

# Himalayan Peaks of Testing Data Pipelines

Ksenia Tomak, Dodo Engineering  
Pasha Finkelshteyn, JetBrains



# Ksenia Tomak

- Tech Lead, Dodo Engineering
- @if\_no\_then\_yes

Industrial IoT, DE, Storages

# Pasha Finkelshteyn

Developer  for Big Data @ JetBrains

@asm0di0

# What is Big Data

- Doesn't fit the single node (or Excel)
- Maybe scaled when growing
- Enough data to make reliable business solutions

# Who are DEs

Plumber of data

Data is produced by

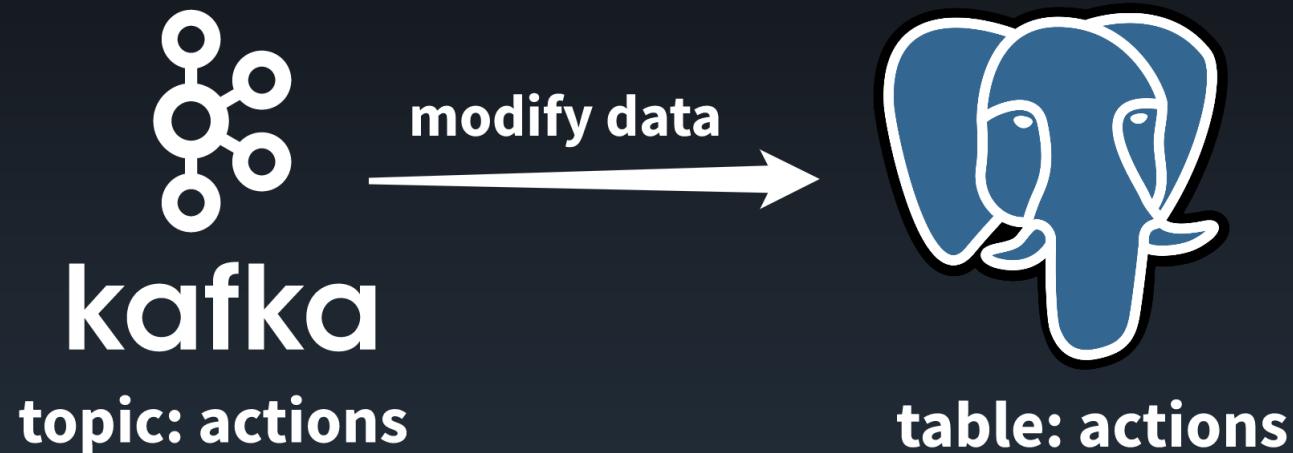
- sensors
- clickstreams
- etc.

# Data sinks

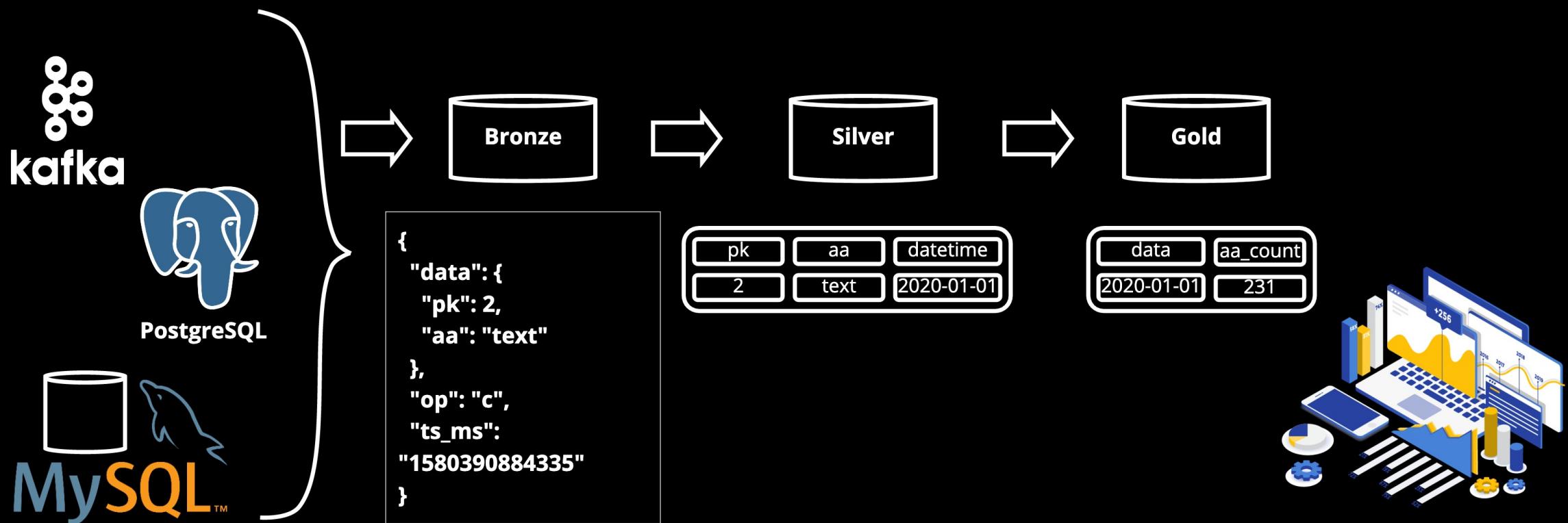
- HDFS
- S3, Azure Blob Storage
- etc.

# What is a data pipeline?

# Data processing



# Data lake?



# Who needs pipelines

- Data Scientists
- Data Analytics
- Marketing
- PO

QA ? = QC

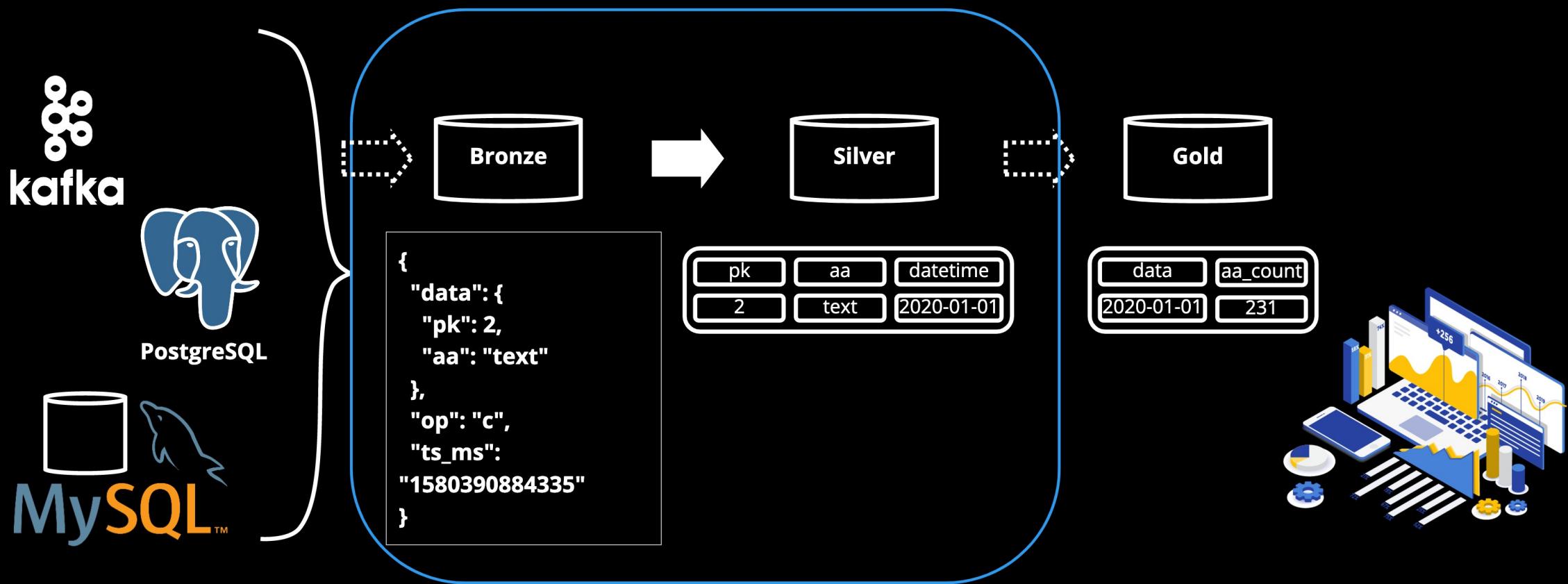
QA ≠ QC

QA is about processes and not only about software quality.

# Pyramid of testing. Unit



# Bronze → Silver pipeline



# Typical pipeline



# Typical pipeline

```
1 from pyspark.sql import DataFrame, functions as F
2 from pyspark.sql.types import *
3
4 data_type = StructType([
5     StructField("pk", LongType(), False),
6     StructField("aa", StringType(), False)
7 ])
8
9 transformed_data = (
10    sample_df
11    .withColumn("json_object", F.get_json_object(F.col("json_value"), "$.data"))
12    .withColumn("parsed_struct", F.from_json(F.col("json_object"), data_type))
13    .selectExpr("parsed_struct.*")
14 )
```

# Typical pipeline

```
1 from pyspark.sql import DataFrame, functions as F
2 from pyspark.sql.types import *
3
4 data_type = StructType([
5     StructField("pk", LongType(), False),
6     StructField("aa", StringType(), False)
7 ])
8
9 transformed_data = (
10    sample_df
11    .withColumn("json_object", F.get_json_object(F.col("json_value"), "$.data"))
12    .withColumn("parsed_struct", F.from_json(F.col("json_object"), data_type))
13    .selectExpr("parsed_struct.*")
14 )
```

# Typical pipeline

```
1 from pyspark.sql import DataFrame, functions as F
2 from pyspark.sql.types import *
3
4 data_type = StructType([
5     StructField("pk", LongType(), False),
6     StructField("aa", StringType(), False)
7 ])
8
9 transformed_data = (
10    sample_df
11    .withColumn("json_object", F.get_json_object(F.col("json_value"), "$.data"))
12    .withColumn("parsed_struct", F.from_json(F.col("json_object"), data_type))
13    .selectExpr("parsed_struct.*")
14 )
```

# Typical pipeline

```
1 from pyspark.sql import DataFrame, functions as F
2 from pyspark.sql.types import *
3
4 data_type = StructType([
5     StructField("pk", LongType(), False),
6     StructField("aa", StringType(), False)
7 ])
8
9 transformed_data = (
10    sample_df
11    .withColumn("json_object", F.get_json_object(F.col("json_value"), "$.data"))
12    .withColumn("parsed_struct", F.from_json(F.col("json_object"), data_type))
13    .selectExpr("parsed_struct.*")
14 )
```

# Typical pipeline

```
1 from pyspark.sql import DataFrame, functions as F
2 from pyspark.sql.types import *
3
4 data_type = StructType([
5     StructField("pk", LongType(), False),
6     StructField("aa", StringType(), False)
7 ])
8
9 transformed_data = (
10    sample_df
11    .withColumn("json_object", F.get_json_object(F.col("json_value"), "$.data"))
12    .withColumn("parsed_struct", F.from_json(F.col("json_object"), data_type))
13    .selectExpr("parsed_struct.*")
14 )
```

# Unit testing of pipeline

What may we test here?

A pipeline should transform data correctly!

*Correctness is a business term*

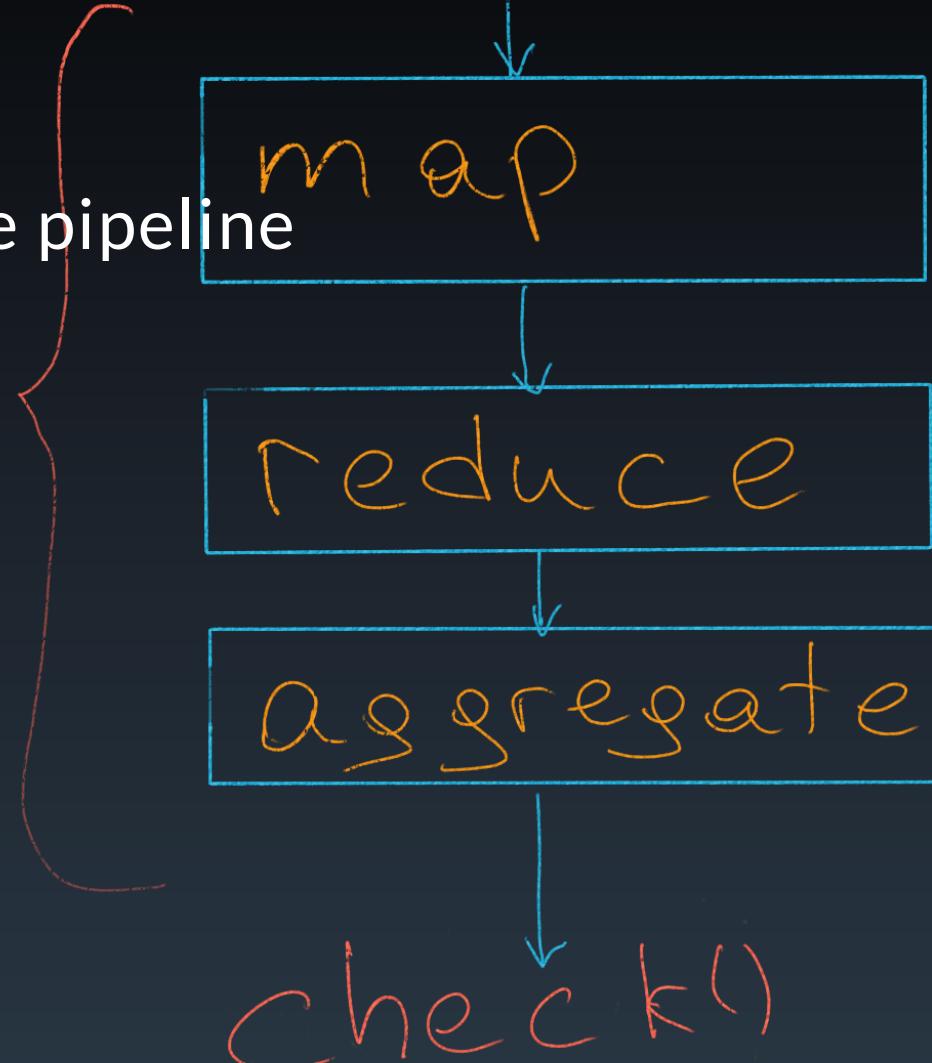
fake data

# Let's paste fakes!

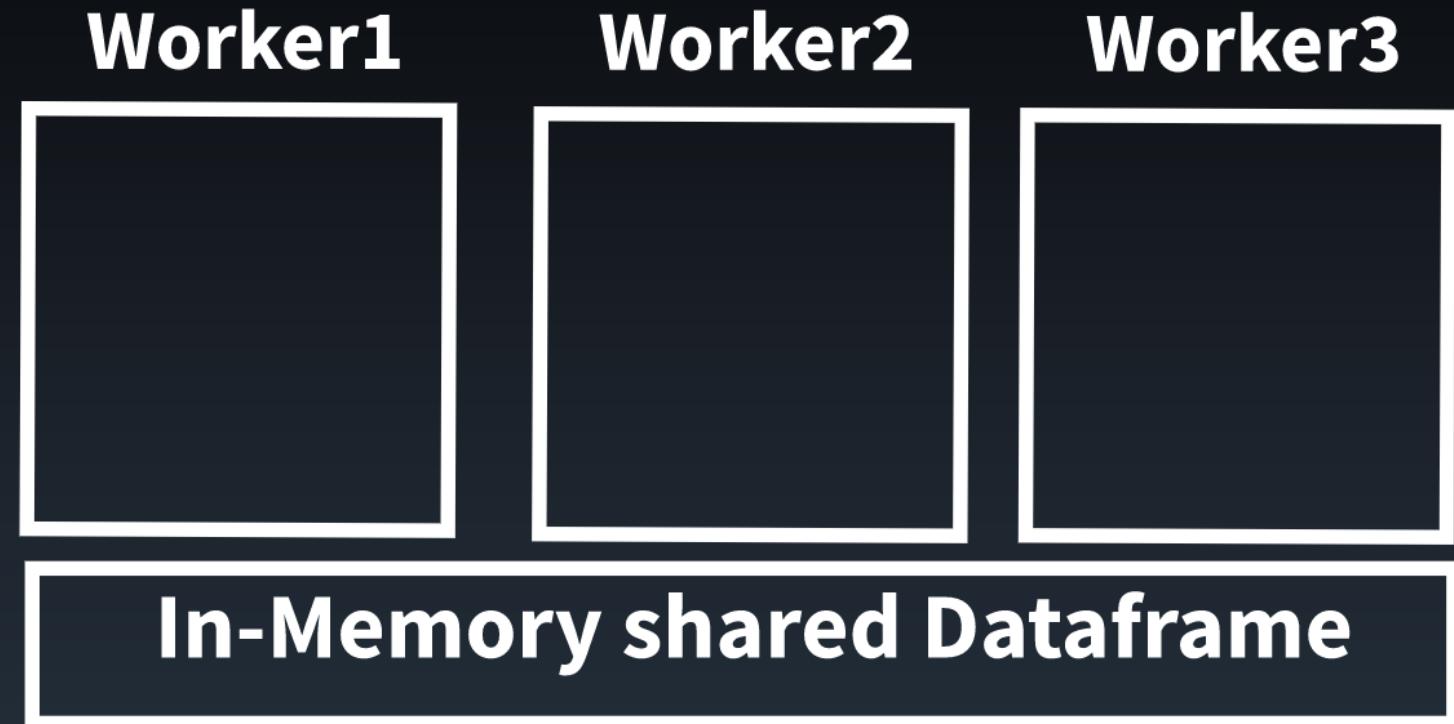
Fake input data

Reference data at the end of the pipeline

Separate  
Function



# Dataframes?

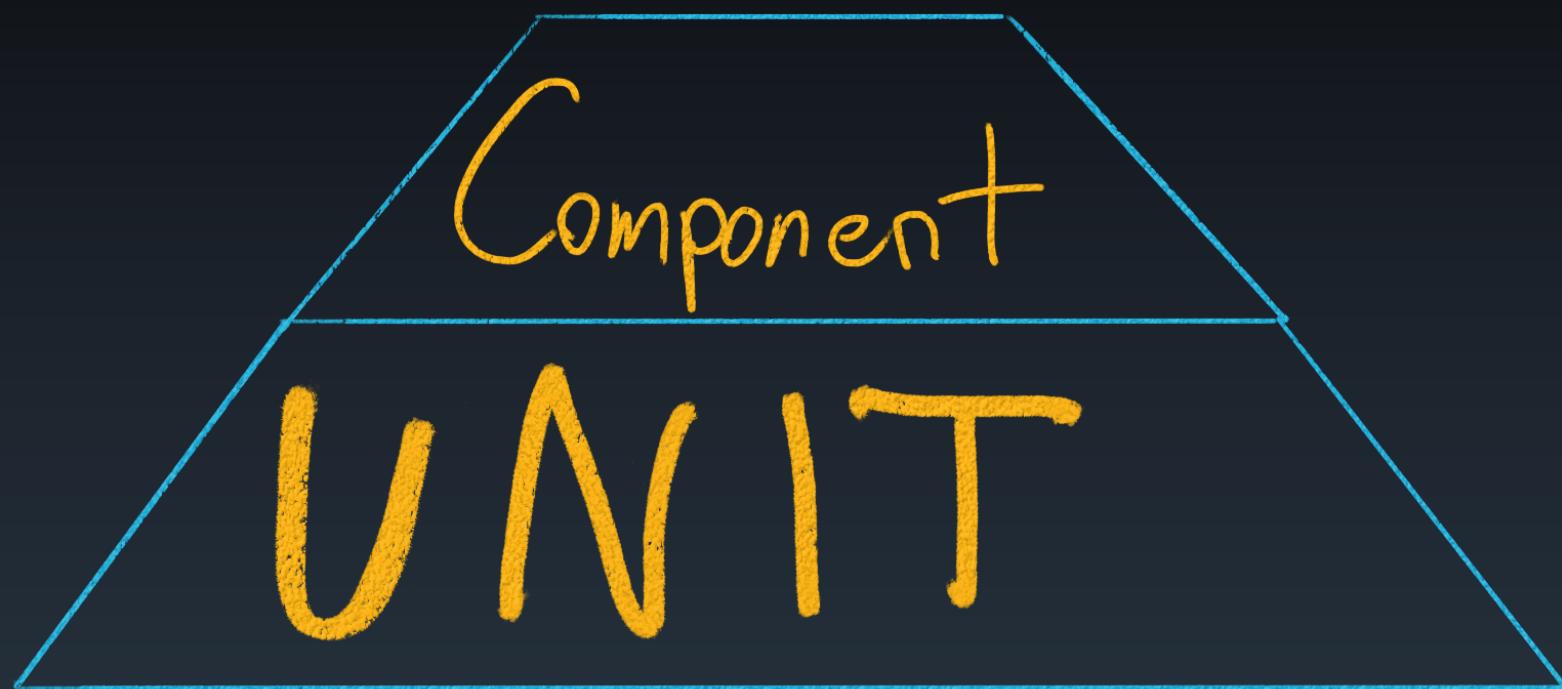


# Tools

[holdenk/spark-testing-base](#) ← Tools to run tests

[MrPowers/spark-daria](#) ← tools to easily create test data

# Component testing

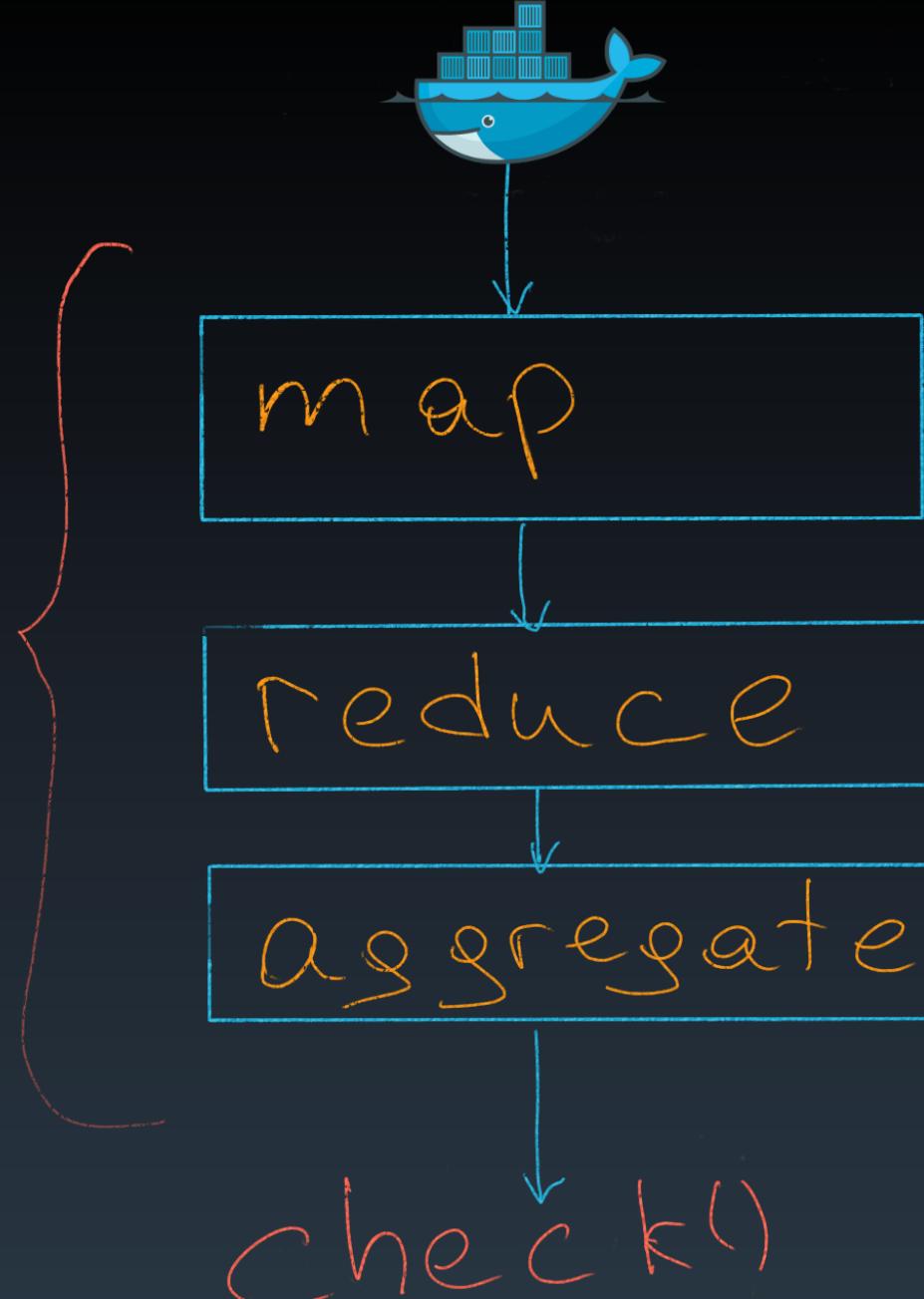




TEST CONTAINERS

# TestContainers

Separate  
Function



# TestContainers

Supported languages:

- Java (and compatibles: Scala, Kotlin, etc.)
- Python
- Go
- Node.js
- Rust
- .NET

# Test Containers

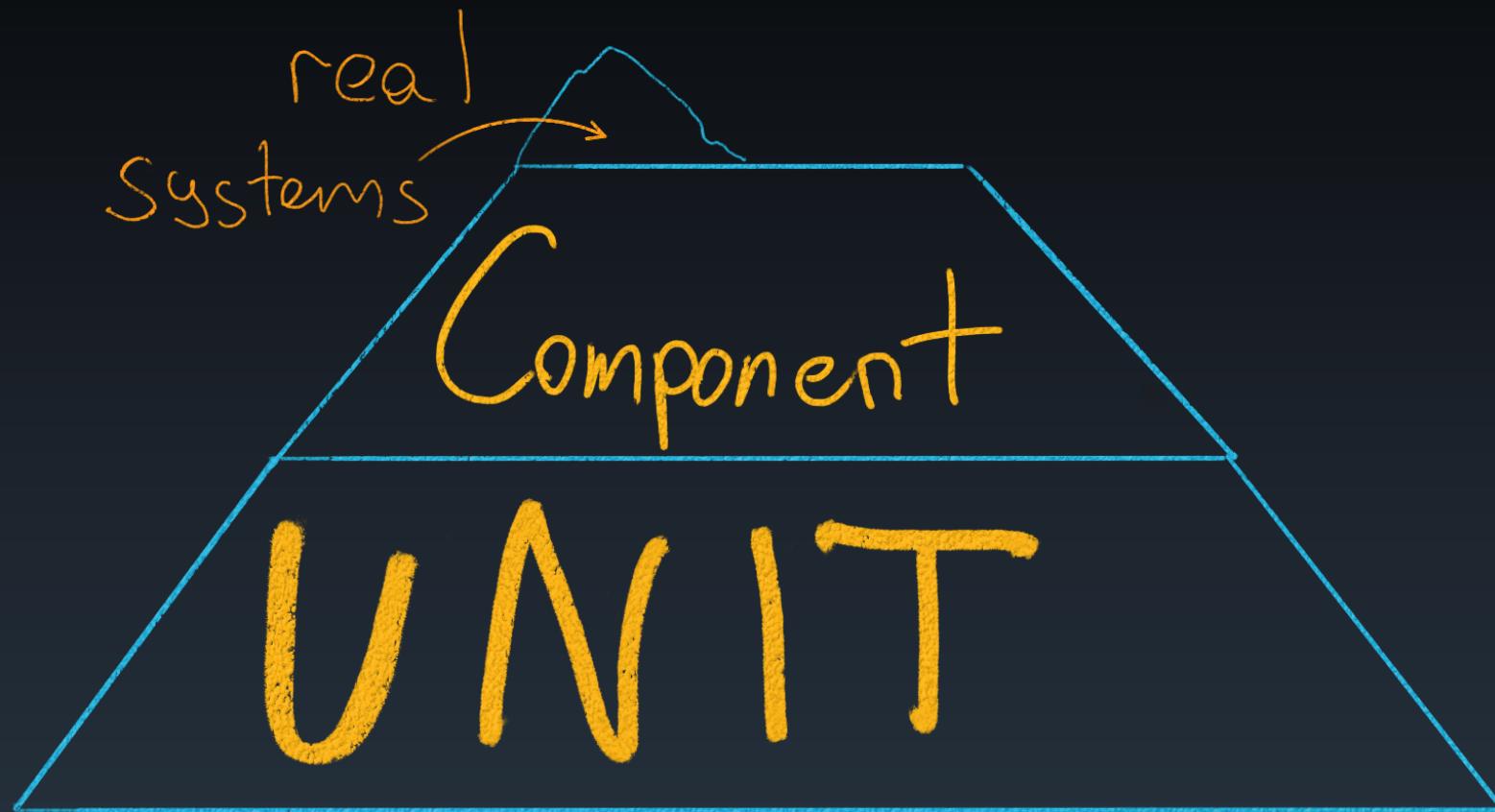
```
1 import sqlalchemy
2 from testcontainers.mysql import MySqlContainer
3
4 with MySqlContainer('mysql:5.7.17') as mysql:
5     engine = sqlalchemy.create_engine(mysql.get_connection_url())
6     version, = engine.execute("select version()").fetchone()
7     print(version) # 5.7.17
```

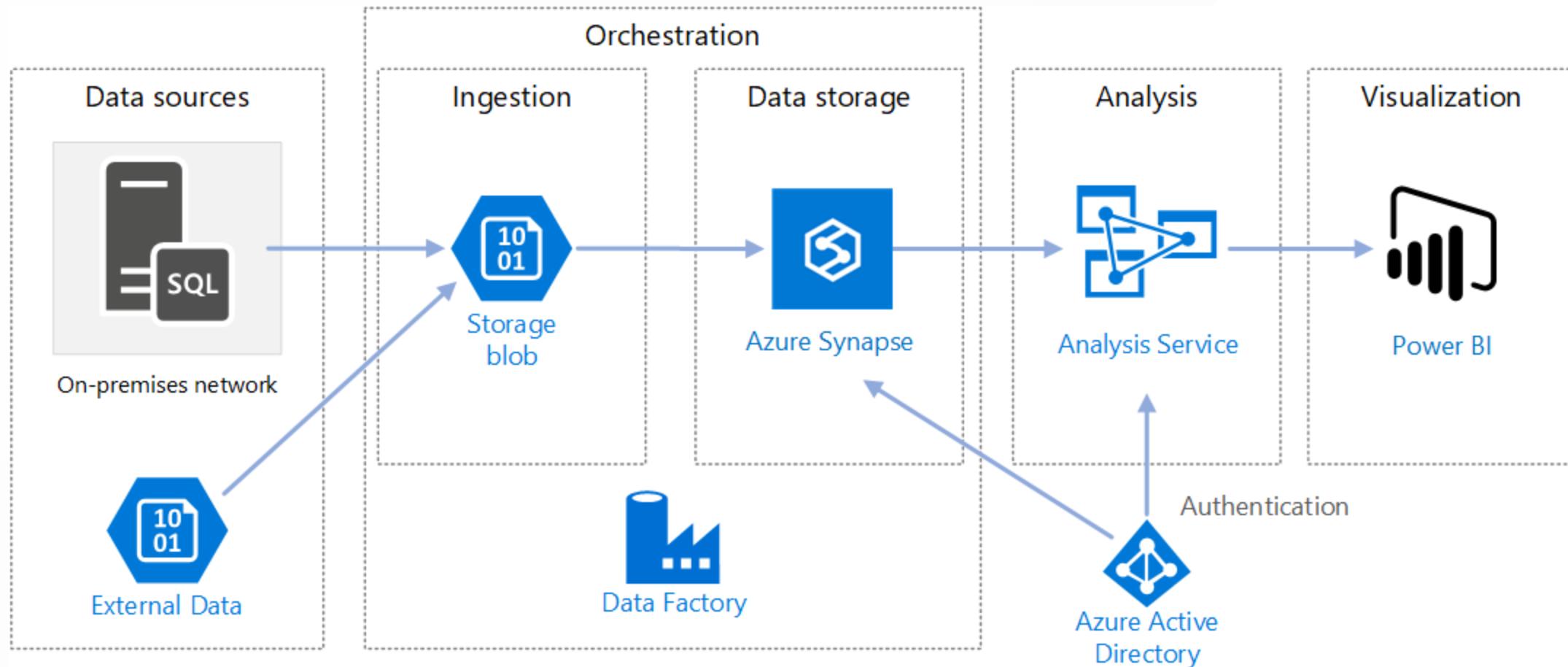
# Test Containers

```
1 import sqlalchemy
2 from testcontainers.mysql import MySqlContainer
3
4 with MySqlContainer('mysql:5.7.17') as mysql:
5     engine = sqlalchemy.create_engine(mysql.get_connection_url())
6     version, = engine.execute("select version()").fetchone()
7     print(version) # 5.7.17
```

# Test Containers

```
1 import sqlalchemy
2 from testcontainers.mysql import MySqlContainer
3
4 with MySqlContainer('mysql:5.7.17') as mysql:
5     engine = sqlalchemy.create_engine(mysql.get_connection_url())
6     version, = engine.execute("select version()").fetchone()
7     print(version) # 5.7.17
```

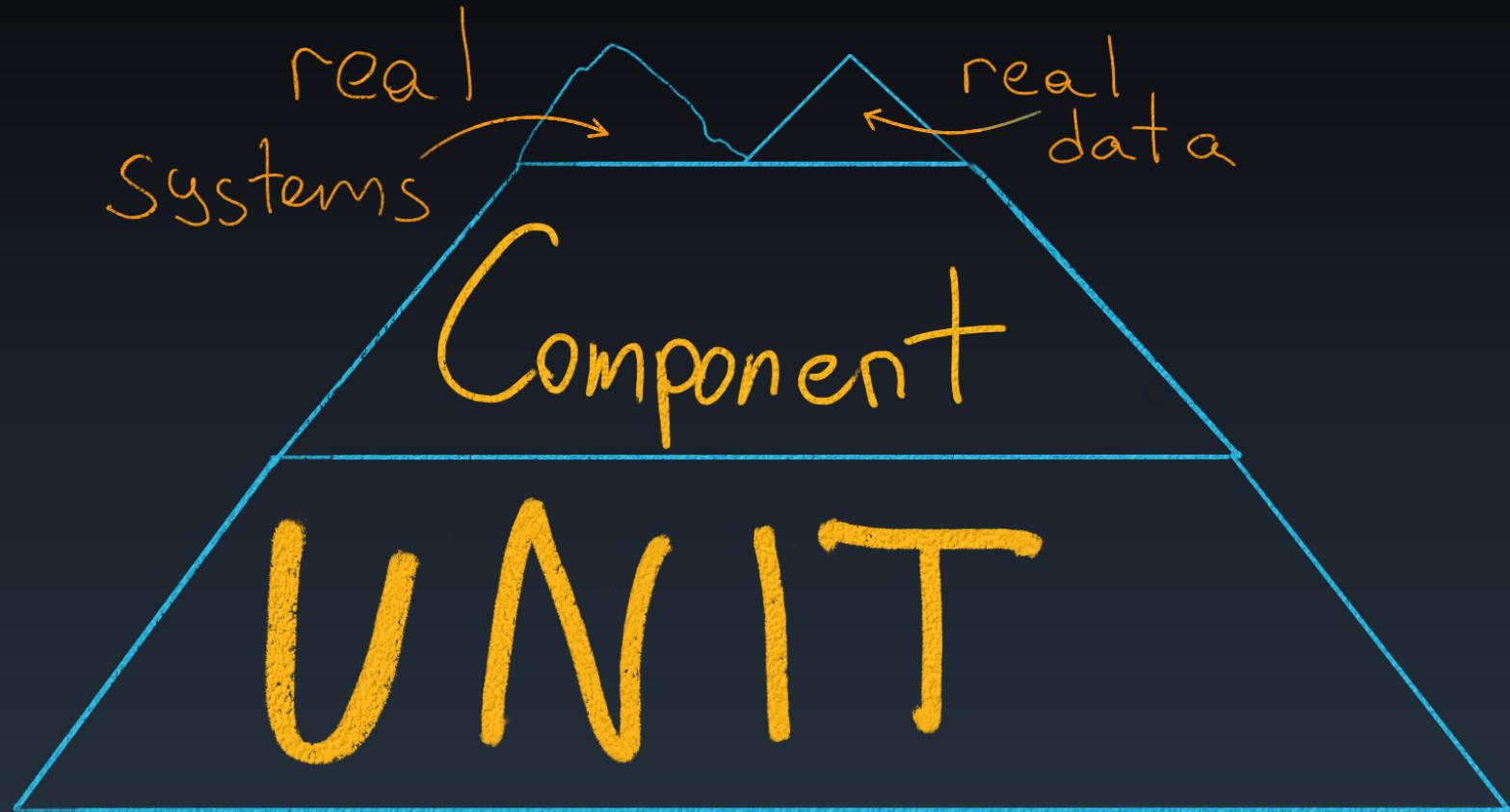




# Real systems

Why are component tests not enough?

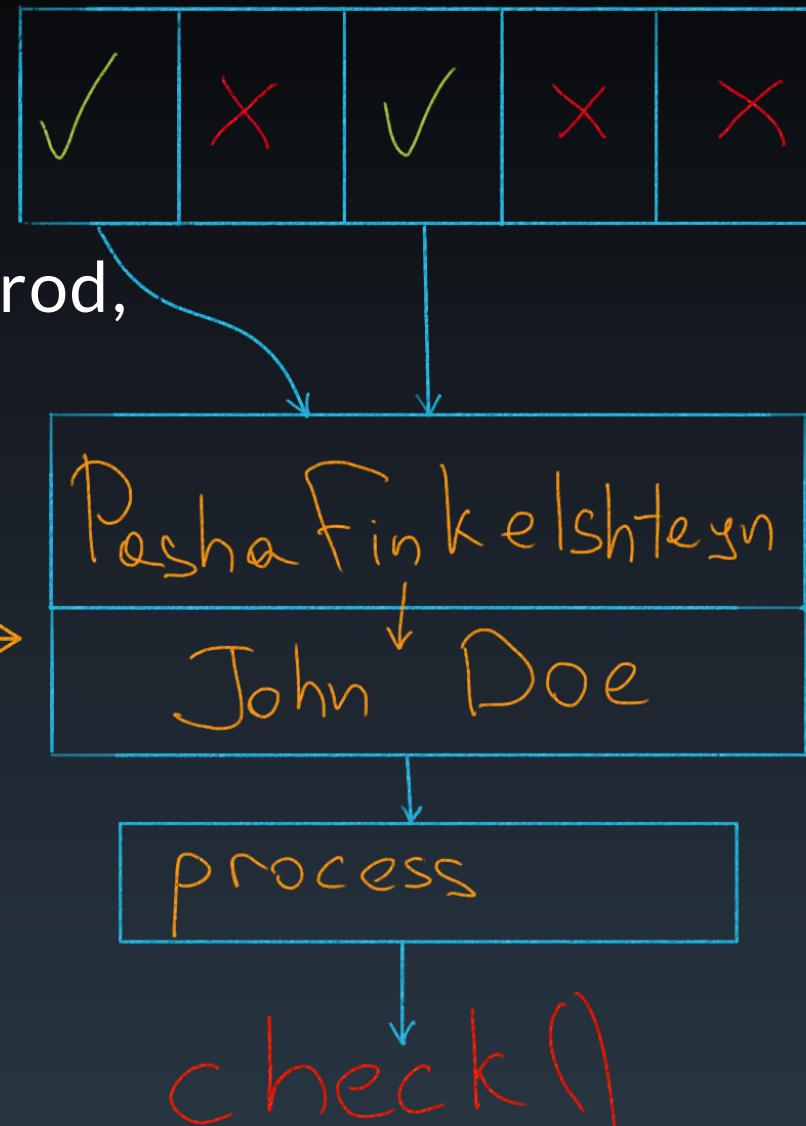
- vendor lock tools (DB, processing, etc.)
- external error handling



# Real data

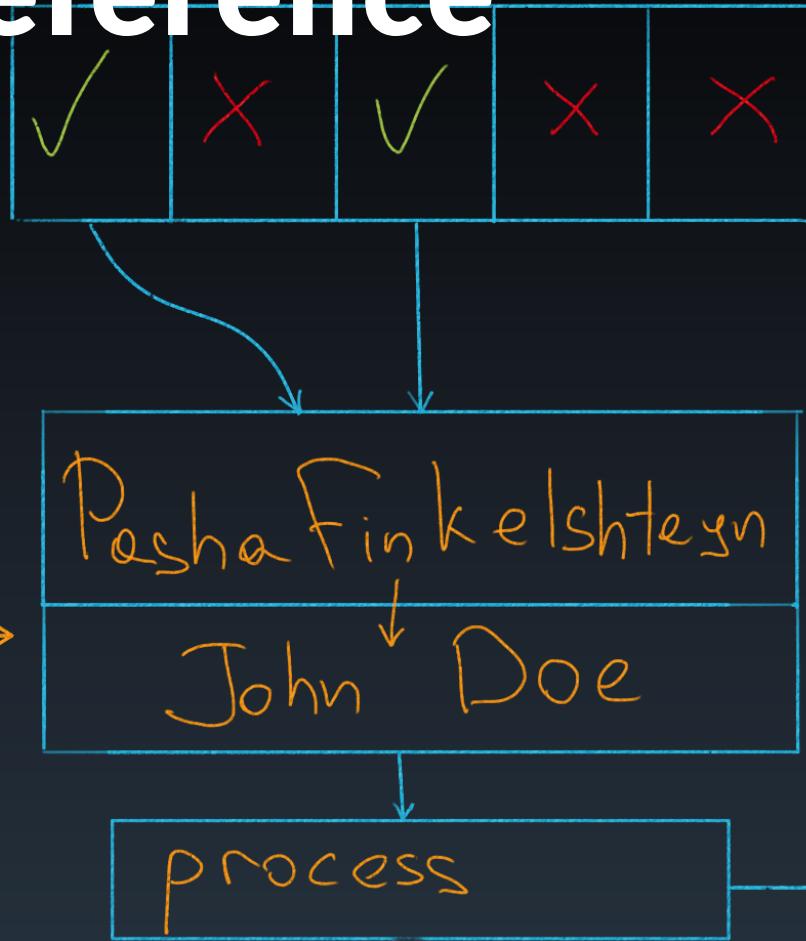
Get data samples from prod,  
anonymize it

Anonymize →



# Compare to reference

Anonymize →



reference  
result

# Real data

Deploy full data backup on stage env,  
anonymize it 😷

**In usual testing you won't trust your code**

**In pipeline testing you won't trust  
both your code and your data**

# Real data expectations

Test:

- no data
- valid data
- invalid data
- illegal data format

# Real data expectations. Tools:

- [great expectations](#),
- [Deequ](#)

```
1 from pyspark.sql.types import Row, StructType  
2 from datetime import datetime  
3  
4 schema = {  
5     "type": "struct",  
6     "fields": [  
7         {"name": "Id", "type": "long", "nullable": False, "metadata": {}},  
8         {"name": "SaleDate", "type": "timestamp", "nullable": False, "metadata": {}},  
9         {"name": "Country", "type": "string", "nullable": False, "metadata": {}},]  
10    }  
11 table_rows = [  
12     Row(1, datetime(2021, 1, 1, 10, 0, 0), "RU"),  
13     Row(2, datetime(1000, 1, 1, 10, 0, 0), "KZ"),  
14     Row(2, datetime(2018, 1, 1, 10, 0, 0), "AU"),  
15     Row(2, datetime(2019, 1, 1, 10, 0, 0), "")],  
16  
17 sample_df = spark.createDataFrame(table_rows, StructType.fromJson(schema))
```

```
1 from pyspark.sql.types import Row, StructType  
2 from datetime import datetime  
3  
4 schema = {  
5     "type": "struct",  
6     "fields": [  
7         {"name": "Id", "type": "long", "nullable": False, "metadata": {}},  
8         {"name": "SaleDate", "type": "timestamp", "nullable": False, "metadata": {}},  
9         {"name": "Country", "type": "string", "nullable": False, "metadata": {}},]  
10    }  
11 table_rows = [  
12     Row(1, datetime(2021, 1, 1, 10, 0, 0), "RU"),  
13     Row(2, datetime(1000, 1, 1, 10, 0, 0), "KZ"),  
14     Row(2, datetime(2018, 1, 1, 10, 0, 0), "AU"),  
15     Row(2, datetime(2019, 1, 1, 10, 0, 0), "")],  
16  
17 sample_df = spark.createDataFrame(table_rows, StructType.fromJson(schema))
```

```
1 from pyspark.sql.types import Row, StructType
2 from datetime import datetime
3
4 schema = {
5     "type": "struct",
6     "fields": [
7         {"name": "Id", "type": "long", "nullable": False, "metadata": {}},
8         {"name": "SaleDate", "type": "timestamp", "nullable": False, "metadata": {}},
9         {"name": "Country", "type": "string", "nullable": False, "metadata": {}},]
10
11 table_rows = [
12     Row(1, datetime(2021, 1, 1, 10, 0, 0), "RU"),
13     Row(2, datetime(1000, 1, 1, 10, 0, 0), "KZ"),
14     Row(2, datetime(2018, 1, 1, 10, 0, 0), "AU"),
15     Row(2, datetime(2019, 1, 1, 10, 0, 0), "")]
16
17 sample_df = spark.createDataFrame(table_rows, StructType.fromJson(schema))
```

```
1 from pyspark.sql.types import Row, StructType  
2 from datetime import datetime  
3  
4 schema = {  
5     "type": "struct",  
6     "fields": [  
7         {"name": "Id", "type": "long", "nullable": False, "metadata": {}},  
8         {"name": "SaleDate", "type": "timestamp", "nullable": False, "metadata": {}},  
9         {"name": "Country", "type": "string", "nullable": False, "metadata": {}},]  
10    }  
11 table_rows = [  
12     Row(1, datetime(2021, 1, 1, 10, 0, 0), "RU"),  
13     Row(2, datetime(1000, 1, 1, 10, 0, 0), "KZ"),  
14     Row(2, datetime(2018, 1, 1, 10, 0, 0), "AU"),  
15     Row(2, datetime(2019, 1, 1, 10, 0, 0), "")],  
16  
17 sample_df = spark.createDataFrame(table_rows, StructType.fromJson(schema))
```

# Great expectations

```
1 from great_expectations.dataset.sparkdf_dataset import SparkDFDataset  
2  
3 ge_sample_df = SparkDFDataset(sample_df)  
4 ge_sample_df.expect_column_values_to_be_in_set("Country", ["RU", "KZ"])
```

# Great expectations

```
1 "result": {  
2     "element_count": 4,  
3     "unexpected_count": 2,  
4     "unexpected_percent": 50.0,  
5     "partial_unexpected_list": ["AU", ""],  
6     "success": false,  
7     "expectation_config": {  
8         "kwargs": {  
9             "column": "Country",  
10            "value_set": ["RU", "KZ"]}}}
```

# Python Deequ

- profilers
- analyzers
- **constraint suggestions**
- constraint verification
- metrics repository

# Python Deequ

```
check = Check(spark, CheckLevel.Warning, "Review Check")

checkResult = (
    VerificationSuite(spark)
        .onData(sample_df)
        .addCheck(
            check.isComplete("Country")
                .isContainedIn("Country", ["RU", "KZ"]))
        .run()
)

checkResult_df = VerificationResult.checkResultsAsDataFrame(spark, checkResult)
checkResult_df.show()
```

[Testing data quality at scale with PyDeequ](#)

[PyDeequ documentation](#)

50

# Python Deequ. C

constraint	constraint_status	constraint_message
CompletenessConstraint	Success	
ComplianceConstraint	Failure	Value: 0.5 does not meet the constraint requirement!

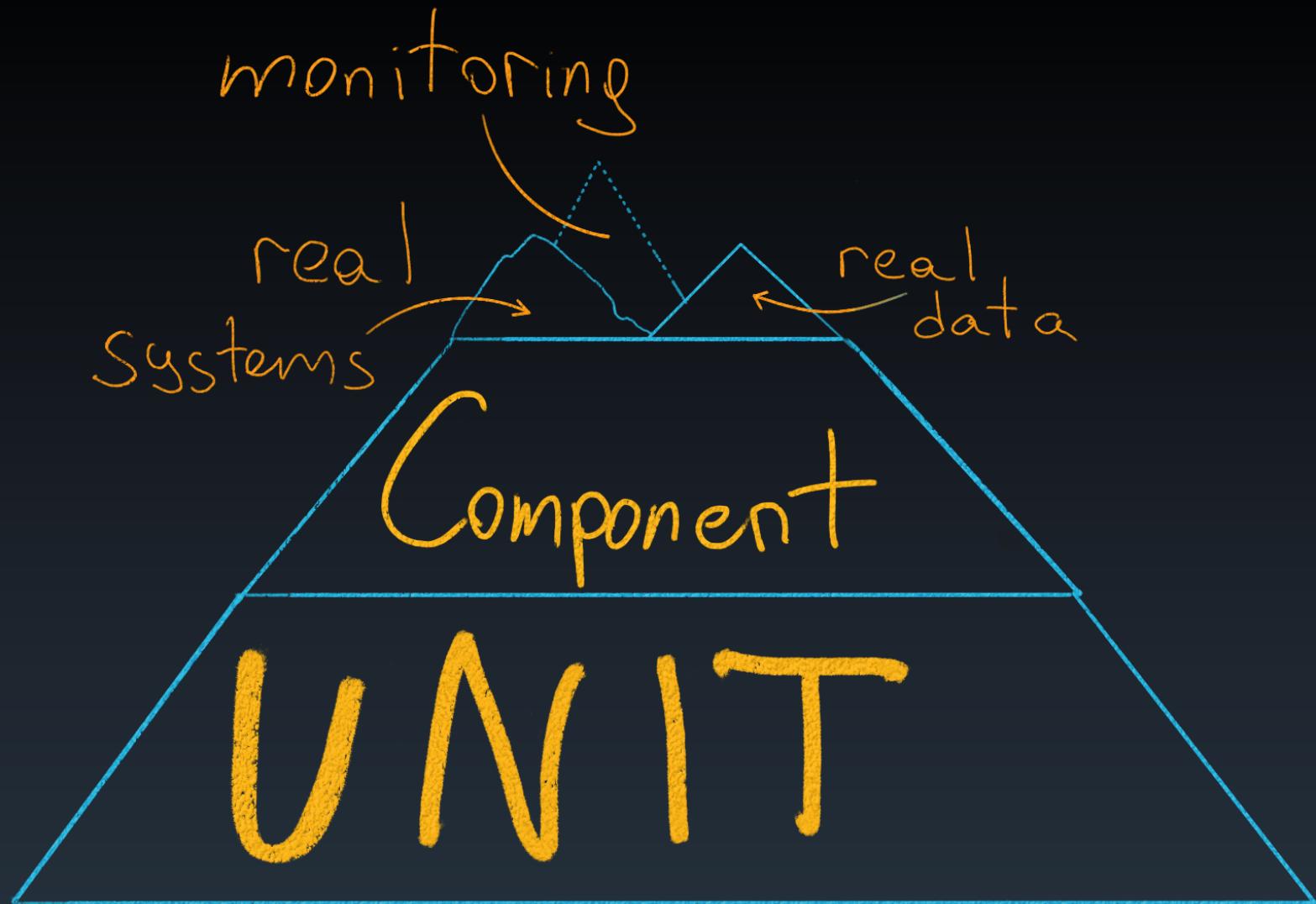
# Python Deequ. Histogram

```
analysisResult = (
    AnalysisRunner(spark)
    .onData(sample_df)
    .addAnalyzer(Histogram("Country"))
    .run()
)

analysisResult_df = AnalyzerContext.successMetricsAsDataFrame(spark, analysisResult)
analysisResult_df.show()
```

# Python Deequ. Histogram

entity	instance	name	value
Column	Country	Histogram.bins	4
Column	Country	<a href="#"><u>Histogram.abs.RU</u></a>	1
Column	Country	<a href="#"><u>Histogram.ratio.RU</u></a>	0.25
Column	Country	<a href="#"><u>Histogram.abs.KZ</u></a>	1
Column	Country	<a href="#"><u>Histogram.ratio.KZ</u></a>	0.25



# Monitoring

## Why?

- The only REAL testing is production
- Data tends to change over time

# Monitoring

What?

- data volumes
- counters
- time
- dead letter queue monitoring
- service health
- business metrics

# Monitoring

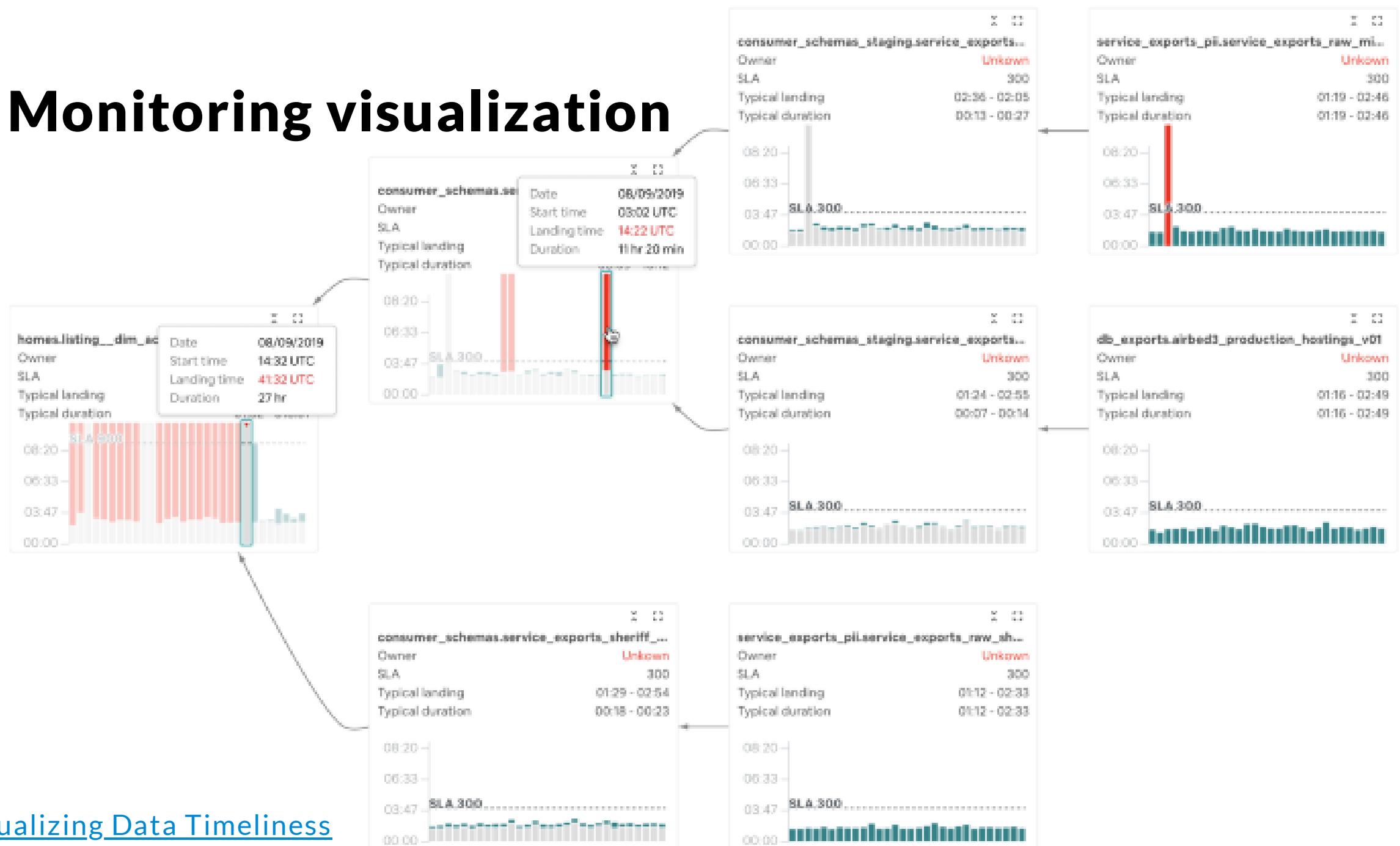
How?

- use Listeners
- use data aggregations

# Data pipelines is always DAG

Monitoring should visualize it

# Monitoring visualization

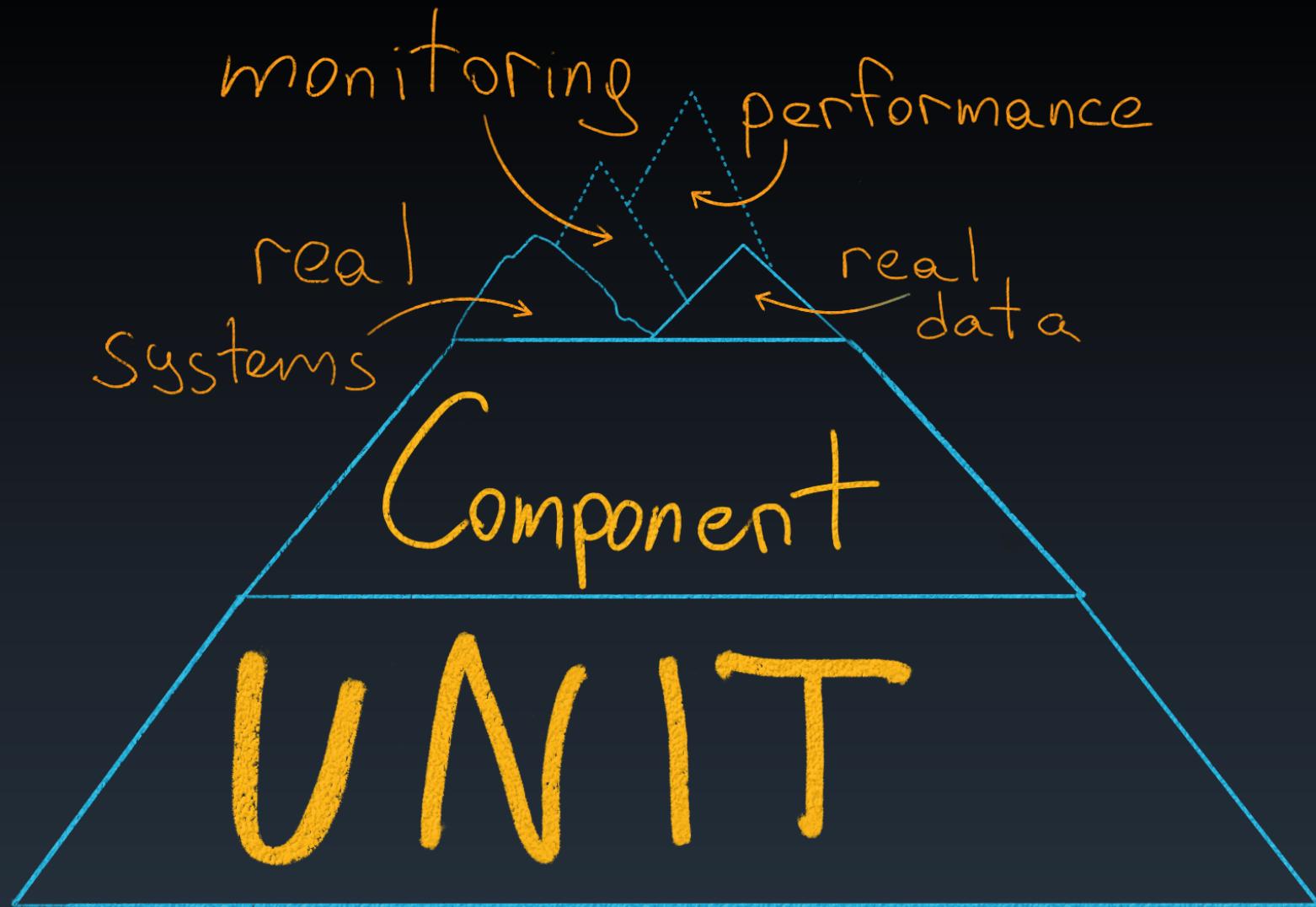


# End-to-End tests

Compare with reports, old DWH

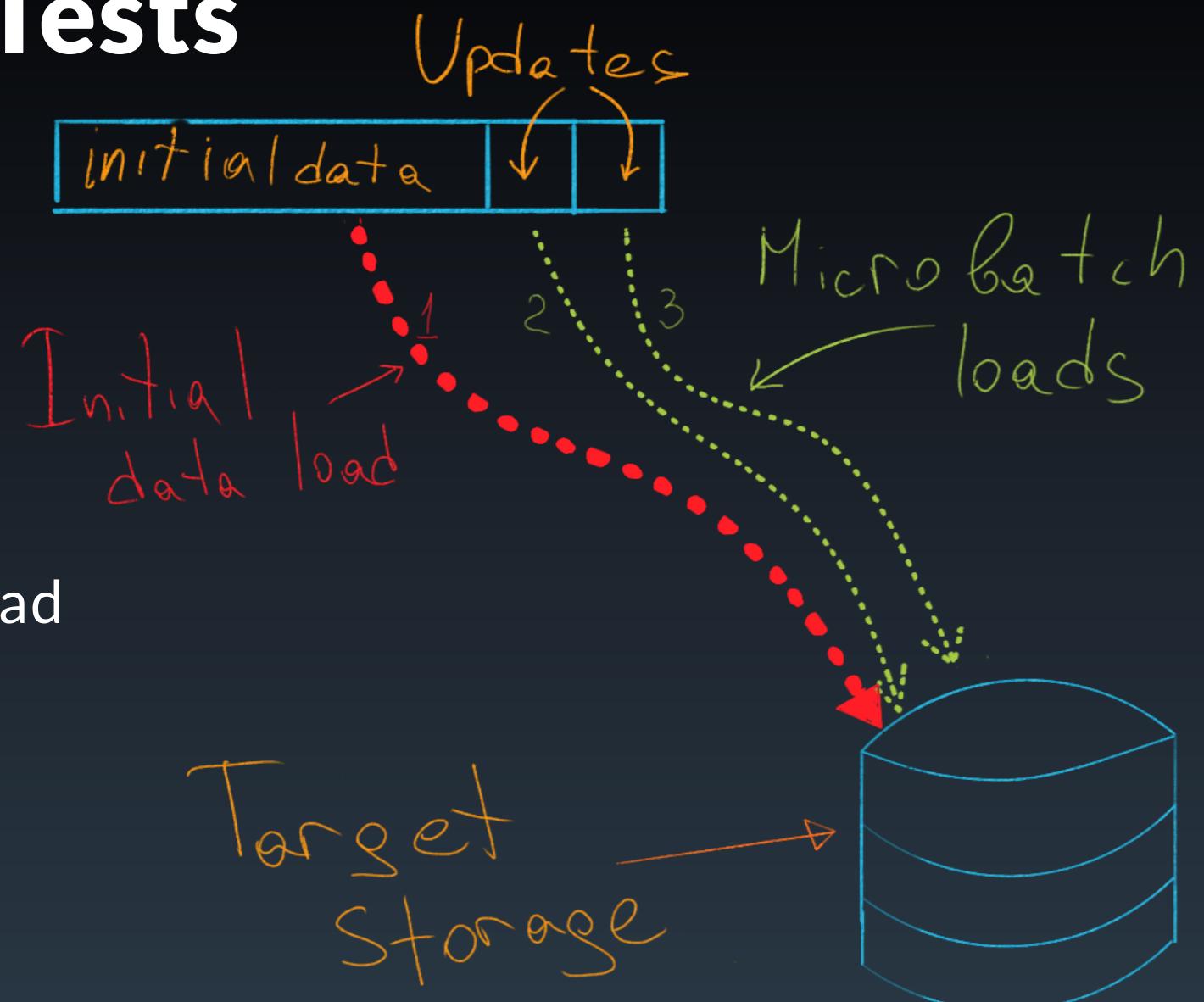
Multiple dimensions:

- data
- data latency
- performance, scalability



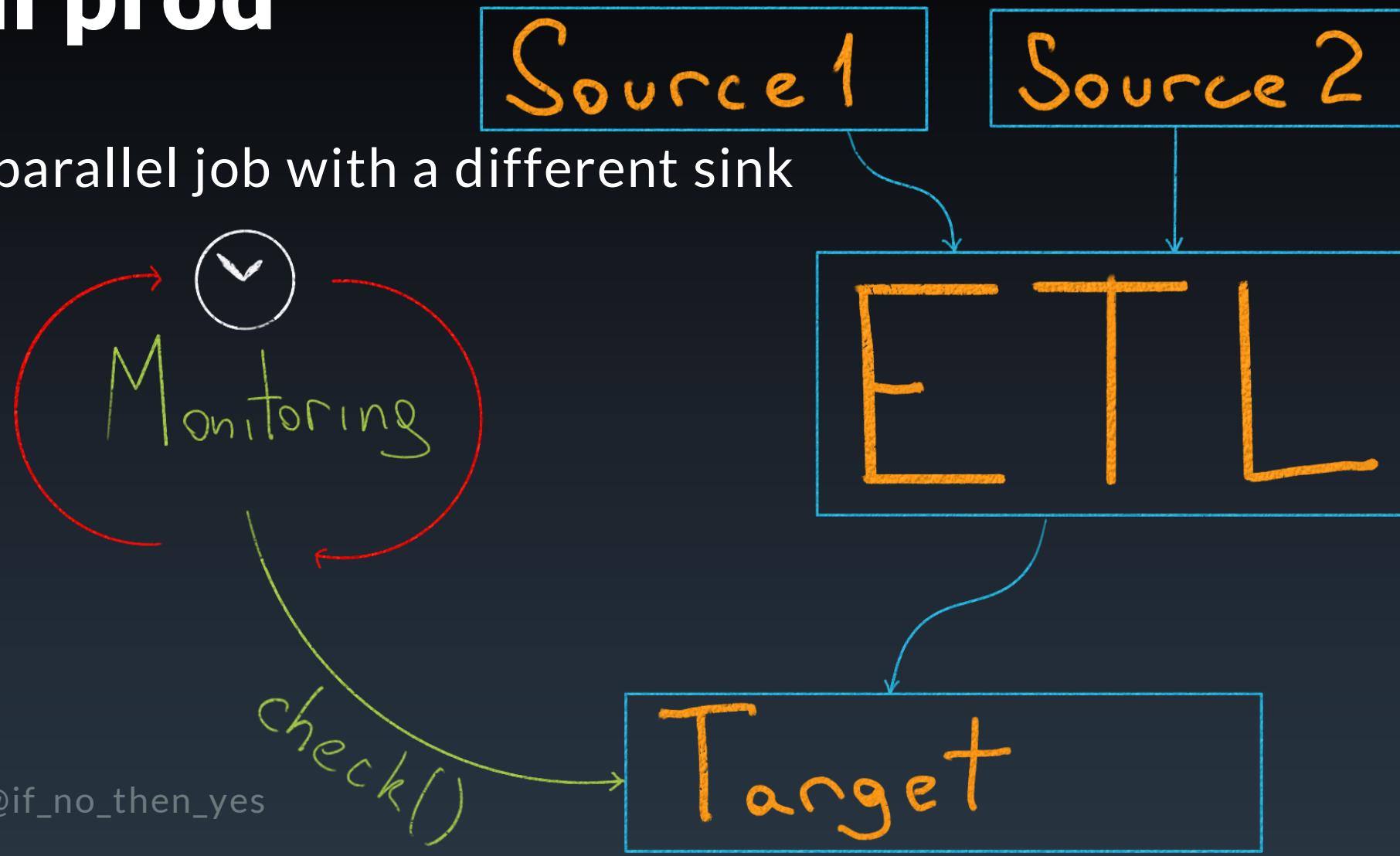
# Performance Tests

- start with SLO
- test your initial data load

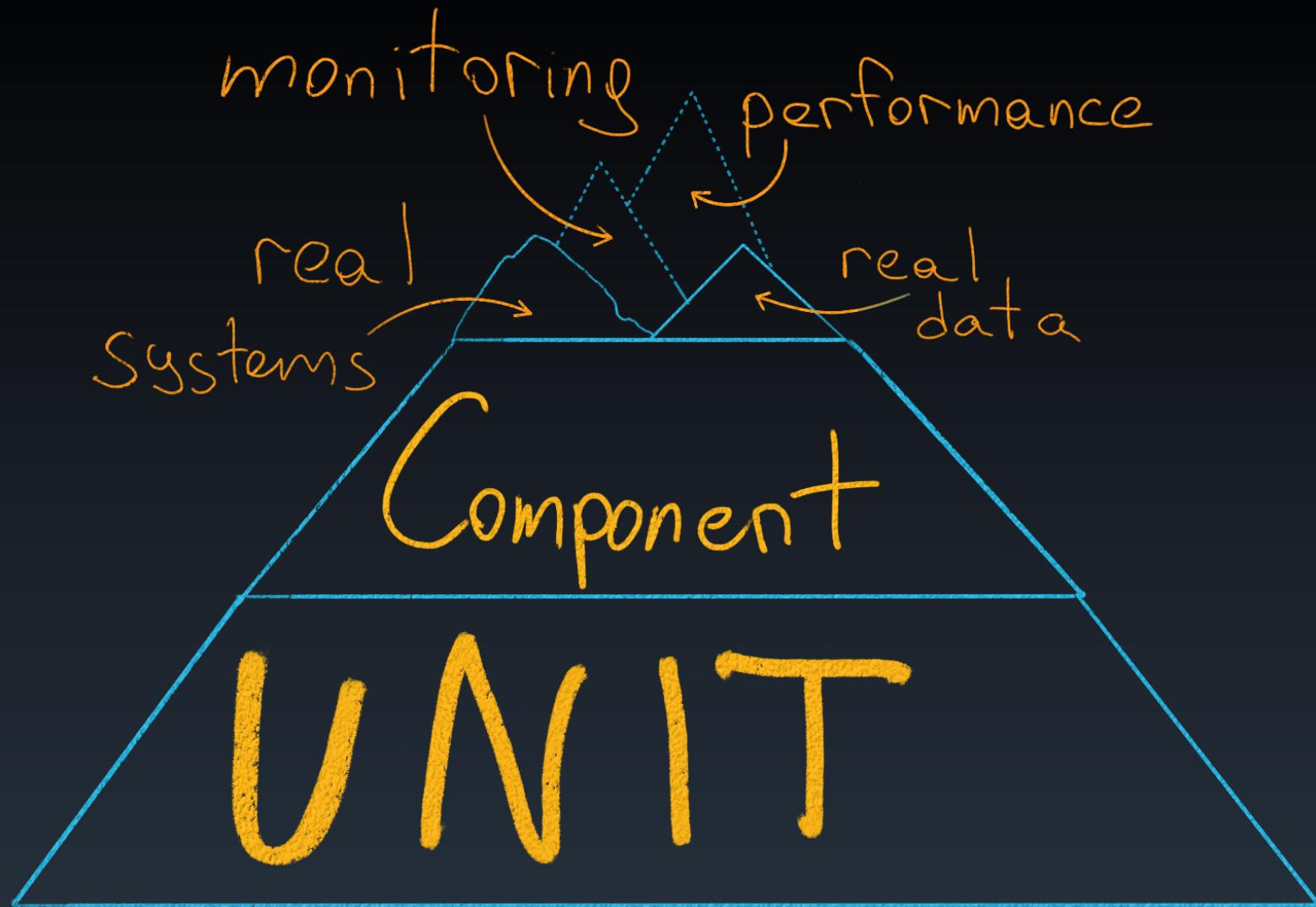


# Real prod

Run a parallel job with a different sink



Using production data for testing in a post GDPR world



# Summary

- Testing pipeline is like testing code
- Testing pipelines is not like testing code
- Pipeline quality is not only about testing
- Sometimes testing outside of production is tricky

# Thanks!

Questions? 

@asm0di0

@if\_no\_then\_yes