

# Himalayan Peaks of Testing Data Pipelines

Ksenia Tomak, HelloFresh  
Pasha Finkelshteyn, JetBrains

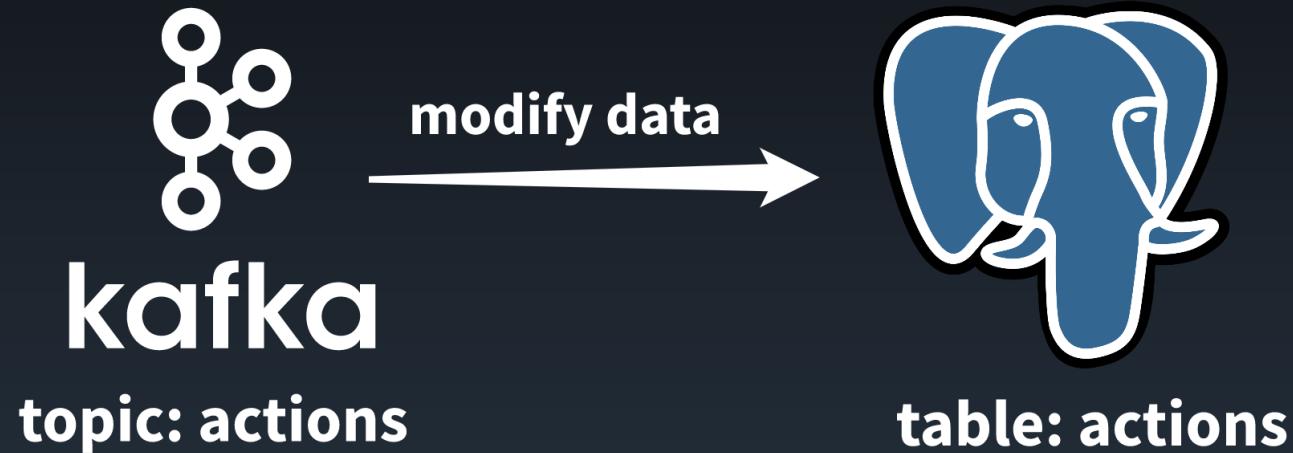
# Pasha Finkelshteyn

Developer  for Big Data @ JetBrains

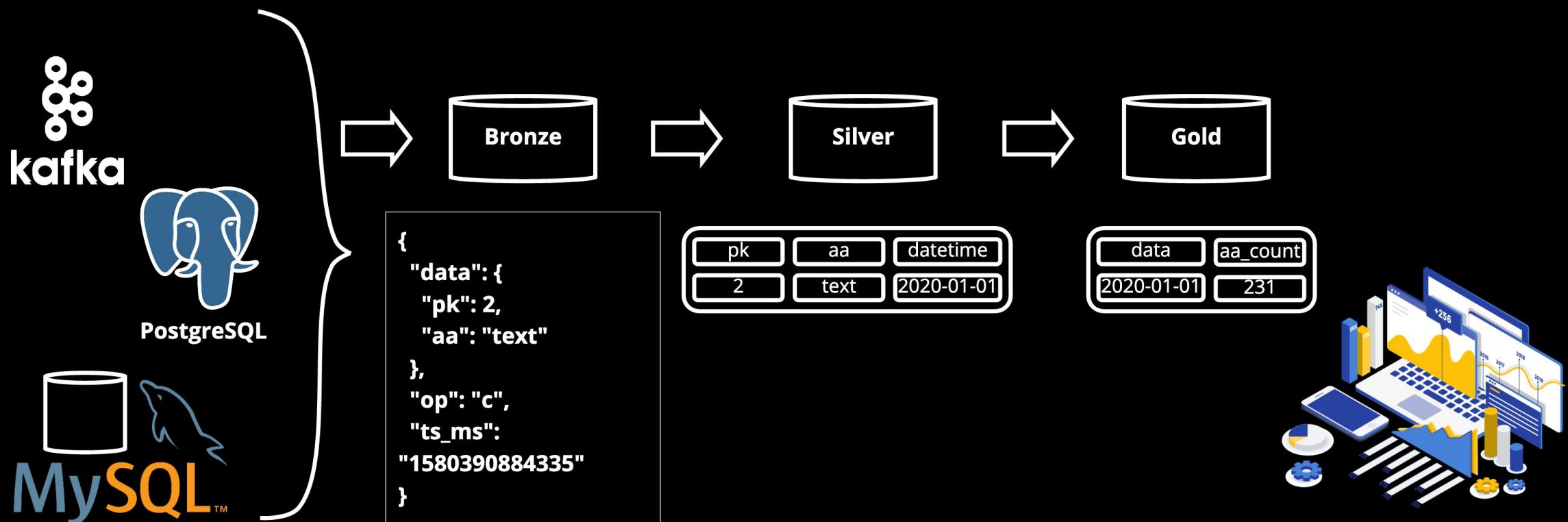
@asm0di0

 @asm0dey@fosstodon.org

# Data processing



# Data lake?



# Who needs pipelines

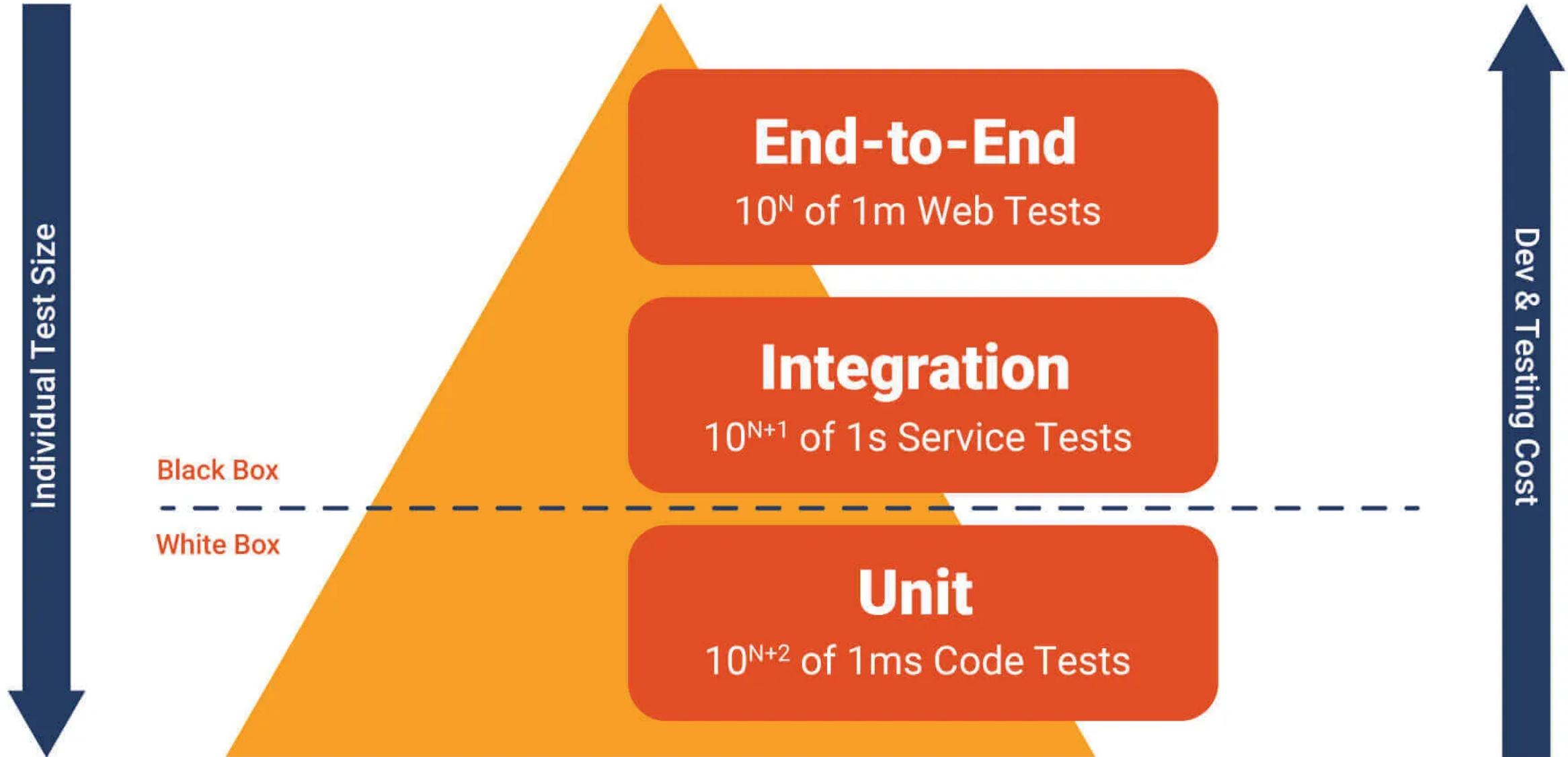
- Data Scientists
- Data Analytics
- Marketing
- PO

# It have to be tested!

# Pyramid of testing?

# Pyramid of testing?

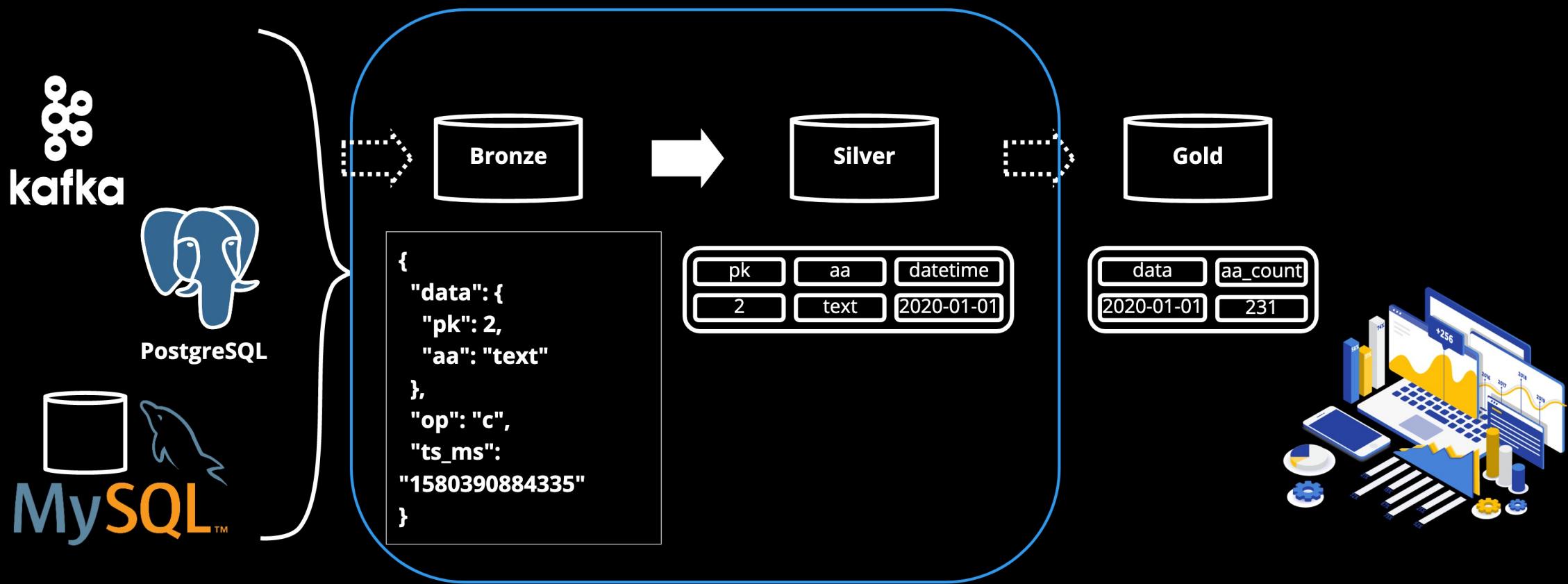




# Pyramid of testing. Unit



# Bronze → Silver pipeline



# Typical pipeline



# Typical pipeline

```
1 StructType schema = new StructType(new StructField[]{  
2     new StructField("pk", DataTypes.LongType, false, Metadata.empty()),  
3     new StructField("aa", new DataTypes.StringType, false, Metadata.empty()),  
4 });  
5  
6 spark.read  
7     .schema(schema)  
8     .csv(/* path */)  
9     .map(/* mapper */)  
10    .show() // terminal operation
```

# Typical pipeline

```
1 StructType schema = new StructType(new StructField[]{  
2     new StructField("pk", DataTypes.LongType, false, Metadata.empty()),  
3     new StructField("aa", new DataTypes.StringType, false, Metadata.empty()),  
4 });  
5  
6 spark.read  
7     .schema(schema)  
8     .csv(/* path */)  
9     .map(/* mapper */)  
10    .show() // terminal operation
```

# Typical pipeline

```
1 StructType schema = new StructType(new StructField[]{  
2     new StructField("pk", DataTypes.LongType, false, Metadata.empty()),  
3     new StructField("aa", new DataTypes.StringType, false, Metadata.empty()),  
4 });  
5  
6 spark.read  
7     .schema(schema)  
8     .csv(/* path */)  
9     .map(/* mapper */)  
10    .show() // terminal operation
```

# Typical pipeline

```
1 StructType schema = new StructType(new StructField[]{  
2     new StructField("pk", DataTypes.LongType, false, Metadata.empty()),  
3     new StructField("aa", new DataTypes.StringType, false, Metadata.empty()),  
4 });  
5  
6 spark.read  
7     .schema(schema)  
8     .csv(/* path */)  
9     .map(/* mapper */)  
10    .show() // terminal operation
```

# Typical pipeline

```
1 StructType schema = new StructType(new StructField[]{  
2     new StructField("pk", DataTypes.LongType, false, Metadata.empty()),  
3     new StructField("aa", new DataTypes.StringType, false, Metadata.empty()),  
4 });  
5  
6 spark.read  
7     .schema(schema)  
8     .csv(/* path */)  
9     .map(/* mapper */)  
10    .show() // terminal operatio
```

# Unit testing of pipeline

What may we test here?

A pipeline should transform data correctly!

*Correctness is a business term*

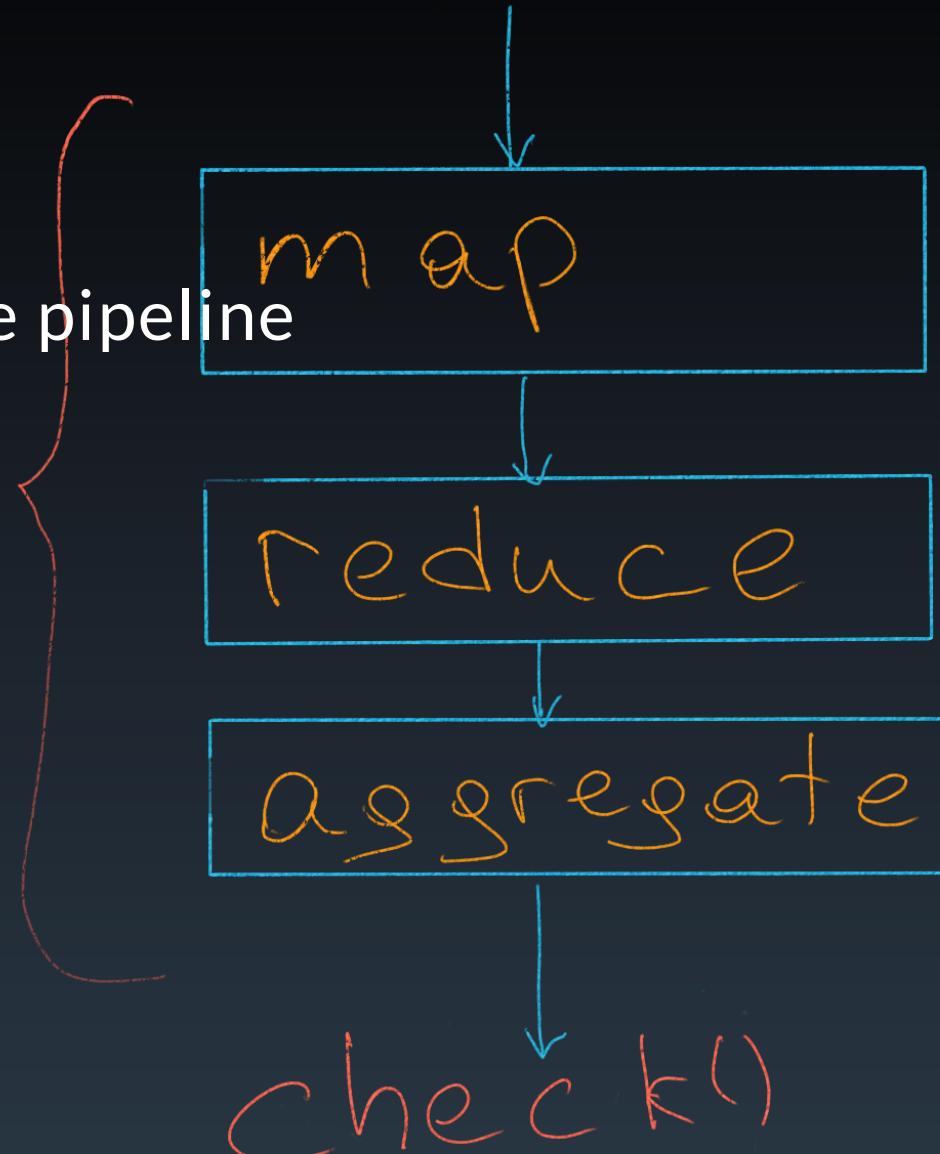
fake data

# Let's paste fakes!

Fake input data

Reference data at the end of the pipeline

Separate  
Function

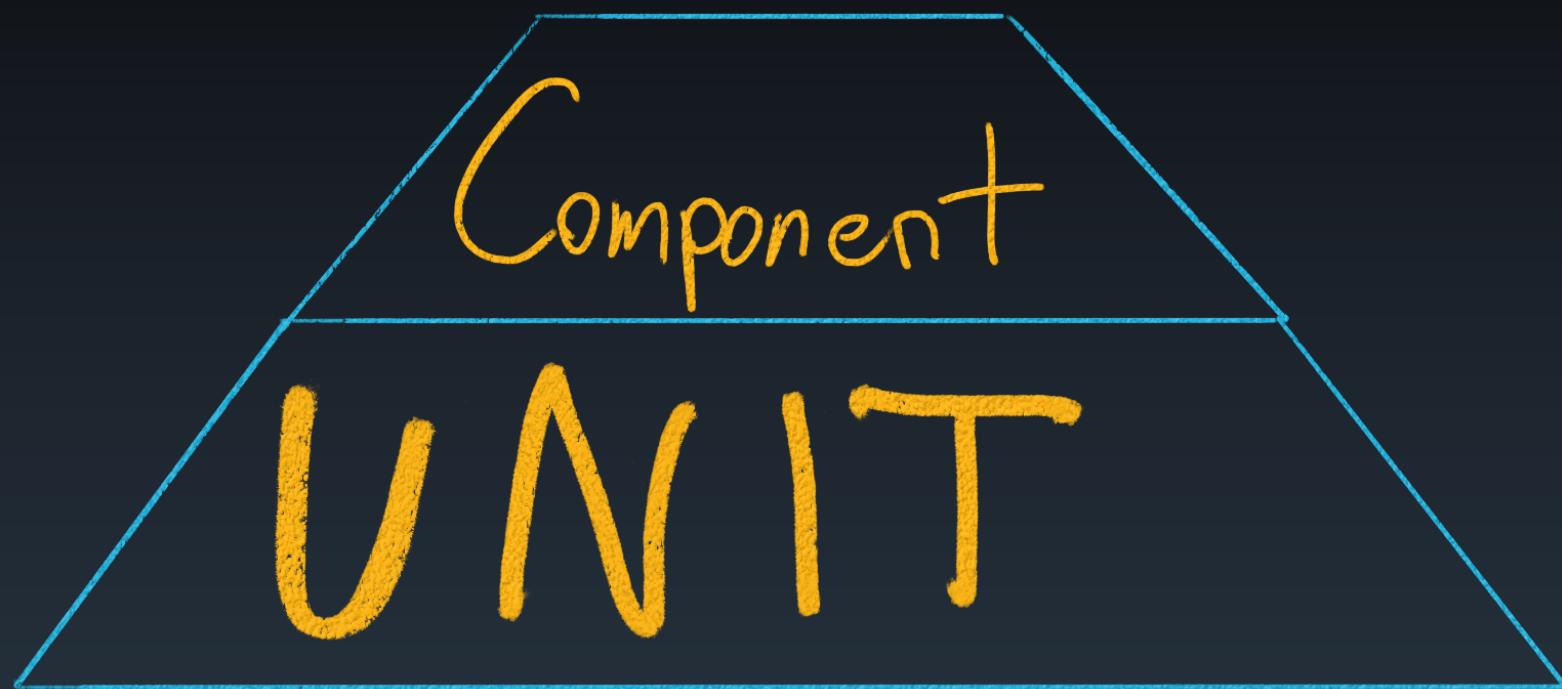


# Tools

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

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

# Component testing

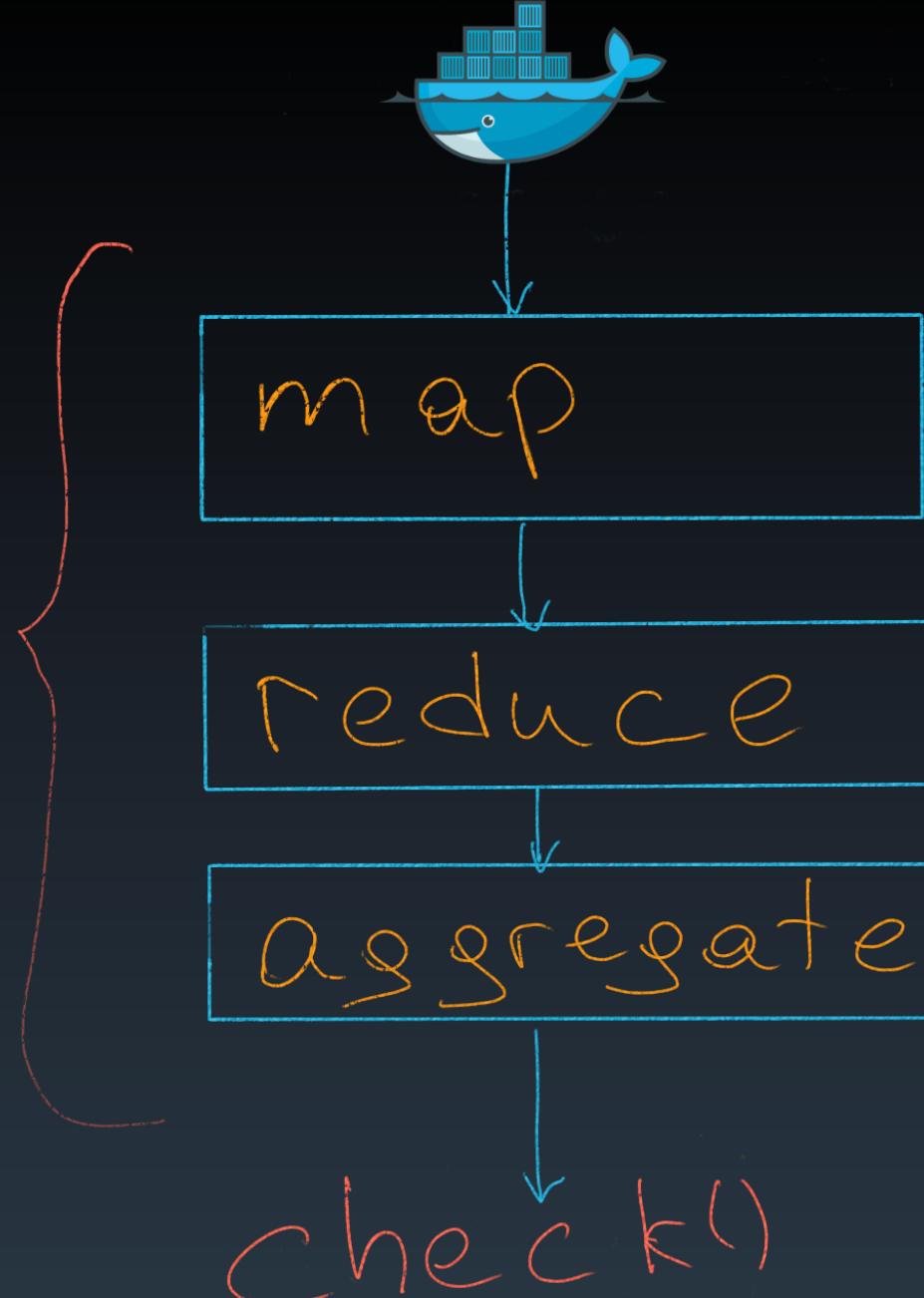




TEST CONTAINERS

# TestContainers

Separate  
Function



# TestContainers

Supported languages:

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

# Test Containers

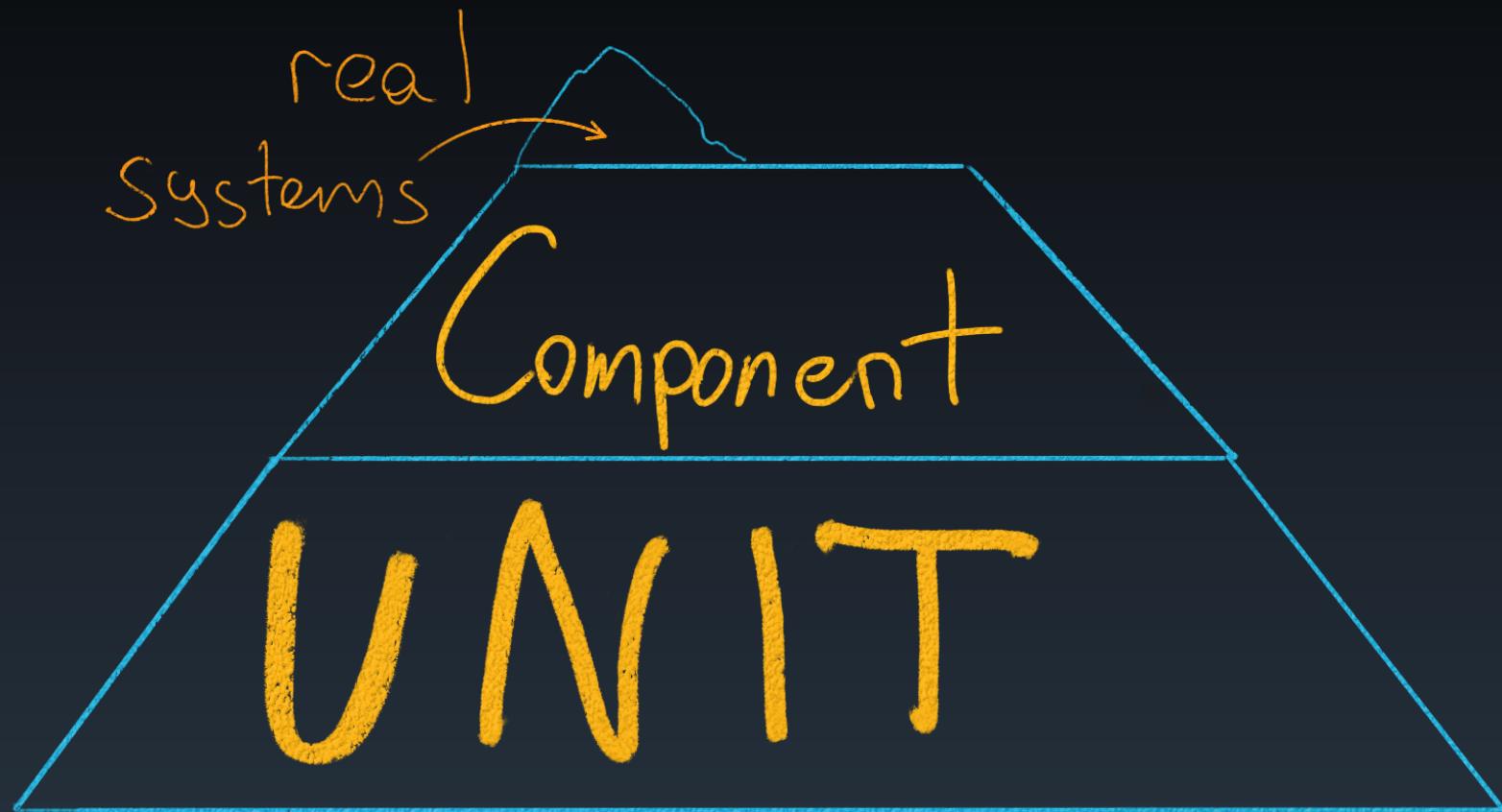
```
1 from testcontainers.postgres import PostgresContainer  
2 import sqlalchemy  
3  
4 postgres_container = PostgresContainer("postgres:9.5")  
5 with postgres_container as postgres:  
6     e = sqlalchemy.create_engine(postgres.get_connection_url())  
7     result = e.execute("select version()")  
8     version, = result.fetchone()  
9 version
```

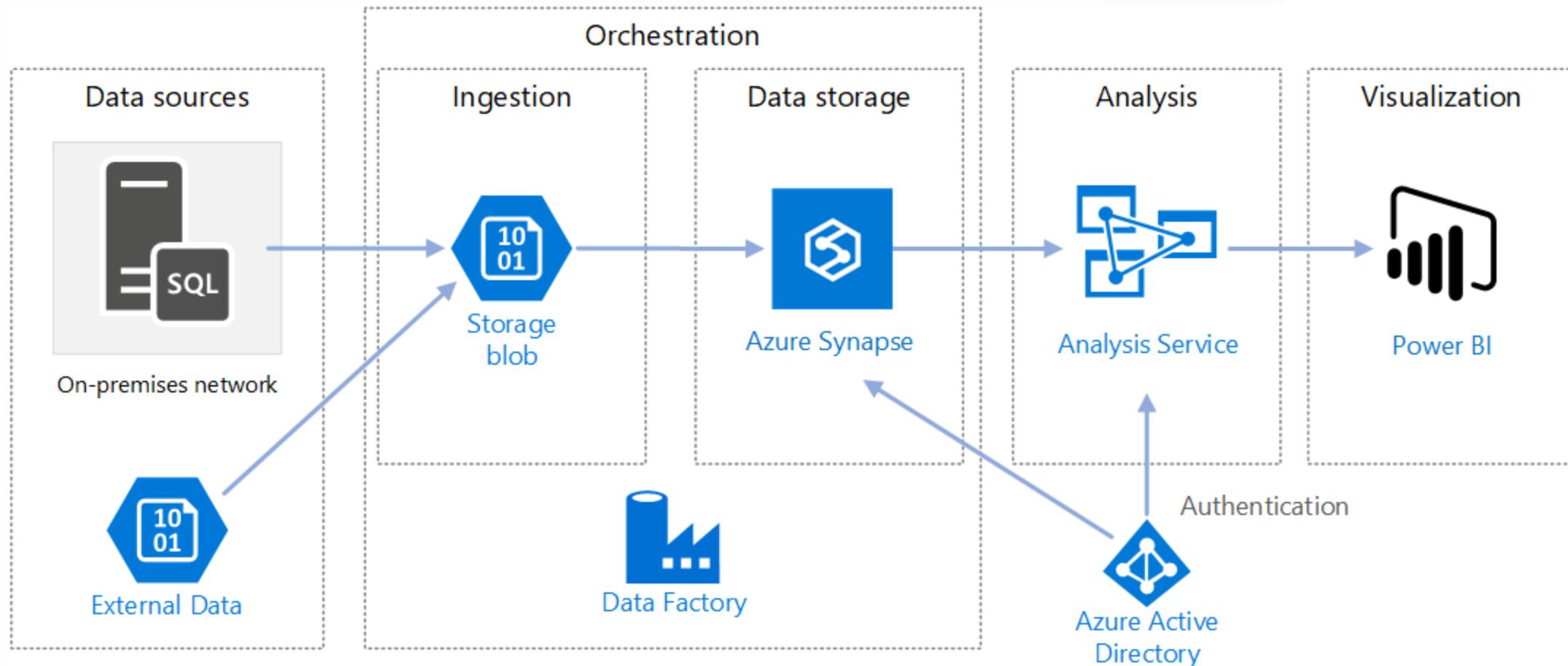
# Test Containers

```
1 from testcontainers.postgres import PostgresContainer  
2 import sqlalchemy  
3  
4 postgres_container = PostgresContainer("postgres:9.5")  
5 with postgres_container as postgres:  
6     e = sqlalchemy.create_engine(postgres.get_connection_url())  
7     result = e.execute("select version()")  
8     version, = result.fetchone()  
9 version
```

# Test Containers

```
1 from testcontainers.postgres import PostgresContainer  
2 import sqlalchemy  
3  
4 postgres_container = PostgresContainer("postgres:9.5")  
5 with postgres_container as postgres:  
6     e = sqlalchemy.create_engine(postgres.get_connection_url())  
7     result = e.execute("select version()")  
8     version, = result.fetchone()  
9 version
```



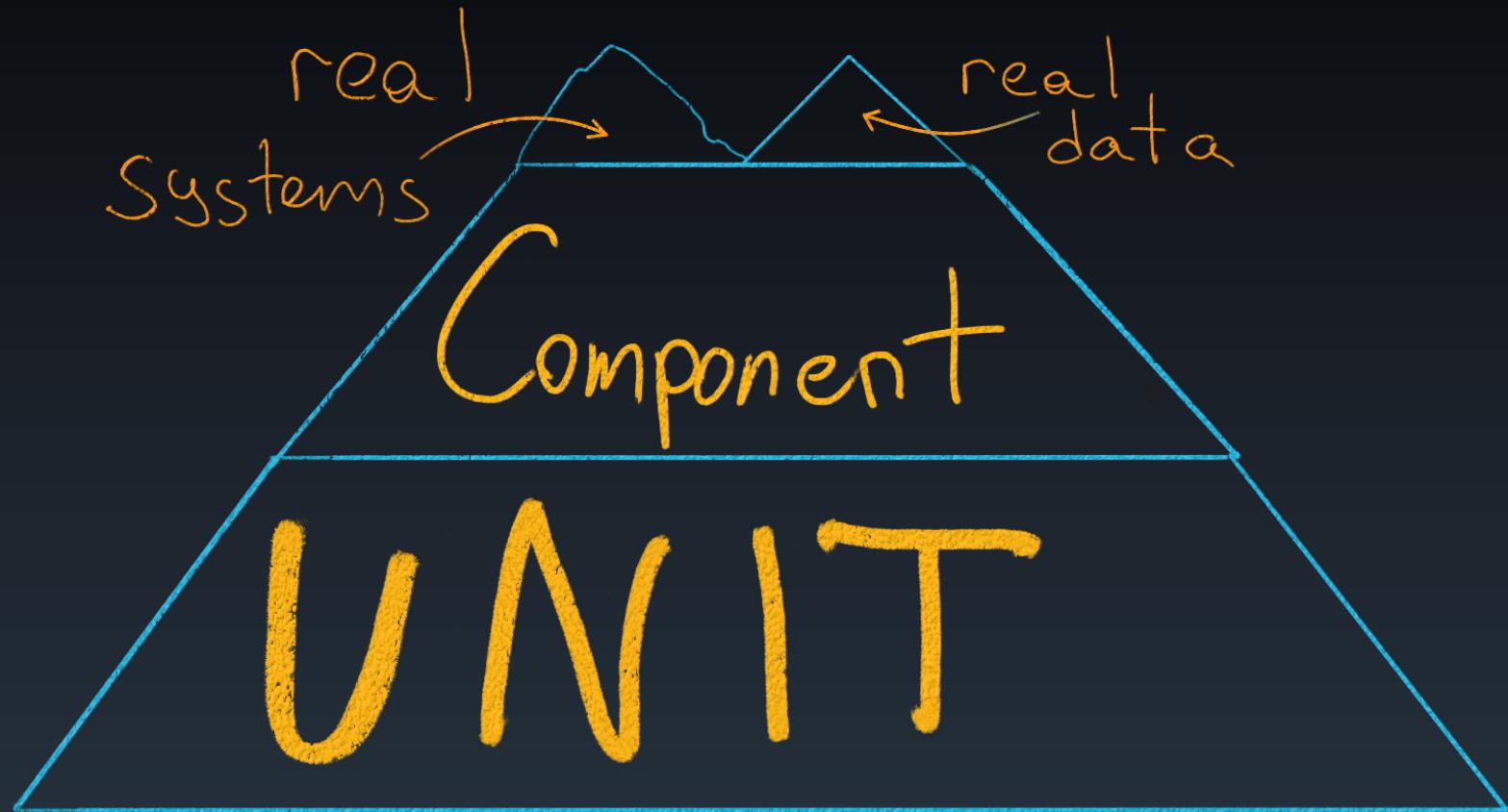


@asm0di0 @if\_no\_then\_yes

# Real systems

Why are component tests not enough?

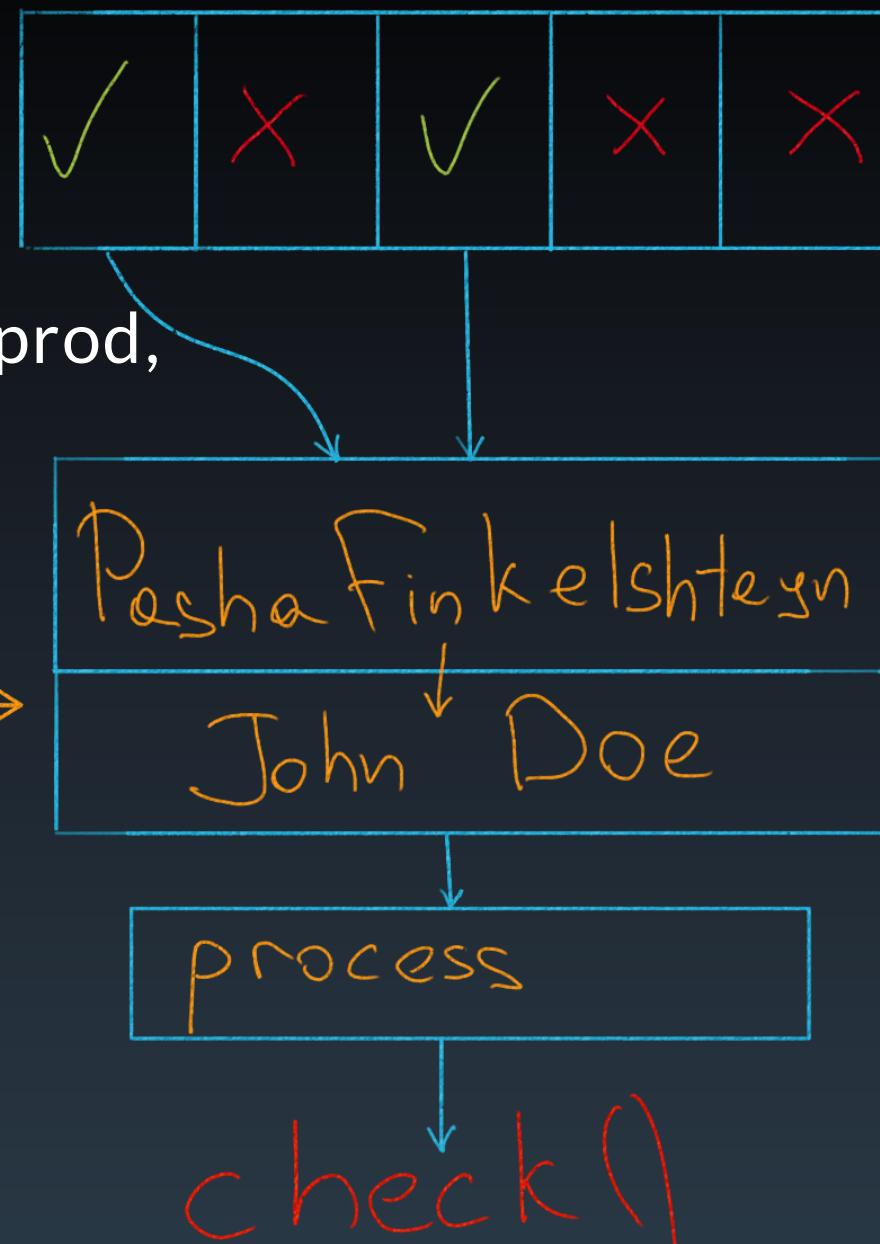
- vendor lock tools (DB, processing, etc.)
- external error handling



# Real data

Get data samples from prod,  
anonymize it

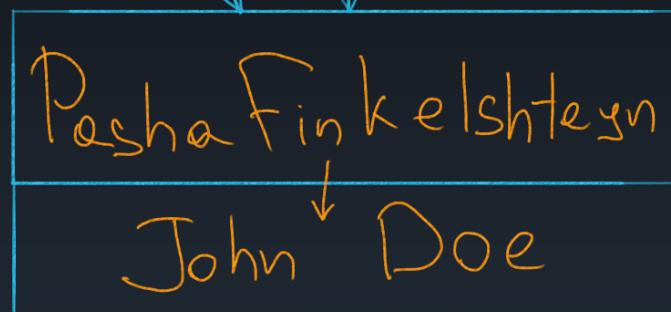
Anonymize →



# Compare to reference

Anonymize →

✓	✗	✓	✗	✗
---	---	---	---	---



process

reference result

+  
Compare  
-

# Comparison example

gender	reference	id	match
m	m	1	TRUE
f	c	2	FALSE
u	u	3	TRUE
c	c	4	TRUE
m	f	5	FALSE

# 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](#)

# Real data expectations

- profilers
- constraint suggestions
- constraint verification
- metrics
- metrics stores

```
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

# 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

```
1 check = Check(spark, CheckLevel.Warning, 'Review Check')
2
3 checkResult = VerificationSuite(spark)
4   .onData(sample_df)
5   .addCheck(check
6     .isComplete('Country')
7     .isContainedIn('Country', ['RU', 'KZ']))
8   .run()
9
10 checkResult_df = VerificationResult.checkResultsAsDataFrame(spark, checkResult)
11 checkResult_df.show()
```

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

[PyDeequ documentation](#)

47

# Python Deequ

```
1 check = Check(spark, CheckLevel.Warning, 'Review Check')
2
3 checkResult = VerificationSuite(spark)
4   .onData(sample_df)
5   .addCheck(check
6     .isComplete('Country')
7     .isContainedIn('Country', ['RU', 'KZ']))
8   .run()
9
10 checkResult_df = VerificationResult.checkResultsAsDataFrame(spark, checkResult)
11 checkResult_df.show()
```

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

[PyDeequ documentation](#)

48

# Python Deequ

```
1 check = Check(spark, CheckLevel.Warning, 'Review Check')
2
3 checkResult = VerificationSuite(spark)
4   .onData(sample_df)
5   .addCheck(check
6     .isComplete('Country')
7     .isContainedIn('Country', ['RU', 'KZ']))
8   .run()
9
10 checkResult_df = VerificationResult.checkResultsAsDataFrame(spark, checkResult)
11 checkResult_df.show()
```

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

[PyDeequ documentation](#)

49

# Python Deequ

```
1 check = Check(spark, CheckLevel.Warning, 'Review Check')
2
3 checkResult = VerificationSuite(spark)
4   .onData(sample_df)
5   .addCheck(check
6     .isComplete('Country')
7     .isContainedIn('Country', ['RU', 'KZ']))
8   .run()
9
10 checkResult_df = VerificationResult.checkResultsAsDataFrame(spark, checkResult)
11 checkResult_df.show()
```

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

[PyDeequ documentation](#)

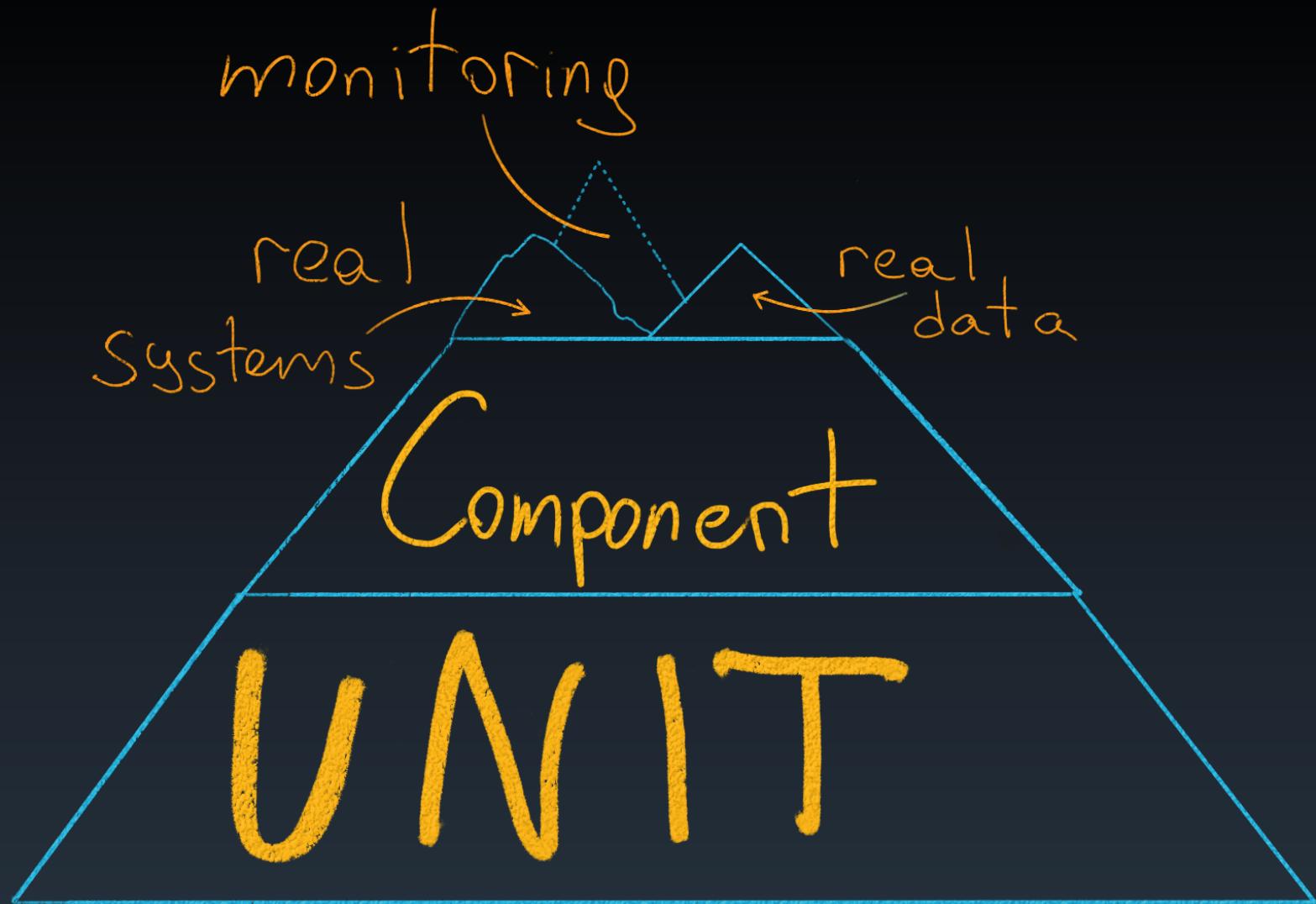
50

# Python Deequ.

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

# Real data expectations. Use cases

- pre-ingestion and post-ingestion data validation
- before pipeline development
- monitoring and alerting



# 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

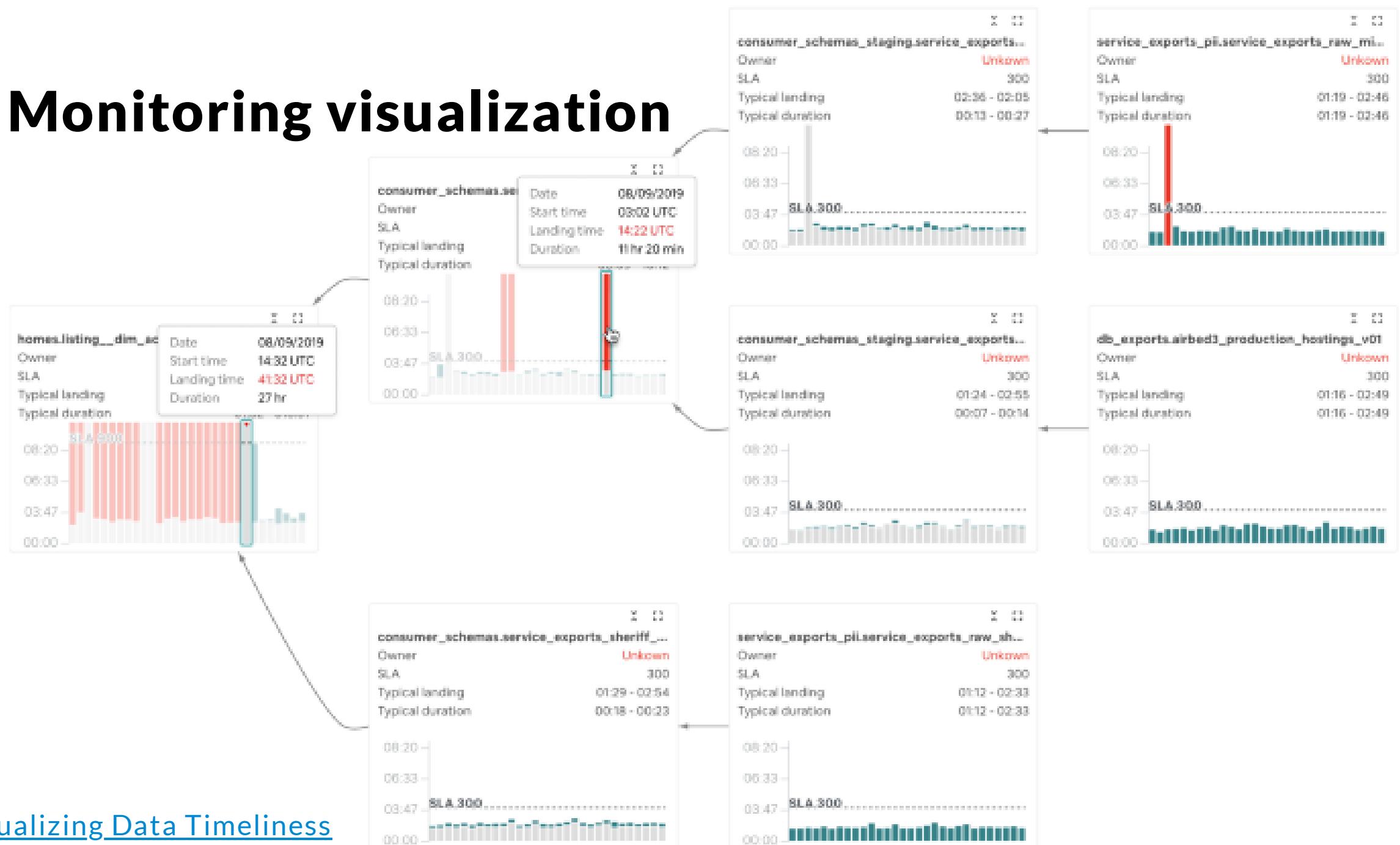
How?

- use Listeners
- use data aggregations
- AirFlow (or another orchestrator)

# Data pipelines is always DAG

Monitoring should visualize it

# Monitoring visualization

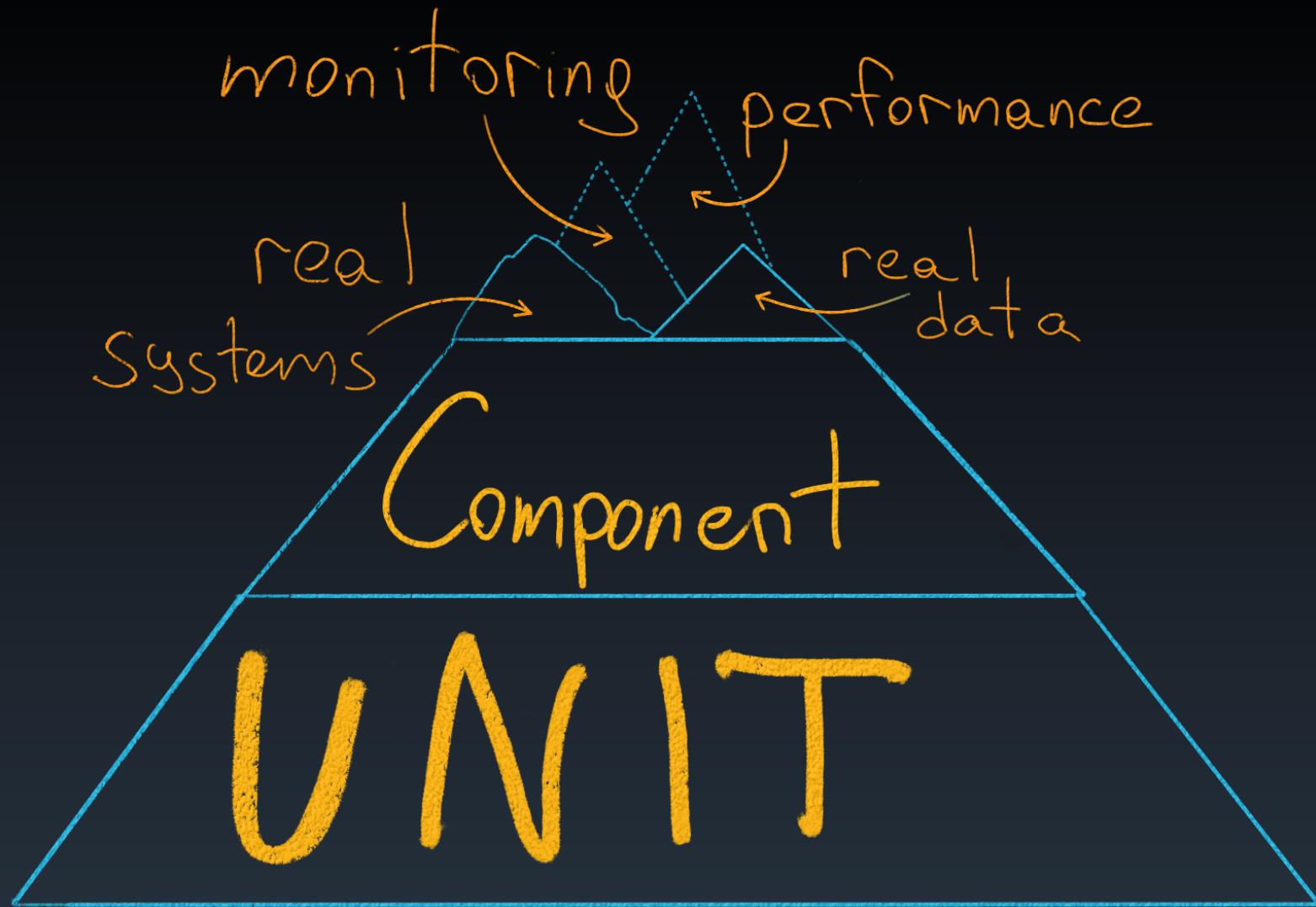


# End-to-End tests

Compare with reports, old DWH

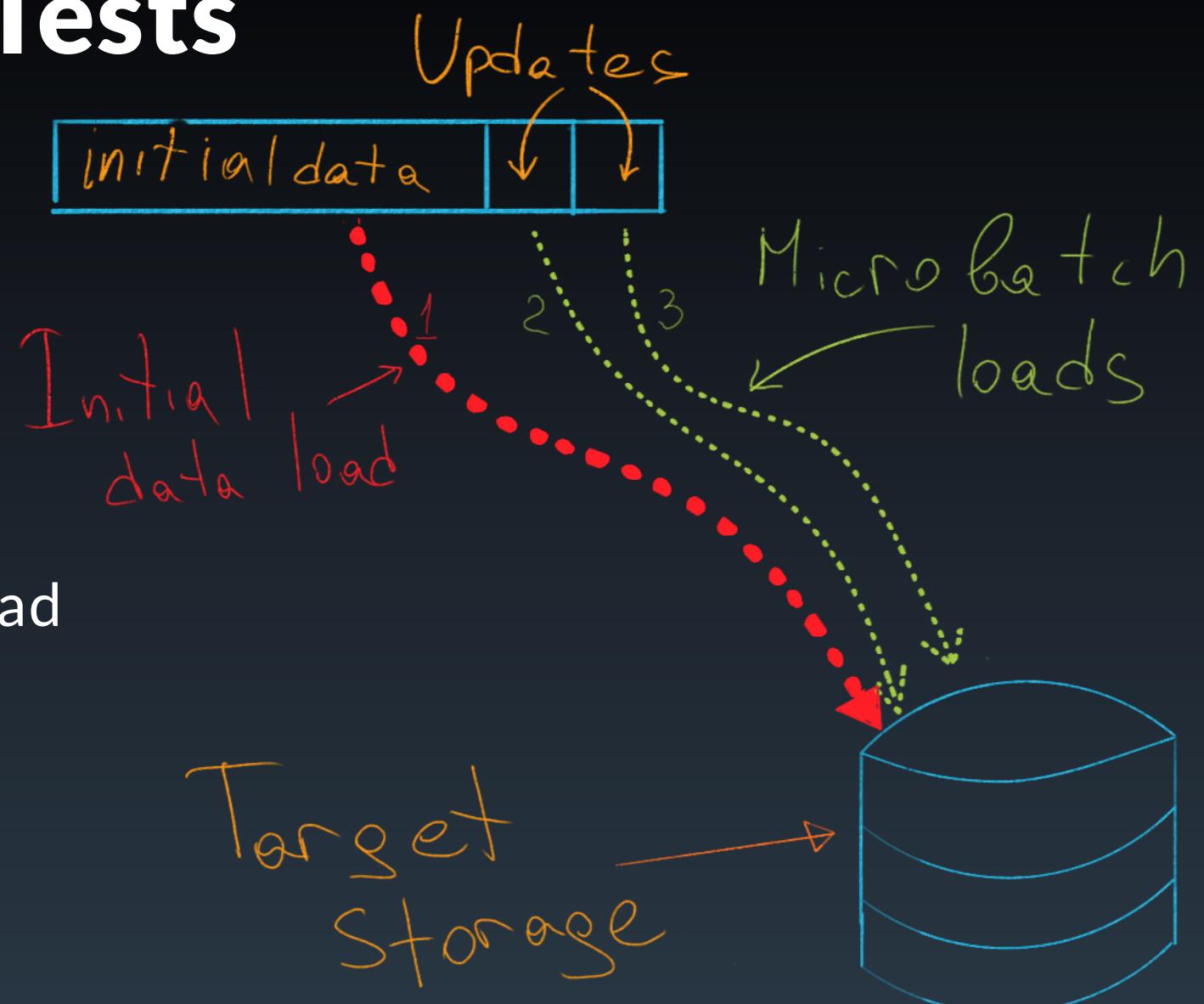
Multiple dimensions:

- data
- data latency
- performance, scalability



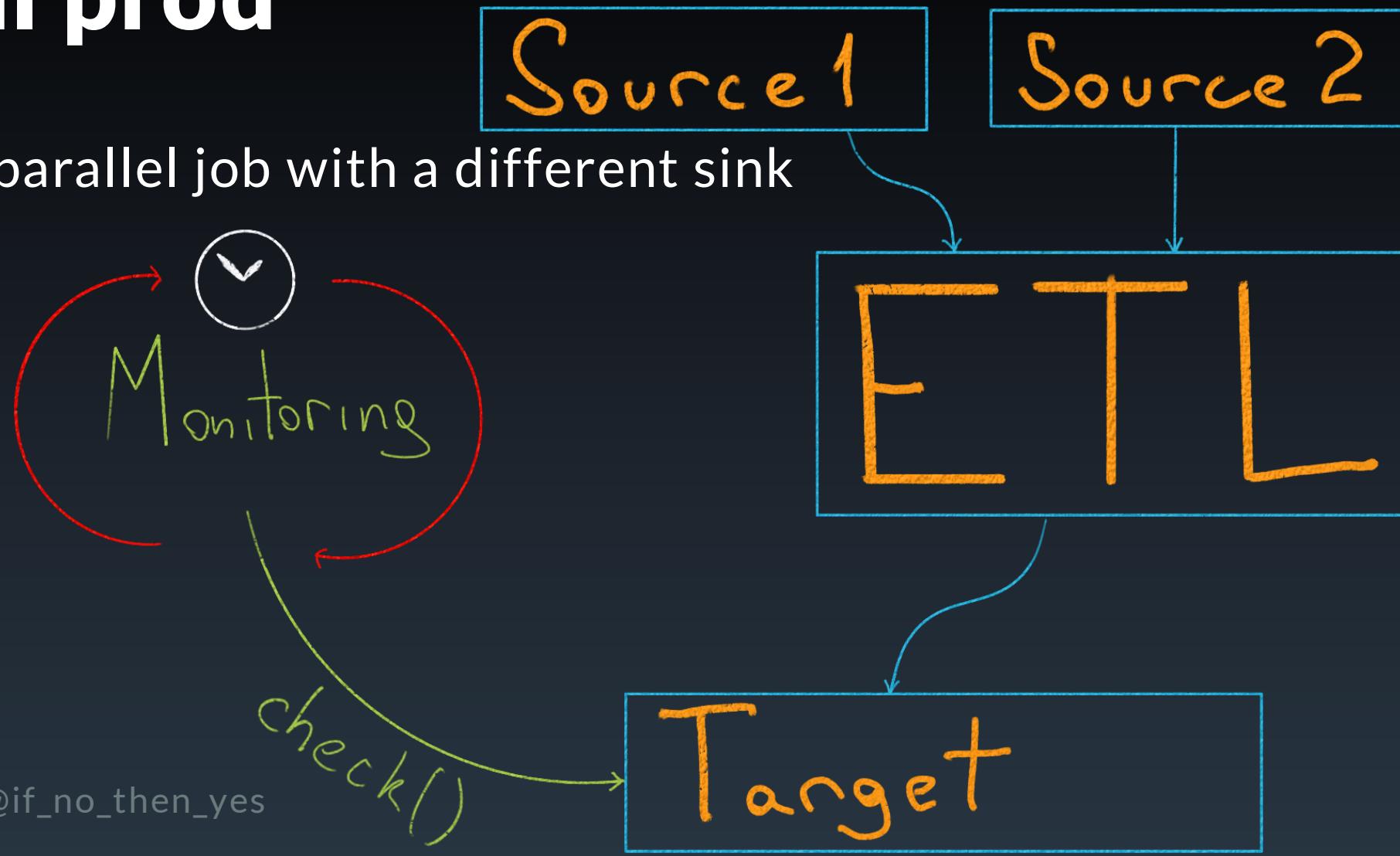
# Performance Tests

- start with SLO
- test your initial data load

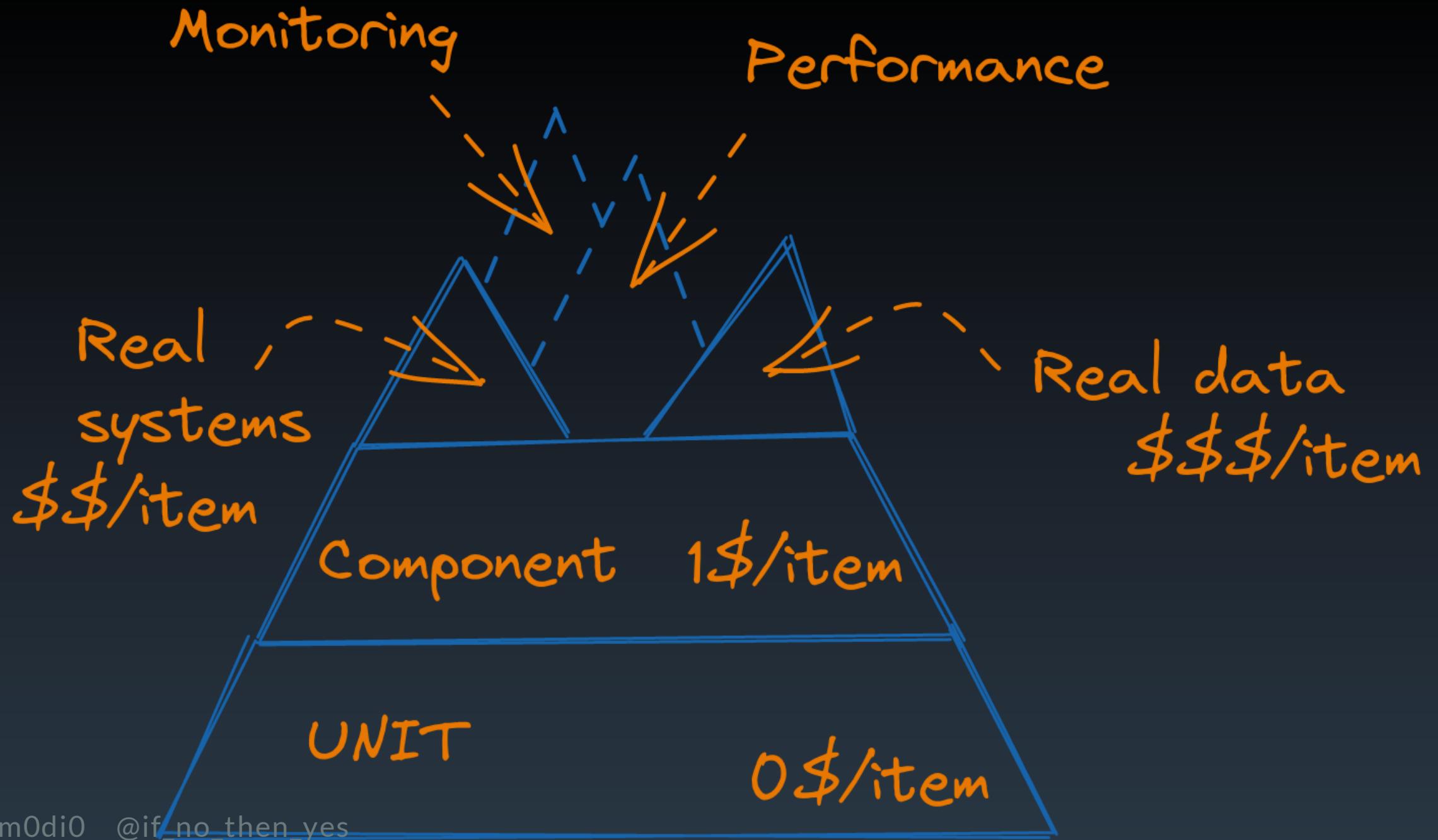


# Real prod

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

 @asm0dey@fosstodon.org