

Scaling data pipelines @Telekom

Aleks & Georg



Agenda

- 01 Scaling data usage within enterprise
Principles of scaling and historical problems
- 02 Technical foundation for scaling
Development process, Modeling, Governance
- 03 Dagster@Telekom
Advantages of asset-based data orchestration

About us



Data expert in academia and industry Magenta Telecom

- meetup organizer and conference speaker
- data architecture, multimodal and complex data challenges



Data engineer at Magenta Telecom

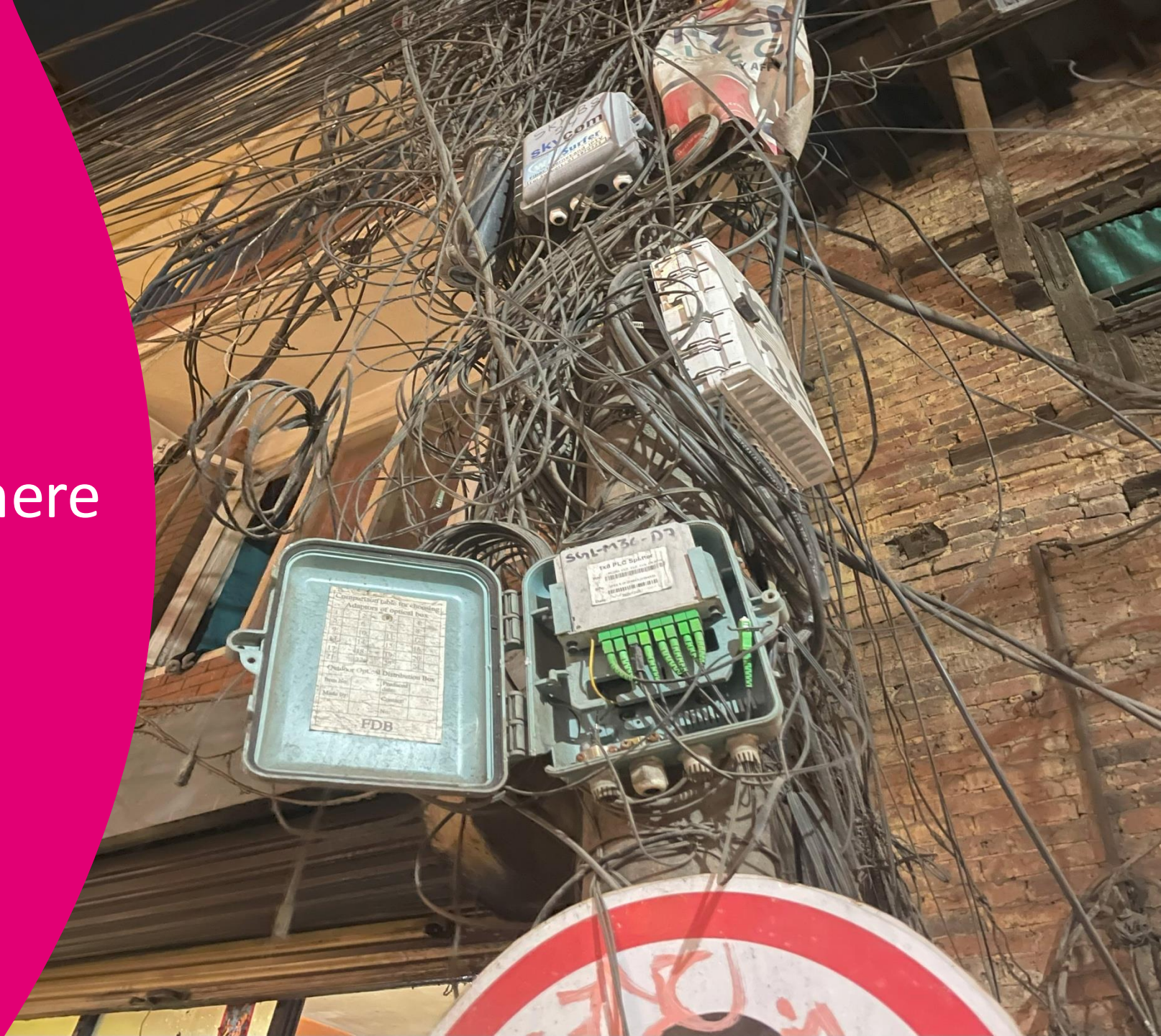
- database internals, data platform engineering

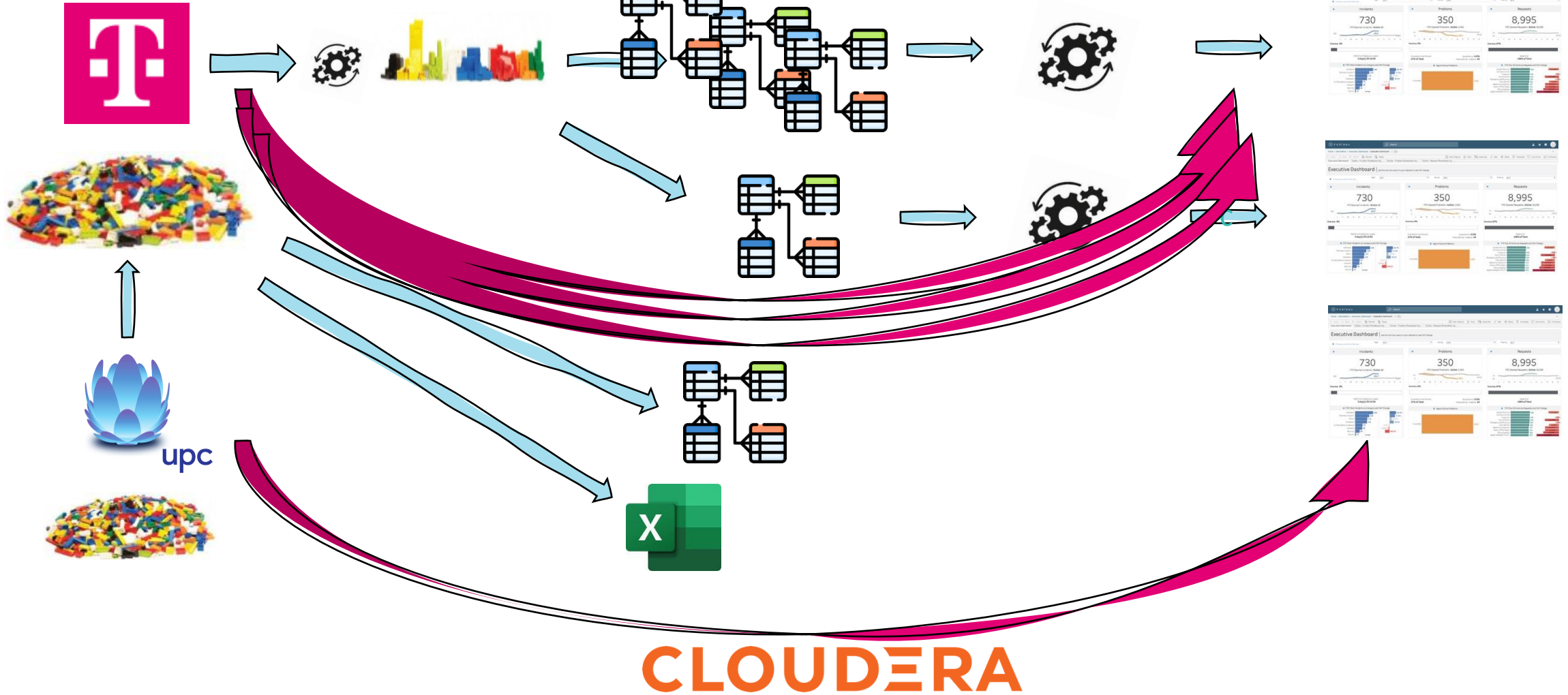
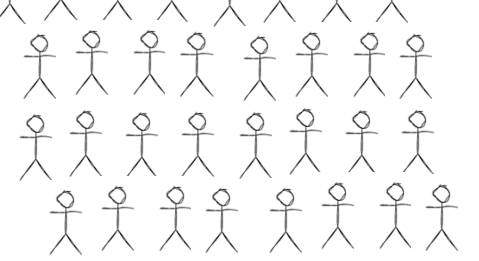


[public live stream about the basics: georgheiler.com/event/magenta-pixi-25](https://yam-united.telekom.com/pages/data-magenta/apps/blog/updates/view/1496666c-c479-4e09-8ef5-0ebfe709e179)

<https://yam-united.telekom.com/pages/data-magenta/apps/blog/updates/view/1496666c-c479-4e09-8ef5-0ebfe709e179>

There is a chaos out there





How did we end here? Time!

business grows (merger)

demand for data grows

methodology and tooling changes

- Missing lineage
- Missing semantics
- Missing collaboration
- High lead times
- Limited quality

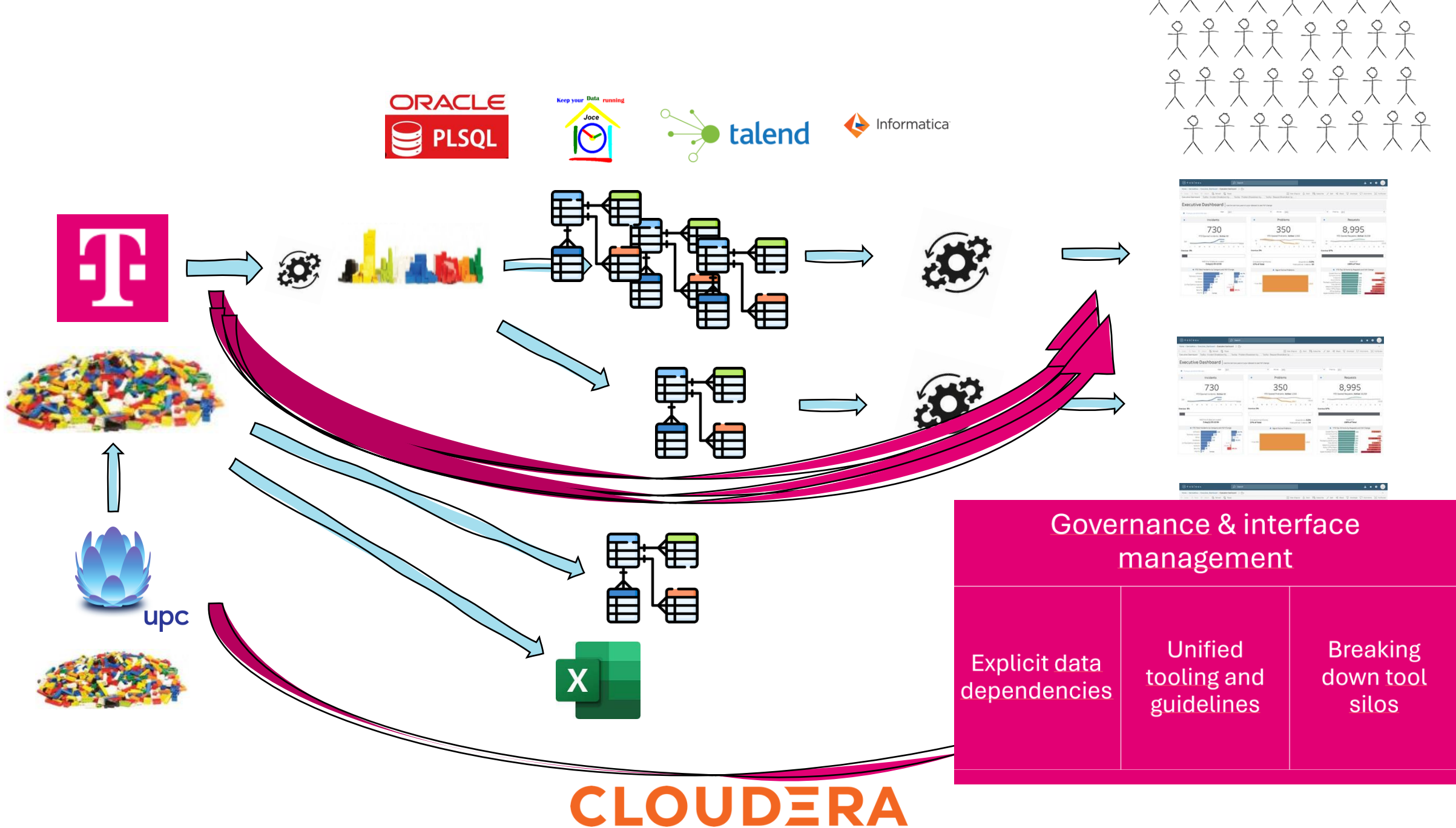


Governance & interface management

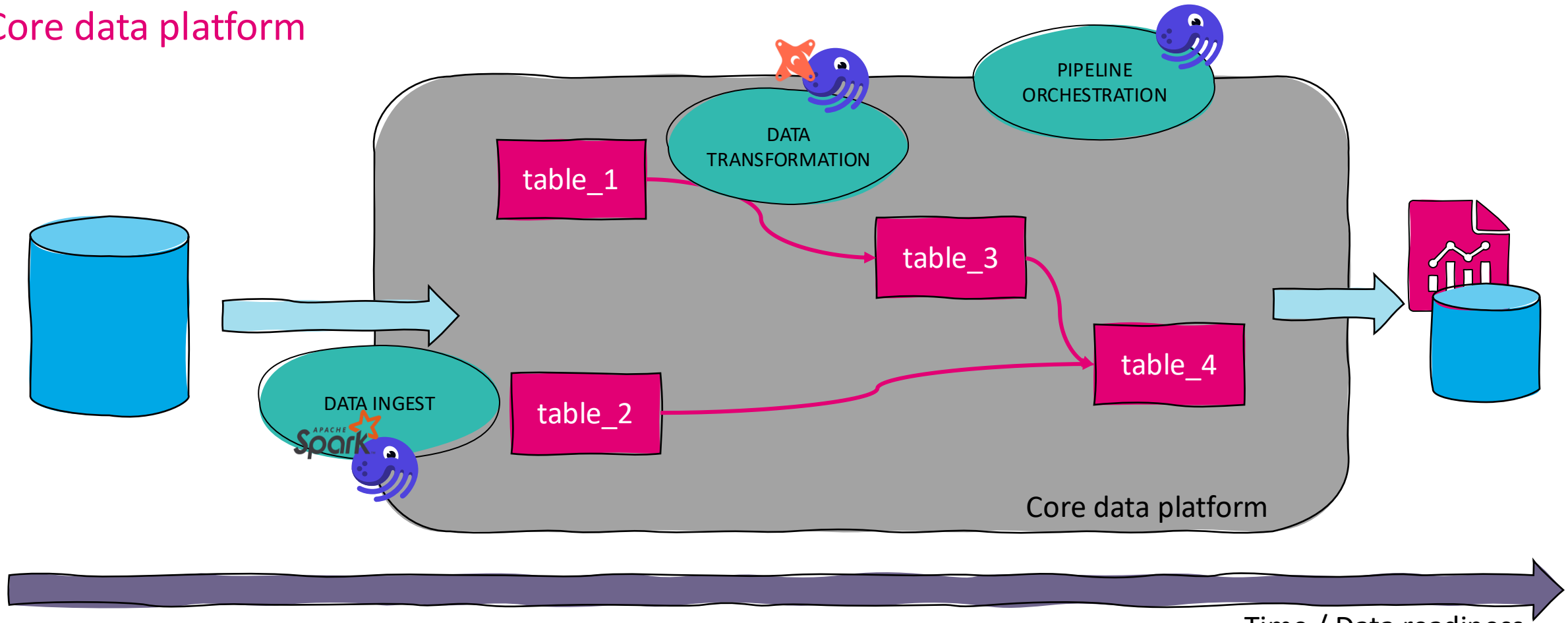
Explicit data
dependencies

Unified
tooling and
guidelines

Breaking
down tool
silos



Core data platform

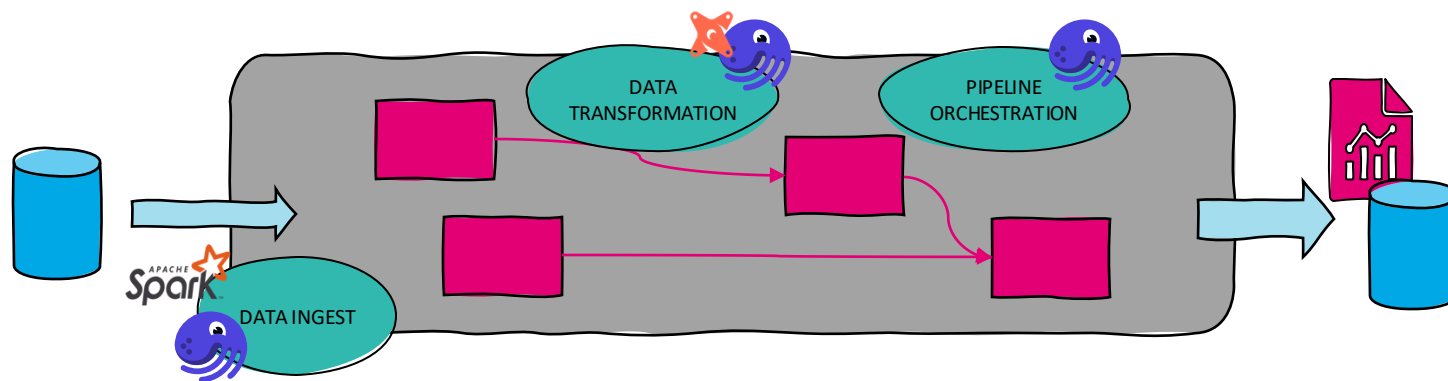


Source data:

- Kafka
- Files
- Database systems

Time / Data readiness

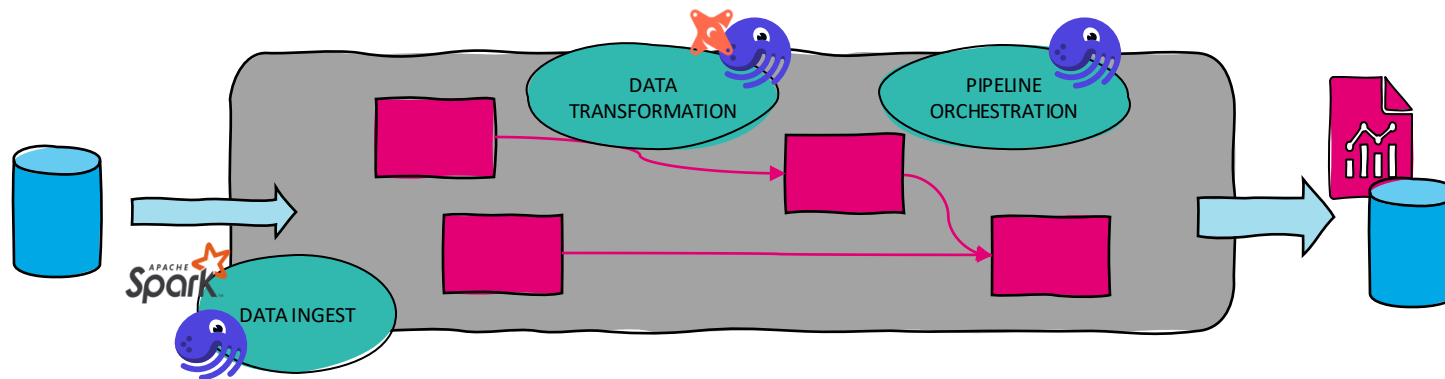
I fear that what we build is very hard to push into the business units



know

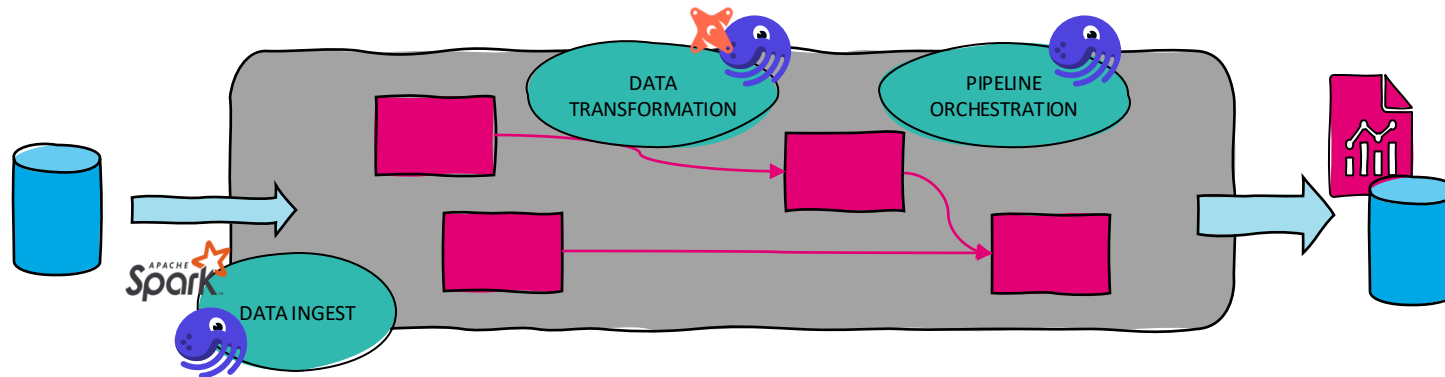
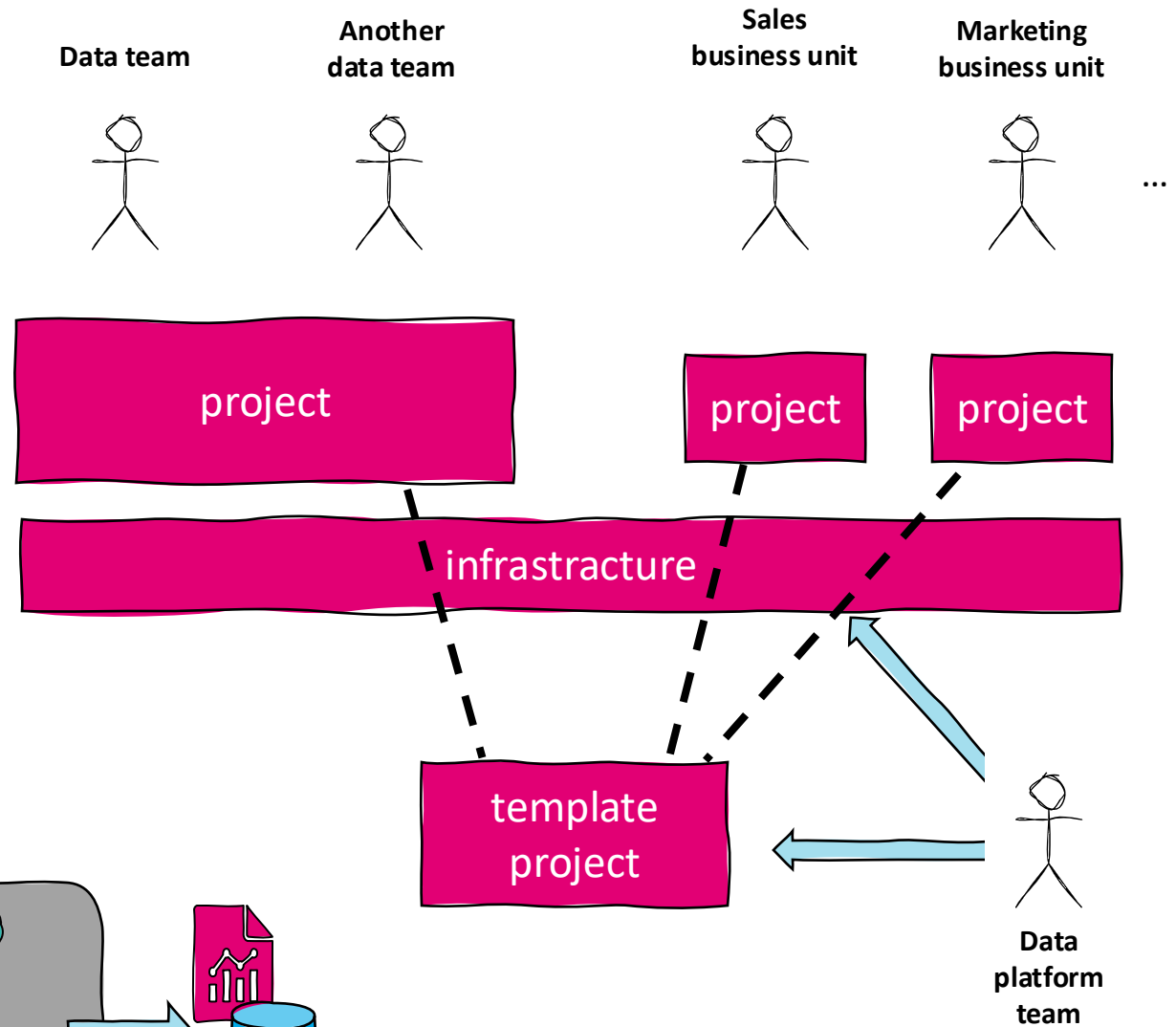
easy

I fear that what we build is very hard to push into the business units



Observation

- Process is straight forward: ingest, transform, use
- Everything we do - we do for business to provide better service
- Hard to scale across company
- Dividing people into **develop framework** and **use framework** groups
- Thinking in a **building block** structure
- Introduce modern tooling supporting software engineering practices: **dbt, dagster, pixi, docker**
- Introduce **new processes, modeling** and **metadata tooling** for better governance

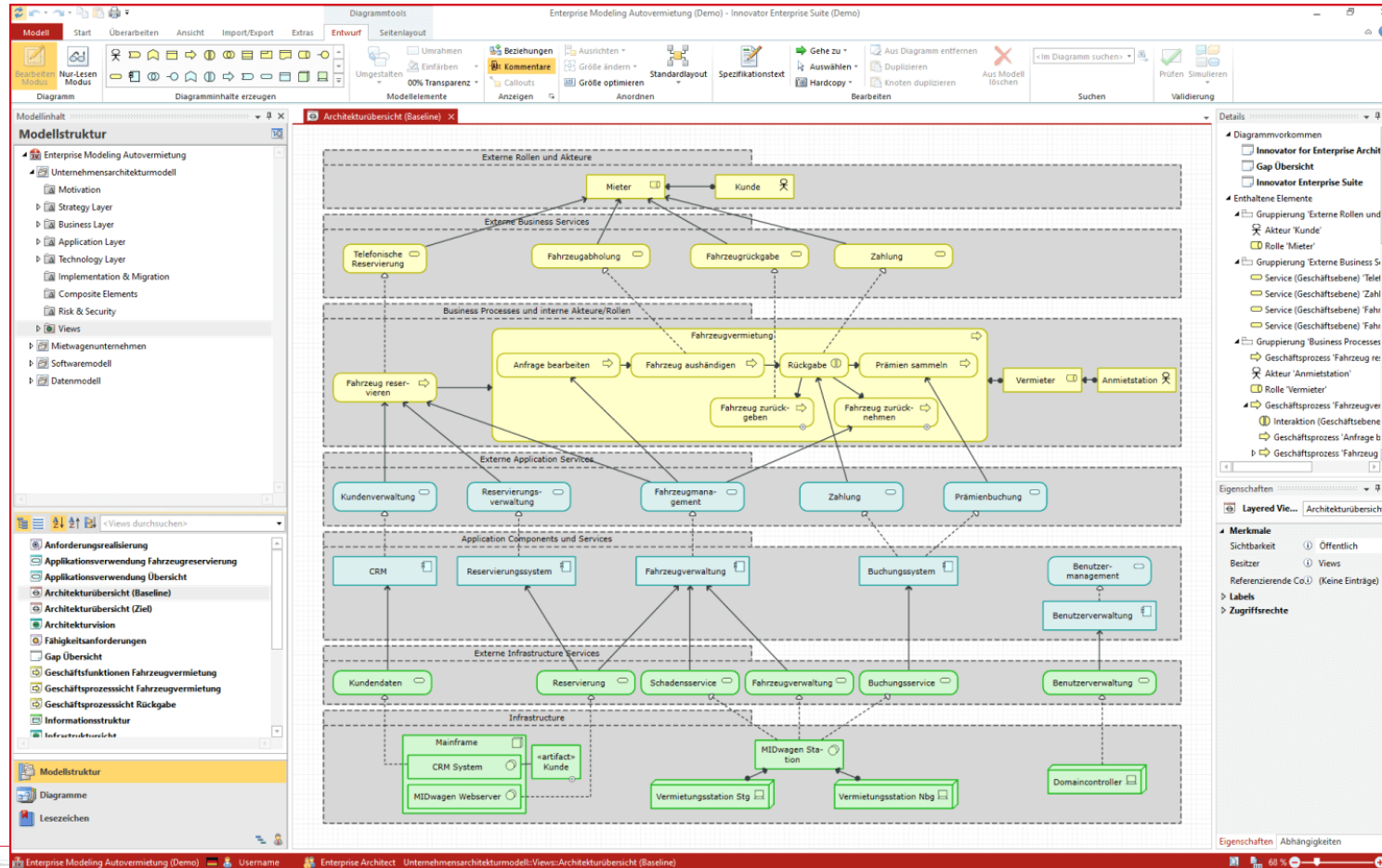
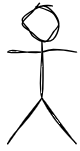


Having a data platform team doesn't mean that your platform scales

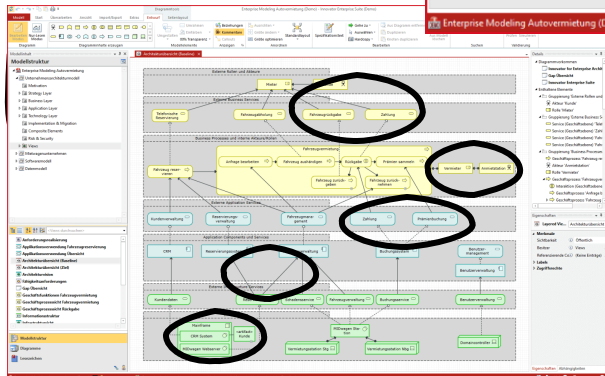
1. Building block with governance, modeling and software-engineering principles
2. Understanding data platform vendor war
3. Understanding tool silos

Break the silos with the building block

Data team

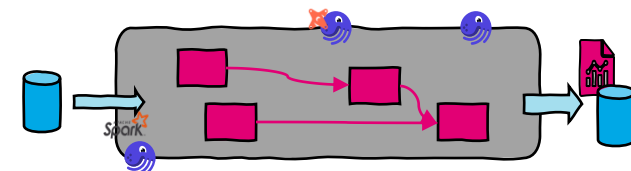


Sales business unit



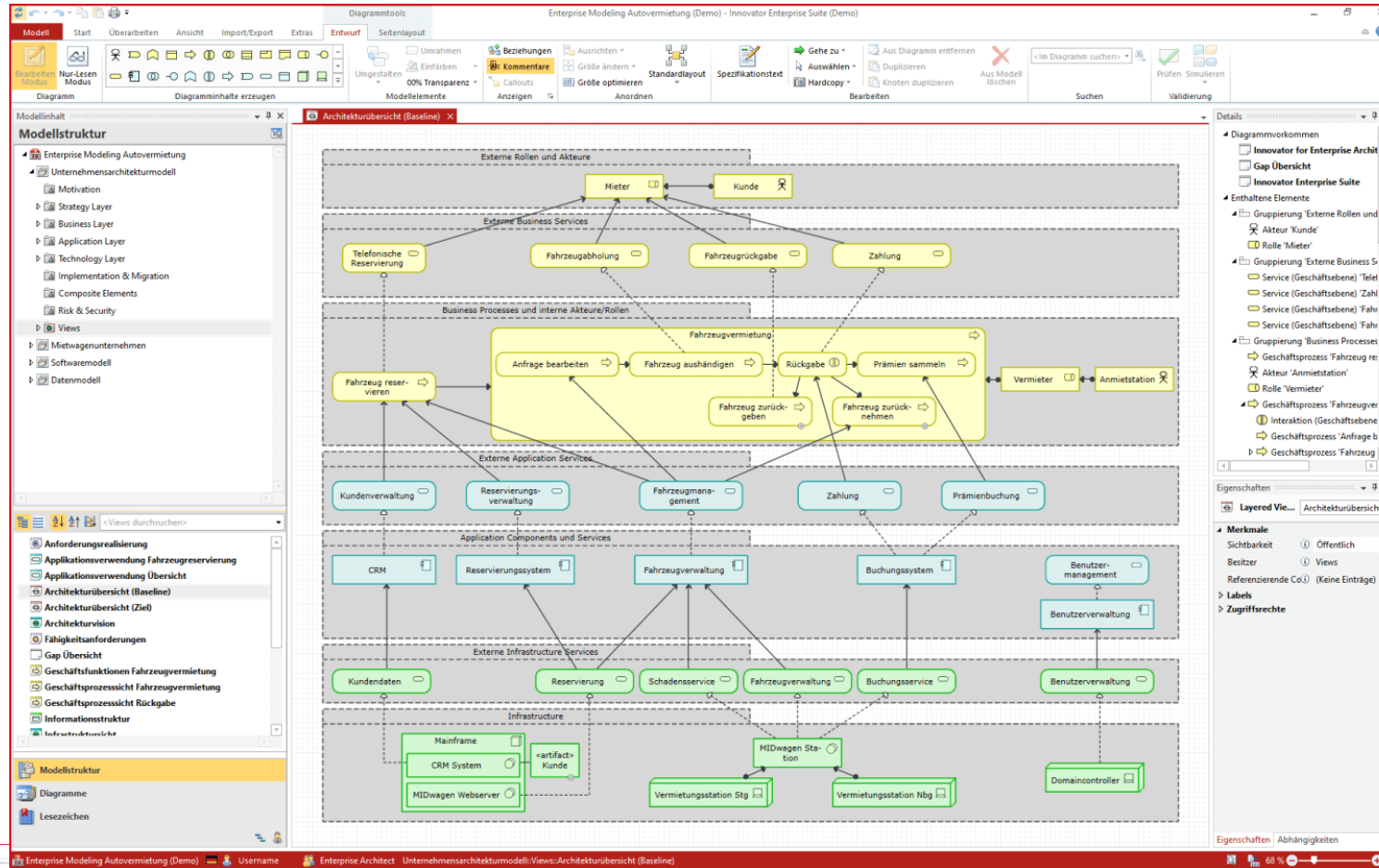
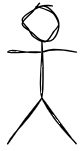
infrastructure

template project

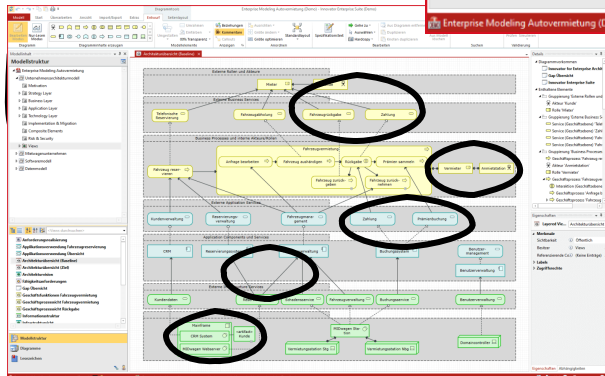


Break the silos with the building block

Data team

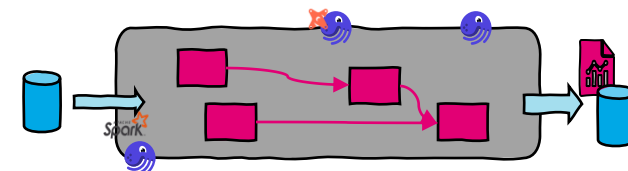


Sales business unit



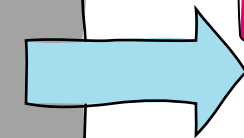
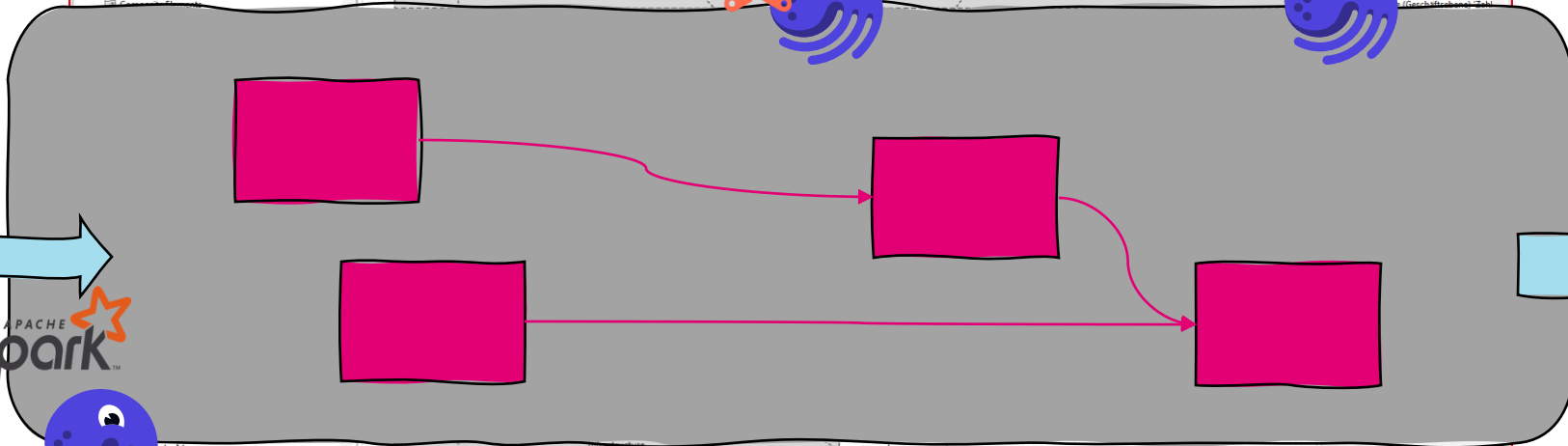
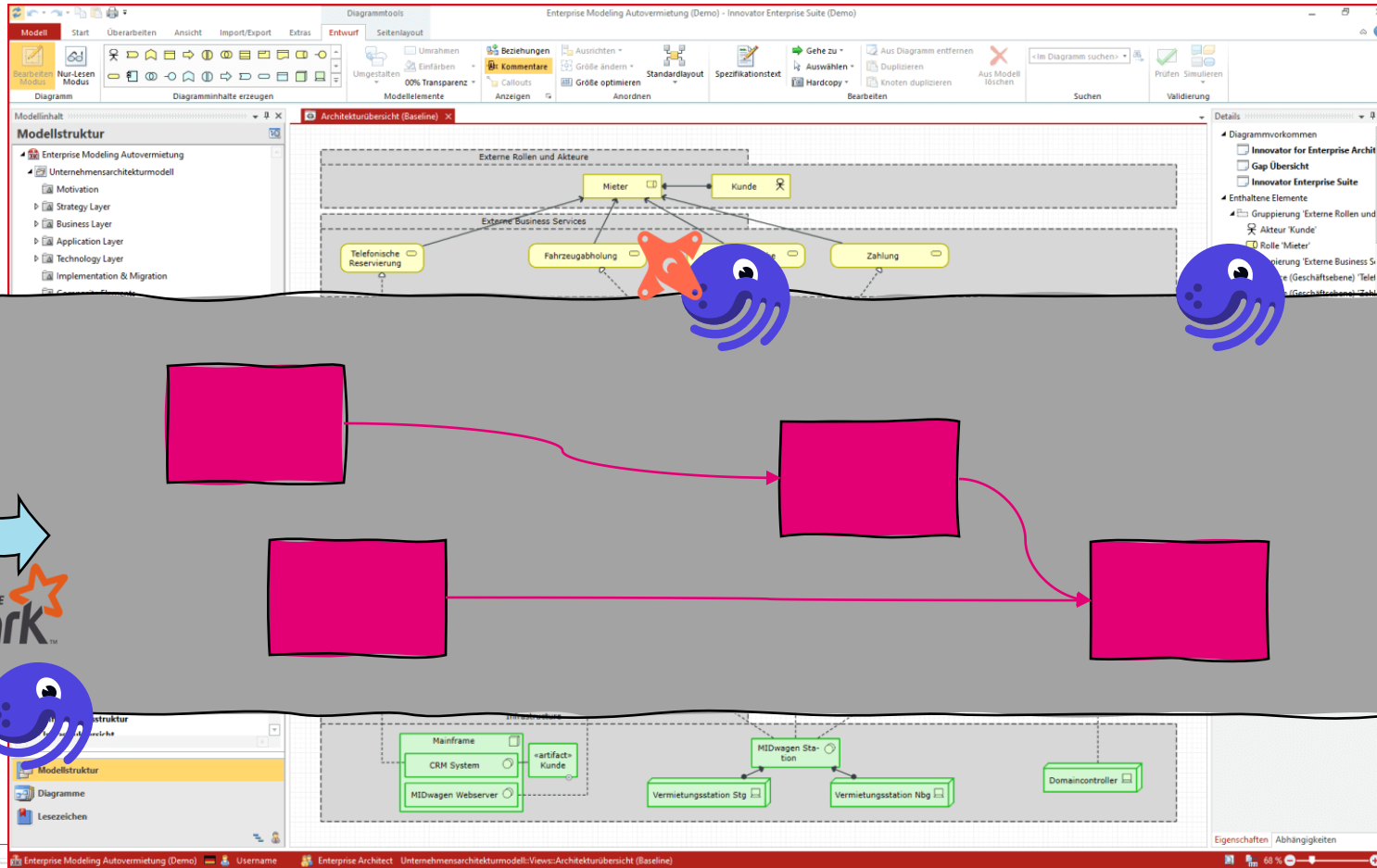
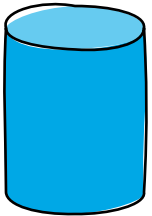
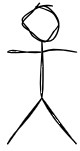
infrastructure

template project

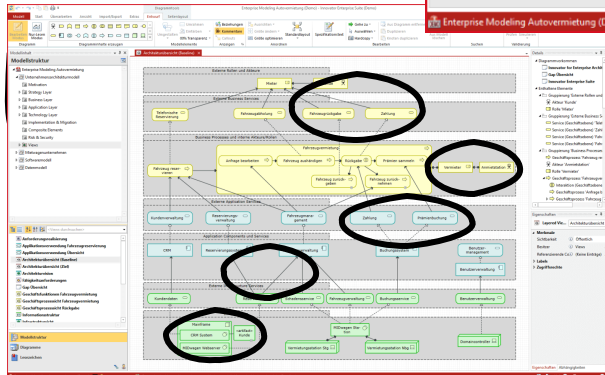
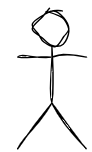
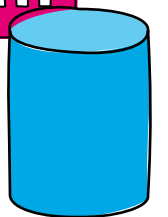


Break the silos with the building block

Data team

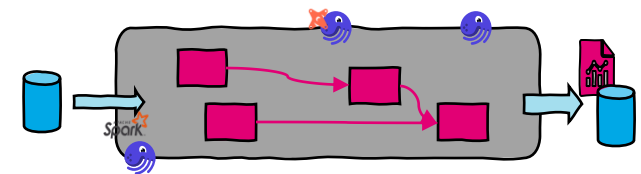


Sales business unit



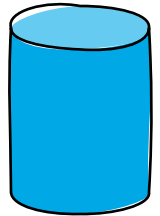
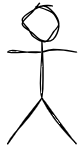
infrastructure

template project

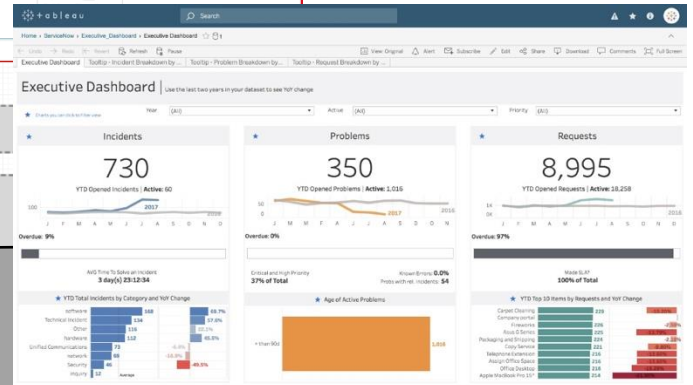
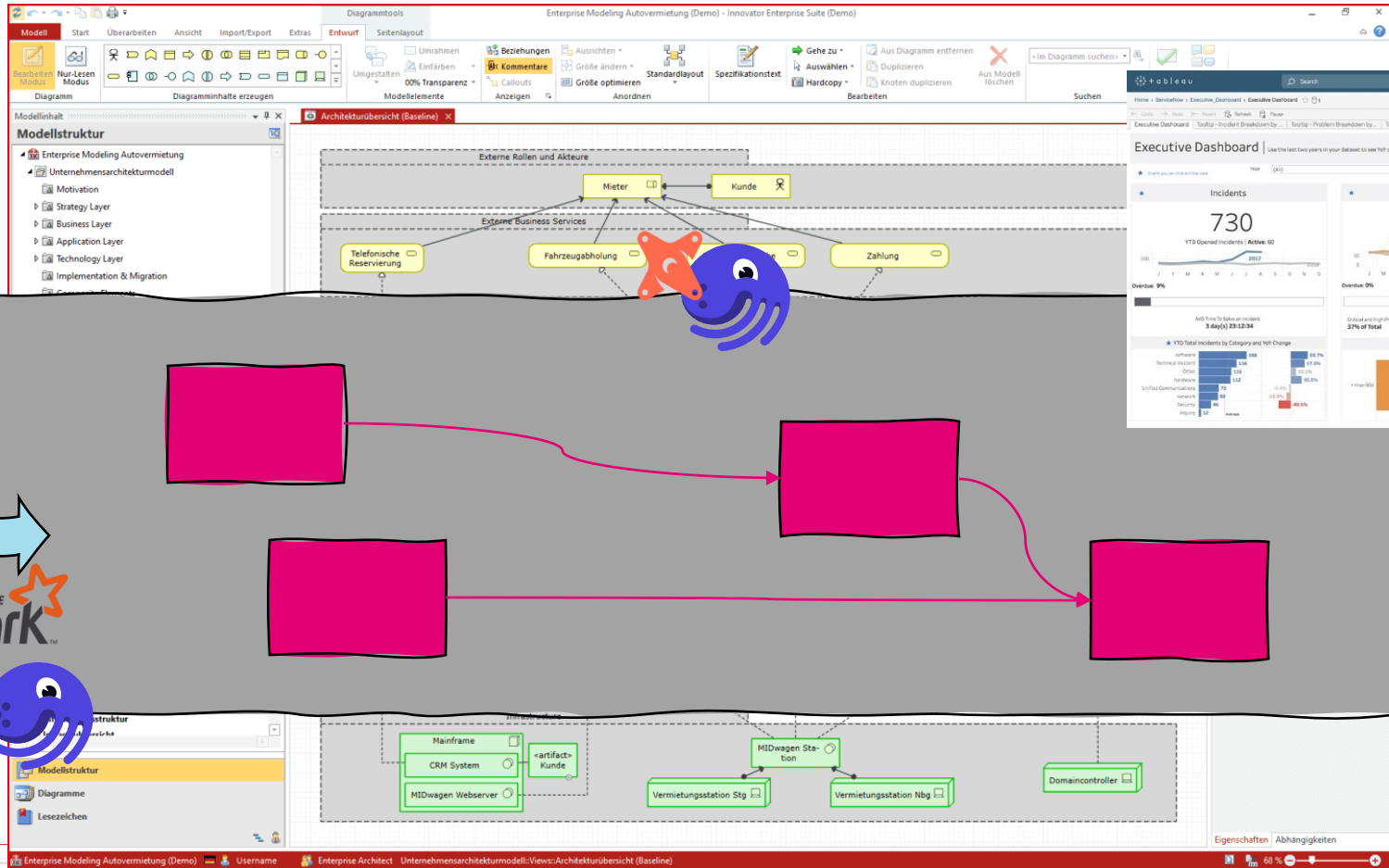


Break the silos with the building block

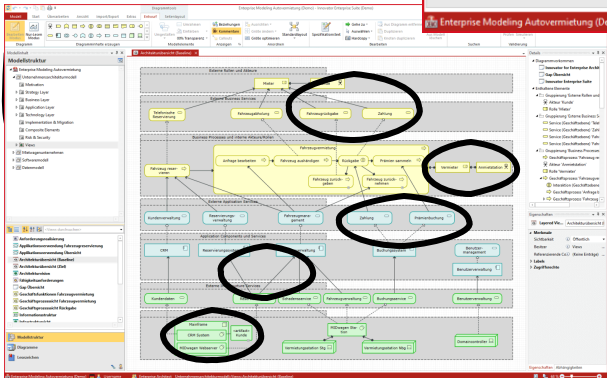
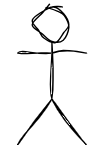
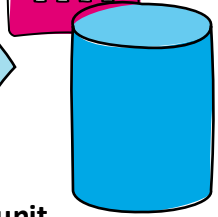
Data team



APACHE Spark

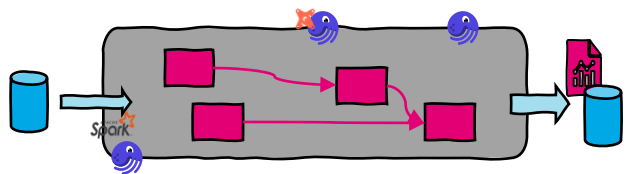


Sales business unit



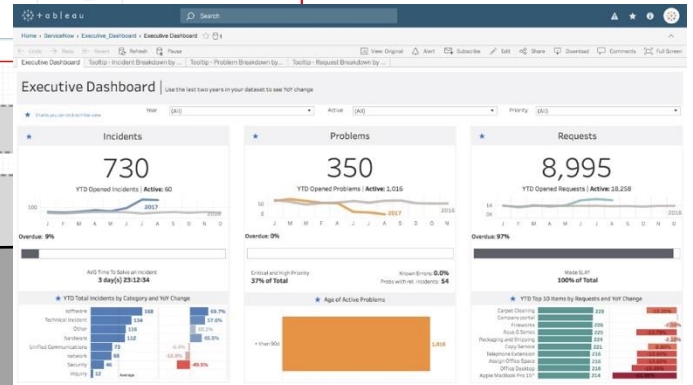
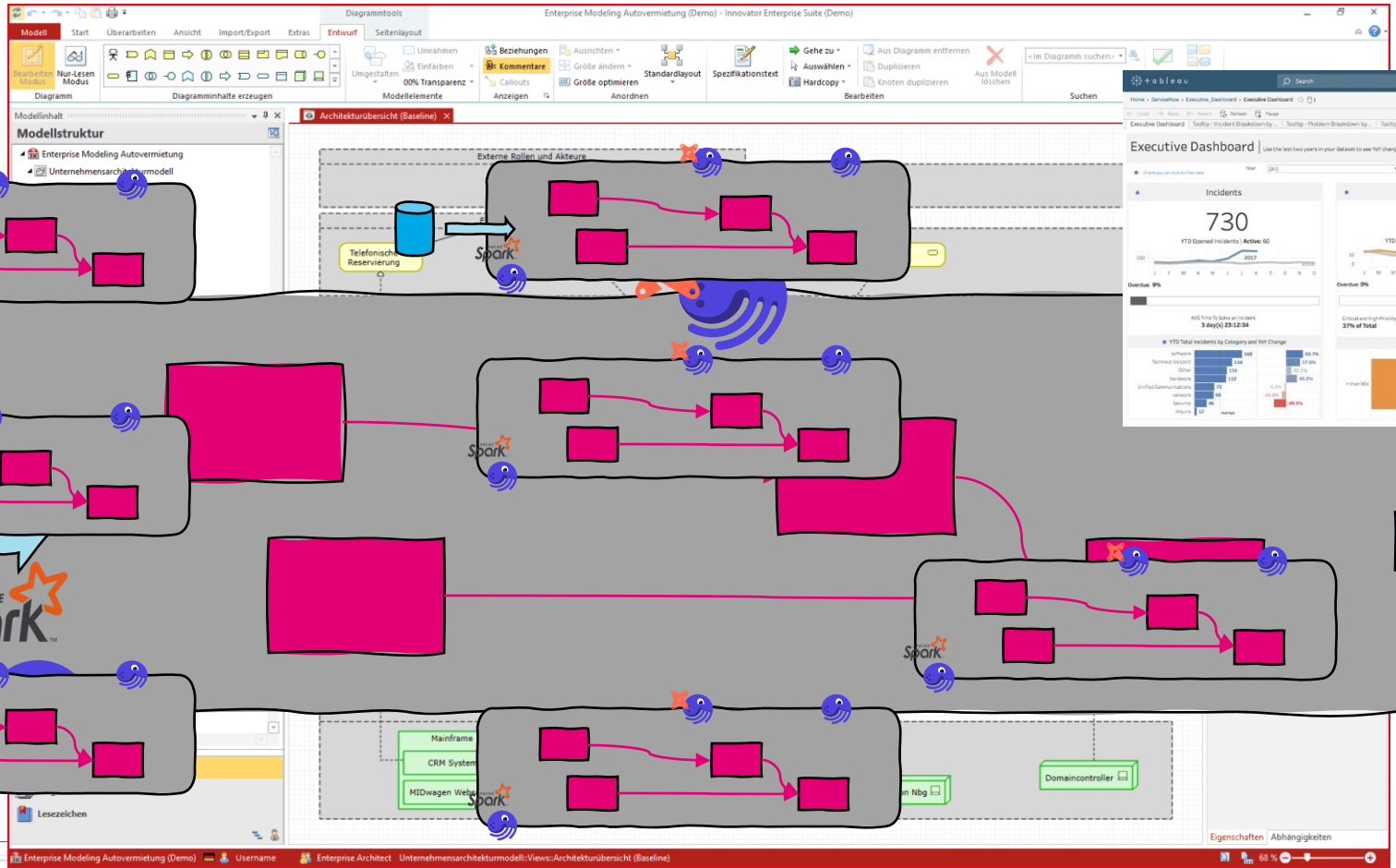
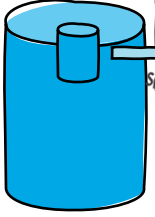
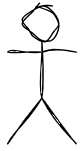
infrastructure

template project

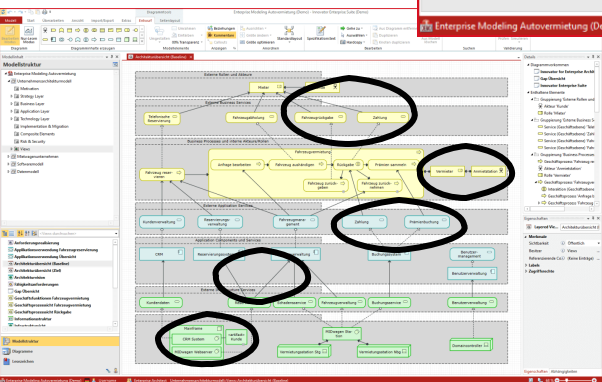
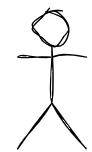


Break the silos with the building block

Data team

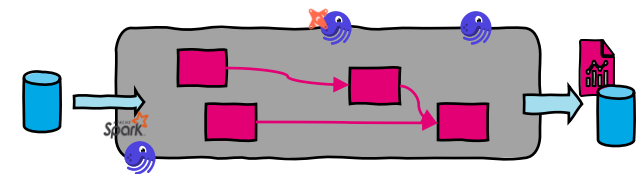


Sales business unit



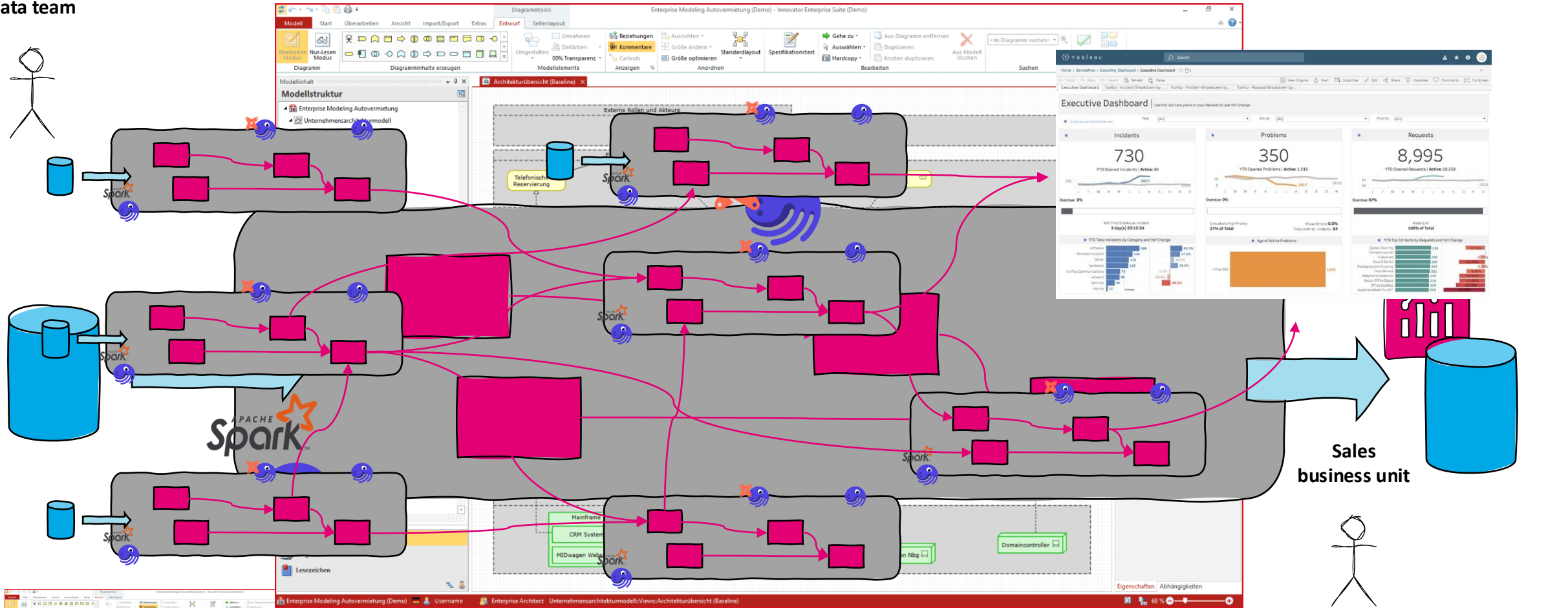
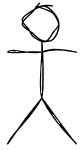
infrastructure

template project

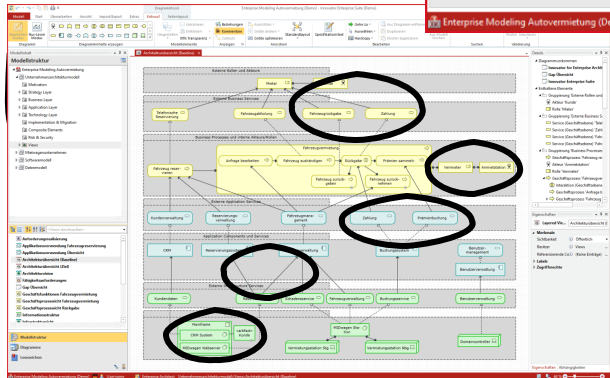
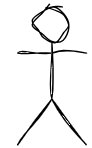


Break the silos with the building block

Data team

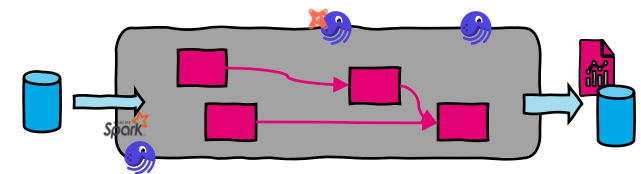


Sales business unit



infrastructure

template project



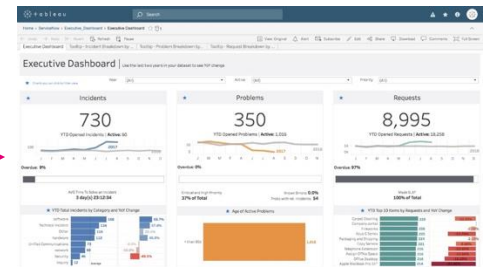
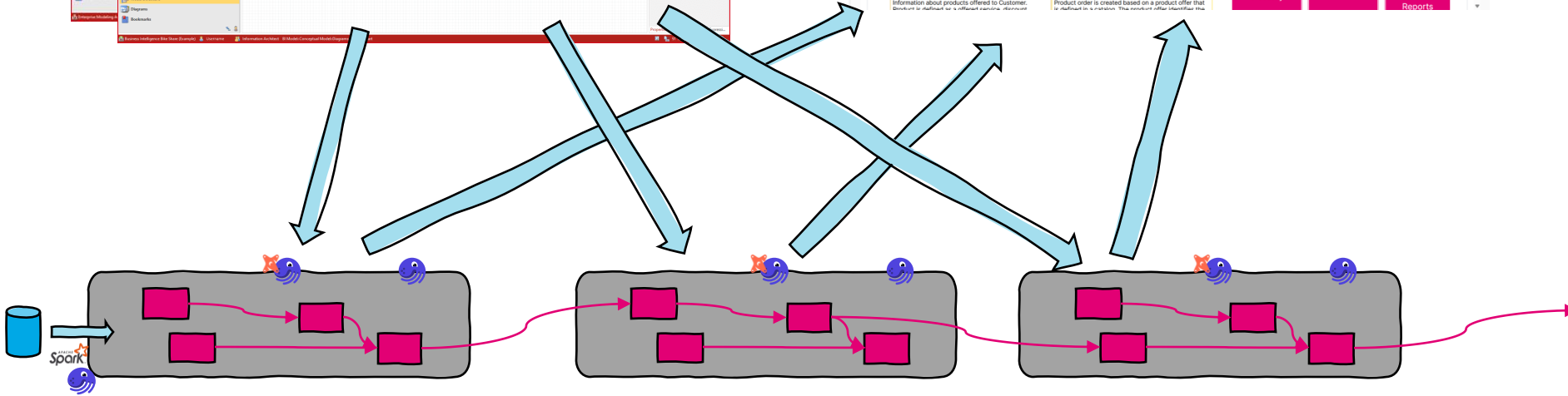
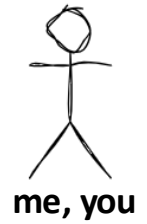
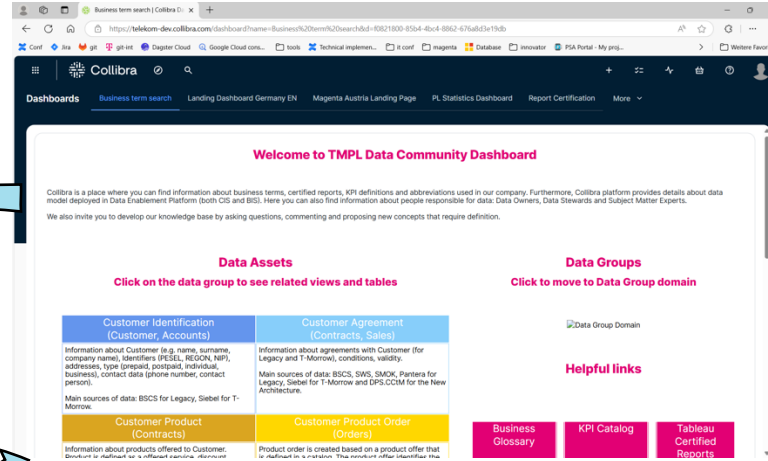
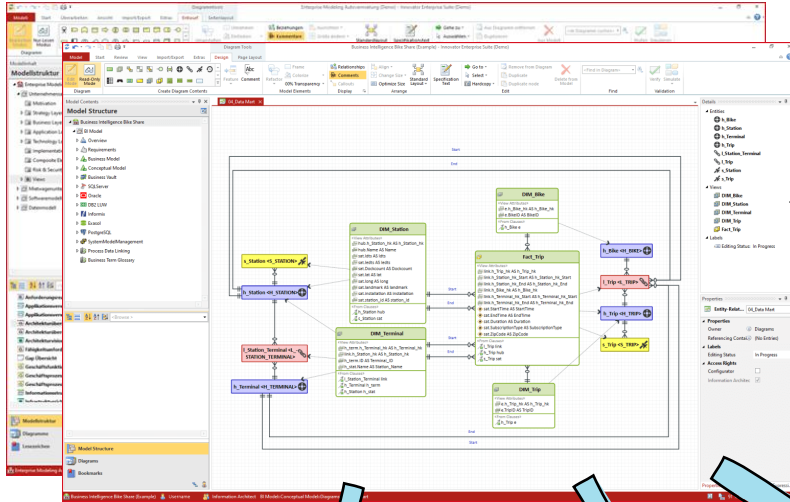
Multi-project setup is a challenge!

1. Dedicated team maintaining infrastructure and template project
2. Governance and modeling tools

Modeling and governance are keys for success

Development phase

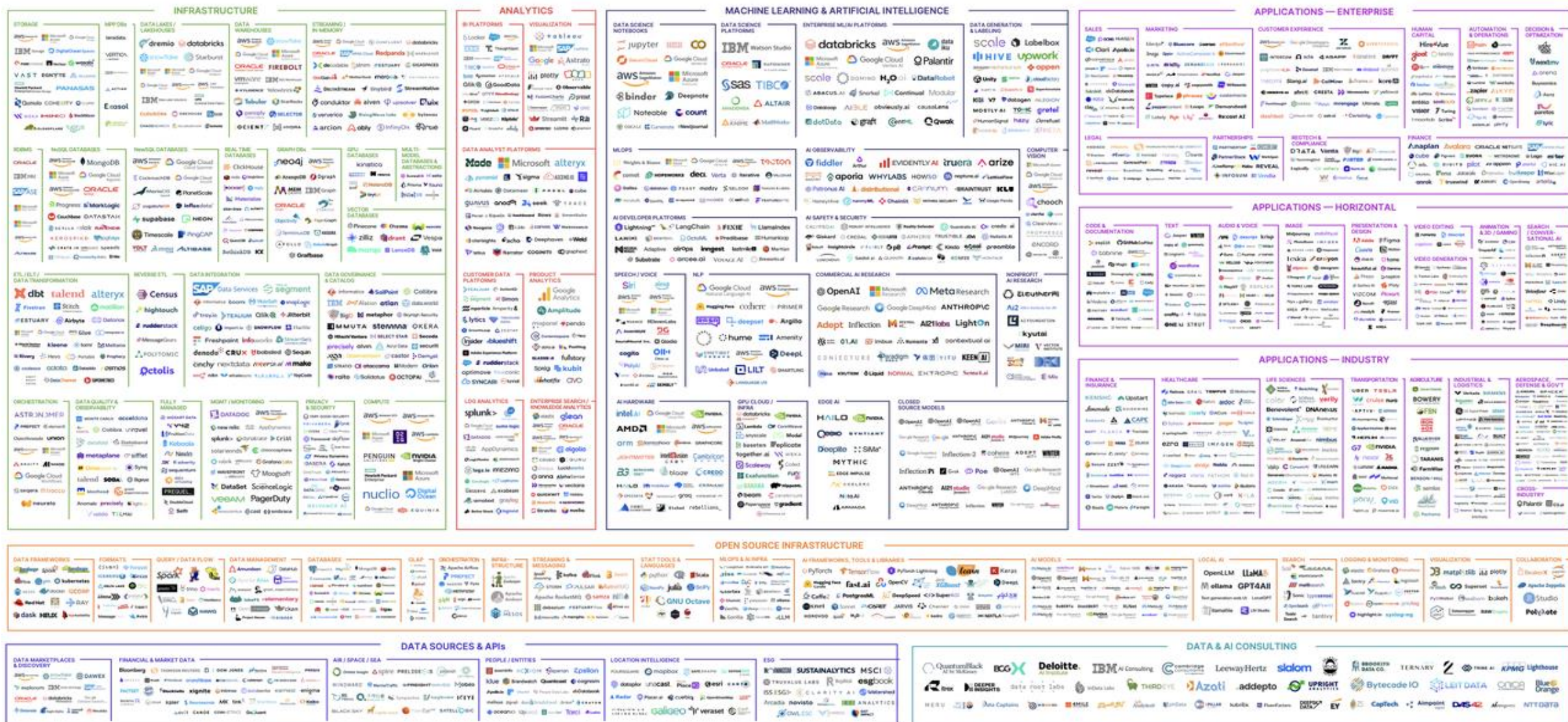
Exploration phase



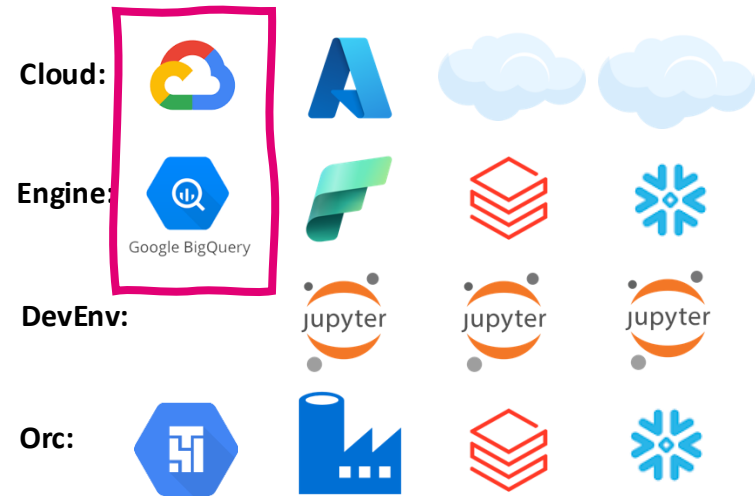
infrastructure

Understanding data platform vendor war

THE 2024 MAD (MACHINE LEARNING, ARTIFICIAL INTELLIGENCE & DATA) LANDSCAPE



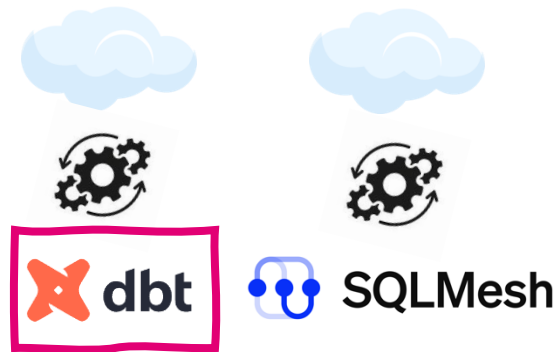
execution engine



- Full stack offered by one vendor
- E2E integration lock in
- Deployment is always a workaround
- No SWE, no local development
- Orchestrator is second class citizen and always task based

- + Frontier in the lakehouse approach
- + Notebooks environment is very convenient
- + Everything on one place

sql transformation framework



- Orchestration is just for DWH/SQL part of the platform

- + Frontier in the SQL development
- + SWE for DWH development

orchestration engine

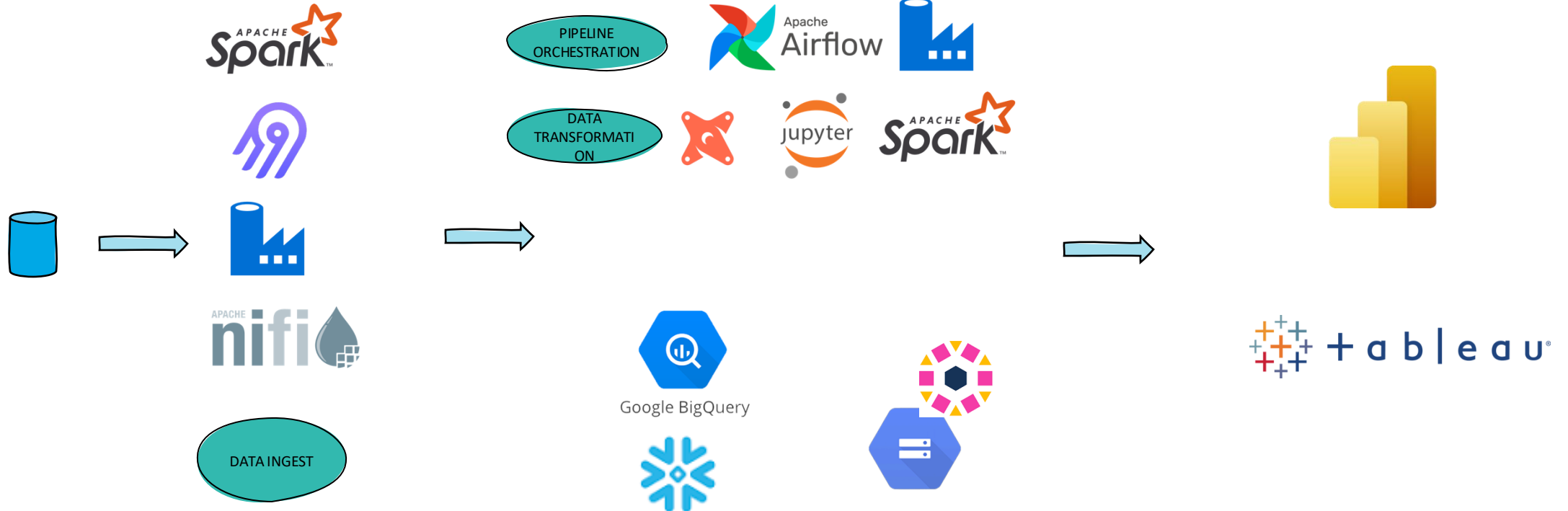
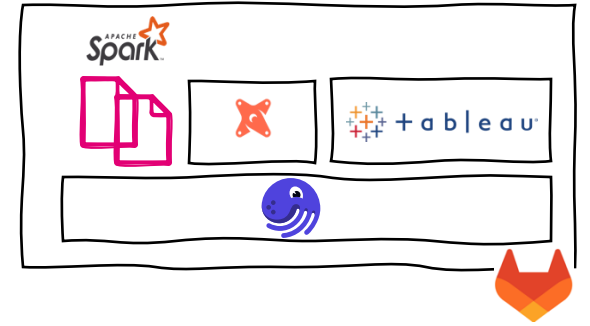
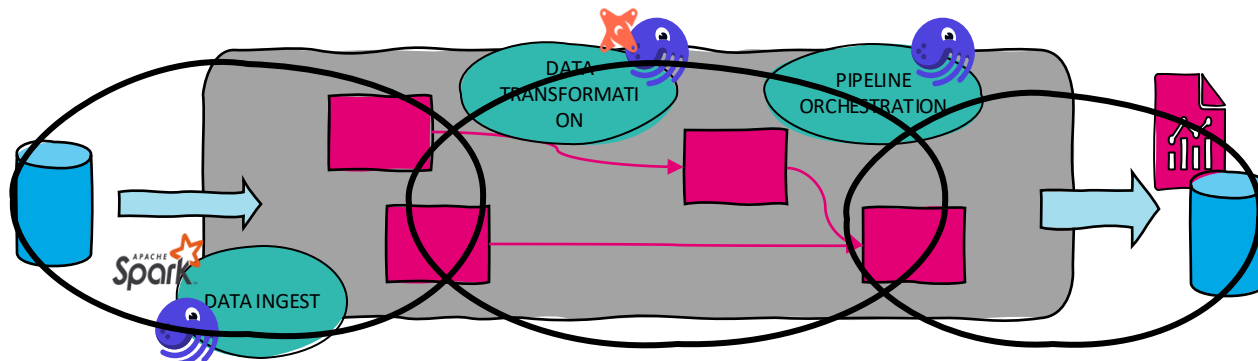
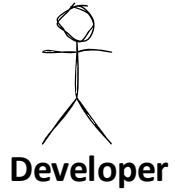


- More technical knowledge needed to setup and use correctly

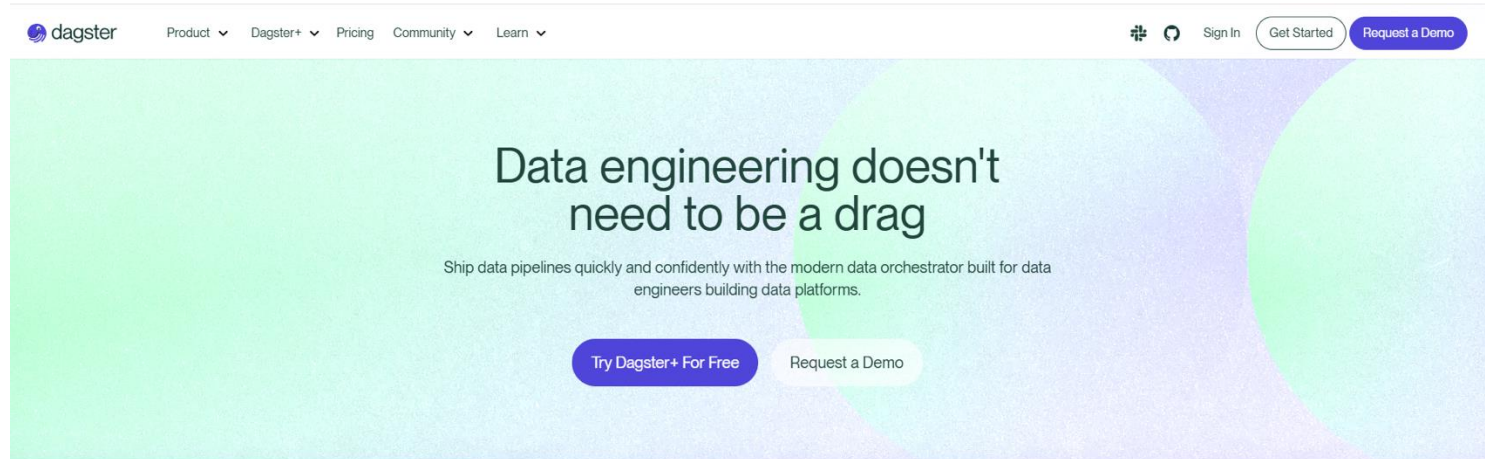
- + Integration is important
- + full SWE and local development

Understanding tool silos

What should i do to get E2E reporting use case done?



Dagster as the core of the platform



Loved by data platform engineers at fast-growing companies

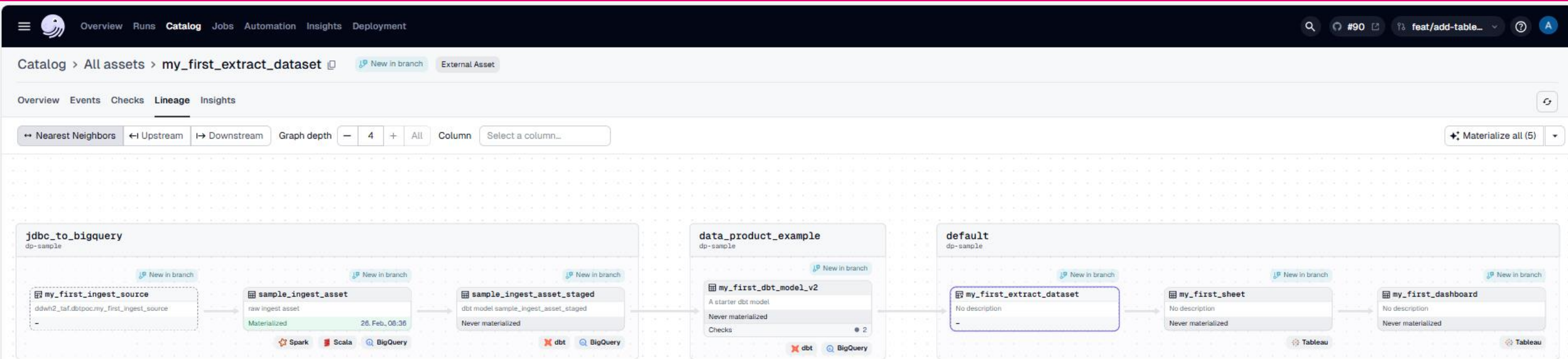
Read our users' success stories



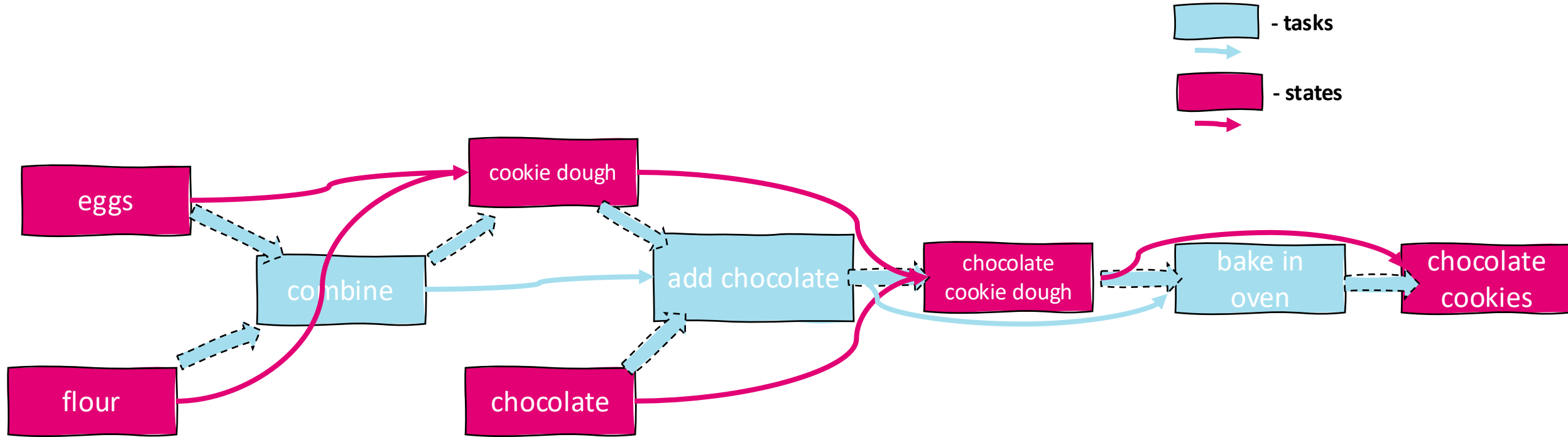
- at Magenta we decided to build around the orchestrator and not around an execution engine
 - hybrid deployment – controlplane SaaS – runtime in our k8s
 - software engineering best practices for project development and deployment
 - asset-based mindset for data flows (graph like a calculator for data dependencies)
- new concepts in orchestration

New enabled concepts

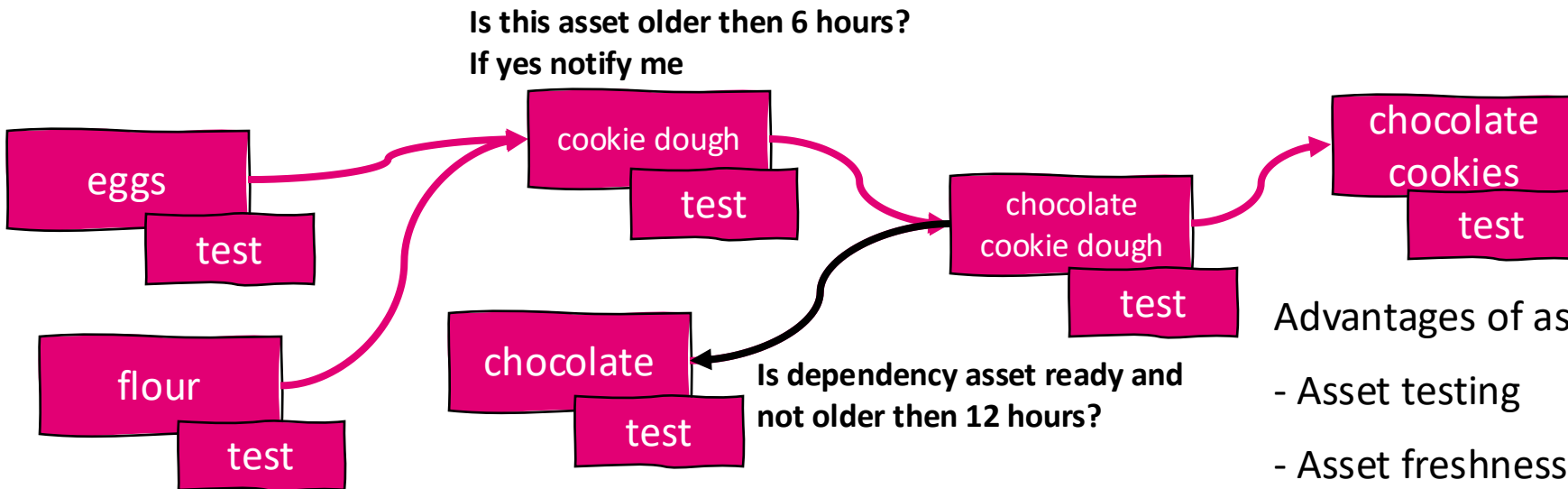
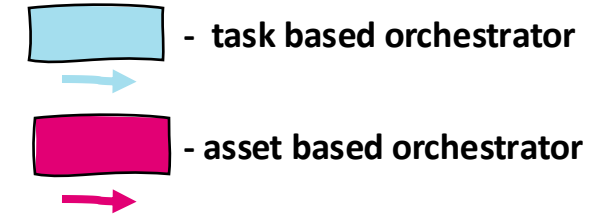
- Asset based graph
- Metadata driven pipeline creation
- Reusable Components
-



Asset and Task based orchestration: Chocolate cookie example



Asset based orchestration

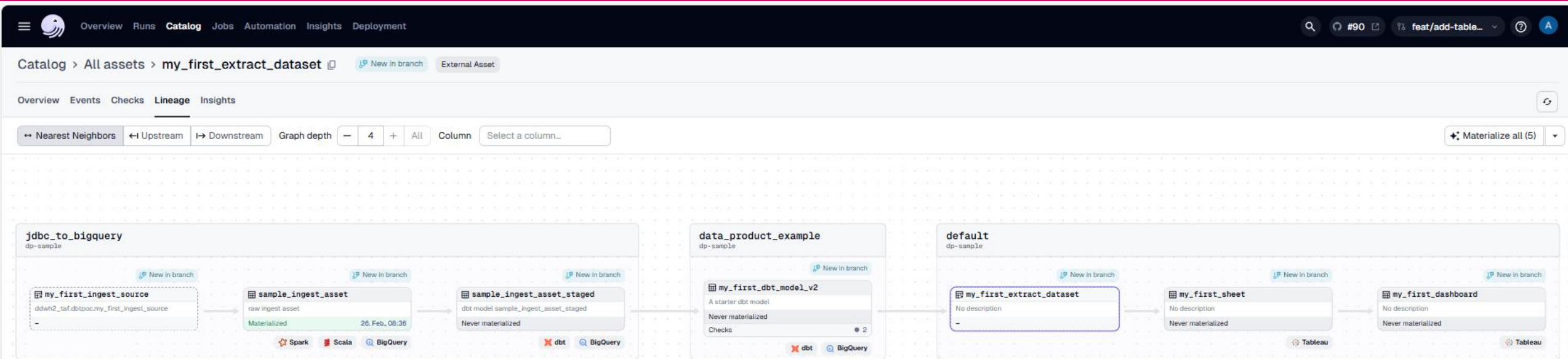


Advantages of asset-based orchestration:

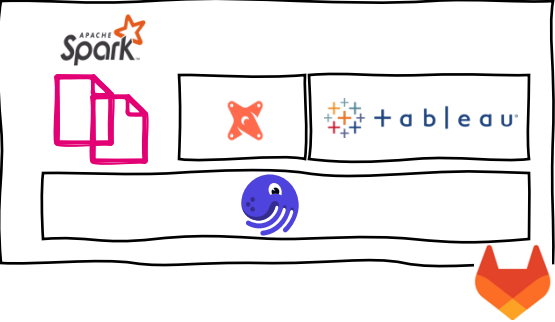
- Asset testing
- Asset freshness
- Asset dependency graph with granular declarative scheduling approach

New enabled concepts

- Asset based graph
- Metadata driven pipeline creation
- Reusable Components
-

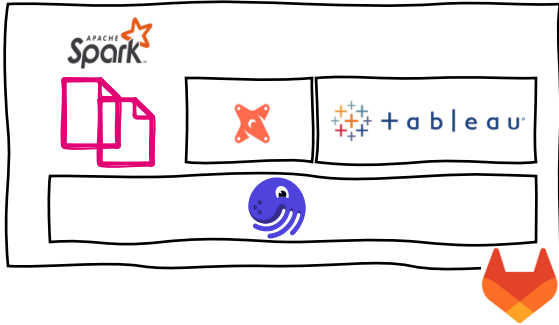


Machine-readable metadata pipeline generation

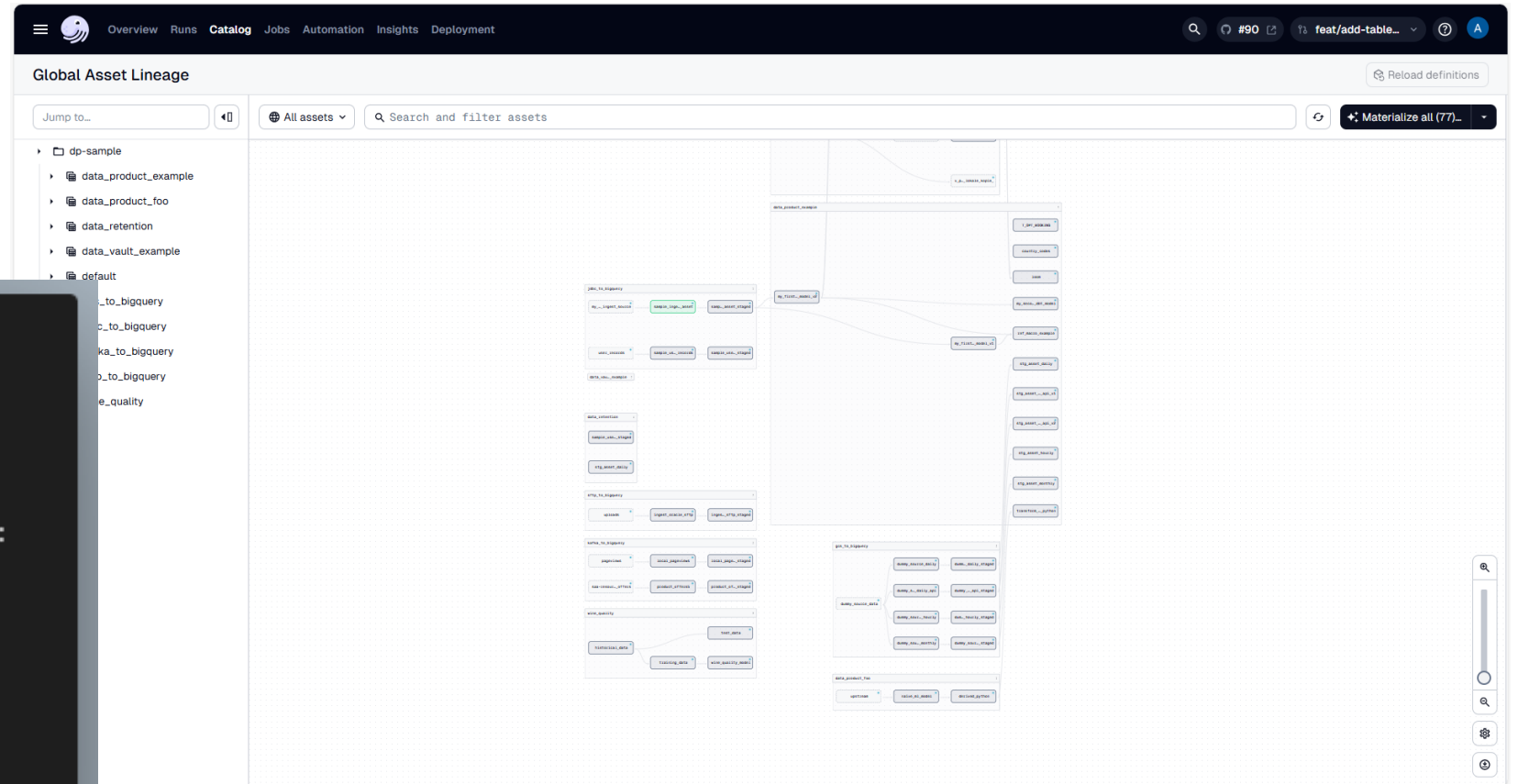


The screenshot shows the Databricks Global Asset Lineage interface. The top navigation bar includes 'Overview', 'Runs', 'Catalog', 'Jobs', 'Automation', 'Insights', and 'Deployment'. The main header is 'Global Asset Lineage' with a search bar and a 'Materialize all (77)' button. The left sidebar lists a tree structure of assets under 'dp-sample', including 'data_product_example', 'data_product_foo', 'data_retention', 'data_vault_example', 'default', 'gcs_to_bigquery', 'jdbc_to_bigquery', 'kafka_to_bigquery', 'sftp_to_bigquery', and 'wine_quality'. The main workspace displays a detailed lineage graph with nodes and arrows, showing the flow of data between various assets. The graph is complex, with many nodes and connections, illustrating the machine-readable metadata pipeline.

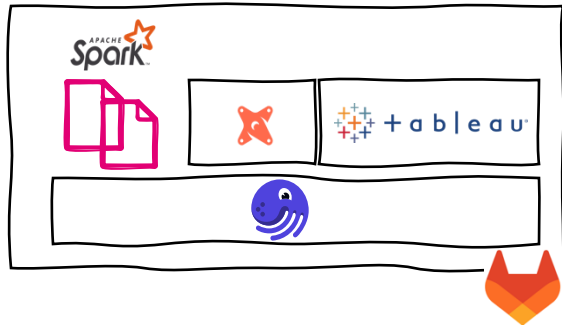
Machine-readable metadata pipeline generation



```
1 metadata = collect_metadata()
2
3 list_of_assets = []
4 for each_metadata_node in metadata:
5     #.. use_metadata
6     @asset
7     def asset():
8         #..use_metadata
9
10    list_of_assets.append(asset)
11
12 Definitions(assets = list_of_assets)
```

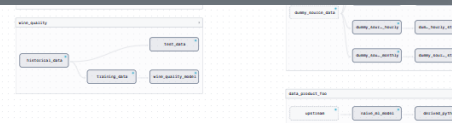


Machine-readable metadata pipeline generation

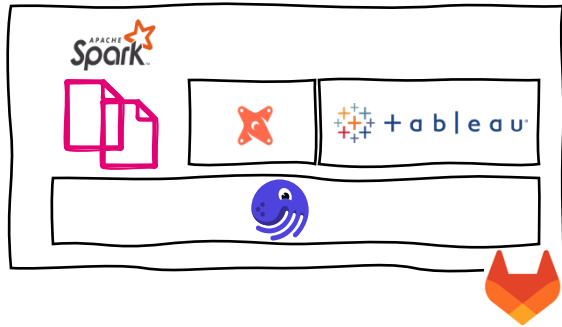


```
1 metadata = collect_metadata()
2
3 list_of_assets = []
4 for each_metadata_node in metadata:
5     #.. use_metadata
6     @asset
7     def asset():
8         #..use_metadata
9
10    list_of_assets.append(asset)
11
12 Definitions(assets = list_of_assets)
```

```
1 configuration_files = read_ingest_configuration_folder(path)
2 list_of_assets = []
3 for each_config_file in configuration_files:
4     config = parse_config(each_config_file)
5
6     @asset(
7         name = config.name
8     )
9     def ingest_asset():
10        df = spark.read(config.source)
11        df.write(config.source)
12        list_of_assets.append(ingest_asset)
13
14 Definitions(assets = list_of_assets)
```



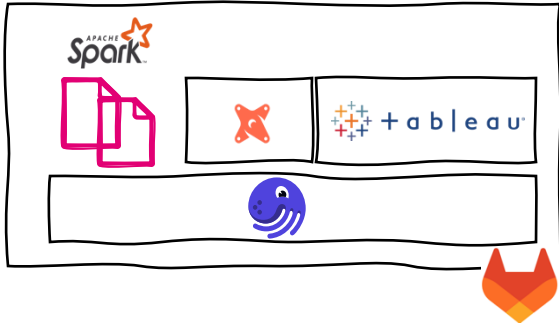
Machine-readable metadata pipeline generation



```
1 metadata = collect_metadata()
2
3 list_of_assets = []
4 for each_metadata_node in metadata:
5     #.. use_metadata
6     @asset
7     def asset():
8         #..use_metadata
9
10    list_of_assets.append(asset)
11
12 Definitions(assets = list_of_assets)
```

```
1 manifest = read_dbt_manifest(path)
2 list_of_assets = []
3 for each_node in manifest:
4     model_name = each_node.name
5     model_deps = each_node.deps
6     @asset(
7         name = model_name,
8         deps = model_deps
9     )
10    def run_dbt_asset(dbt: DbtClient)
11        dbt.run(f"--select {model_name}")
12
13    list_of_assets.append(run_dbt_asset)
14
15 Definitions(assets = list_of_assets)
```

Machine-readable metadata pipeline generation



```
1 metadata = collect_metadata()
2
3 list_of_assets = []
4 for each_metadata_node in metadata:
5     #.. use_metadata
6     @asset
7     def asset():
8         #..use_metadata
9
10    list_of_assets.append(asset)
11
12 Definitions(assets = list_of_assets)
```

```
1 conf
2 list
3 for
4
5
6
7
8
9
10
11
12
13
14 Defi
```

```
1 m
2 l
3 f
4
5
6
7
8
9
10
11
12
13
14
15 D
```

```
1 tableau_server = TableauServer(creds)
2 #send rest api call to tableau server and get information
3 tableau_metadata = tableau_server.get_metadata()
4 list_of_assets = []
5 for each_tableau_object in tableau_metadata:
6     tableau_object_deps = each_tableau_object.deps
7     tableau_object_name = each_tableau_object.deps
8     if each_tableau_object.type == extract_datasource:
9         @asset(
10             name = tableau_object_name,
11             deps = tableau_object_deps
12         )
13         def refresh_extract(tableau_server):
14             #send api call to refresh object
15             tableau_server.refresh_extract(tableau_object_name)
16             list_of_assets.append(run_dbt_asset)
17     else:
18         @asset(name = tableau_object_deps)
19         def asset():
20             pass
21             list_of_assets.append(asset)
22
23 Definitions(assets = list_of_assets)
```

Reusable components

- Define once, test & reuse
- Resources → Encapsulate complex logic to interact with external systems
- IO manager → Make complex IO interactions substitutable & testable
- Benefits
 - Dependency injection
 - Day 1 productivity: Scale the data pipeline down to a single laptop
 - Increase self-service: Business/DS focus not required to handle complex IO

```
2 @asset(  
3     io_manager_key="bigquery_io_manager",  
4 )  
5 def awesome_ml_model(context, reference_addresses: pd.DataFrame, bigquery: BigQueryResource) -> pd.DataFrame:  
6     # simple normal python code here  
7     # IO is abstracted  
8     context.log.info(f"from source: \n{reference_addresses.head()}")  
9     # auth & complexity (imagine web API) is abstracted  
0     with bigquery.get_client() as client:  
1         job = client.query("select * from example.upstream")  
2         query_result = job.result().to_dataframe()  
3         context.log.info(f"direct query: \n{query_result.head()}")  
4     return pd.DataFrame({"foo": [1,2,3]})
```

Takeaways

- Integrated asset-based graph is key (from ingest, transformation, reporting, tests – to AI)
 - Event driven connection
 - Better collaboration (scaling)
- Software engineering principles enable business self service
 - Blueprint
 - Automate all the things: CI/CD (stateful & stateless)
 - DRY: build tested foundation – dependency injection
 - Make business departments part of the key processes and pipelines
- Executable specification (metadata, contracts)
 - Interface Mangement
 - Preserve semantics
 - Preserve compliance (security classification, PII, retention)

Last things

The screenshot shows a GitHub repository for 'local-data-stack' with a pull request titled '[dagster-tableau] Exploring embedded data sources #27218'. The pull request is merged and was created 2 weeks ago. The repository has 1 branch and 0 tags. The pull request summary includes a 'Summary & Motivation' section with two points: 1. Current implementation was fetching limited metadata from tableau which was only limited to id and names, but have added few more fields like upstreamTables and databases details and many more fields. 2. Earlier we were only showing published data sources and ignoring embedded data sources. With this changes we are showing embedded data sources in case published data sources are not present. The 'How I Tested These Changes' section states 'Tested on local system with the help of docker desktop'. The pull request has 1 thumbs up and 3 commits added last month. The repository files list includes doc, img, {{ cookiecutter.project_slug }}, .gitignore, README.md, cookiecutter.json, pixi.lock, pyproject.toml, and yamllintconfig.yaml. The pull request is reviewed by maximearmstrong and has the label 'integration: tableau'.

Data platform is team work and we are very proud and excited about the journey ahead

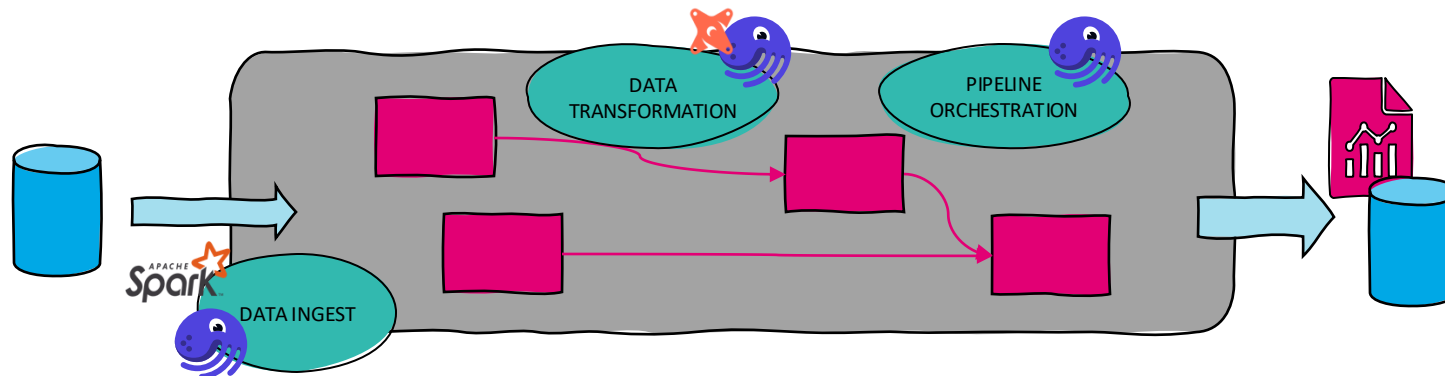
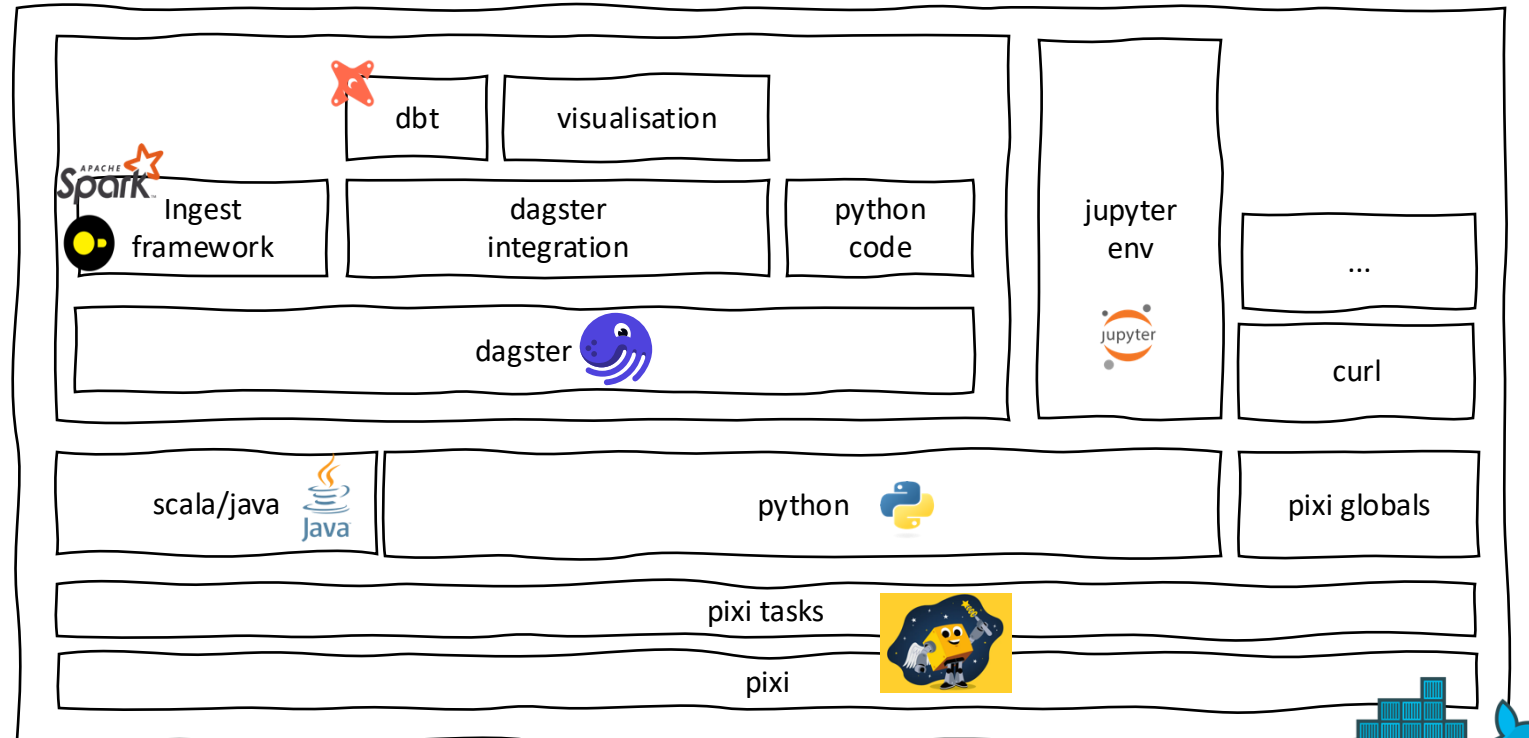
Scaling data pipelines @Telekom

Aleks & Georg



Building block

- Dagster is the core
- Need for java because of custom spark code
- Development tooling for testing and code quality
- Pixi is holding everything together as the environment manager



What should I do now?

