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 produces by

- sensors

Big Data Storage Formats

- CSV
- ORC
- Parquet

What is a pipeline?

Who needs pipelines

- Data Scientists
- Data Analysts
- Marketing
- PO

QA ?= QC

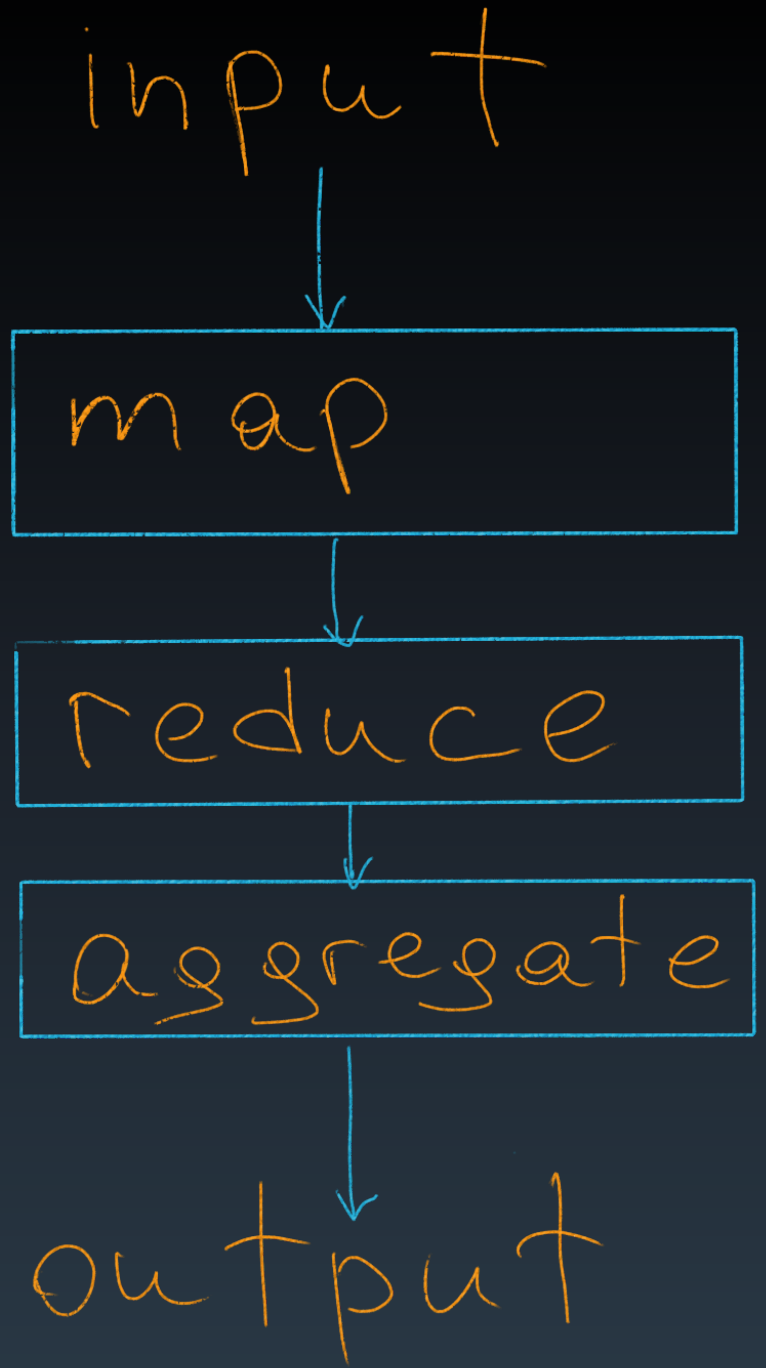
QA of pipeline

QA ≠ QC

QA is about processes and not only about software quality.

Pyramid of testing. Unit

Typical pipeline



Typical pipeline

Unit testing of pipeline

What may we test here?

A pipeline should transform data correctly!

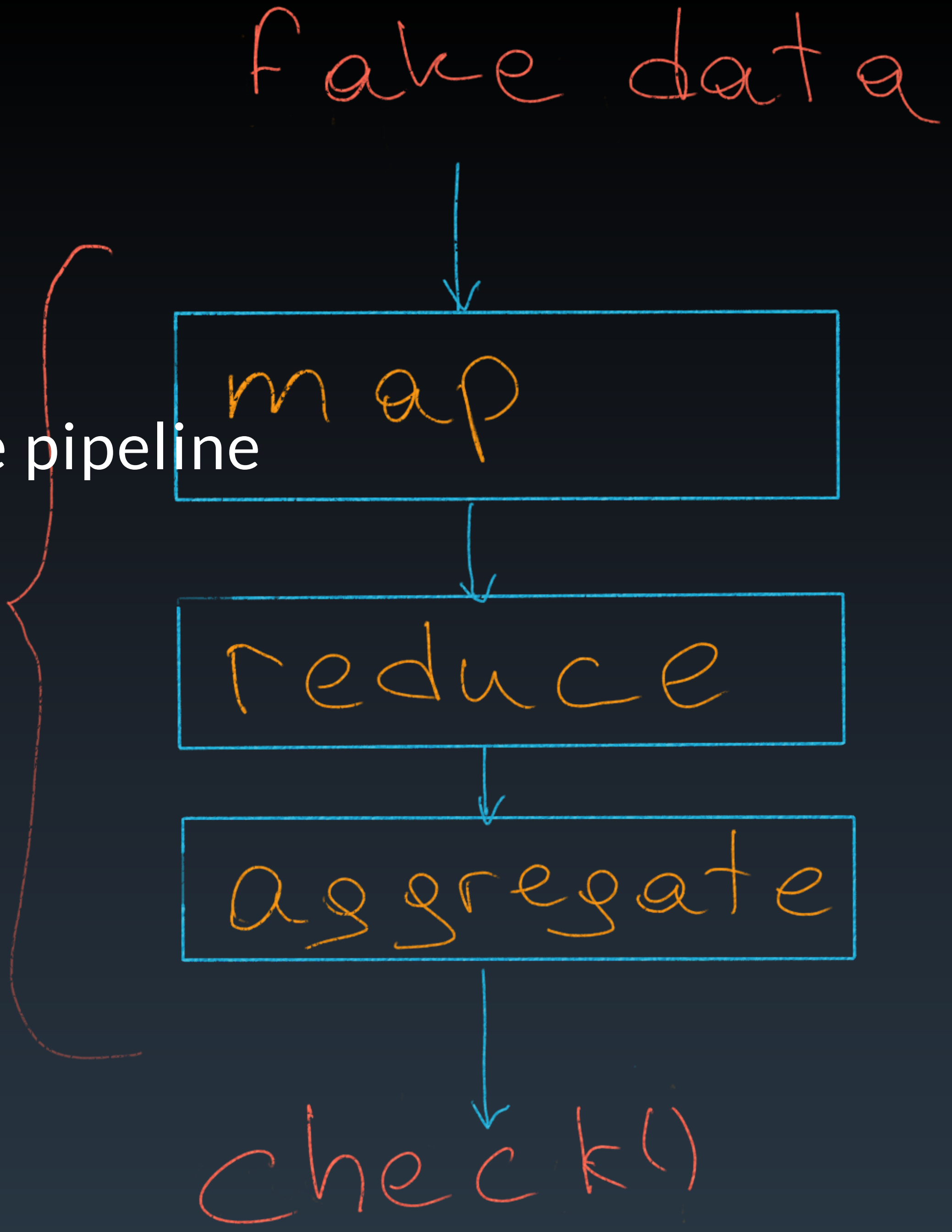
Correctness is a business term

Let's paste fakes!

Fake/mock input data

Reference data at the end of the pipeline

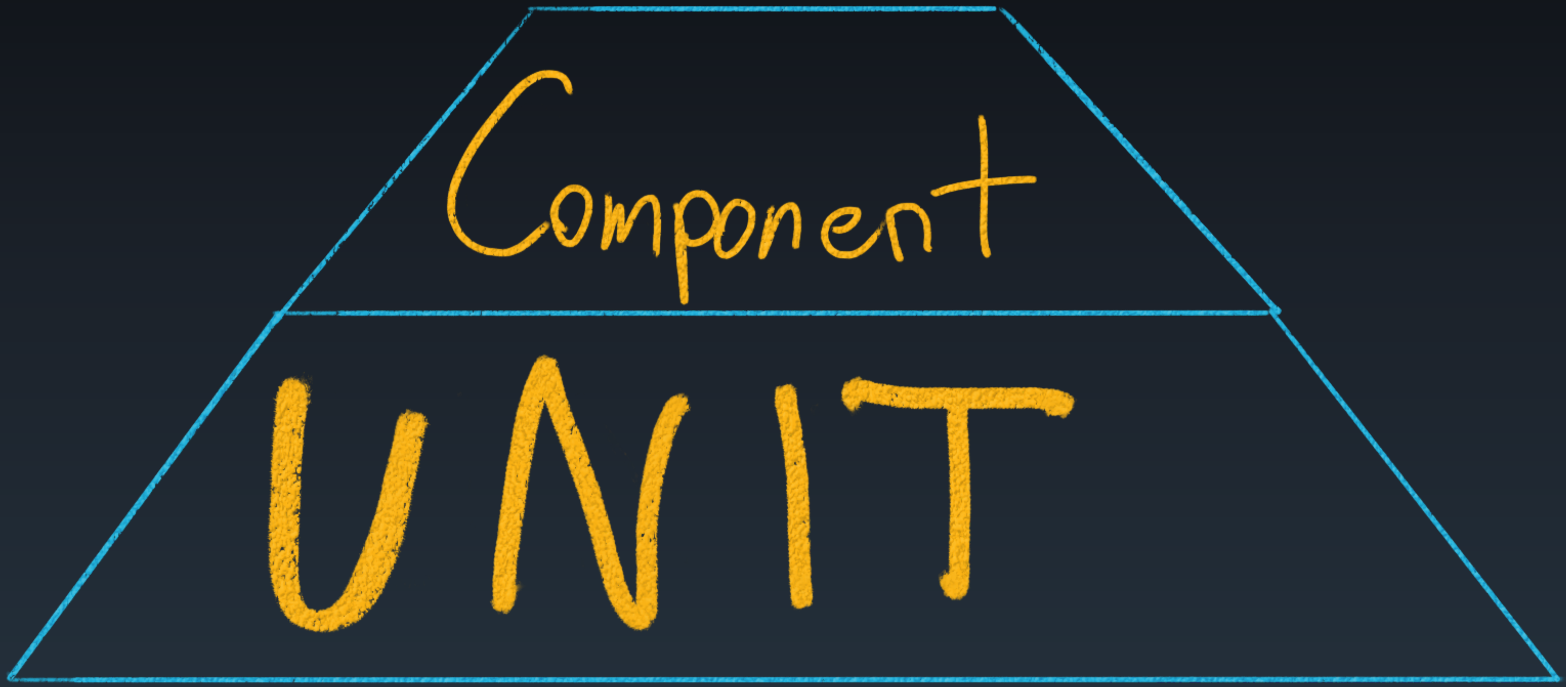
Separate function



Tools

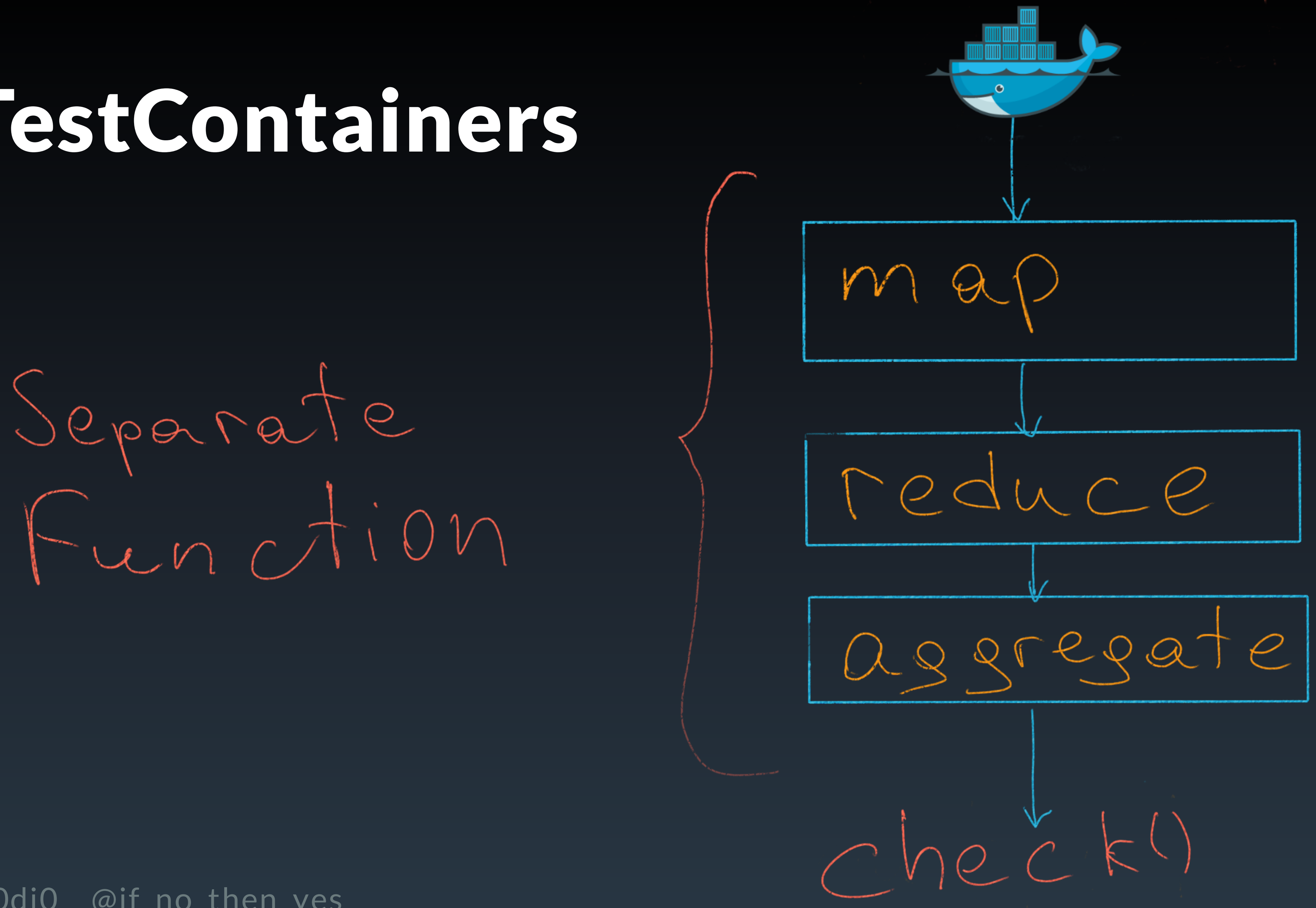
[holdenk/spark-testing-base](#) ← Tools to run tests
[MrPowers/spark-daria](#) ← tools to easily create test data

Component testing



TESTCONTAINERS

TestContainers



TestContainers

Supported languages:

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

Test Containers

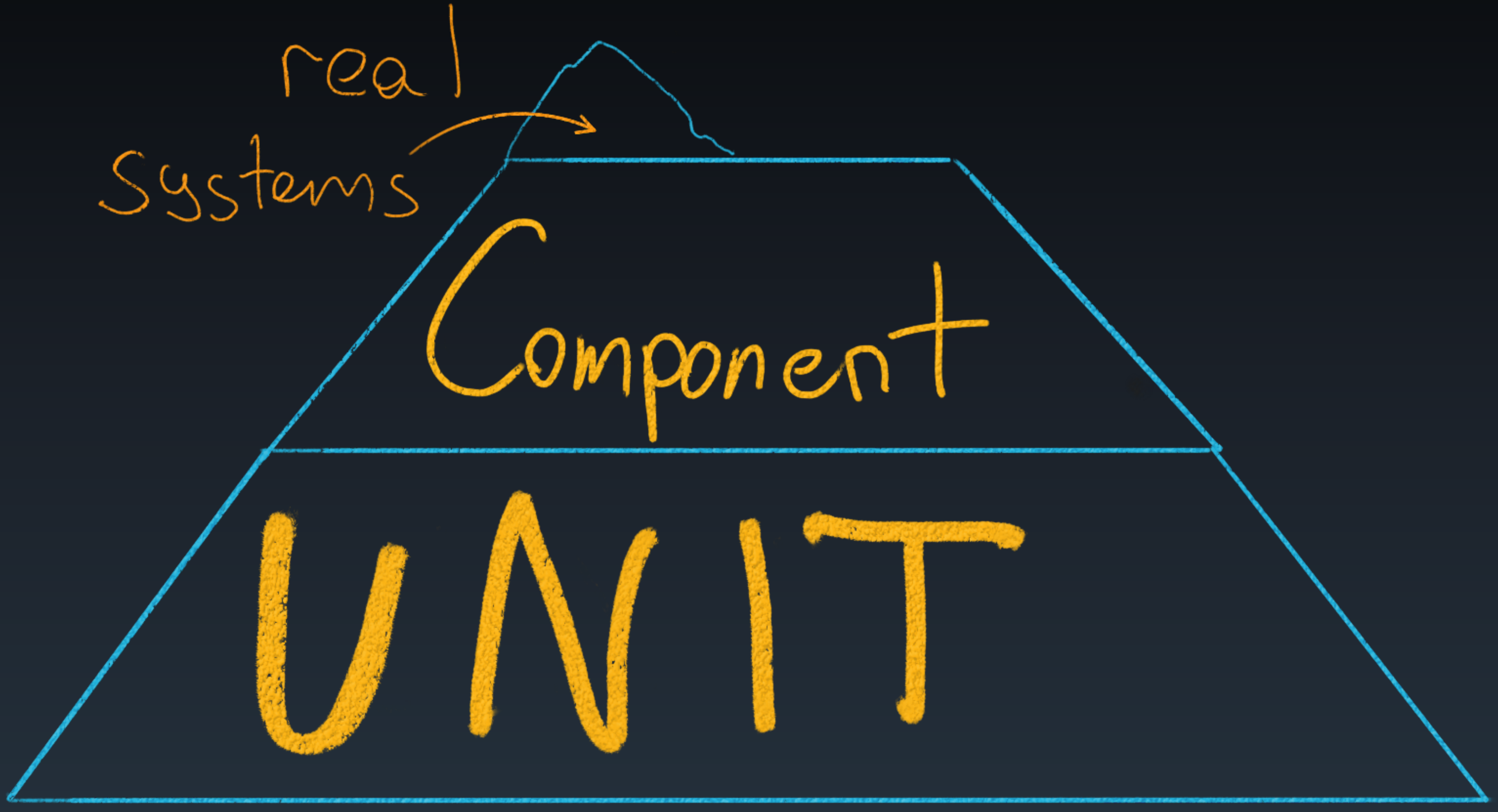
```python
import sqlalchemy
from testcontainers.mysql import MySqlContainer

with MySqlContainer('mysql:5.7.17') as mysql:
    engine = sqlalchemy.create_engine(mysql.get_connection_url())
    version, = engine.execute("select version()").fetchone()
    print(version)  # 5.7.17
```

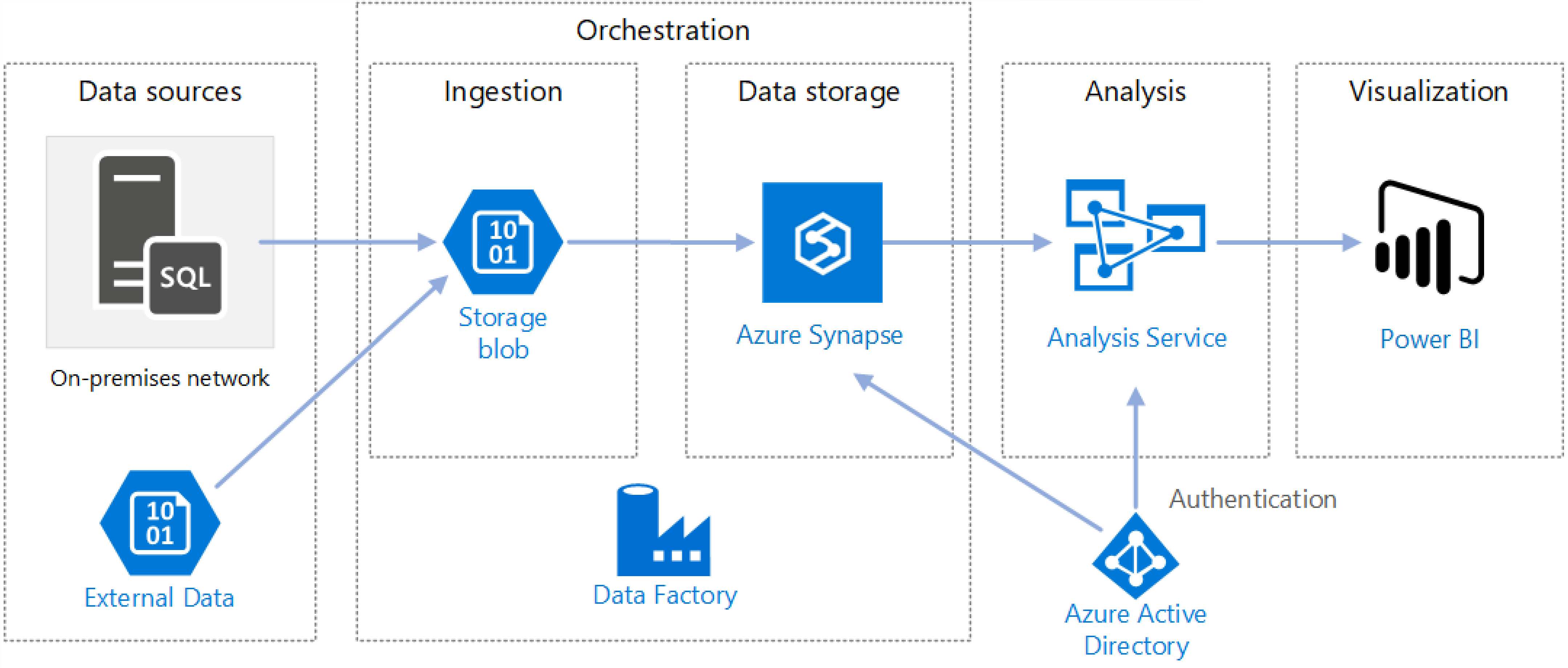
Test Containers

```python
import sqlalchemy
from testcontainers.mysql import MySqlContainer

with MySqlContainer('mysql:5.7.17') as mysql:
    engine = sqlalchemy.create_engine(mysql.get_connection_url())
    version, = engine.execute("select version()").fetchone()
    print(version)  # 5.7.17
```

Test Containers

```python
import sqlalchemy
from testcontainers.mysql import MySqlContainer

with MySqlContainer('mysql:5.7.17') as mysql:
    engine = sqlalchemy.create_engine(mysql.get_connection_url())
    version, = engine.execute("select version()").fetchone()
    print(version)  # 5.7.17
```


real
systems
Component
UNIT

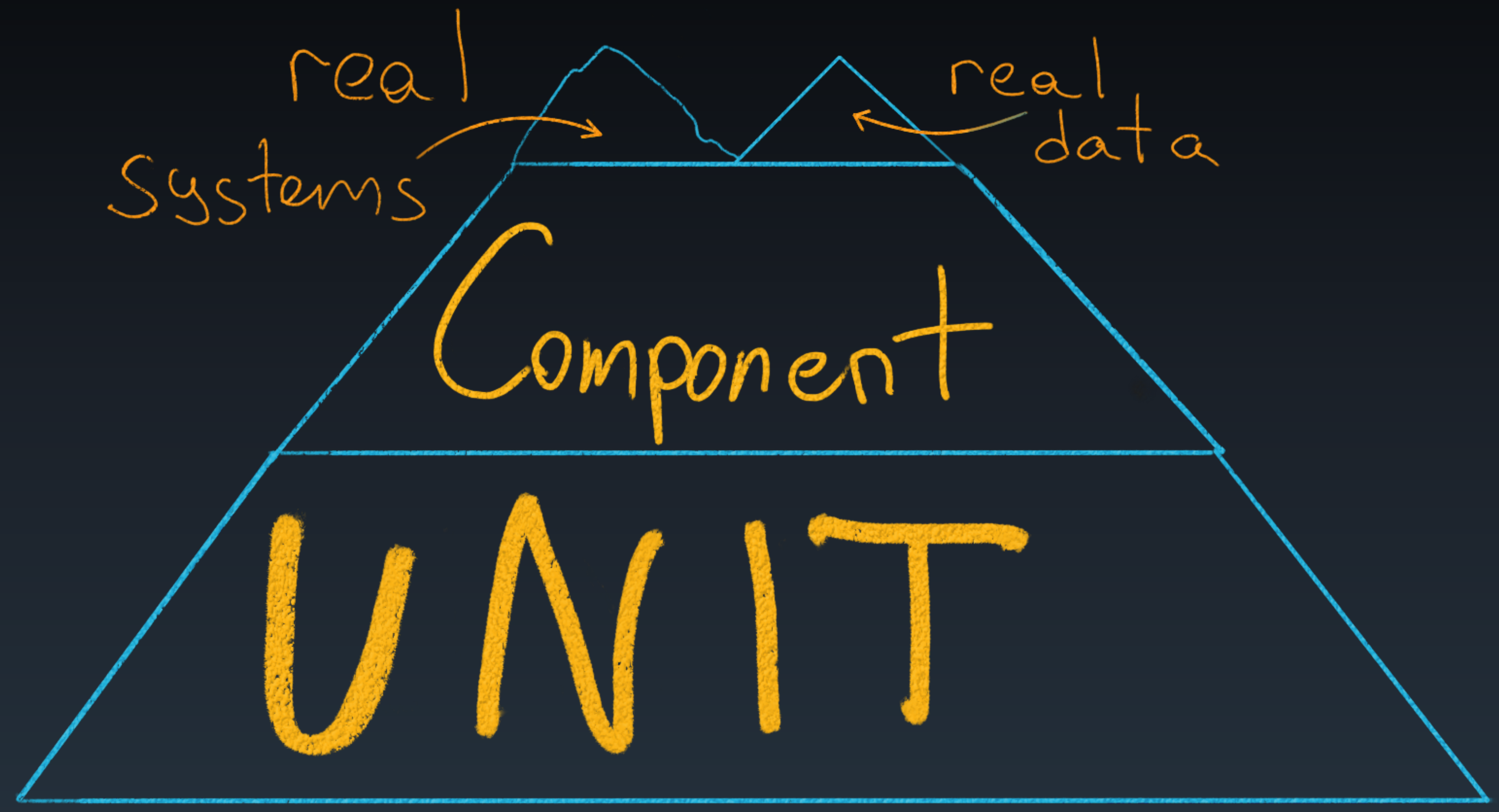


@asm0di0 @if_no_then_yes

Real systems

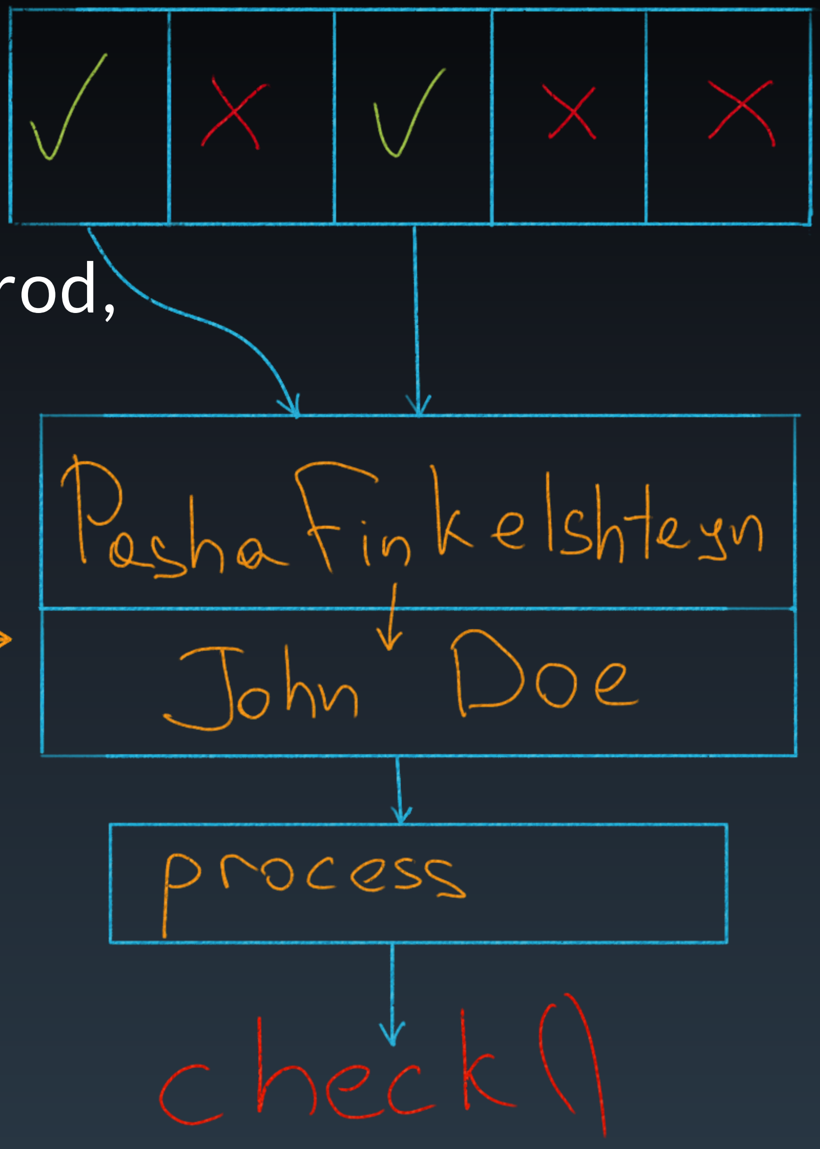
Why are component tests not enough?

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

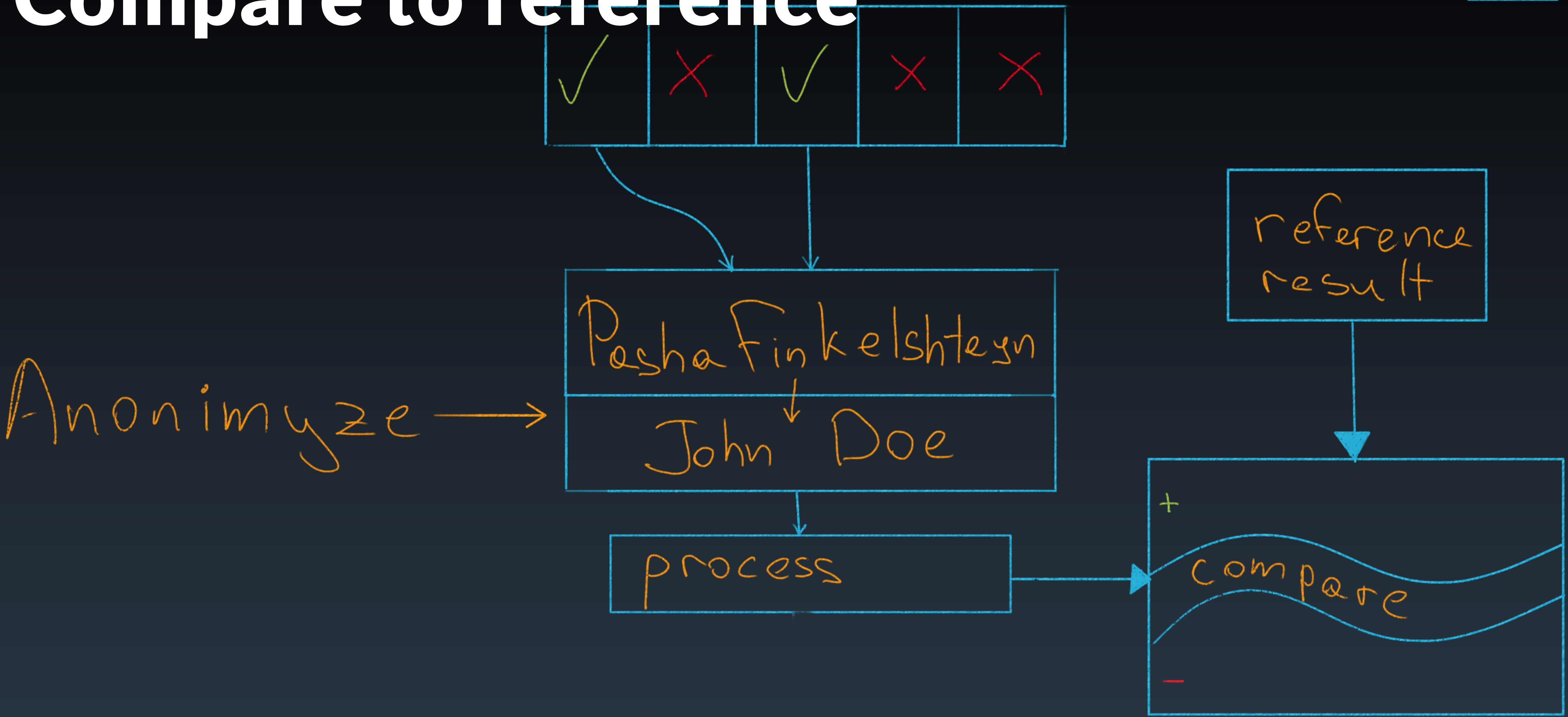

real
systems
real
data
Component
UNIT

Real data

Get data samples from prod, anonymize it



Compare to reference



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

Automated Testing For Protecting Data Pipelines from Undocumented Assumptions

```python
from pyspark.sql.types import Row, StructType
from datetime import datetime

schema = {
  "type": "struct",
  "fields": [
    {"name": "Id", "type": "long", "nullable": False, "metadata": {}},
    {"name": "SaleDate", "type": "timestamp", "nullable": False, "metadata": {}},
    {"name": "Country", "type": "string", "nullable": False, "metadata": {}},]}

table_rows = [
        Row(1, datetime(2021, 1, 1, 10, 0, 0), "RU" ),
        Row(2, datetime(1000, 1, 1, 10, 0, 0), "KZ"),
        Row(2, datetime(2018, 1, 1, 10, 0, 0), "AU"),
        Row(2, datetime(2019, 1, 1, 10, 0, 0), ""),]

sample_df = spark.createDataFrame(table_rows, StructType.fromJson(schema))
```

```python
from pyspark.sql.types import Row, StructType
from datetime import datetime

schema = {
  "type": "struct",
  "fields": [
    {"name": "Id", "type": "long", "nullable": False, "metadata": {}},
    {"name": "SaleDate", "type": "timestamp", "nullable": False, "metadata": {}},
    {"name": "Country", "type": "string", "nullable": False, "metadata": {}},]}

table_rows = [
        Row(1, datetime(2021, 1, 1, 10, 0, 0), "RU" ),
        Row(2, datetime(1000, 1, 1, 10, 0, 0), "KZ"),
        Row(2, datetime(2018, 1, 1, 10, 0, 0), "AU"),
        Row(2, datetime(2019, 1, 1, 10, 0, 0), ""),]

sample_df = spark.createDataFrame(table_rows, StructType.fromJson(schema))
```

```python
from pyspark.sql.types import Row, StructType
from datetime import datetime

schema = {
  "type": "struct",
  "fields": [
    {"name": "Id", "type": "long", "nullable": False, "metadata": {}},
    {"name": "SaleDate", "type": "timestamp", "nullable": False, "metadata": {}},
    {"name": "Country", "type": "string", "nullable": False, "metadata": {}},]}

table_rows = [
        Row(1, datetime(2021, 1, 1, 10, 0, 0), "RU" ),
        Row(2, datetime(1000, 1, 1, 10, 0, 0), "KZ"),
        Row(2, datetime(2018, 1, 1, 10, 0, 0), "AU"),
        Row(2, datetime(2019, 1, 1, 10, 0, 0), ""),]

sample_df = spark.createDataFrame(table_rows, StructType.fromJson(schema))
```

```python
from pyspark.sql.types import Row, StructType
from datetime import datetime

schema = {
  "type": "struct",
  "fields": [
    {"name": "Id", "type": "long", "nullable": False, "metadata": {}},
    {"name": "SaleDate", "type": "timestamp", "nullable": False, "metadata": {}},
    {"name": "Country", "type": "string", "nullable": False, "metadata": {}},]}

table_rows = [
        Row(1, datetime(2021, 1, 1, 10, 0, 0), "RU" ),
        Row(2, datetime(1000, 1, 1, 10, 0, 0), "KZ"),
        Row(2, datetime(2018, 1, 1, 10, 0, 0), "AU"),
        Row(2, datetime(2019, 1, 1, 10, 0, 0), ""),]

sample_df = spark.createDataFrame(table_rows, StructType.fromJson(schema))
```

Great expectations

```
from great_expectations.dataset.sparkdf_dataset import SparkDFDataset

ge_sample_df = SparkDFDataset(sample_df)
ge_sample_df.expect_column_values_to_be_in_set("Country", ["RU", "KZ"])
```

Great expectations

```
"result": {
  "element_count": 4,
  "unexpected_count": 2,
  "unexpected_percent": 50.0,
  "partial_unexpected_list": ["AU",""]},
"success": false,
"expectation_config": {
  "kwargs": {
    "column": "Country",
    "value_set": ["RU", "KZ"]}}
```

Python Deequ

```
# No Spark 3.0 support yet
from pydeequ.checks import *
from pydeequ.verification import *

check = Check(spark, CheckLevel.Warning, "Country Check")
checkResult =(
  VerificationSuite(spark)
    .onData(sample_df)
    .addCheck(
        check.isContainedIn("Counrty", ["RU","KZ"]))
   .run()
)
checkResult_df = VerificationResult.checkResultsAsDataFrame(spark, checkResult)
checkResult_df.show()
```

[Testing data quality at scale with PyDeequ]()

Python Deequ

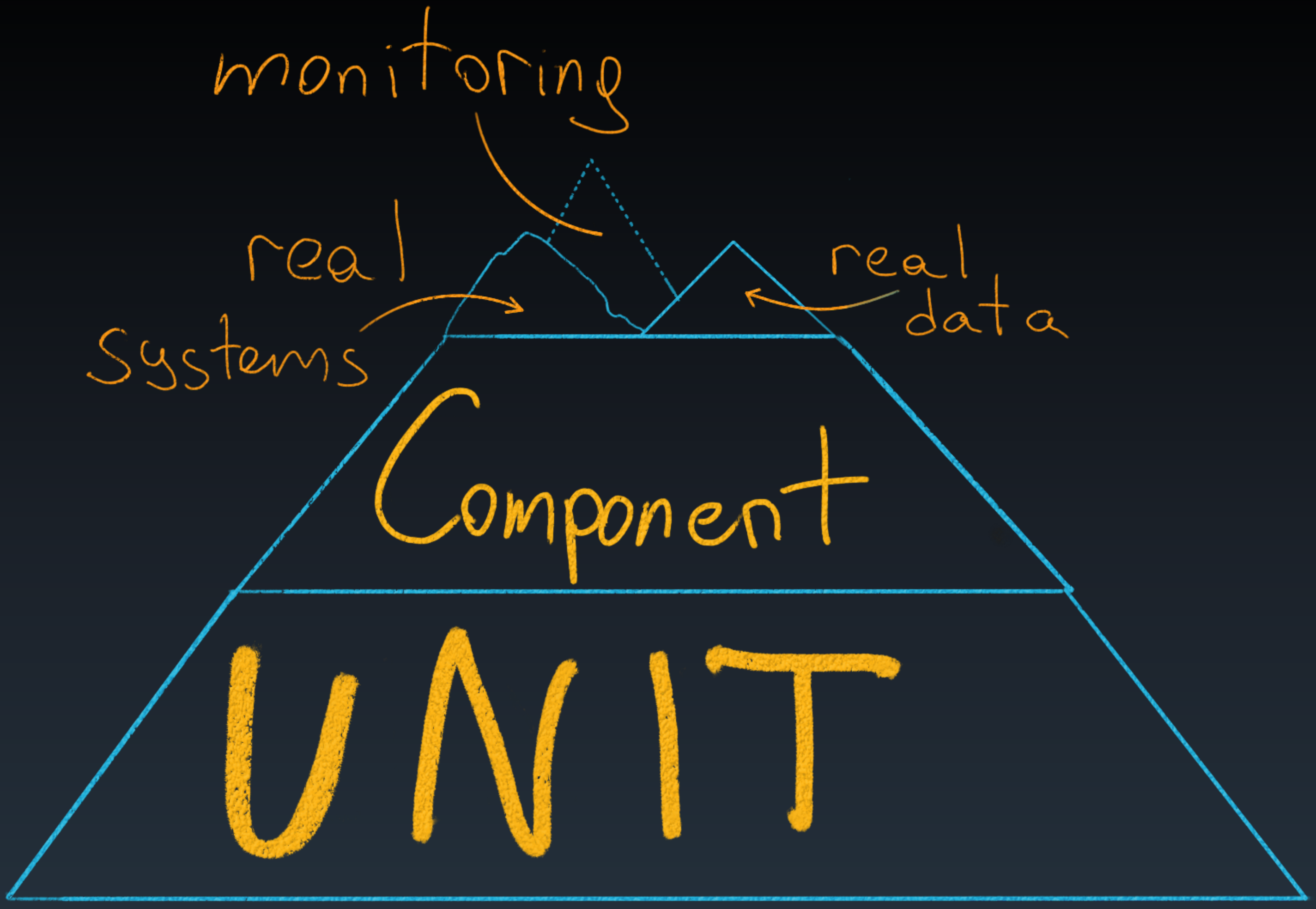
```
# No Spark 3.0 support yet
from pydeequ.checks import *
from pydeequ.verification import *

check = Check(spark, CheckLevel.Warning, "Country Check")
checkResult =(
  VerificationSuite(spark)
    .onData(sample_df)
    .addCheck(
        check.isContainedIn("Counrty", ["RU","KZ"]))
   .run()
)
checkResult_df = VerificationResult.checkResultsAsDataFrame(spark, checkResult)
checkResult_df.show()
```

[Testing data quality at scale with PyDeequ]

Python Deequ

```
# No Spark 3.0 support yet
from pydeequ.checks import *
from pydeequ.verification import *

check = Check(spark, CheckLevel.Warning, "Country Check")
checkResult =(
  VerificationSuite(spark)
    .onData(sample_df)
    .addCheck(
        check.isContainedIn("Counrty", ["RU","KZ"]))
   .run()
)
checkResult_df = VerificationResult.checkResultsAsDataFrame(spark, checkResult)
checkResult_df.show()
```

[Testing data quality at scale with PyDeequ]


monitoring
real systems
real data
Component
UNIT

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

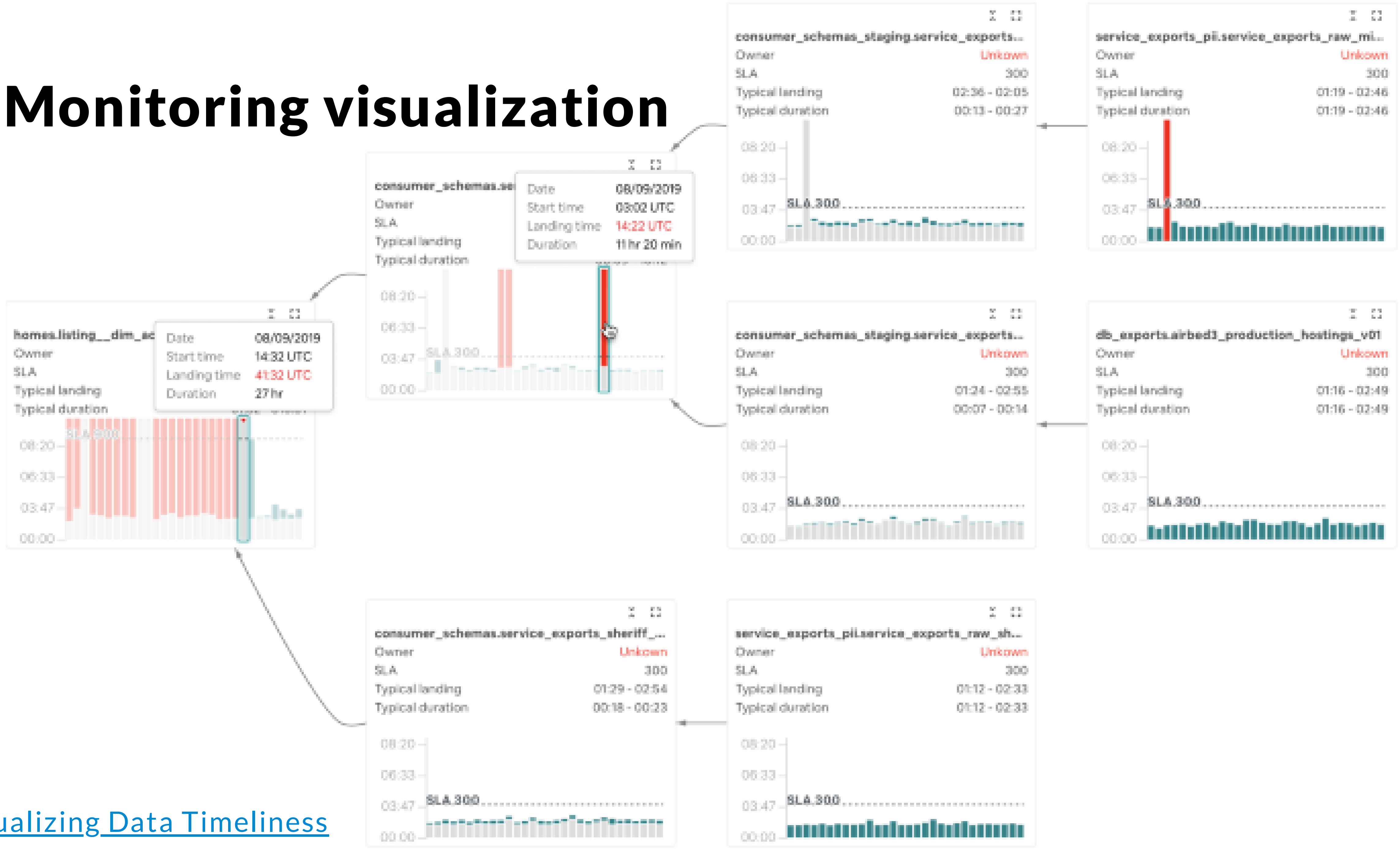
How?

- use Listeners
- use data aggregations

Data pipelines is always DAG

Monitoring should visualize it

Monitoring visualization



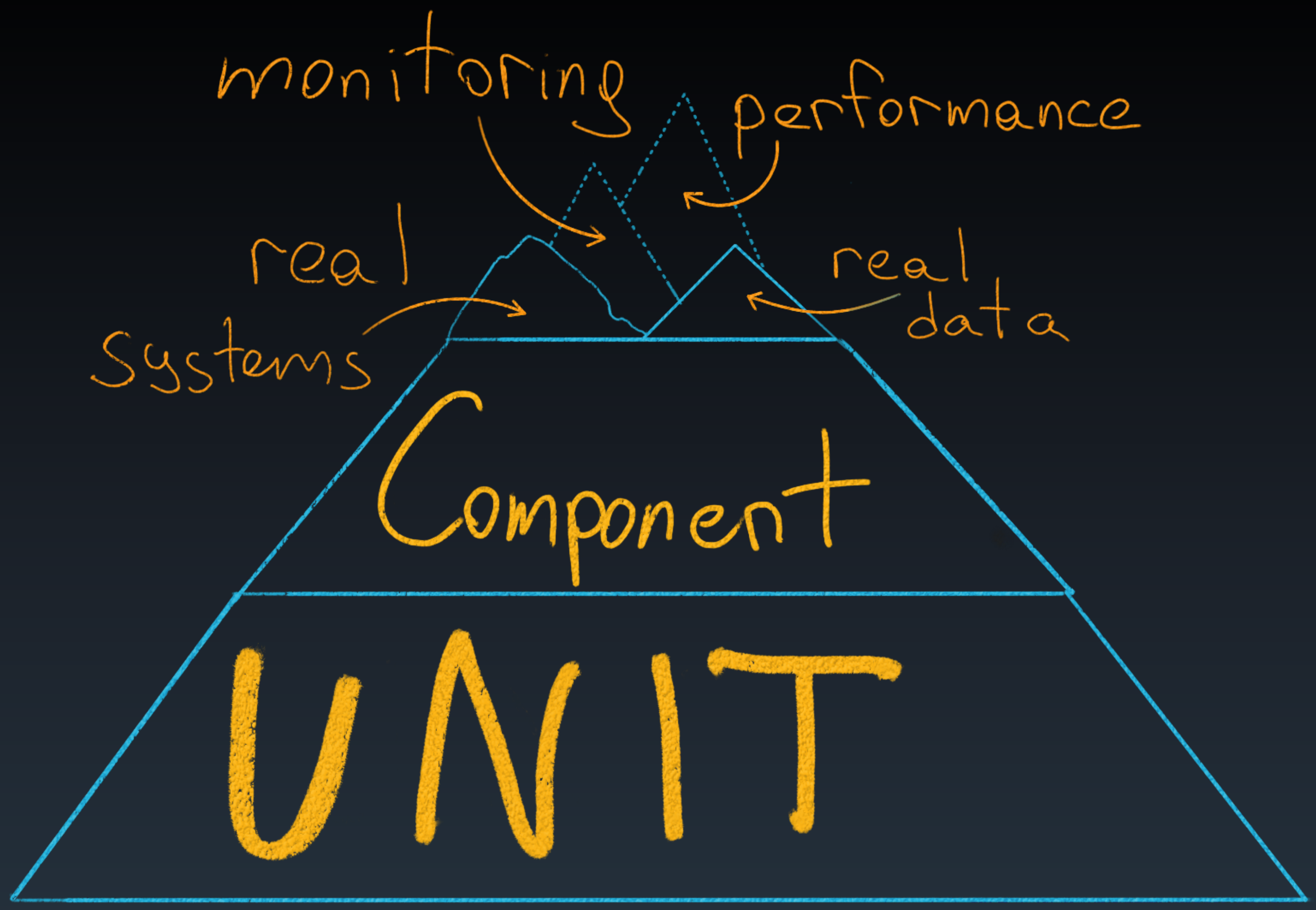
[Visualizing Data Timeliness](#)

End-to-End tests

Compare with reports, old DWH

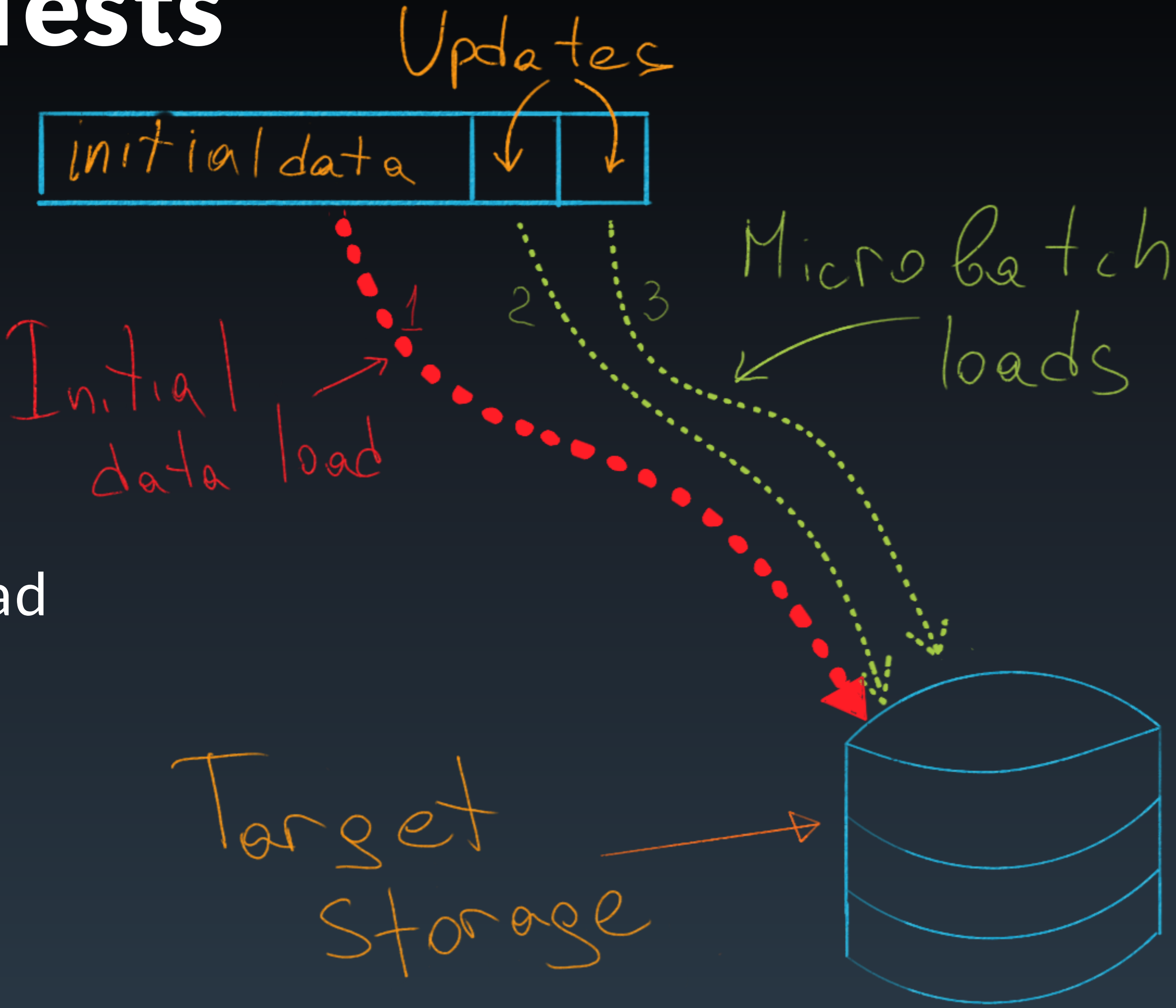
Multiple dimensions:

- data
- data latency
- performance, scalability


monitoring
performance
real
systems
real
data
Component
UNIT

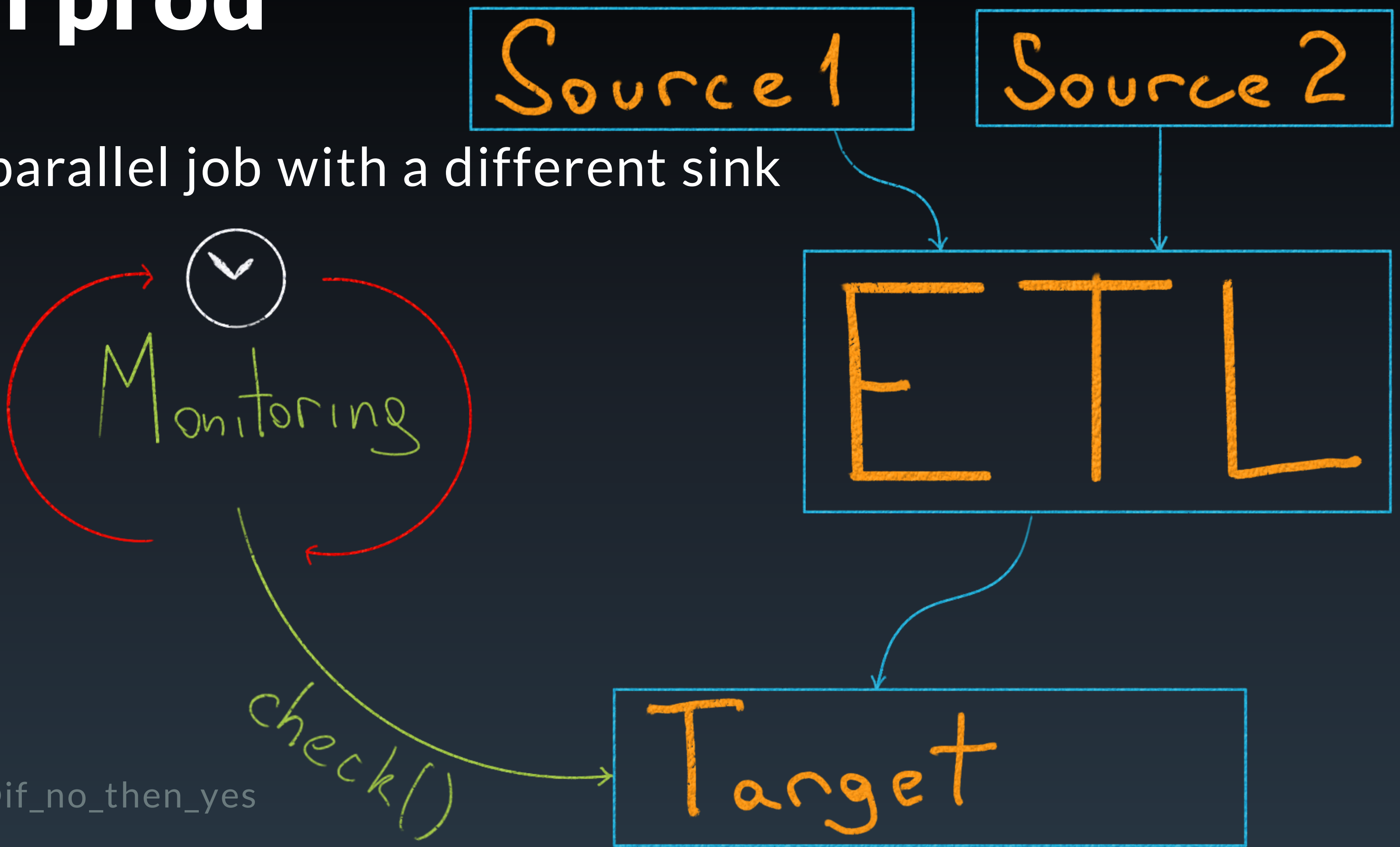
Performance Tests

- start with SLA
- 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]