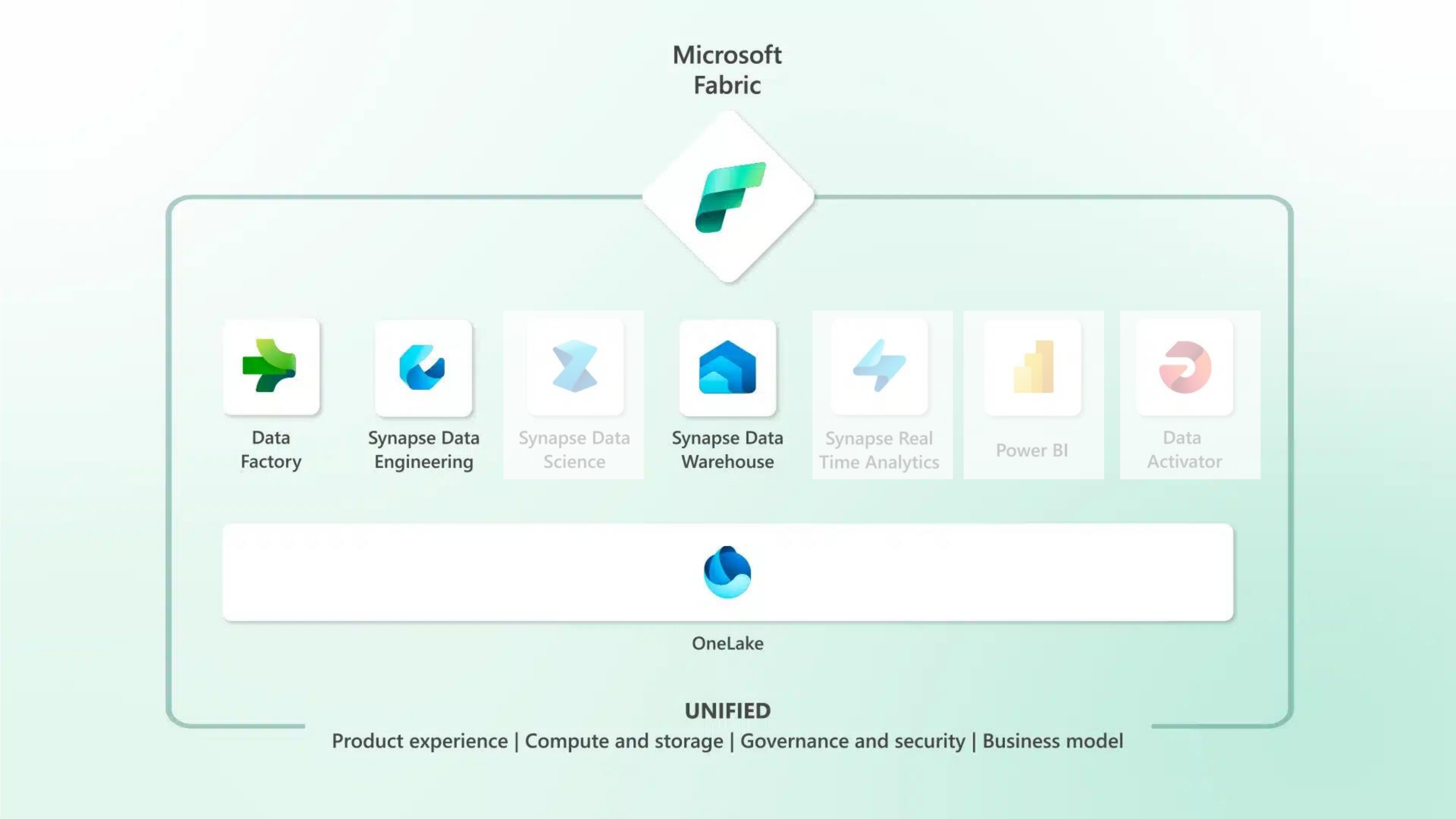
Today

- MS Fabric for SQL Data Engineers!

1. Data warehouse design patterns
2. Battle of the engines: SQL vs Spark
3. Code reusability

- Your first data warehouse with Fabric

What the F?



Microsoft Fabric
Data Factory
Synapse Data Engineering
Synapse Data Science
Synapse Data Warehouse
Synapse Real Time Analytics
Power BI
Data Activator
OneLake
UNIFIED
Product experience | Compute and storage | Governance and security | Business model

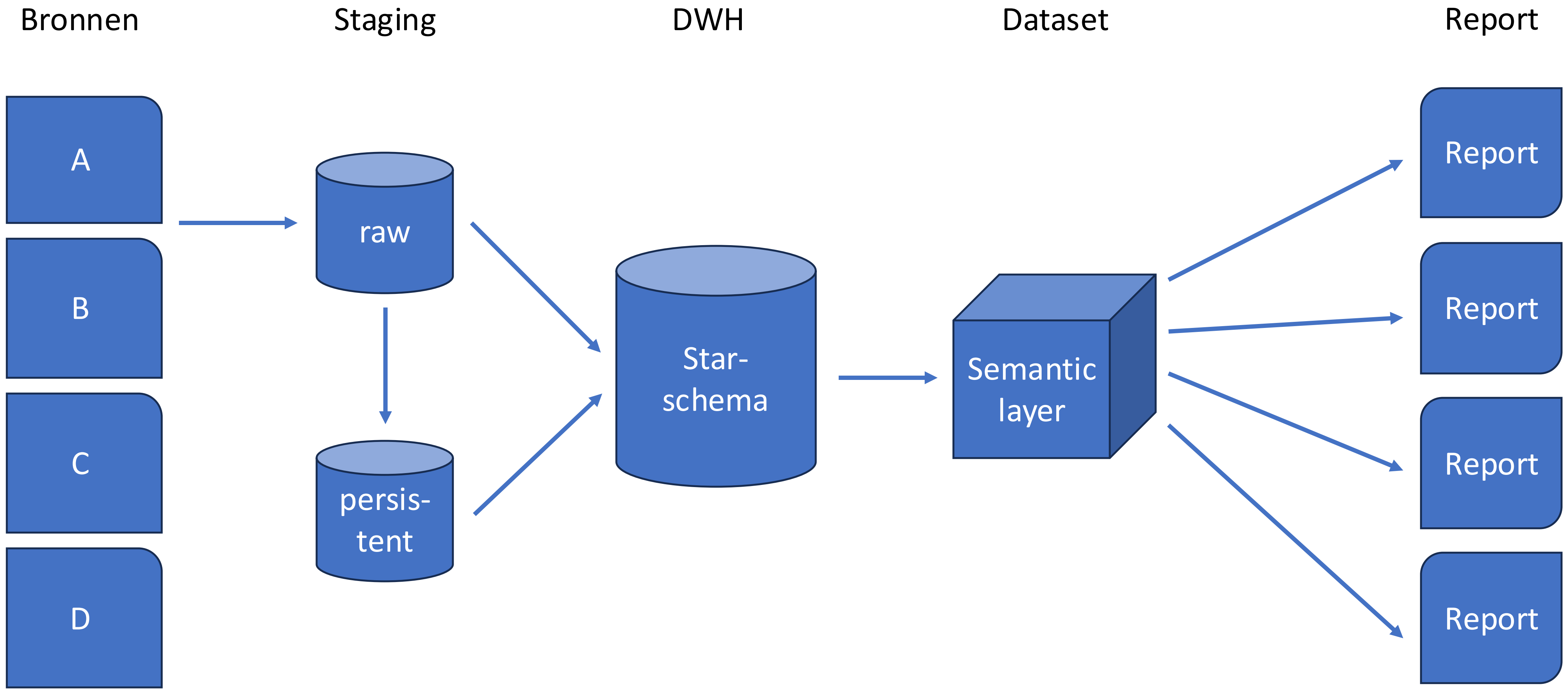
Data warehouse – general design principles

- Historical data archive from operational applications
- Combined data from multiple applications
- Prepared / precalculated transformations and business logic
- Single source of truth

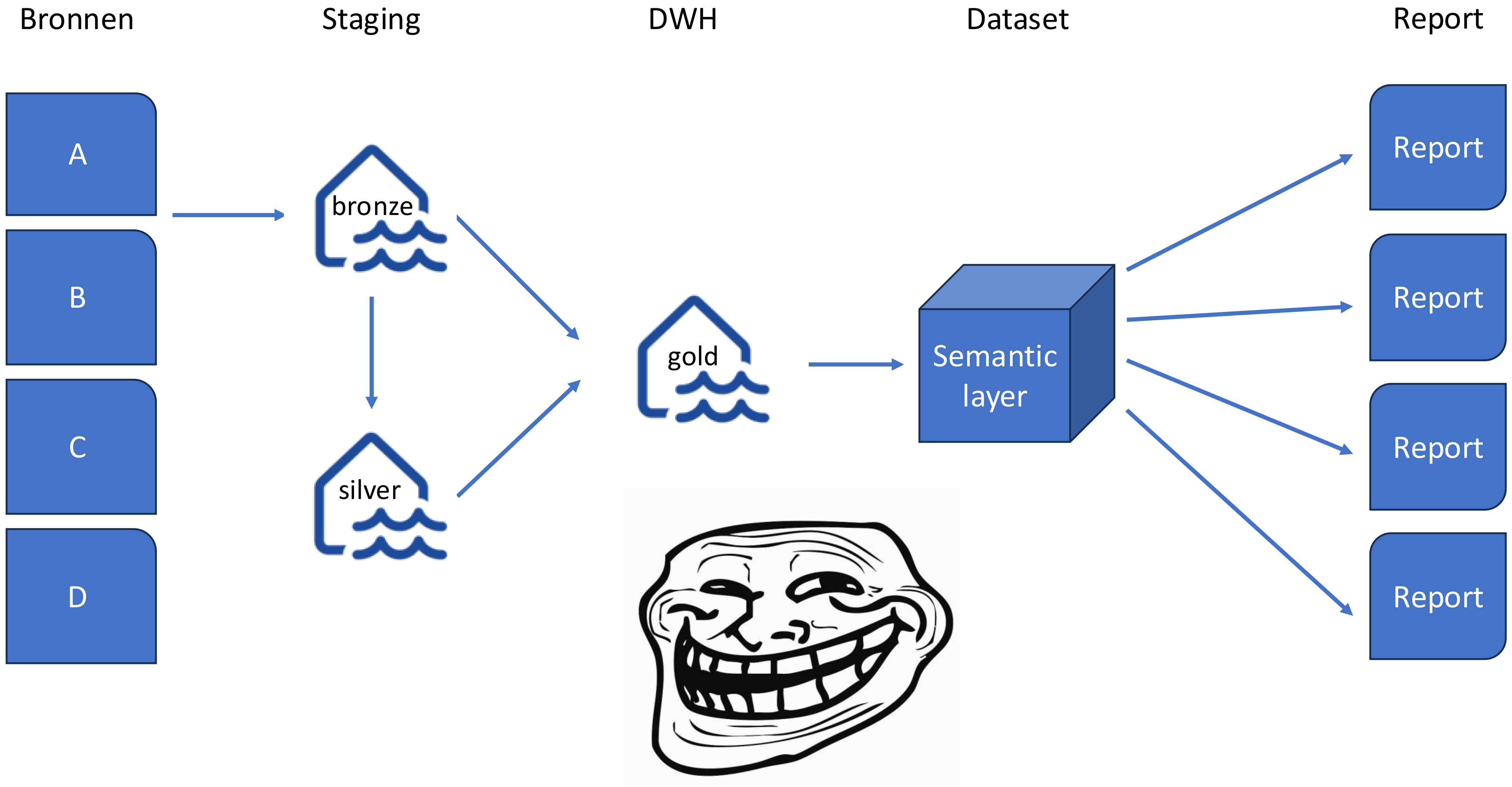
Design patters

Topic 1/3

General architecture in the SQL world

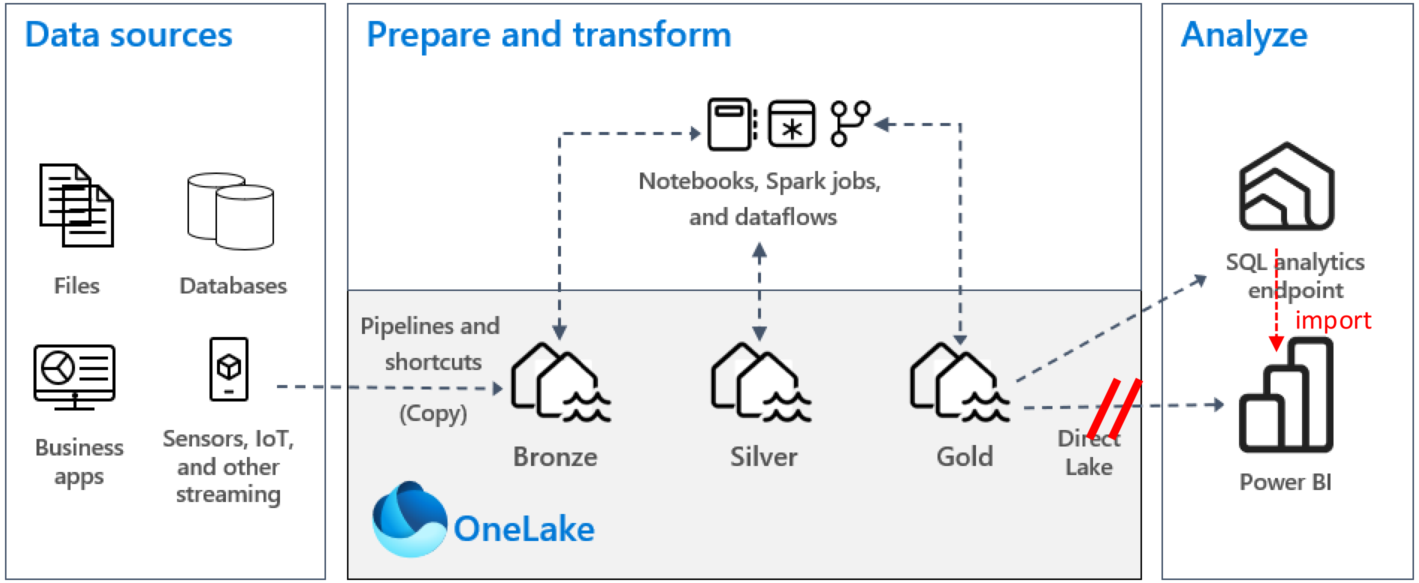


General architecture in the Lakehouse world



Medaillon architecture

- Not a Fabric-specific concept!
- Made popular by Databricks
- Describes three layers of data enrichment (bronze, silver, gold)



What we do:

- Bronze = raw data
- Silver = history, persistent stage, time travel, data types
- Gold = star schema

So… lakehouse vs database?

- We all know databases, right?
  - Structured data, with tables, columns, data types

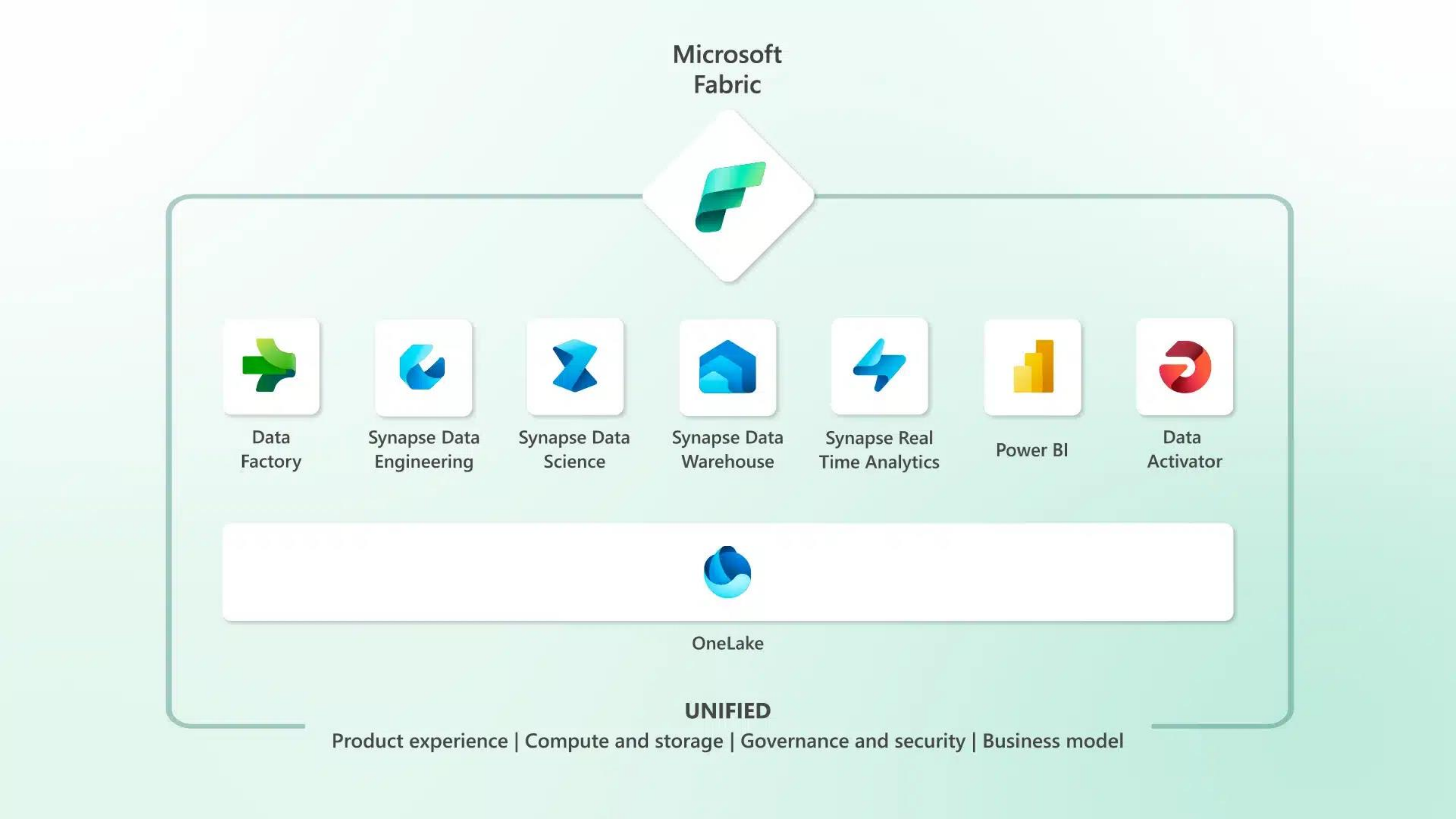
- And data lakes:
  - Structured and unstructured data, such as csv, parquet, jpg, pdf, etc

- Lakehouse:
  - Best of both worlds?
  - Data lake storage, delta parquet tables with structures

Engines: SQL vs Spark

Topic 2/3

Remember this one?



Today’s scope

Synapse Data Engineering

Spark engine, OneLake storage (unstructured files, structured delta tables)

Synapse Data Warehouse

T-SQL engine, OneLake storage (structured delta tables), also, <u>we don’t use this one</u>…
(perhaps only for configuration data)

OneLake

Storage account, shortcuts, security

Fabric Warehouse

- Not used in Kimura – we only use Lakehouse

- Lakehouse = file (parquet / delta) storage with Spark engine
- Warehouse = file storage with Spark and T-SQL APIs

- T-SQL is amazing, but not full feature parity with SQL Server
- Plus, it's an abstraction on top of Spark

- Conclusion: learn to use (Py)Spark and go all-in on Lakehouses ☺

- With a bit of flexibility you can use SQL in your Lakehouses

Spark engine for data engineering

- First differences between SQL and Spark – processing languages
- In a database, it’s SQL
- In Spark, we can choose whatever fits our requirements
  - Very popular: PySpark (Python for Spark) and SparkSQL

Spark engine: functional concepts

- Functional concepts don’t change much

- For our Silver layer (previous: persistent stage), we now do Schema On Read instead of Schema On Write
  - This saves development time and lots of configurations

Spark engine: technical concepts

- Technical concepts do change a lot:

- Storage: from relational database to delta tables
- Data structure: from fixed schema tables to delta tables with transaction logs
- Language: T-SQL to Spark (variants)
- Style: transaction-based processing to distributed parallel processing

Spark languages

- PySpark (python) and SparkSQL (SQL) are a higher abstraction of native Spark
- You can work with DataFrames, and also with all the Python libraries you can think of
- Fabric has more than 100 available Python libraries, but you can also develop and install your own

PySpark notebooks for processing

- Kimura Data Framework: interaction between:
  - Kimura Python Library (data processing)
  - PySpark notebooks (data content, business rules, etc)

```python
#Parameters
full_load = False
#if full_load = True then travel through all timestamps ignoring the timestamp param. This is not implemented yet.
timestamp = "20240229161122"#"20240206090005" #20240201055008
```

✓ - Apache Spark session ready in 3 min 13 sec 439 ms. Command executed in 2 sec 474 ms by admin-kimura on 5:23:05 PM, 2/29/24

```python
#Imports
import kimura as k
import ast
from datetime import datetime
import json

#Base stuff
base_path = f"Files/Landing/Full"
```

✓ - Apache Spark session ready in 2 min 35 sec 458 ms. Command executed in 2 sec 64 ms by admin-kimura on 6:13:37 PM, 2/29/24

```python
json_path = f"{base_path}/{timestamp}/metadata.json"
json_df = spark.read.json(json_path)
json_content = json_df.collect()[0].asDict()
table_list = json.loads(json_content['TableList'])
# display(table_list)

#Load data to Bronze layer delta tables
for table in table_list:
    target_table = table['TargetTable']
    source_system = table['SourceSystem']
    file_type = table['FileType']
    table_id = table['Id']
    incremental_field = table['IncrementalField']
    json_root_array = table['JSONRootArray']
    table_key_list = ast.literal_eval(table['TableKey']) #

    #Start logging
    start_datetime = datetime.now()

    #Execute
    print(f"Starting with table: {target_table}")
    count_rows = k.bronze_create_table(
        timestamp = timestamp,
        sourcesystem = source_system,
        tablename = target_table,
        tablekeys = table_key_list,
        base_path = base_path,
        file_type = file_type,
        table_id = table_id,
        incremental_field = incremental_field,
        json_root_array = json_root_array,
        spark_session = spark
    )

    #End logging
    end_datetime = datetime.now()
    delta = end_datetime - start_datetime
    print(f"Table: {target_table}, rowcount: {count_rows}, duration in sec: {delta.total_seconds()}")
```

⊗ - Command executed in 1 min 29 sec 923 ms by admin-kimura on 5:24:36 PM, 2/29/24

Fabric notebooks

- Jupyter notebooks

- Support for Python, Scala, SparkSQL and R
  - Can be mixed!
  - Examples on the following slides

PySpark notebooks - PySpark

```
# load data using python, into a spark dataframe
spark_df = spark.read.format('delta').load('Tables/silver_dimDates')
display(spark_df)
```

17 sec - Command executed in 16 sec 775 ms by DWHDEV | Kimura on 1:56:40 PM, 4/20/24

Spark jobs (2 of 2 succeeded) Resources Log

Table Chart Showing rows 1 - 1000

	Date	123 Year	123 MonthNumber	ABC MonthName	123 DayOfMonth	123 DayOfWeek	ABC Day
1	2018-01-01	2018	1	January	1	2	Monday
2	2018-01-08	2018	1	January	8	2	Monday
3	2018-01-15	2018	1	January	15	2	Monday
4	2018-01-22	2018	1	January	22	2	Monday
5	2018-01-29	2018	1	January	29	2	Monday
6	2018-02-05	2018	2	February	5	2	Monday
7	2018-02-12	2018	2	February	12	2	Monday
8	2018-02-19	2018	2	February	19	2	Monday
9	2018-02-26	2018	2	February	26	2	Monday
10	2018-03-05	2018	3	March	5	2	Monday
11	2018-03-12	2018	3	March	12	2	Monday
12	2018-03-19	2018	3	March	19	2	Monday

PySpark notebooks – PySpark & SQL

```python
# load data using python, into a spark dataframe, using a SQL string

query = "SELECT * FROM Silver.silver_dimDates LIMIT 10"

spark_df = spark.sql(query)
display(spark_df)
```

✓ 5 sec - Command executed in 5 sec 36 ms by DWHDEV | Kimura on 1:58:06 PM, 4/20/24

> Spark jobs (2 of 2 succeeded) Resources Log

Table Chart Showing rows 1 - 10

	Date	123 Year	123 MonthNumber	ABC MonthName	123 DayOfMonth	123 DayOfWeek	ABC Day
1	2018-01-01	2018	1	January	1	2	Monday
2	2018-01-08	2018	1	January	8	2	Monday
3	2018-01-15	2018	1	January	15	2	Monday
4	2018-01-22	2018	1	January	22	2	Monday
5	2018-01-29	2018	1	January	29	2	Monday
6	2018-02-05	2018	2	February	5	2	Monday
7	2018-02-12	2018	2	February	12	2	Monday
8	2018-02-19	2018	2	February	19	2	Monday
9	2018-02-26	2018	2	February	26	2	Monday

PySpark notebooks – SparkSQL

```
%%sql
-- query data using SQL

SELECT
    t.*
FROM Silver.silver_dimDates t
LIMIT 10
```

✓ 2 sec - Command executed in 1 sec 573 ms by DWHDEV | Kimura on 1:59:44 PM, 4/20/24

Spark jobs (2 of 2 succeeded)   Resources

Table   Chart

	Date	123 Year	123 MonthNumber	ABC MonthName	123 DayOfMonth	123 DayOfWeek	ABC Day
1	2018-01-01	2018	1	January	1	2	Monday
2	2018-01-08	2018	1	January	8	2	Monday
3	2018-01-15	2018	1	January	15	2	Monday
4	2018-01-22	2018	1	January	22	2	Monday
5	2018-01-29	2018	1	January	29	2	Monday
6	2018-02-05	2018	2	February	5	2	Monday
7	2018-02-12	2018	2	February	12	2	Monday
8	2018-02-19	2018	2	February	19	2	Monday
9	2018-02-26	2018	2	February	26	2	Monday

PySpark notebooks – SparkSQL -> PySpark

```
%%sql
CREATE OR REPLACE TEMPORARY VIEW my_temp_view AS
SELECT
    t.*
FROM Silver.silver_dimDates t
LIMIT 10
```

3 sec - Command executed in 2 sec 405 ms by DWHDEV | Kimura on 2:18:05 PM, 4/20/24

Log

No data available

```
spark_df_from_view = spark.sql("select * from my_temp_view")
display(spark_df_from_view)
```

2 sec - Command executed in 1 sec 588 ms by DWHDEV | Kimura on 2:18:32 PM, 4/20/24

Spark jobs (2 of 2 succeeded)   Resources   Log

Table   Chart   Showing rows 1 - 10

	Date	123 Year	123 MonthNumber	ABC MonthName	123 DayOfMonth	123 DayOfWeek
1	2018-01-01	2018	1	January	1	2
2	2018-01-08	2018	1	January	8	2
3	2018-01-15	2018	1	January	15	2
4	2018-01-22	2018	1	January	22	2
5	2018-01-29	2018	1	January	29	2
6	2018-02-05	2018	2	February	5	2
7	2018-02-12	2018	2	February	12	2

PySpark notebooks – PySpark -> SparkSQL

```
# load data using python, into a spark dataframe
spark_df = spark.read.format('delta').load('Tables/silver_dimDates')
spark_df.createOrReplaceTempView('my_temp_view')
```

✓ 1 sec - Command executed in 838 ms by DWHDEV | Kimura on 2:21:11 PM, 4/20/24

> Log

```
%%sql
SELECT
    t.*
FROM my_temp_view t
LIMIT 10
```

✓ 2 sec - Command executed in 1 sec 489 ms by DWHDEV | Kimura on 2:21:16 PM, 4/20/24

> Spark jobs (2 of 2 succeeded) Resources Log

Table Chart

	Date	123 Year	123 MonthNumber	ABC MonthName	123 DayOfMonth	123 DayOfWeek	ABC Day
1	2018-01-01	2018	1	January	1	2	Monday
2	2018-01-08	2018	1	January	8	2	Monday
3	2018-01-15	2018	1	January	15	2	Monday
4	2018-01-22	2018	1	January	22	2	Monday
5	2018-01-29	2018	1	January	29	2	Monday
6	2018-02-05	2018	2	February	5	2	Monday
7	2018-02-12	2018	2	February	12	2	Monday
8	2018-02-19	2018	2	February	19	2	Monday

Code reusability

Topic 3/3

Code reusability

- Manual work vs Code generation vs Code reusability

- Manual work: NOPE
- Code generation: cool, done that a lot, but it’s not needed anymore
- Code reusability: write code once, execute many times

- Examples on the next slides

Lazy is good

- In Fabric, we try to be <u>lazy data engineers</u>

- The more work we do manually, the less customers we can help and the more errors we make

- The general rule is; everything you do twice, you put into our Kimura Data Framework

- DRY: Don’t Repeat Yourself

Generated code

```sql
SELECT
    ISNULL([t].id, '') AS [Id]
,   ISNULL([t].internalName, '') AS [InternalName]
,   ISNULL([t].externalName, '') AS [ExternalName]
,   ISNULL([t].carrierId, 0) AS [CarrierId]
,   ISNULL([t].country, '') AS [Country]
,   ISNULL([t].batchNumber, 0) AS [BatchNumber]
,   ISNULL([t].deletedAt, '1900-01-01') AS [DeletedAt]
,   ISNULL(getDate(), '1900-01-01') AS [Sys_FirstLoadTimeStamp]
,   ISNULL(getDate(), '1900-01-01') AS [Sys_LastUpdateTimeStamp]
,   ISNULL('I', '') AS [Sys_Operation]
,   ISNULL('1900-01-01', '1900-01-01') AS [Sys_ValidFrom]
,   ISNULL('2099-12-31', '1900-01-01') AS [Sys_ValidTo]
,   ISNULL(1, 1) AS [Sys_CurrentRow]
INTO #temp
FROM [pers].[Homerr-production_Depots] as t
where t.Sys_ValidFrom <= @date and t.Sys_ValidTo > @date

create index ix_temp on #temp ([Id])

MERGE INTO [dwh].[dimDepots] AS target
USING #temp AS source
ON target.[Id] = source.[Id]
```

```sql
    WHEN MATCHED AND (
        target.[Id] <> source.[Id]
OR      target.[InternalName] <> source.[InternalName]
OR      target.[ExternalName] <> source.[ExternalName]
OR      target.[CarrierId] <> source.[CarrierId]
OR      target.[Country] <> source.[Country]
OR      target.[BatchNumber] <> source.[BatchNumber]
OR      target.[DeletedAt] <> source.[DeletedAt]
)THEN
    UPDATE SET
    target.[Id] = source.[Id]
,   target.[InternalName] = source.[InternalName]
,   target.[ExternalName] = source.[ExternalName]
,   target.[CarrierId] = source.[CarrierId]
,   target.[Country] = source.[Country]
,   target.[BatchNumber] = source.[BatchNumber]
,   target.[DeletedAt] = source.[DeletedAt]
,   target.[Sys_LastUpdateTimeStamp] = source.[Sys_LastUpdateTimeStamp]
,   target.[Sys_Operation] = 'U'
,   target.[Sys_StageDB] = source.[Sys_StageDB]

    WHEN NOT MATCHED THEN
    INSERT (
    [Id]
,   [InternalName]
,   [ExternalName]
,   [CarrierId]
,   [Country]
,   [BatchNumber]
,   [DeletedAt]
,   [Sys_FirstLoadTimeStamp]
,   [Sys_LastUpdateTimeStamp]
,   [Sys_Operation]
,   [Sys_StageDB]
,   [Sys_ValidFrom]
,   [Sys_ValidTo]
,   [Sys_CurrentRow]
    )
    VALUES(
    source.[Id]
,   source.[InternalName]
,   source.[ExternalName]
,   source.[CarrierId]
,   source.[Country]
,   source.[BatchNumber]
,   source.[DeletedAt]
,   source.[Sys_FirstLoadTimeStamp]
,   source.[Sys_LastUpdateTimeStamp]
,   'I'
,   source.[Sys_StageDB]
,   source.[Sys_ValidFrom]
,   source.[Sys_ValidTo]
,   source.[Sys_CurrentRow]
    );
```

Reusable functions…

```python
#Generic path names for Silver tables
def silver_target_path(silver_table: str) -> str:
    path = f"Tables/silver_{silver_table}"
    return path

#Create Sys fields
def add_sys_fields(df: sdf, business_key_columns: list) -> sdf:
    df_with_sys_fields = \
        df.withColumn("Sys_ID", sha2(concat_ws("~", *business_key_columns), 256)) \
          .withColumn("Sys_ValidFrom", to_timestamp(lit("1900-01-01 00:00:00.000"), "yyyy-MM-dd HH:mm:ss.SSS")) \
          .withColumn("Sys_ValidTo", to_timestamp(lit("2999-12-31 00:00:00.000"), "yyyy-MM-dd HH:mm:ss.SSS")) \
          .withColumn("Sys_IsActive", lit(1).cast(BooleanType())) \
          .withColumn("Sys_LoadDatetime", current_timestamp())
    return df_with_sys_fields

#Upsert into delta table logic
def silver_create_table(silver_table: str, df_mapping: sdf, business_key_columns: list, spark_session: SparkSession) -> str:
    """
    This function performs an upsert into a delta table in Silver. If the table doesn't exist, it will be created.
    """

    result = "" #hiermee kunnen we bijvoorbeeld logging output geven

    df_to_merge = add_sys_fields(df_mapping, business_key_columns)

    target_path = silver_target_path(silver_table)
    exclude_from_update = ['Sys_ID', 'Sys_ValidFrom', 'Sys_ValidTo', 'Sys_IsActive', 'Sys_LoadDatetime']
    update_cols = {col: f"s.`{col}`" for col in df_to_merge.columns if col not in exclude_from_update}

    if DeltaTable.isDeltaTable(spark_session, target_path):
        silver_table = DeltaTable.forPath(spark_session, target_path)

        #Upsert dataframe to delta
        update_condition = " AND ".join(f"t.`{key_col}` = s.`{key_col}`" for key_col in business_key_columns)
        silver_table.alias("t").merge(
            source = df_to_merge.alias("s"),
            condition = update_condition
        ).whenMatchedUpdate(set=update_cols).whenNotMatchedInsertAll().execute()
    else:
        #if silver table doesn't exist yet
        df_to_merge.write.format('delta').mode('overwrite').save(target_path)

    return result
```

…make for super easy notebooks

```python
import kimura as k

#Mapping logic
silver_table = "dimHandlingTypes"
business_key_columns = ['HandlingTypeId']

query = """
select DISTINCT
    t.CARGOHANDLINGTYPE as HandlingTypeId
    ,e.MEMBERNAME as Description
from Bronze.bronze_BYOD_FlxUS_CargoTransHandlingStaging t
left join Bronze.bronze_BYOD_FlxUS_EnumValueTableStaging e
    on t.CARGOHANDLINGTYPE = e.VALUE
    and e.ENUMNAME = 'FlxUs_CargoHandlingType'
"""

df_mapping = spark.sql(query)

k.silver_create_table(
    silver_table = silver_table,
    df_mapping = df_mapping,
    business_key_columns = business_key_columns,
    spark_session = spark
)
```

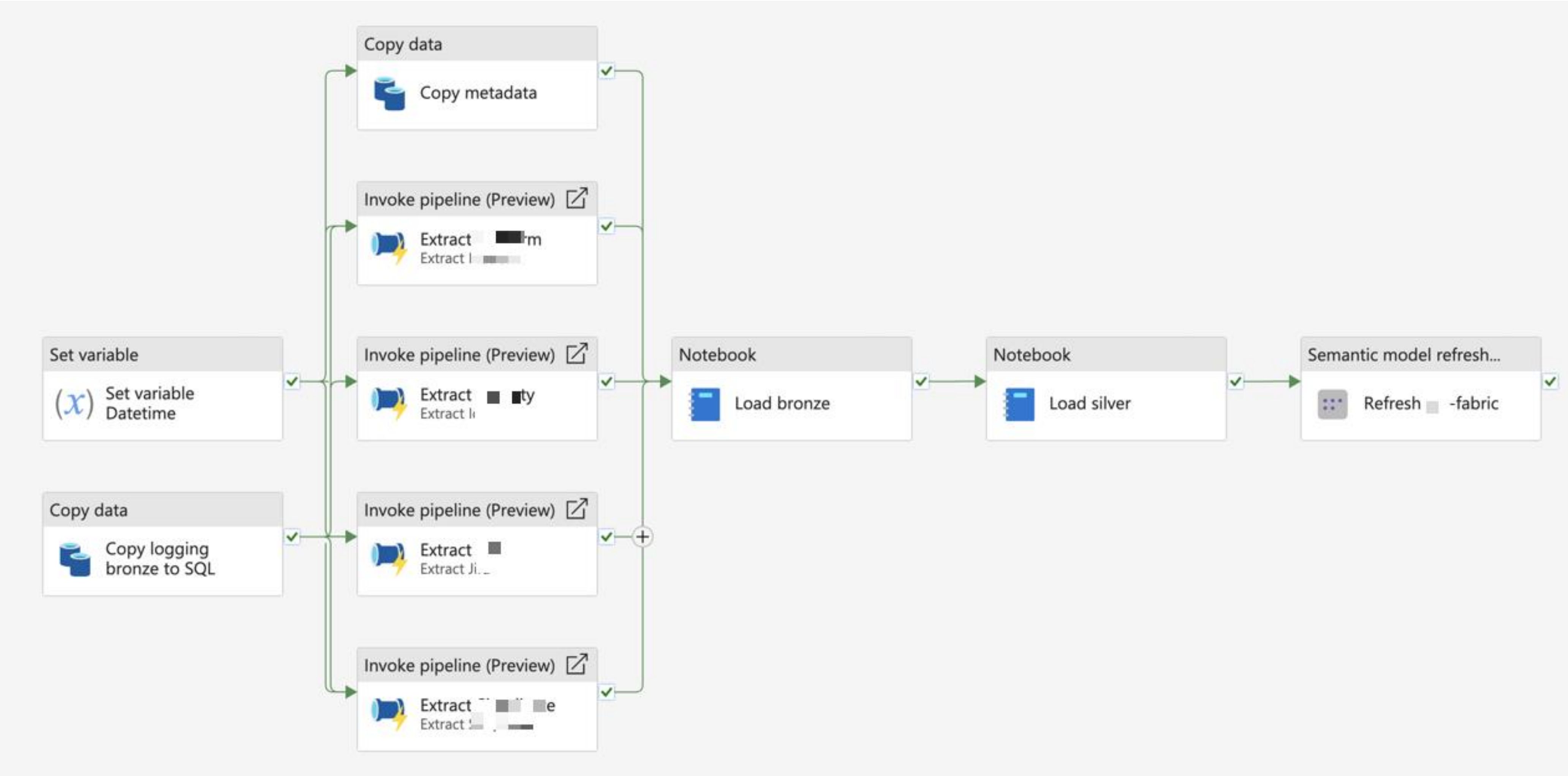
Your first lakehouse

End-to-end overview

Rulebook

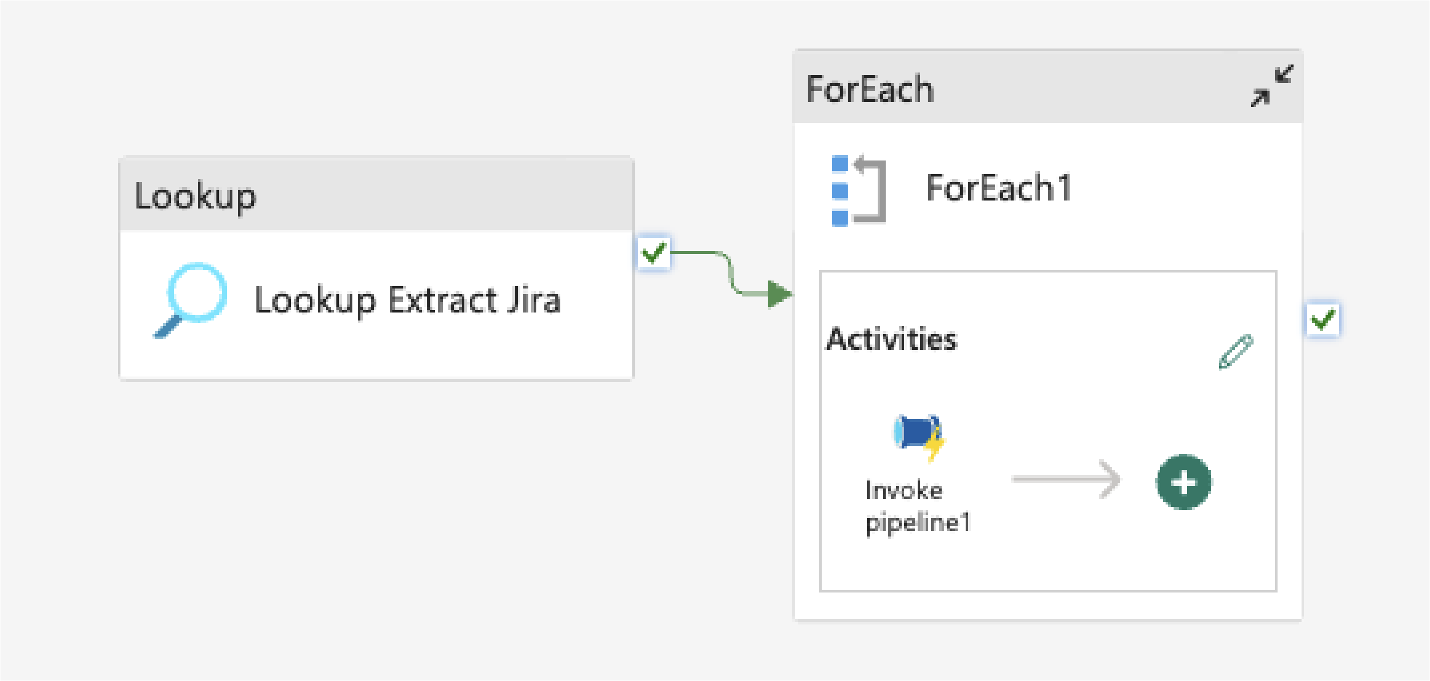
- Use Data Factory Pipelines for ingest and orchestration
  - Parametrised pipelines using Azure SQL (or Fabric Warehouse?) config database
- Use Lakehouse for storage (files in bronze, delta in silver + gold)
- PySpark and SparkSQL in notebooks for processing
  - Reusable code with custom Python library
- Power BI semantic model using Import Mode and SQL endpoint on Gold Lakehouse

Main pipeline (simplified)



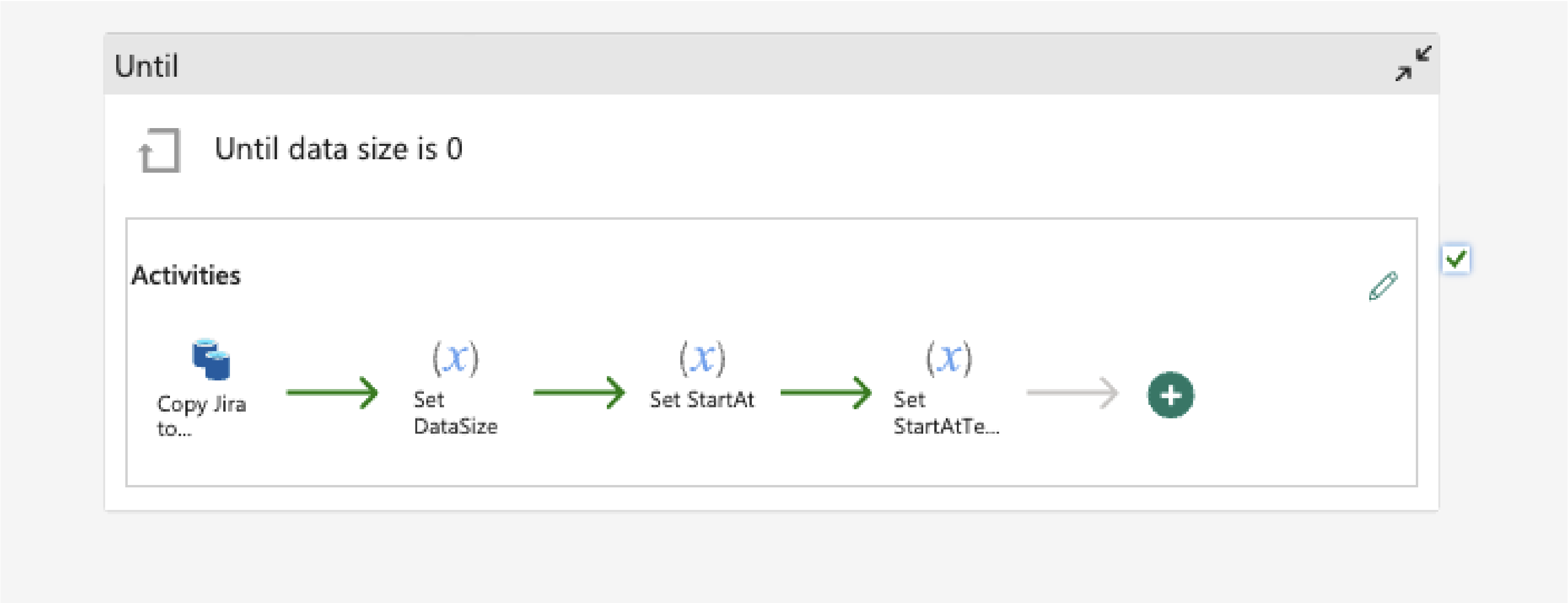
Ingest / extract example

- Pipeline + data lake folder structure



Ingest / extract example

- Pipeline + data lake folder structure



Ingest / extract example

- Pipeline + data lake folder structure

Pipeline expression builder

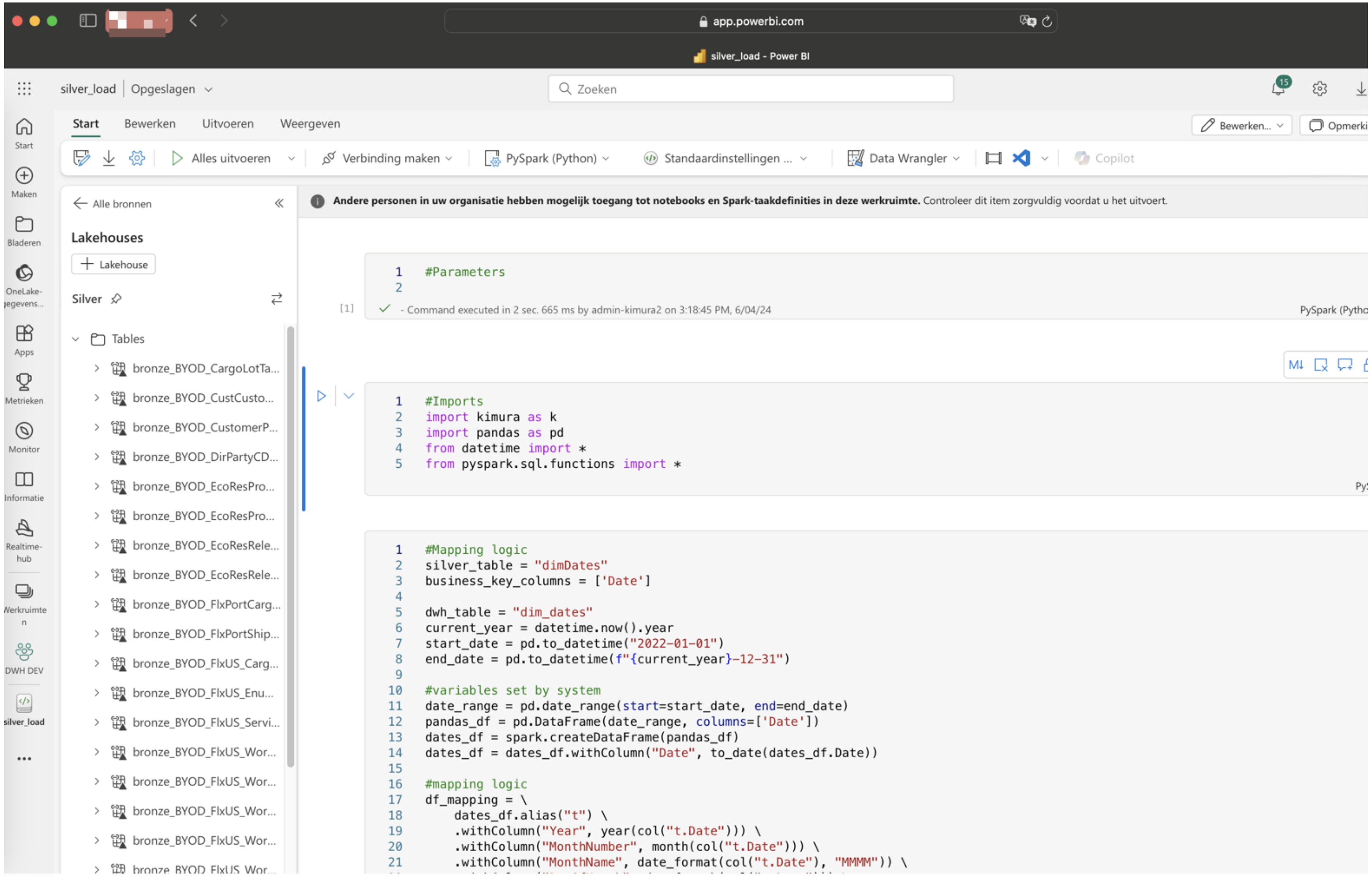
Add dynamic content below using any combination of expressions, functions and system variables.

```
@concat(
    'Landing/',
    '/Full/',
    pipeline().parameters.Datetime,
    '/',
    pipeline().parameters.SourceSystem,
    '/',
    pipeline().parameters.TargetTable
)
```

Clear contents

Notebooks

- Starting with a very light weight setup, from your browser:



Notebooks

- Starting with a very light weight setup, from your browser:

- You can worry about Visual Studio Code later ☺

Notebook Bronze -> Silver

```python
json_path = f"{base_path}/{timestamp}/metadata.json"
json_df = spark.read.json(json_path)
json_content = json_df.collect()[0].asDict()
table_list = json.loads(json_content['TableList'])
# display(table_list)

#Load data to Bronze layer delta tables
for table in table_list:
    target_table = table['TargetTable']
    source_system = table['SourceSystem']
    file_type = table['FileType']
    table_id = table['Id']
    incremental_field = table['IncrementalField']
    json_root_array = table['JSONRootArray']
    table_key_list = ast.literal_eval(table['TableKey']) #

    #Start logging
    start_datetime = datetime.now()

    #Execute
    print(f"Starting with table: {target_table}")
    count_rows = k.bronze_create_table(
        timestamp = timestamp,
        sourcesystem = source_system,
        tablename = target_table,
        tablekeys = table_key_list,
        base_path = base_path,
        file_type = file_type,
        table_id = table_id,
        incremental_field = incremental_field,
        json_root_array = json_root_array,
        spark_session = spark
    )

    #End logging
    end_datetime = datetime.now()
    delta = end_datetime - start_datetime
    print(f"Table: {target_table}, rowcount: {count_rows}, duration in sec: {delta.total_seconds()}")
```

```python
#Parameters
full_load = False
#if full_load = True then travel thr
timestamp = "20240223131249"#"202402
```

✓ - Apache Spark session ready in 3 min 30 sec 150 ms. Comman

```python
#Imports
import kimura as k
import ast
from datetime import datetime
import json

#Base stuff
base_path = f"Files/Landing/Full"
```

✓ - Command executed in 270 ms by DWHDEV | Kimura on 3:57:

Notebook Silver -> Gold

```
#Mapping logic
silver_table = "dimProjects"
business_key_columns = ['ProjectId']

query = """
select
    t.`data.id` as ProjectId
    ,t.`data.name` as ProjectName
    ,regexp_replace(t.`data.project_status.label`,'tab_','') as Status
    ,t.`data.organization_id` as OrganisationId
    ,t.`data.start_date` as StartDate
    ,t.`data.end_date` as EndDate
    ,c.Sys_ID as FK_dimCustomers_Sys_ID
from Bronze.bronze_Simplicate_ProjectsProject t
left join Silver.silver_dimCustomers c
    on t.`data.organization.id` = c.SimplicateId
where 1=1
"""

df_mapping = spark.sql(query)
# display(df_mapping)
output = k.silver_create_table(silver_table = silver_table, df_mapping = df_map
```

Recap

- Fabric Lakehouses are very powerful
- Spark (especially SparkSQL) is not hard to learn for SQL Engineers
- Code reusability is much easier than in SQL
- So… start using it tomorrow?

That’s it!

Questions?

Kimura Data Intelligence B.V.

“Wij zoeken dringend een lead data consultant. Ben jij diegene?”

Fonteinkruid 6b
3931 WX, Woudenberg
The Netherlands

www.kimura.nl
info@kimura.nl

Kimura
data intelligence