

# Tables as Code

*From Ad-hoc Scripts to Maintainable ETL Workflows at Booking.com*



**Bram** van den Akker



**Jon** Smith



**Maurits** Kroese



**João** Laia



**Alina** Solovjova



**Jeroen** Schmidt



**Niek** Tax

# Data @ Booking.com

**10x** teams working on data

**100x** data practitioners

**1000x** data workflows

**10000x** data assets

In this talk, we tackle a common data design **headache**

**Let me explain.**

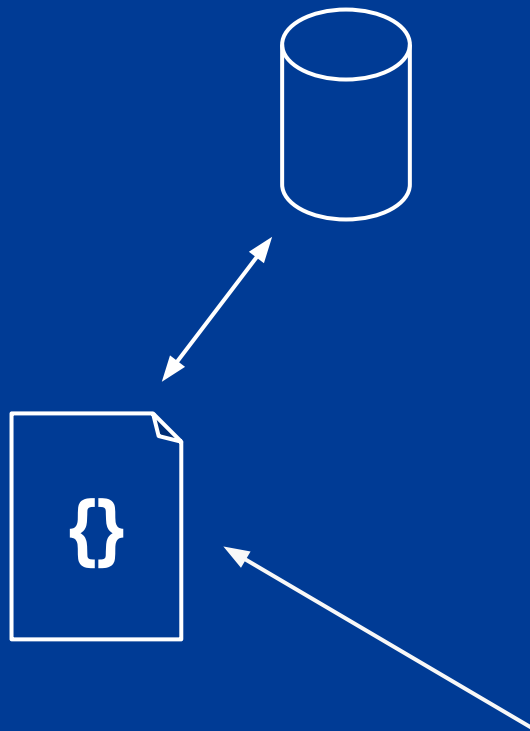


**Data**

S3/HDFS, Snowflake, Hive, Presto...

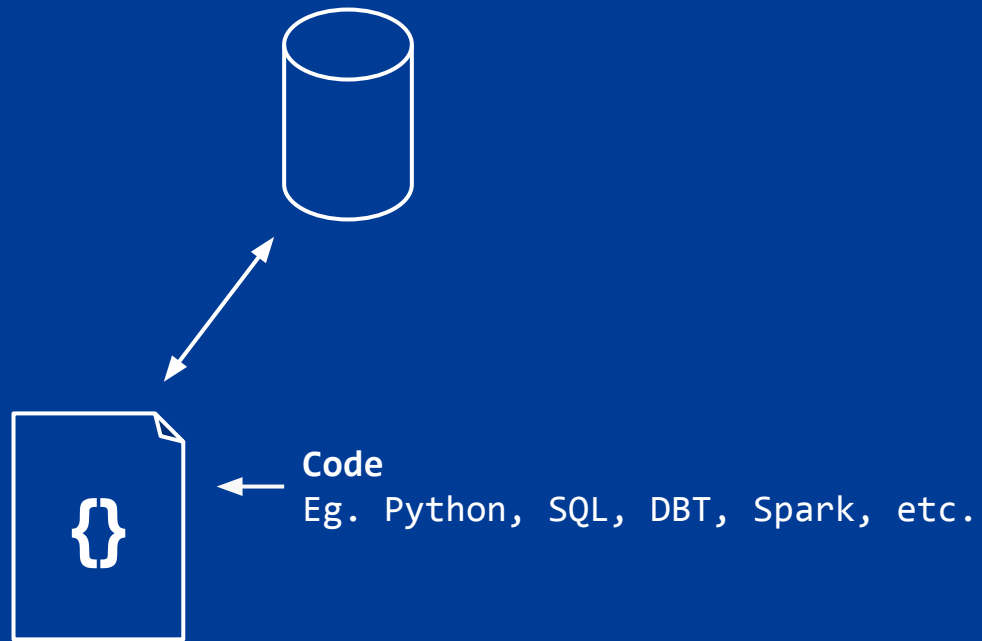


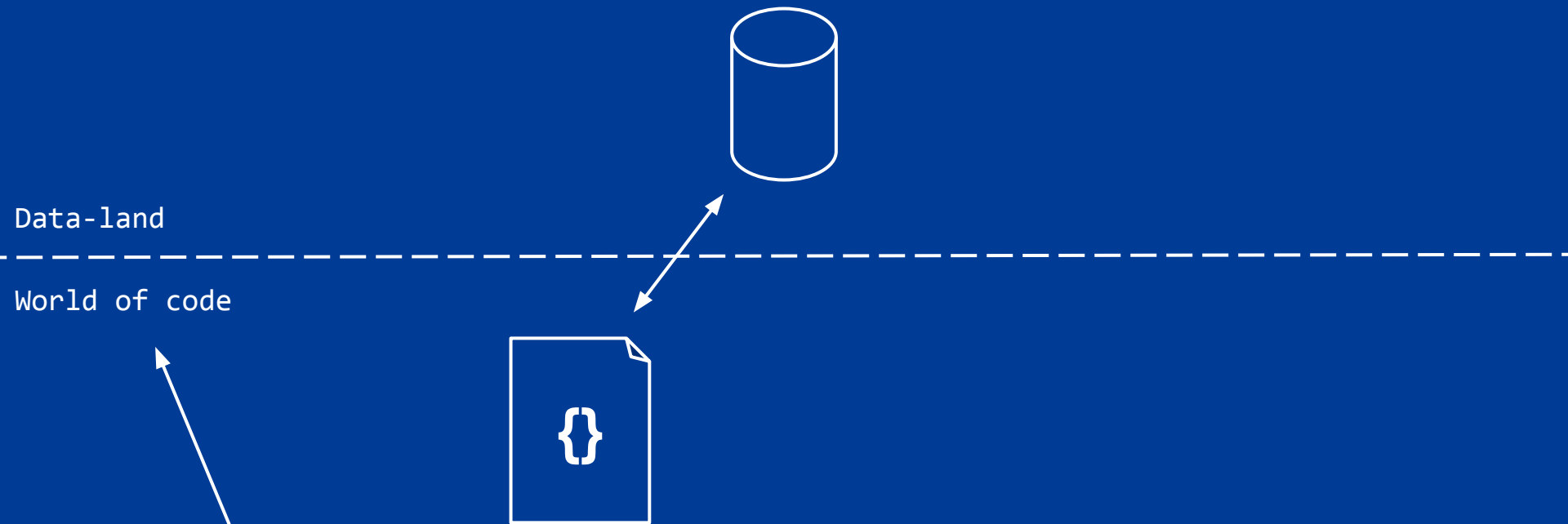
It all starts with our persistent data



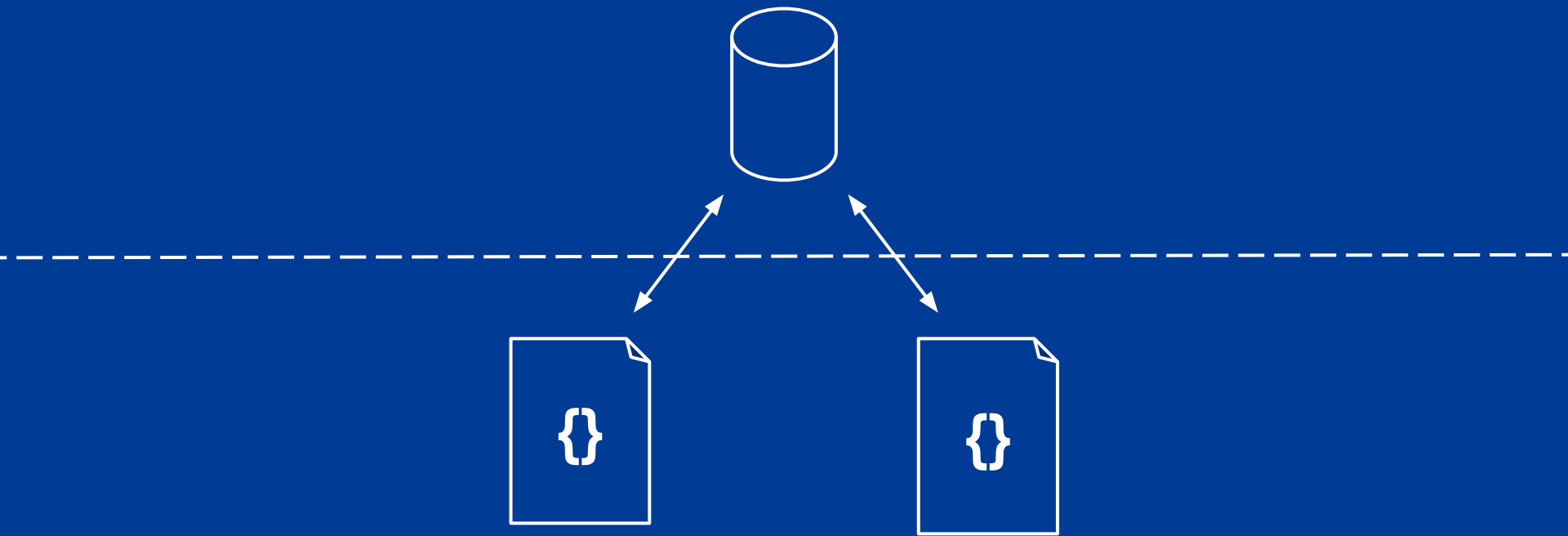
Which is produced and consumed by code







Both the code and data exist at different levels

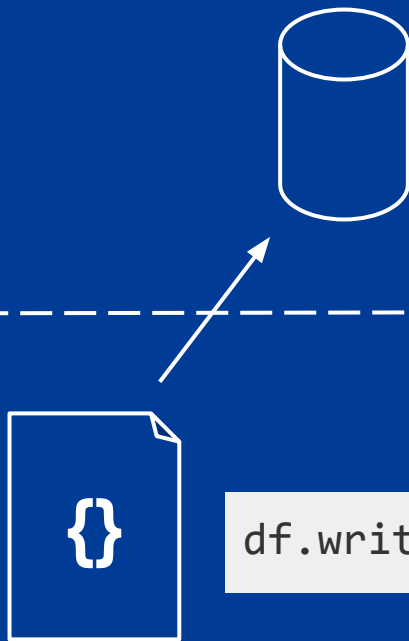


Producers and ...

... communicate through ...

... consumers *only* ...

... the persistent data



```
df.write.saveAsTable("dbimports.hotels")
```

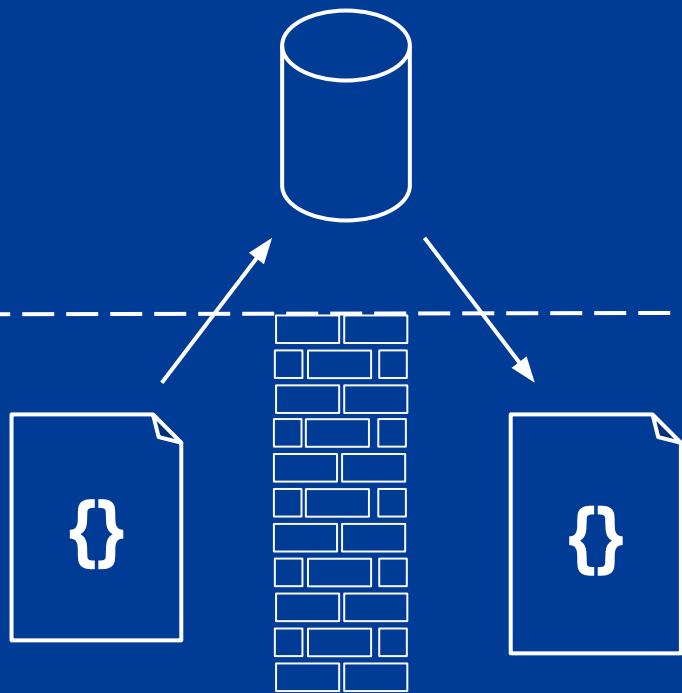
For example, writing to a table in PySpark only requires the string location.



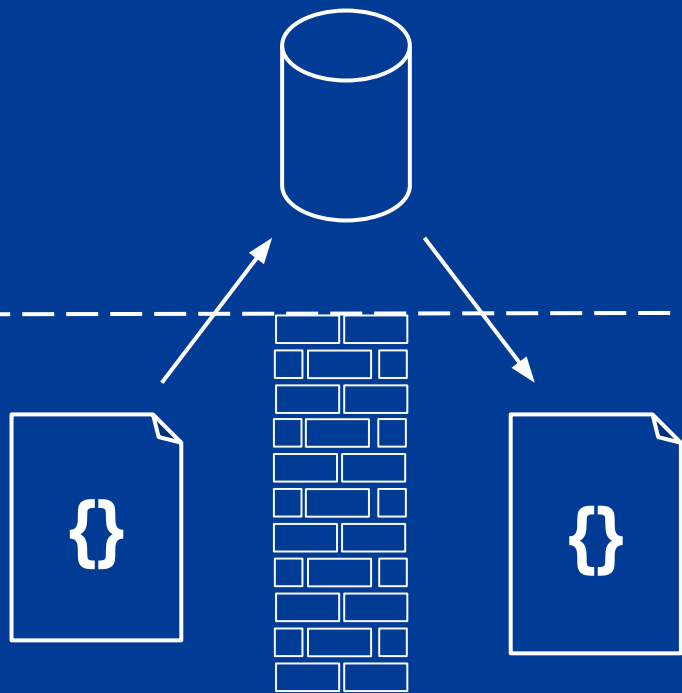
```
df = spark.table("dbimports.hotels")
```



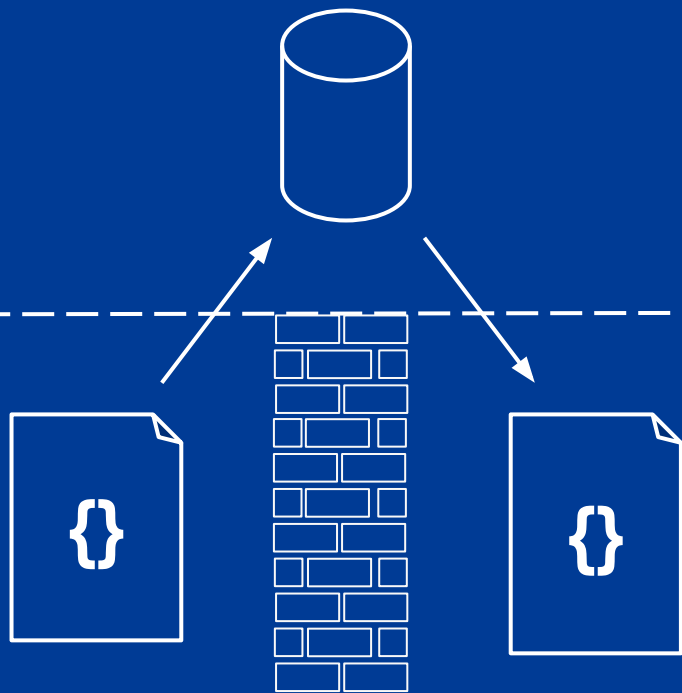
**However, when reading nothing guarantees what comes back.**



As a result, changes are only really tested with the persistent data...



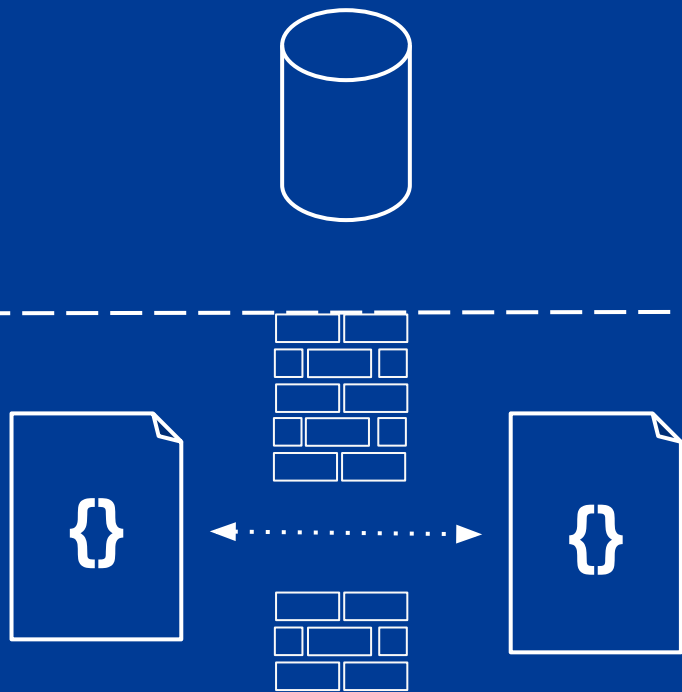
As a result, changes are only really tested with the persistent data...  
...in production.



As a result, changes are only really tested with the persistent data..  
...in production.

#YOLO





What we need is **some contract** between the consumer and producer

Bringing us to the title of this talk

Bringing us to the title of this talk

# Tables as Code

## The structure of this talk:

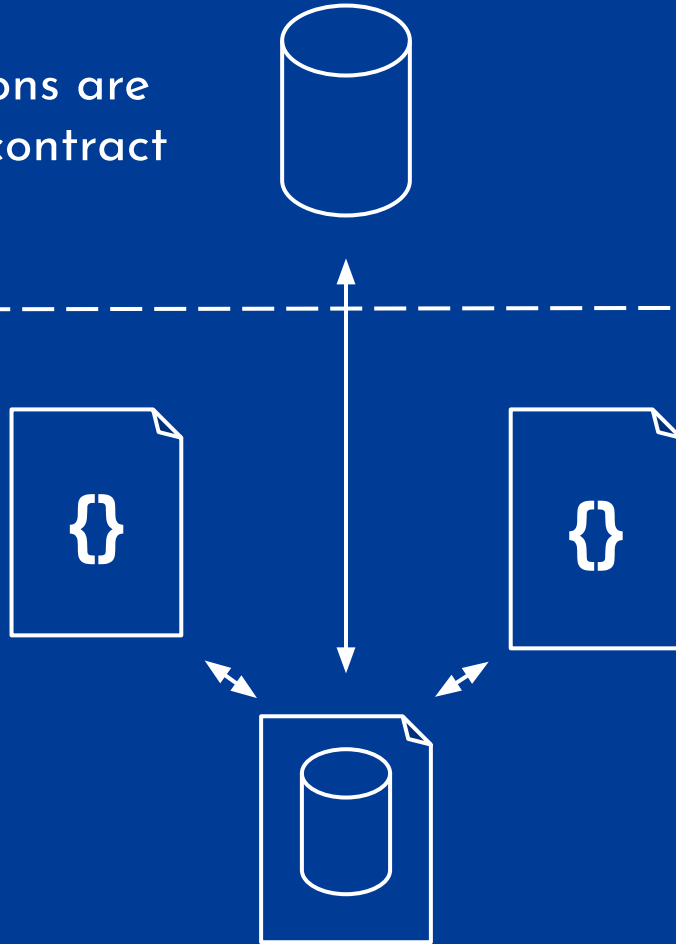
- What are “tables as code”
- Advantages
  - Customization of data access
  - Testing made easy
  - Builds & deployments made easy
  - Scaling builds
- Scaling the community



Tables as Code, are contracts  
in the target language



All read/write operations are  
executed through this contract



```
class hotels(HiveTable):  
    schema = StructType([  
        StructField("hotel_id", LongType()),  
        StructField("is_closed", BooleanType()),  
    ])  
  
    format = "orc"  
    partition_by = ["date"]
```

Each contract is a **class**, containing metadata



```
class hotels(HiveTable):  
    schema = StructType([  
        StructField("hotel_id", LongType()),  
        StructField("is_closed", BooleanType()),  
    ])  
  
    format = "orc"  
    partition_by = ["date"]
```

Each contract is a **class**, containing metadata  
such as the schema





```
class hotels(HiveTable):  
    schema = StructType([  
        StructField("hotel_id", LongType()),  
        StructField("is_closed", BooleanType()),  
    ])
```

Note: This could be coupled with a schema registry (eg. confluence Avro)

```
partition_by = ["date"]
```

Each contract is a **class**, containing metadata  
such as the schema



```
class hotels(HiveTable):  
    schema = StructType([  
        StructField("hotel_id", LongType()),  
        StructField("is_closed", BooleanType()),  
    ])  
  
    format = "orc"  
    partition_by = ["date"]
```


Each contract is a **class**, containing metadata such as the schema, format, and more



As developers, we can now control the interface to the underlying data


```
def write(cls, df, atomic=False):  
    df = cls._check_schema_compatibility(df)  
    cls._validate_partition_by()  
    df = cls._execute_custom_hooks(df)  
    cls._write(df, atomic=atomic)
```

For example, we can run schema validation



```
def write(cls, df, atomic=False):  
    df = cls._check_schema_compatibility(df)  
    cls._validate_partition_by()  
    df = cls._execute_custom_hooks(df)  
    cls._write(df, atomic=atomic)
```

```
def write(cls, df, atomic=False):  
    df = cls._check_schema_compatibility(df)  
    cls._validate_partition_by()  
    df = cls._execute_custom_hooks(df)  
    cls._write(df, atomic=atomic)
```



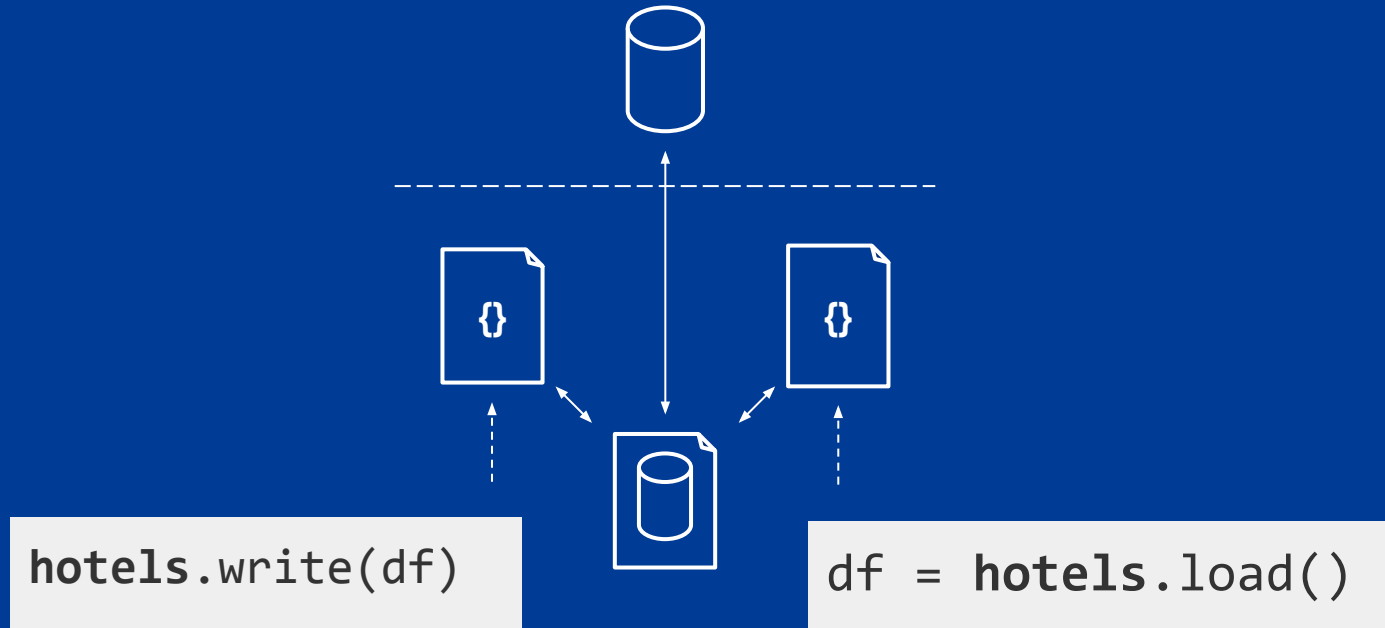
**Or any other arbitrary methods, both custom and predefined**

These types of methods can also include

- In-code data validation
- Source/target redirects
  - eg. writing staging output to personal schemas
- Custom write strategies
  - eg. atomic writes, incremental inserts

```
from tables.hive import hotels
```

Instead of relying on materialization of the data, we rely on a shared code representation of the table



And this object, can now become the interface to the persisted data.



All of these changes might sound somewhat interesting.

However, the real benefits come with...

# The Monorepo

An orange clapperboard graphic is positioned on the left side of the slide. It consists of a rectangular body and a hinged top bar that is tilted upwards at an angle. The text is centered within the rectangular body.

# Monorepo vs. Polyrepo Debate

# Monorepo vs. Polyrepo Debate

For now...



A diagram on a blue background showing three orange rounded rectangles. Two are at the top, labeled 'Local Testing' and 'Continuous Integration'. A third is centered below them, labeled 'Bazel and Test Caching'. All three are enclosed in a white dashed rounded rectangle.

Local Testing

Continuous  
Integration

Bazel and Test  
Caching

*The Monorepo*



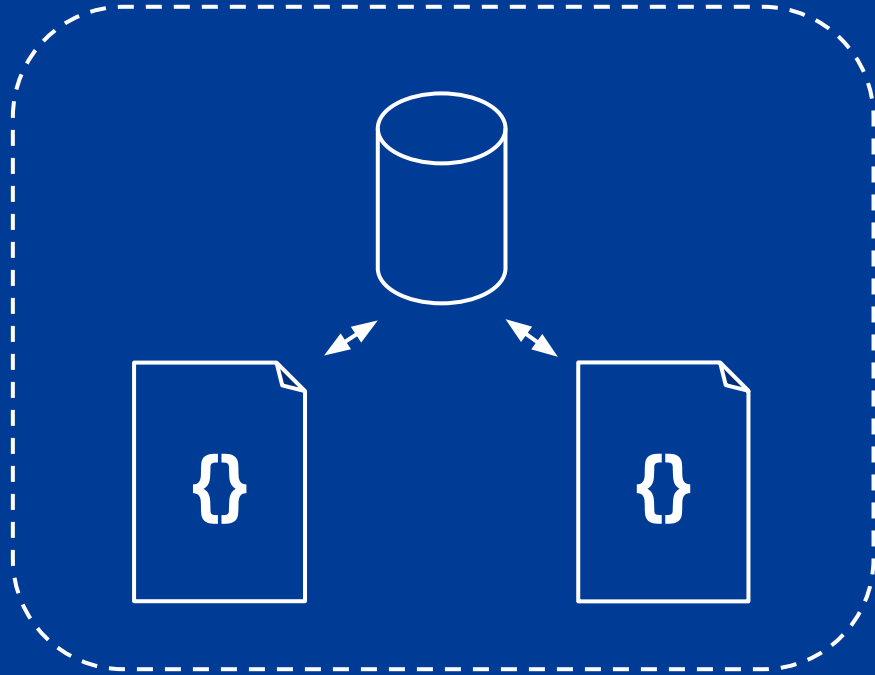
A diagram titled "The Monorepo" showing three components within a dashed white rounded rectangle on a blue background. The components are: "Local Testing" in an orange box, "Continuous Integration" in a dark blue box, and "Bazel and Test Caching" in a dark blue box.

Local Testing

Continuous  
Integration

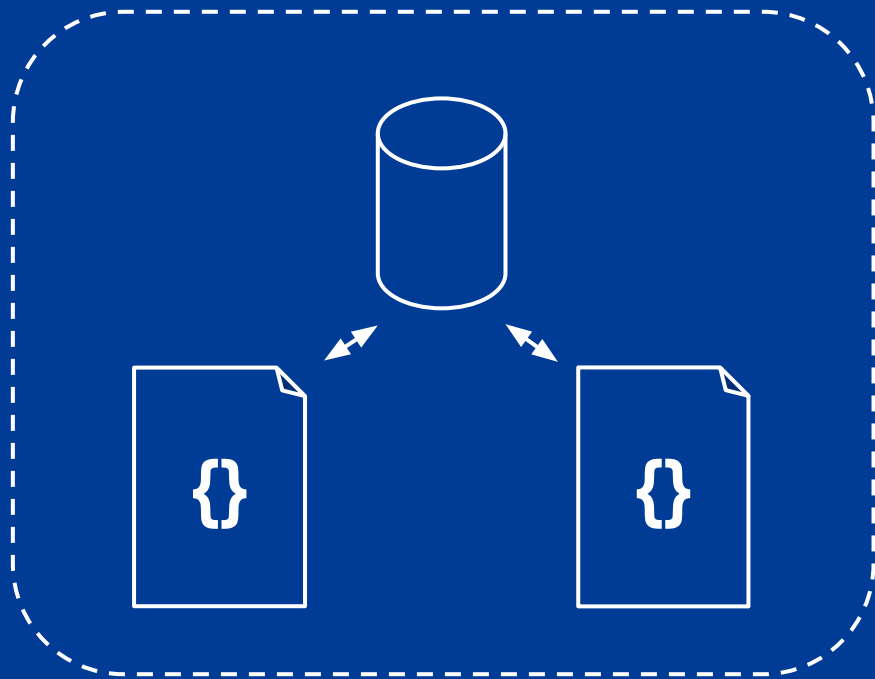
Bazel and Test  
Caching

*The Monorepo*



Validate code/data contract...

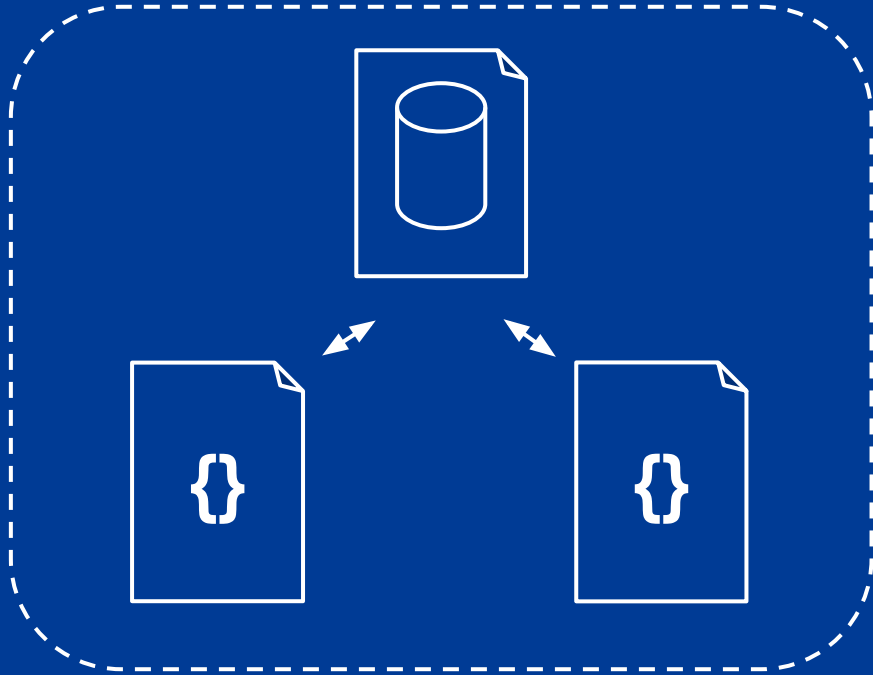




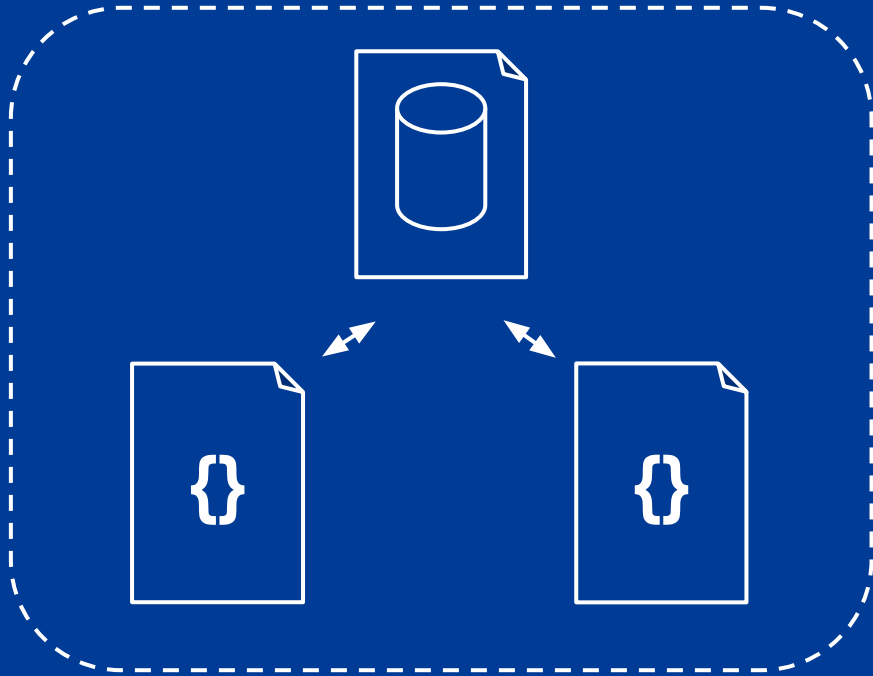
Validate code/data contract  
by testing in prod



Surprise 'ColumnNotFound'  
exception after 45min



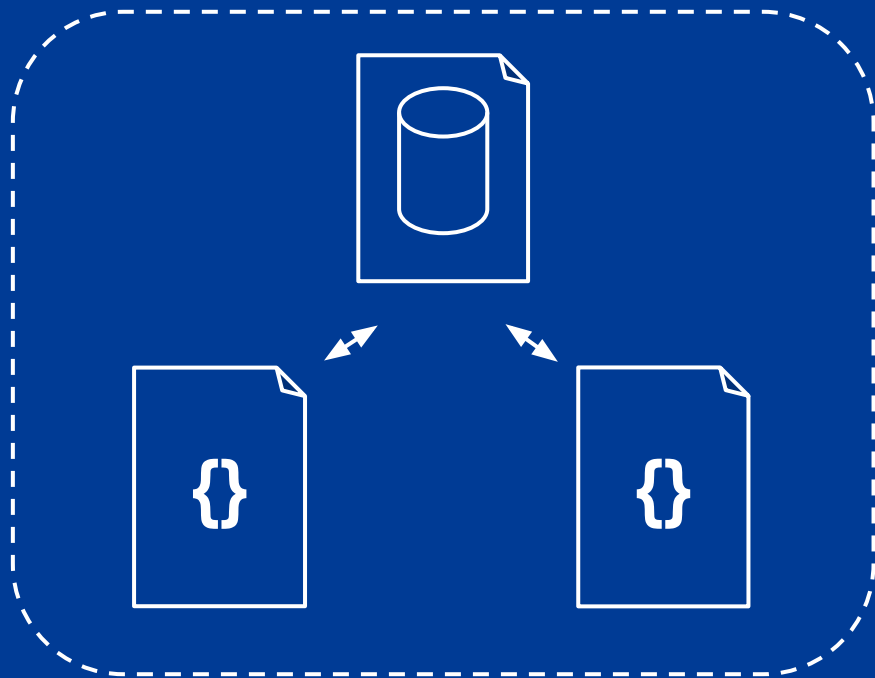
Validate code/data contract  
*locally*



Validate code/data contract  
*locally*



No surprise exceptions in  
prod



Validate code/data contract  
*locally*



No surprise exceptions in  
prod



(Faster development cycle)

# Testing

de/data contract

exceptions in

(Faster development cycle)

# Testing

Setting up  
local Spark

# Testing

Setting up  
local Spark

# Testing

Creating local  
database



Setting up  
local Spark

Creating mock  
data

# Testing

Creating local  
database

Setting up  
local Spark

Creating mock  
data

# Testing

Mocking  
interfaces

Creating local  
database

Setting up  
local Spark

Creating mock  
data

Setup /  
teardown

# Testing

Mocking  
interfaces

Creating local  
database

Setting up  
local Spaw

mock

Setup /  
teardown

# Handled

Mocking  
interfaces

```
from tables.hive import hotels
```

```
def test_process_hotels(hive_database):  
    hive_database.create_table_if_not_exists(hotels)  
    ...
```

```
from tables.hive import hotels
```

```
def test_process_hotels(hive_database):
```

```
    hive_database.create_table_if_not_exists(hotels)
```

```
    ...
```

```
from tables.hive import hotels

def test_process_hotels(hive_database):
    hive_database.create_table_if_not_exists(hotels)
    ...
```

- pytest fixture, globally available
- local Hive, erased after every test

```
from tables.hive import hotels
```

```
def test_process_hotels(hive_database):
```

```
    hive_database.create_table_if_not_exists(hotels)
```

```
    ...
```



```
from tables.hive import hotels

def test_process_hotels(hive_database):
    hive_database.create_table_if_not_exists(hotels)
    ...
```

No need to specify schema or metadata

```
from tables.hive import hotels

def test_process_hotels(hive_database):
    hive_database.create_table_if_not_exists(hotels)
    ...
```

No need to specify schema or metadata

Reduced chance of typos

```
from tables.hive import hotels, available_rooms
```

```
def process_hotel_pipeline():
```

```
    hotel_df = hotels.load()
```

Let's say we've got an ETL pipeline function

```
from tables.hive import hotels
```

```
from pipelines import process_hotel_pipeline
```

```
def test_execution_plan(hive_database):
```

```
    hive_database.create_table_if_not_exists(hotels)
```

```
    process_hotel_pipeline()
```

```
from tables.hive import hotels, available_rooms

def process_hotel_pipeline():
    hotel_df = hotels.load()
    available_rooms_df = get_available_rooms(hotel_df)
    available_rooms.write(available_rooms_df)
```

---

```
from tables.hive import hotels
from pipelines import process_hotel_pipeline

def test_execution_plan(hive_database):
    hive_database.create_table_if_not_exists(hotels)
    process_hotel_pipeline()
```

```
from tables.hive import hotels, available_rooms
```

```
def p
```

This function accesses a `num_rooms` column...

```
hotel_df = hotels.load()
```

```
available_rooms_df = get_available_rooms(hotel_df)
```

```
available_rooms.write(available_rooms_df)
```

```
from tables.hive import hotels
```

```
from pipelines import process_hotel_pipeline
```

```
def test_execution_plan(hive_database):
```

```
    hive_database.create_table_if_not_exists(hotels)
```

```
    process_hotel_pipeline()
```

```
from tables.hive import hotels, available_rooms

def process_hotel_pipeline():
    hotel_df = hotels.load()
    available_rooms_df = get_available_rooms(hotel_df)
    available_rooms.write(available_rooms_df)
```

```
from tables.hive import hotels
from tables.hive import available_rooms
```

Create empty table with predefined schema...

```
def test_execution_plan(hive_database):
    hive_database.create_table_if_not_exists(hotels)
    process_hotel_pipeline()
```

```
from tables.hive import hotels
```

```
from tables.hive.schemas import available_rooms
```

*Surprisingly powerful test!*

```
def process_hotel_pipeline():
```

```
    hotel_df = hotels.load()
```

```
    available_rooms_df = get_available_rooms(hotel_df)
```

```
    available_rooms.write(available_rooms_df)
```

```
from tables.hive import hotels
```

```
from pipelines import process_hotel_pipeline
```

```
def test_execution_plan(hive_database):
```

```
    hive_database.create_table_if_not_exists(hotels)
```

```
    process_hotel_pipeline()
```

```
from tables.hive import hotels
```

```
from tables.hive.reports import available_rooms
```

*Surprisingly powerful test!*

```
def process_hotel_pipeline():
```

Fails if table has no `num\_rooms` column

```
    available_rooms.write(available_rooms_df)
```

```
from tables.hive import hotels
```

```
from pipelines import process_hotel_pipeline
```

```
def test_execution_plan(hive_database):
```

```
    hive_database.create_table_if_not_exists(hotels)
```

```
    process_hotel_pipeline()
```



```
from tables.hive import hotels
```

```
from tables.hive.rooms import available_rooms
```

*Surprisingly powerful test!*

```
def process_hotel_pipeline():
```

Fails if table has no `num\_rooms` column

No data needed!

```
from tables.hive import hotels
```

```
from pipelines import process_hotel_pipeline
```

```
def test_execution_plan(hive_database):
```

```
    hive_database.create_table_if_not_exists(hotels)
```

```
    process_hotel_pipeline()
```

```
from tables.hive import hotels
from pyspark.sql import Row

def test_process_hotels_handles_closed_hotel(hive_database):
    hive_database.write([Row(is_closed=1)], hotels)
    ...
```

```
from tables.hive import hotels
from pyspark.sql import Row

def test_process_hotels_handles_closed_hotel(hive_database):
    hive_database.write([Row(is_closed=1)], hotels)
    ...
```

- write custom data to table
- don't need to write every column

A diagram showing three components of a monorepo workflow: 'Local Testing' (dark blue), 'Continuous Integration' (orange), and 'Bazel and Test Caching' (dark blue). These components are enclosed in a dashed white rounded rectangle. The 'Continuous Integration' box is highlighted in orange, while the others are dark blue. The entire diagram is set against a solid blue background.

Local Testing

Continuous  
Integration

Bazel and Test  
Caching

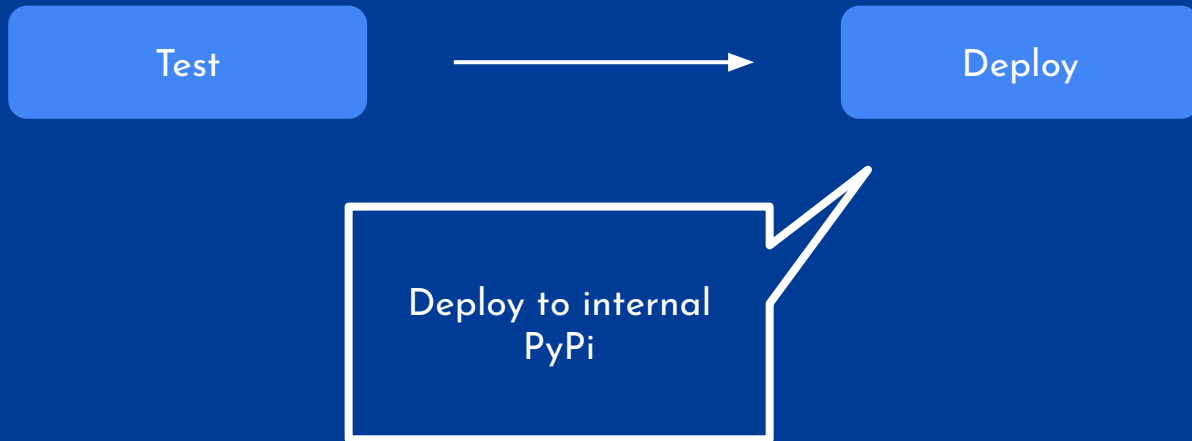
*The Monorepo*

Test

- Unit tests
- Integration tests
- Coverage check



GitLab



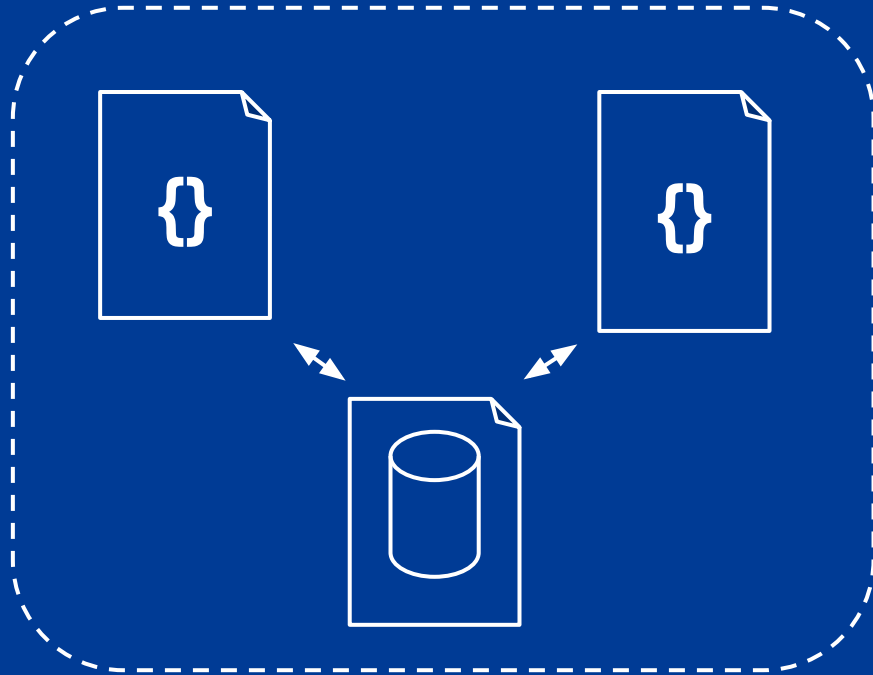


all tests must pass before merge



100% (line) test coverage

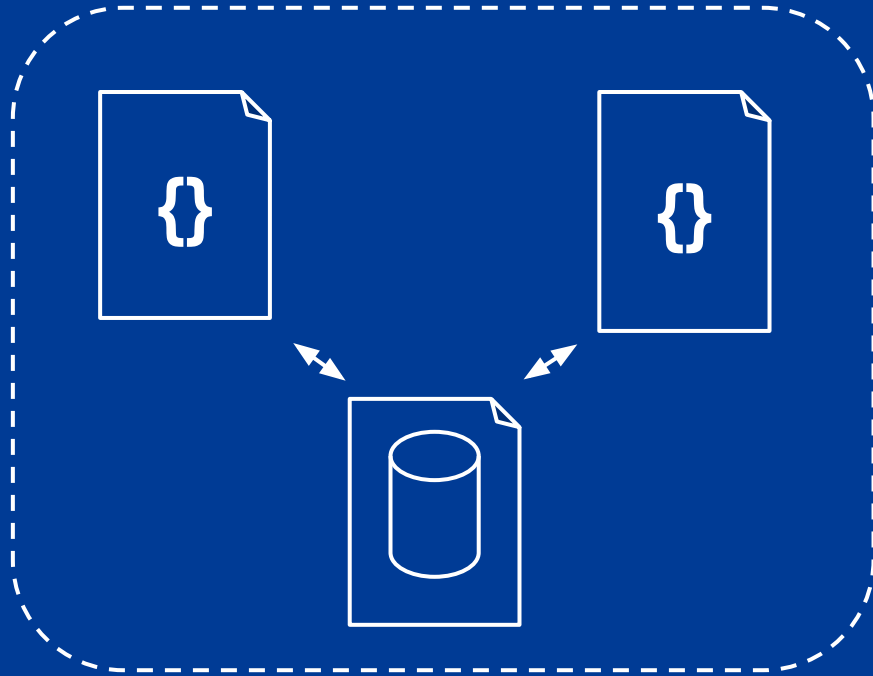




100% (line) test coverage



Code/data contract validated



100% (line) test coverage



Code/data contract validated



Friday afternoon deployments



A diagram illustrating the components of a Monorepo. It features a large dashed white rounded rectangle on a blue background. Inside this rectangle, there are three rounded rectangular boxes. At the top, there are two dark blue boxes: 'Local Testing' on the left and 'Continuous Integration' on the right. Below these, centered, is a single orange box labeled 'Bazel and Test Caching'.

Local Testing

Continuous  
Integration

Bazel and Test  
Caching

*The Monorepo*

Test



Deploy

- 1000s of tests, big and small
- Can take >1hr on one machine

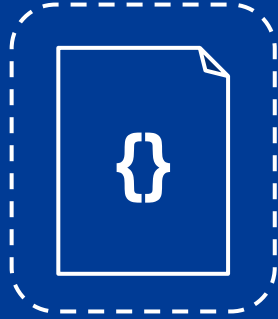
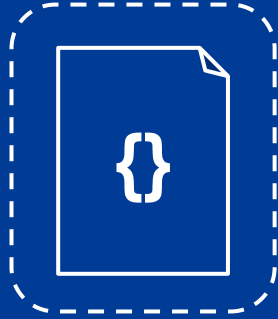
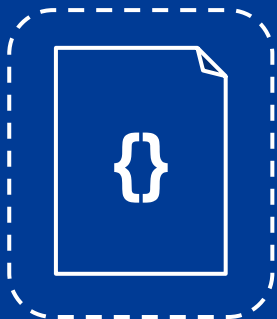
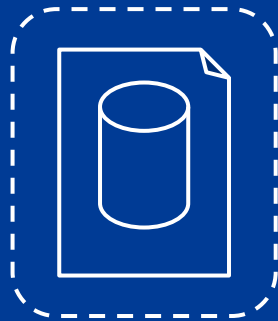
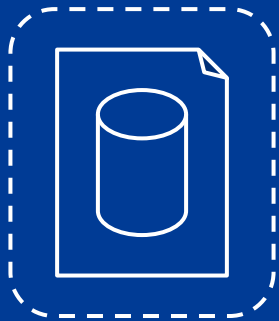
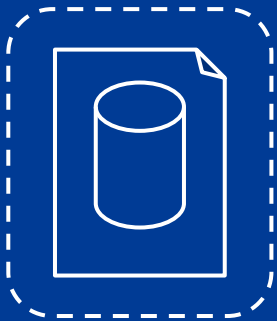


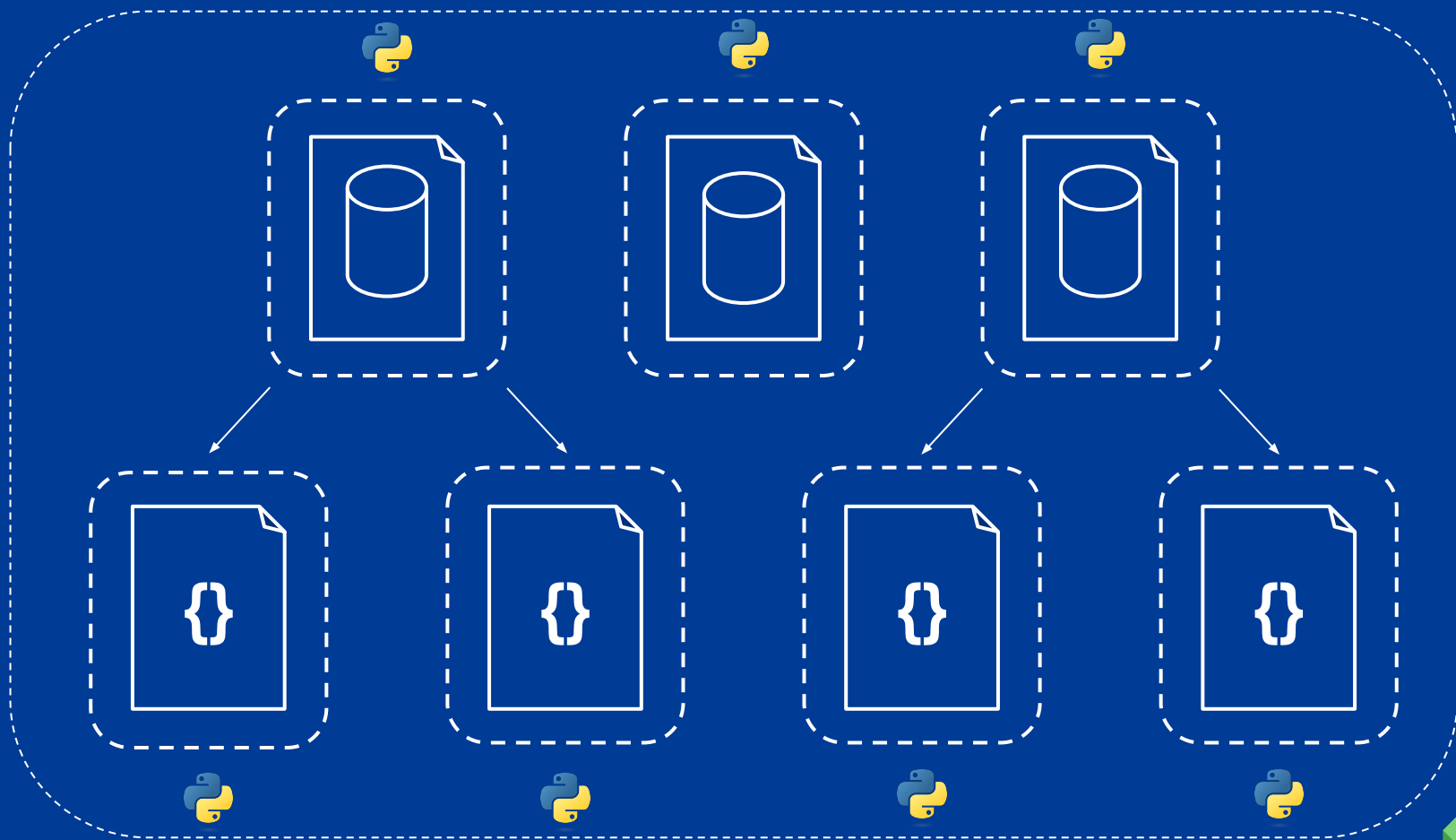
GitLab



Bazel builds

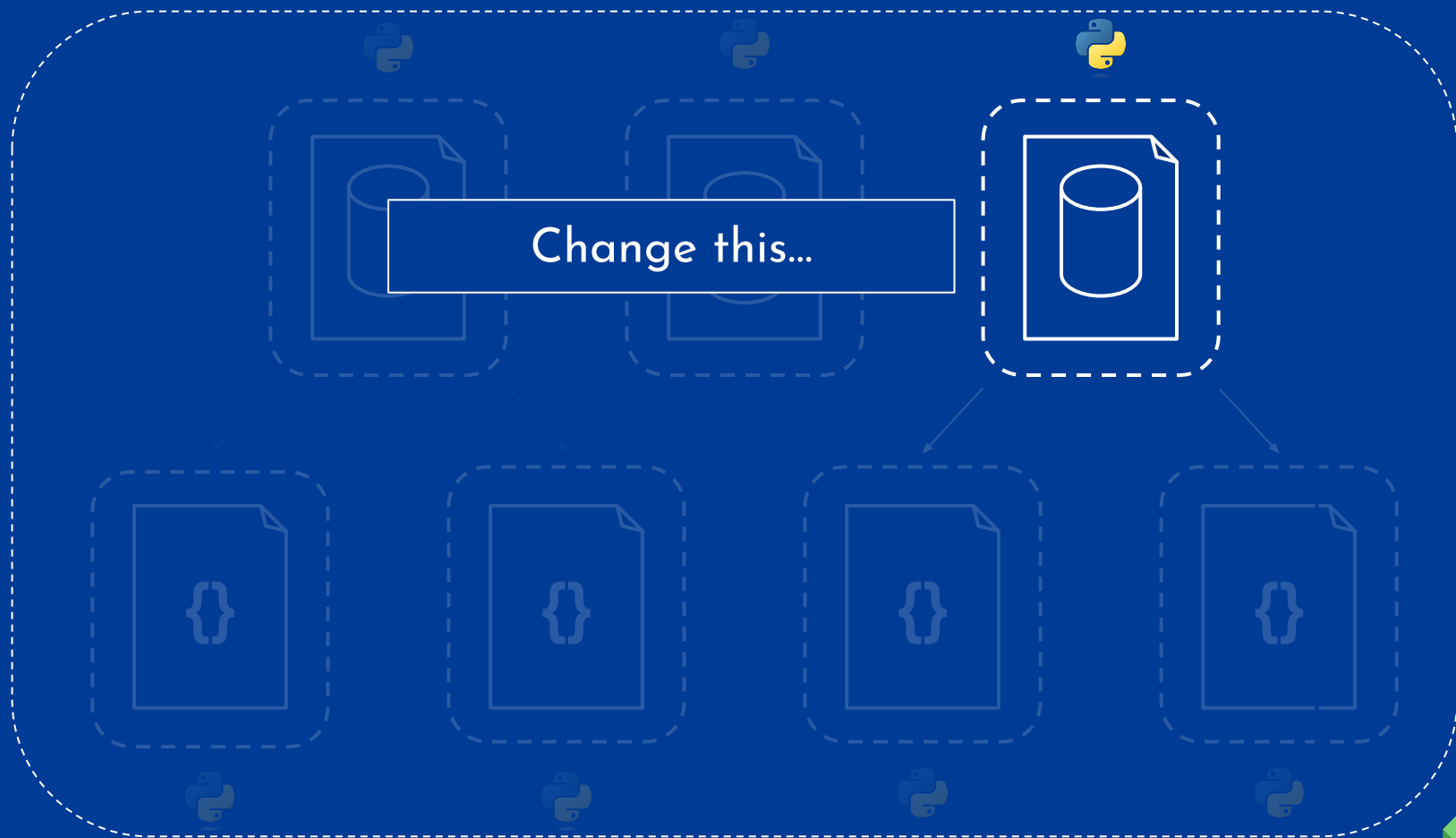






Bazel



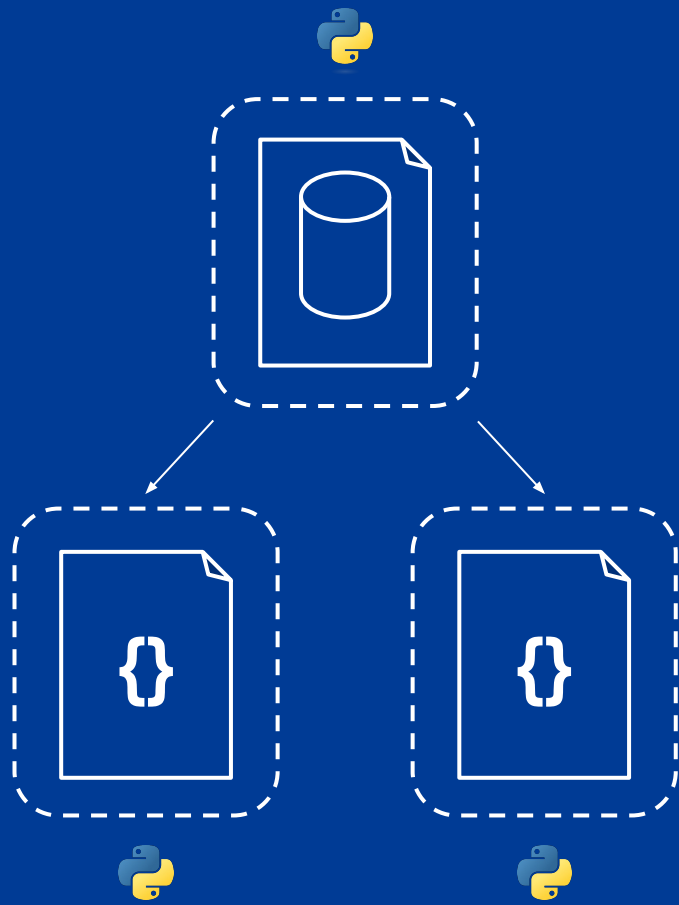
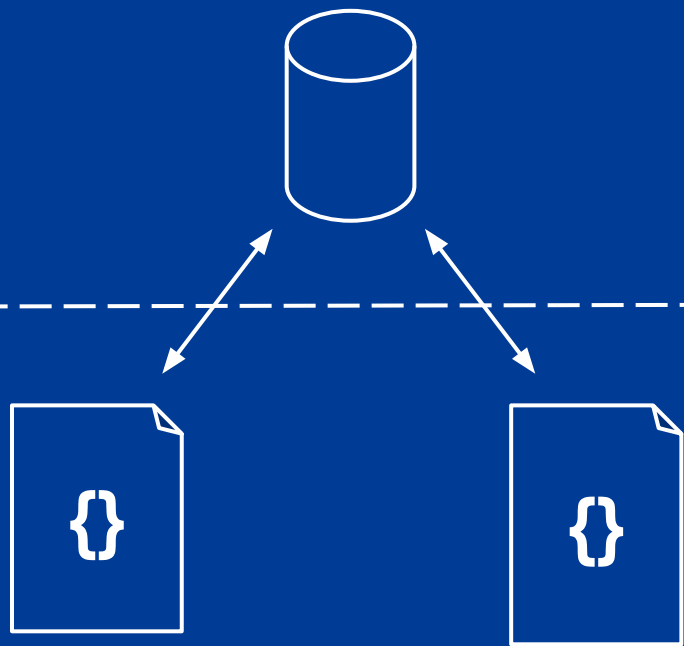




The diagram is set against a dark blue background. A large, rounded rectangle with a dashed white border contains the main elements. At the top, three Python logos are arranged horizontally. Below them, a row of three document icons (a rectangle with a folded top-right corner) is shown. The middle document is faded, while the left and right ones are white. A white rectangular box with the text "Change this..." is positioned over the middle document. Arrows point from the left and right documents down to two more documents below. These two bottom documents are also white and contain a white curly brace icon. Below these two documents are two more Python logos. The entire diagram is enclosed in the dashed white border.

Change this...

And only test these!



old → new

Community



Began as part of a single product team

 Dogfooding! 



Began as part of a single product team



```
class hotels(HiveTable):  
    schema = StructType([  
        StructField("hotel_id", LongType()),  
        StructField("is_closed", BooleanType()),  
    ])  
  
    format = "orc"
```

**dbimports.py**



```
> ./cli generate-table dbimports.hotels
```

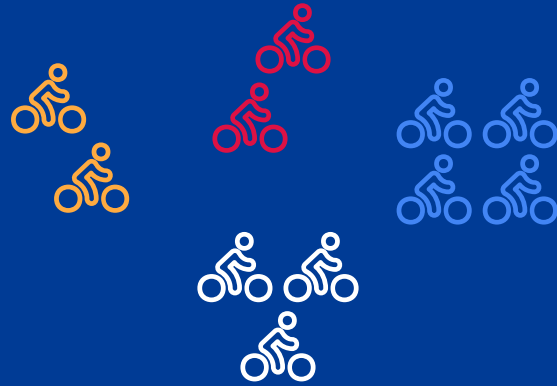
```
class hotels(HiveTable):  
    schema = StructType([  
        StructField("hotel_id", LongType()),  
        StructField("is_closed", BooleanType()),  
    ])  
  
    format = "orc"
```

**dbimports.py**



```
> ./cli any-common-operation
```





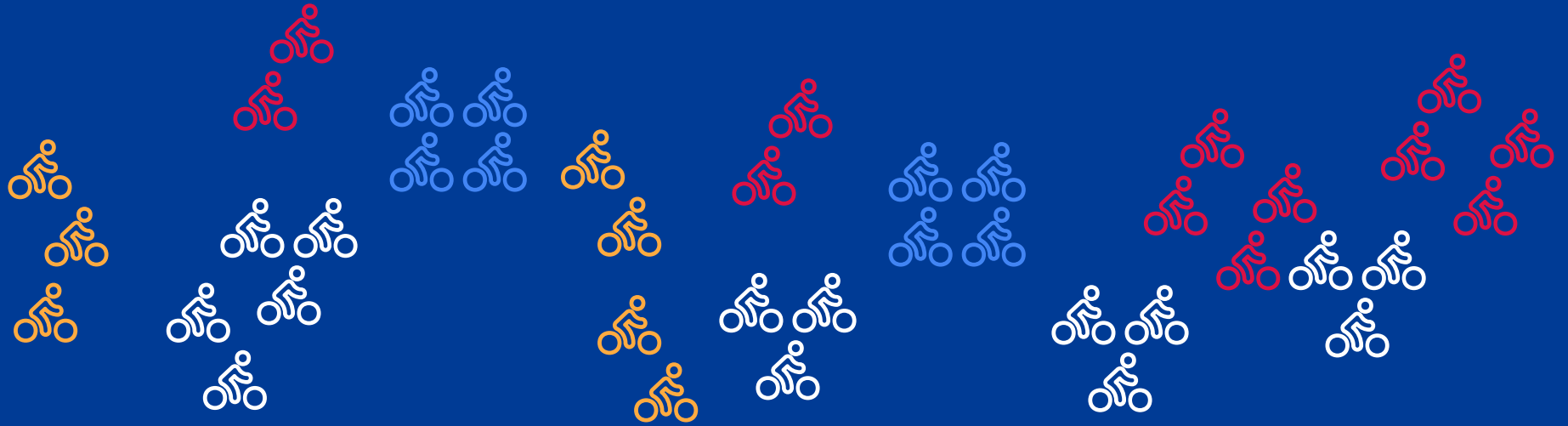
Grew organically across teams



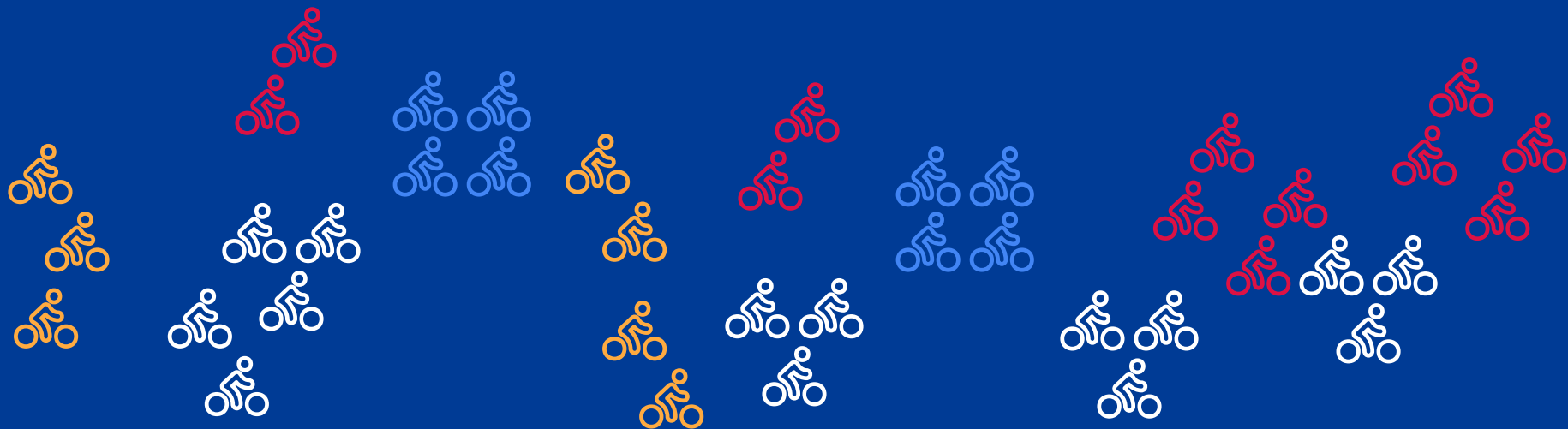
Testing

# Testing





Gradually reaches critical mass -> network effects



*How to support growth and community?*

## *How to support growth and community?*



Communication and  
Support



Treat As Internal Open  
Source



Internal Training  
Courses

# Current State

- 384 ETL pipelines
- Supports 25 teams
- Includes ML model training pipelines
- Alerting and monitoring libraries



# Future State

- Continue onboarding teams
- Onboard more data sources
- Open source?

