Tables as Code

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

Booking.com

Bram van den Akker

Jon Smith

Booking.com

Maurits Kroese

João Laia

Alina Solovjova

Jeroen Schmidt

Niek Tax

Booking.com

Data @ Booking.com

10x teams working on data

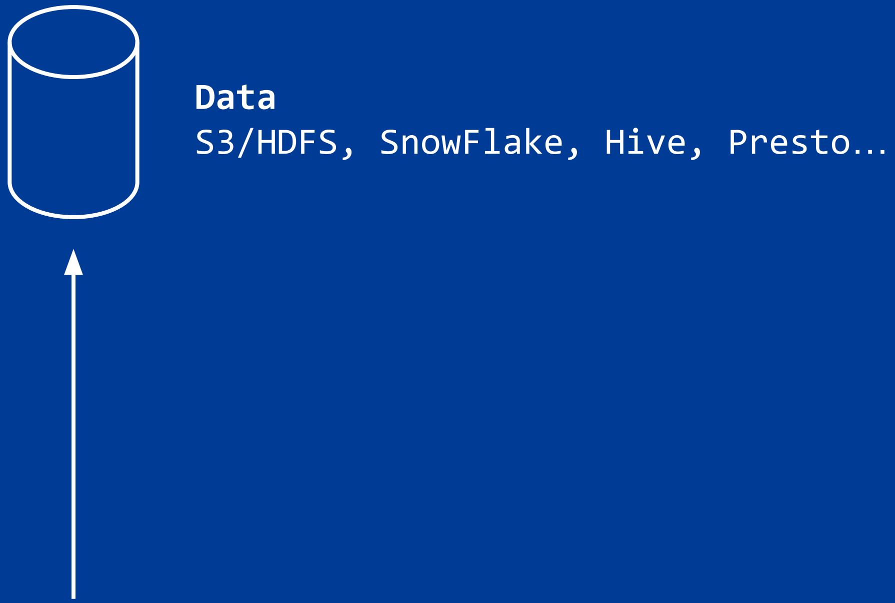
100x data practitioners

1000x data workflows

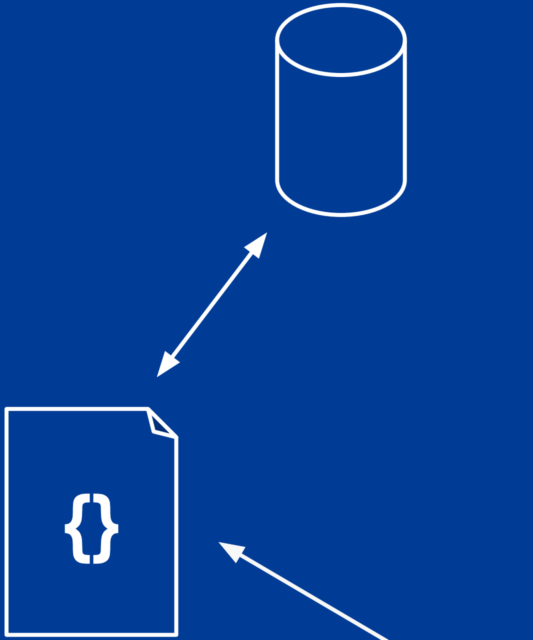
10000x data assets

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

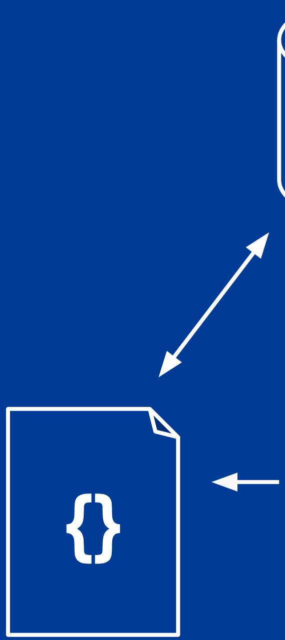
Let me explain.



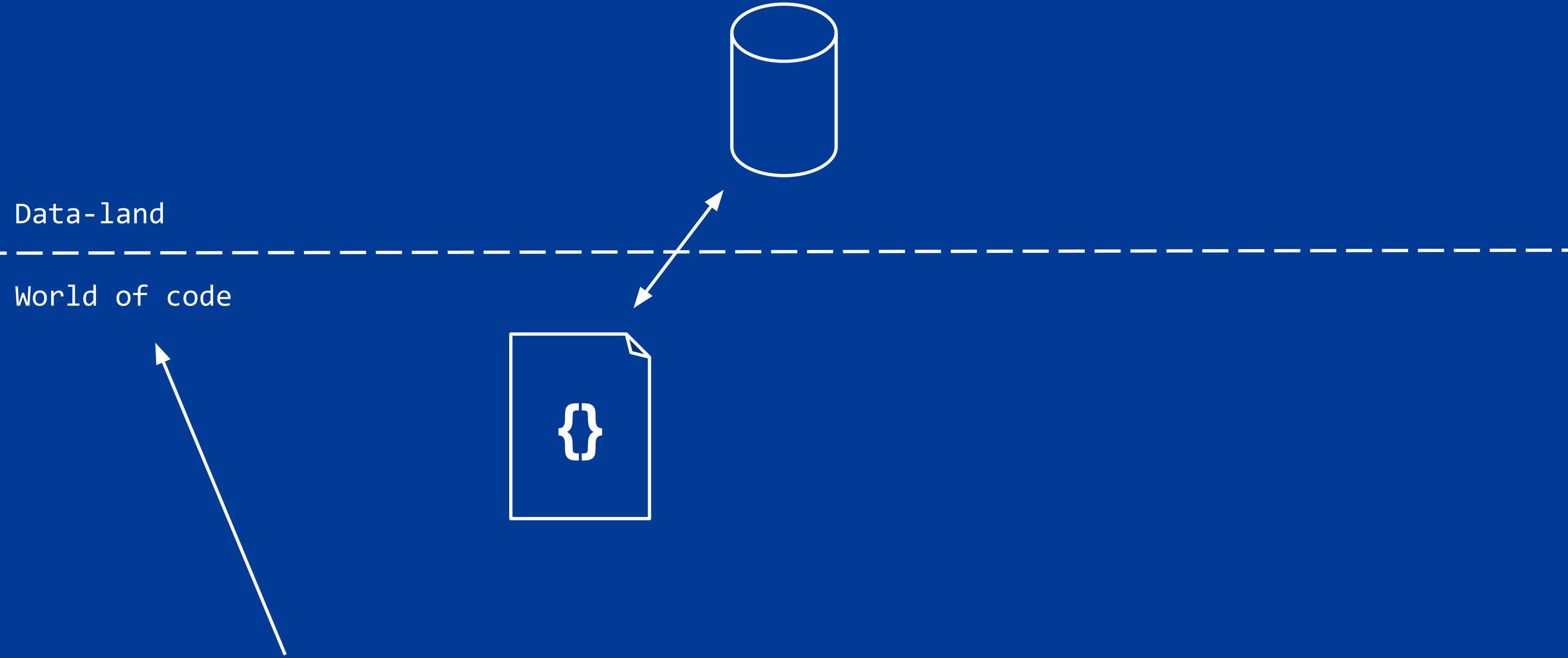
It all starts with our persistent data



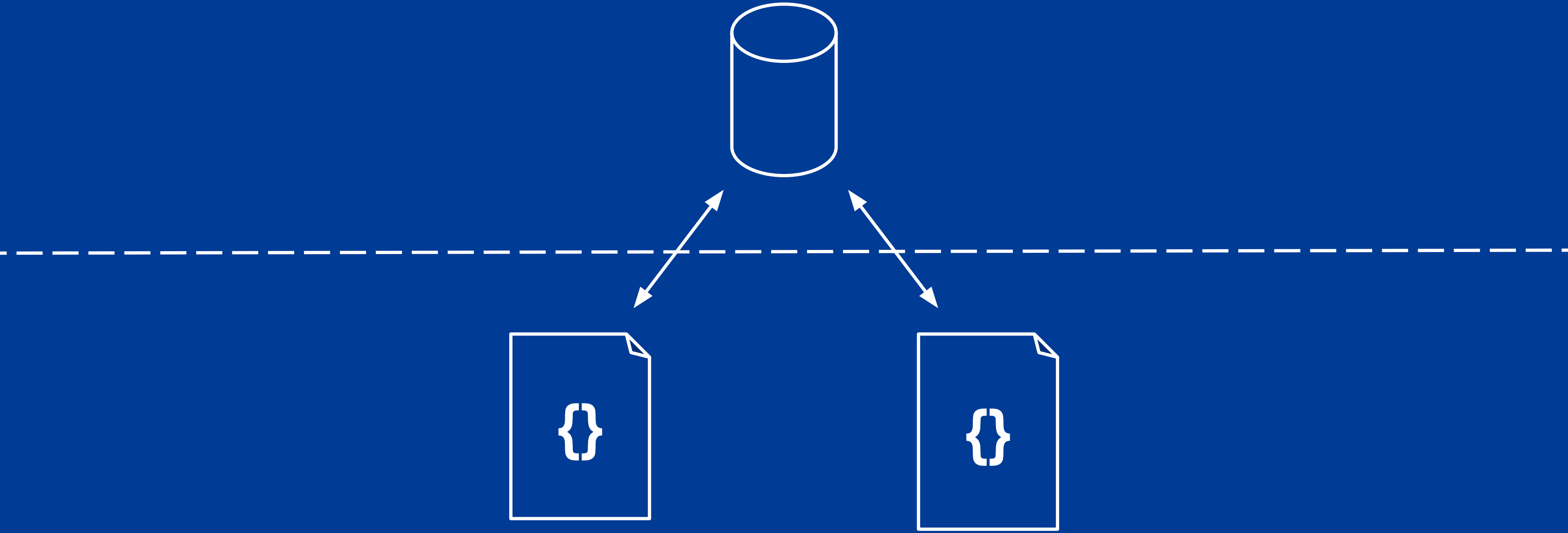
Which is produced and consumed by code



Code
Eg. Python, SQL, DBT, Spark, etc.



Both the code and data exist at different levels

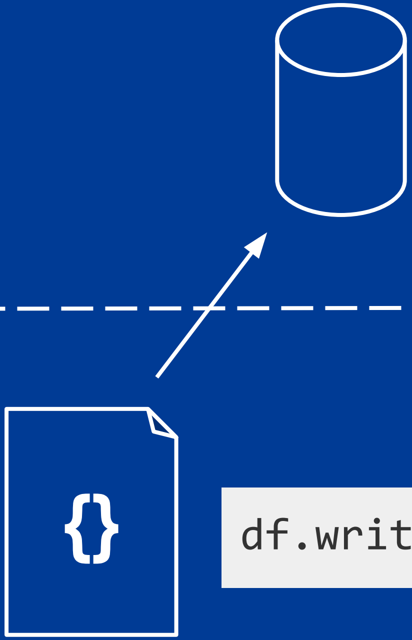


Producers and …

… communicate through …

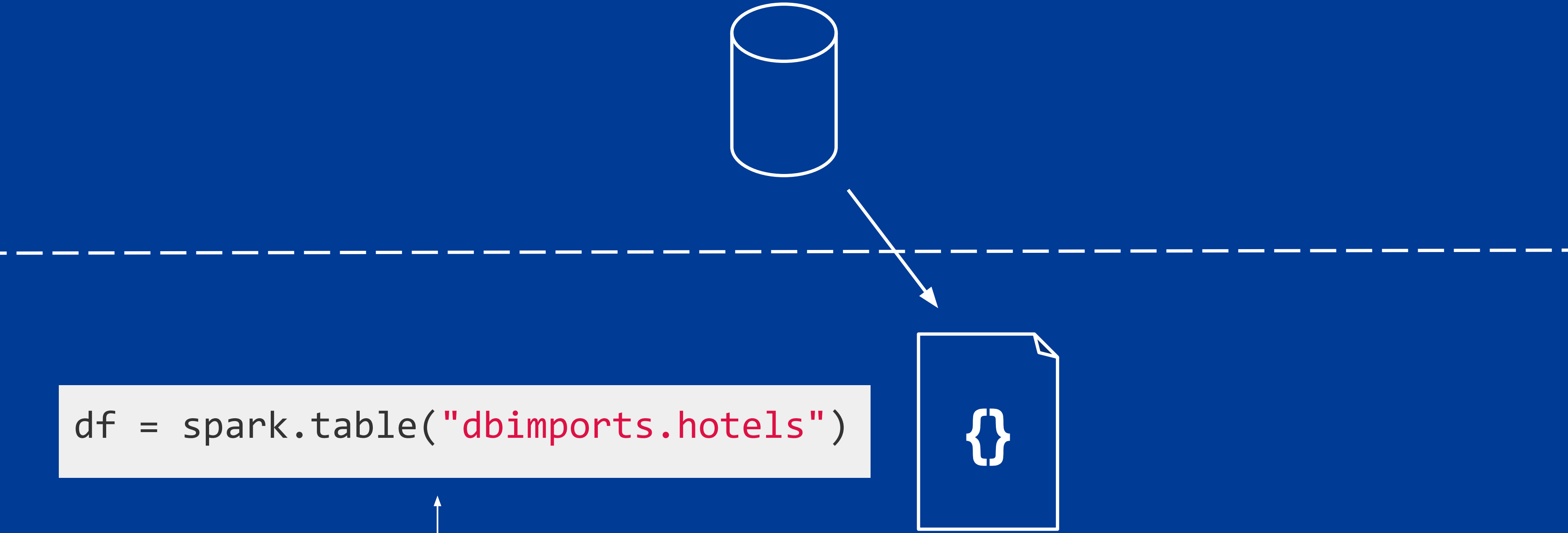
… consumers *only* …

… the persistent data



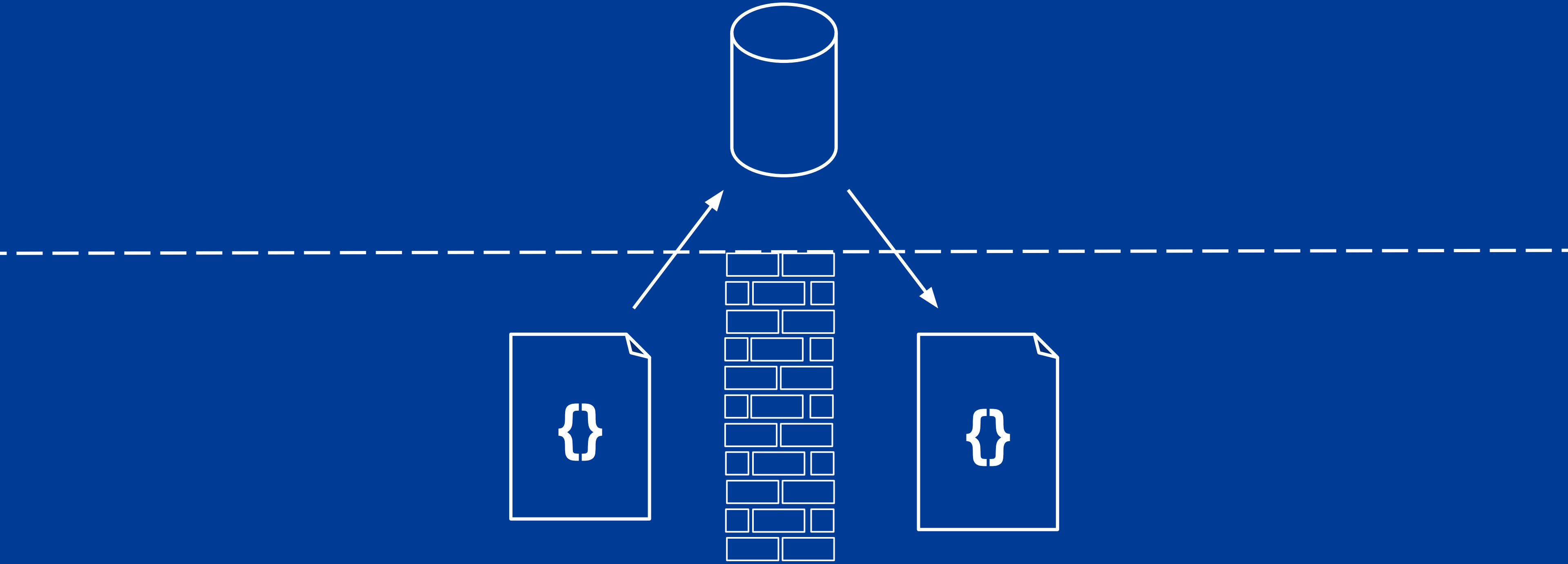
```
df.write.saveAsTable("dbimports.hotels")
```

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

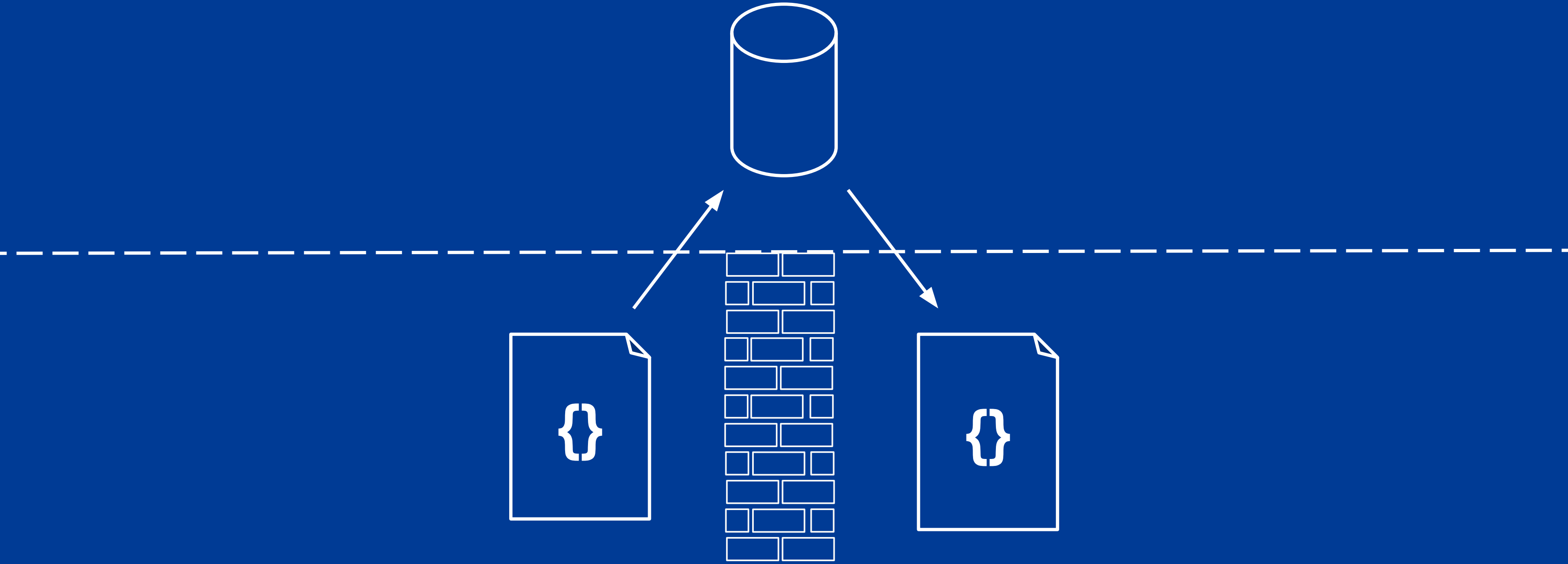


```python
df = spark.table("dbimports.hotels")
```

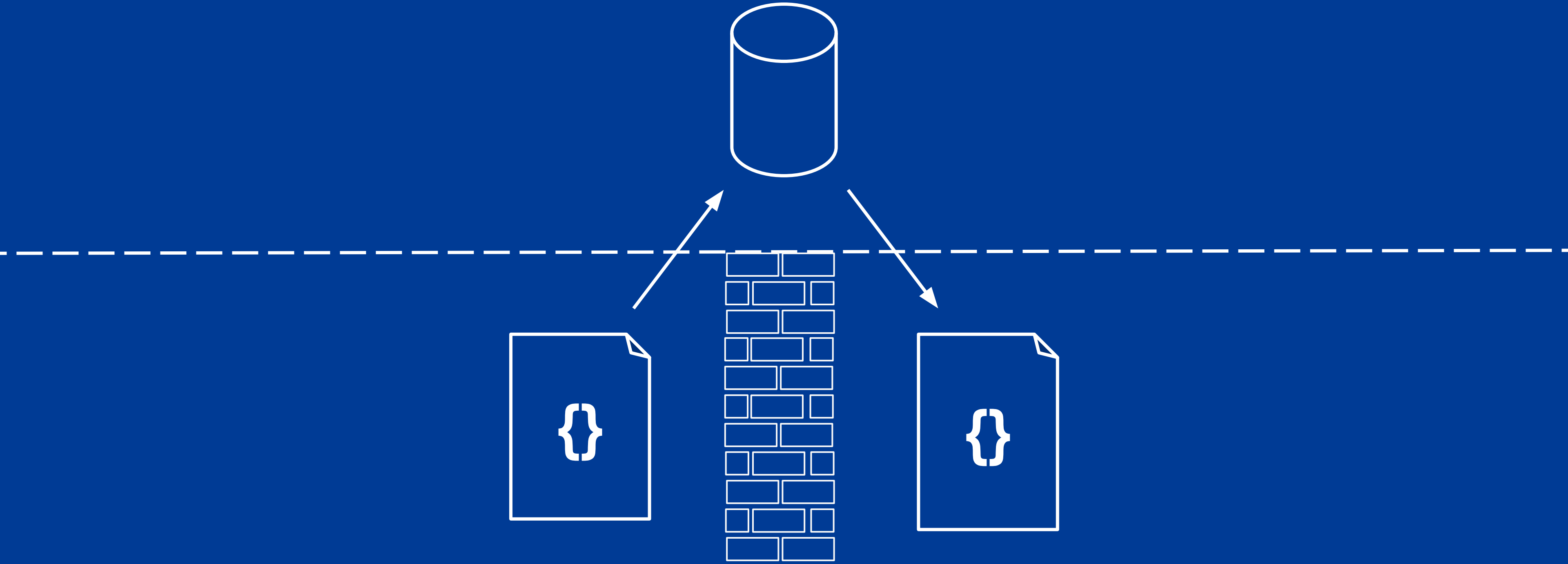
However, when reading nothing guarantees what comes back.



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

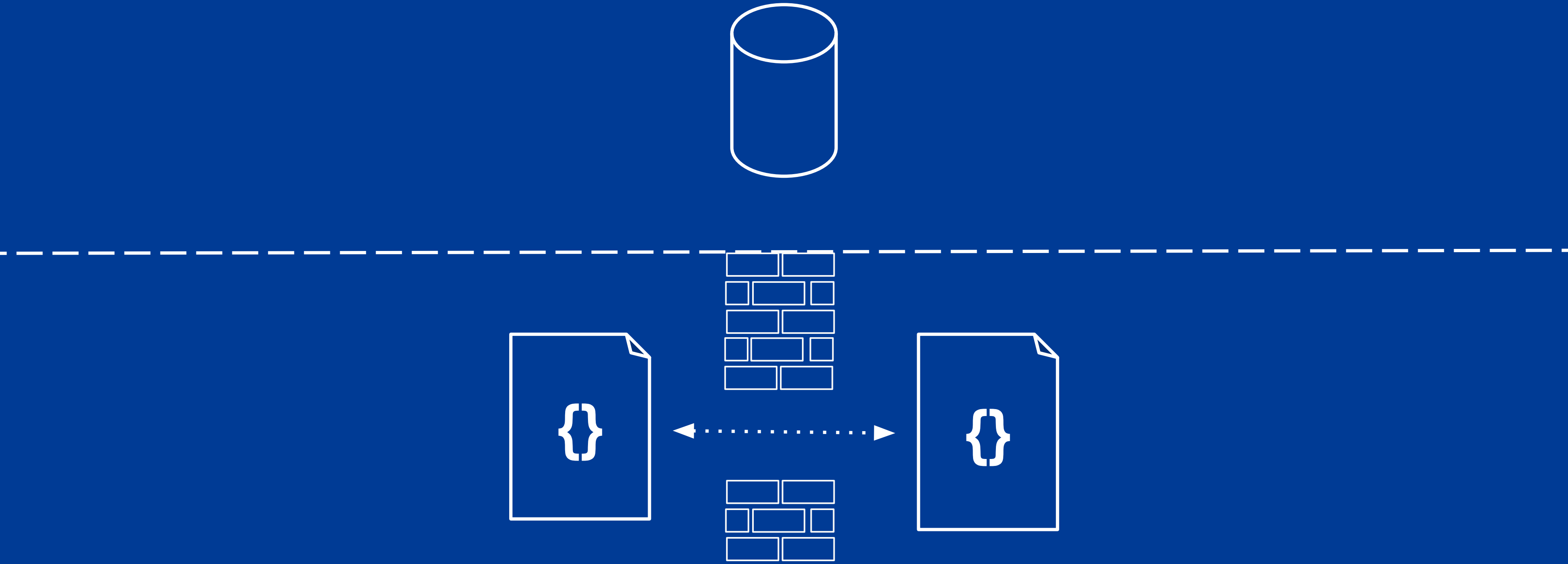


As a result, changes are only really tested with the persistent data...
...in production.



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

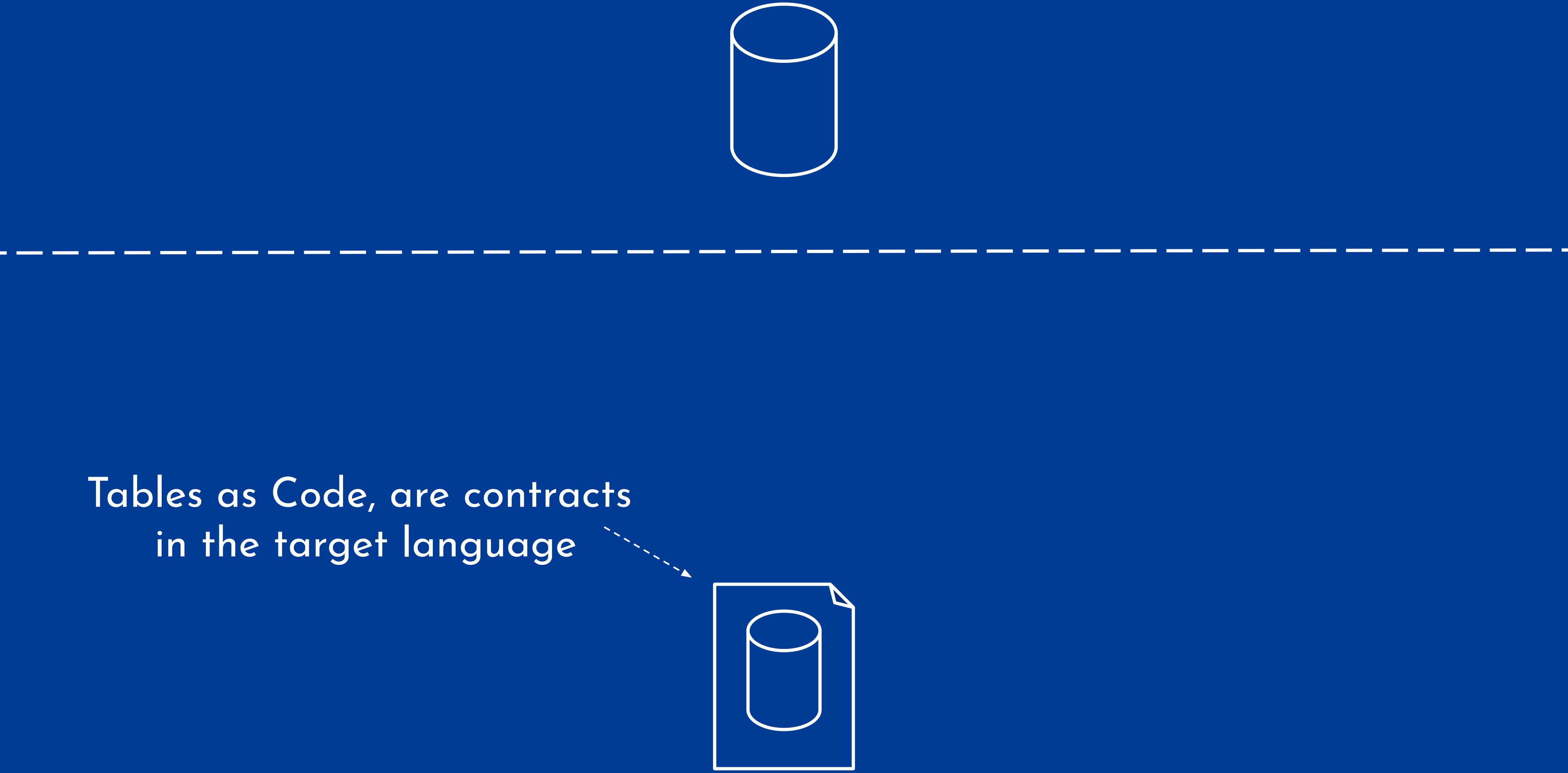
Bringing us to the title of this talk

Bringing us to the title of this talk

Tables as Code

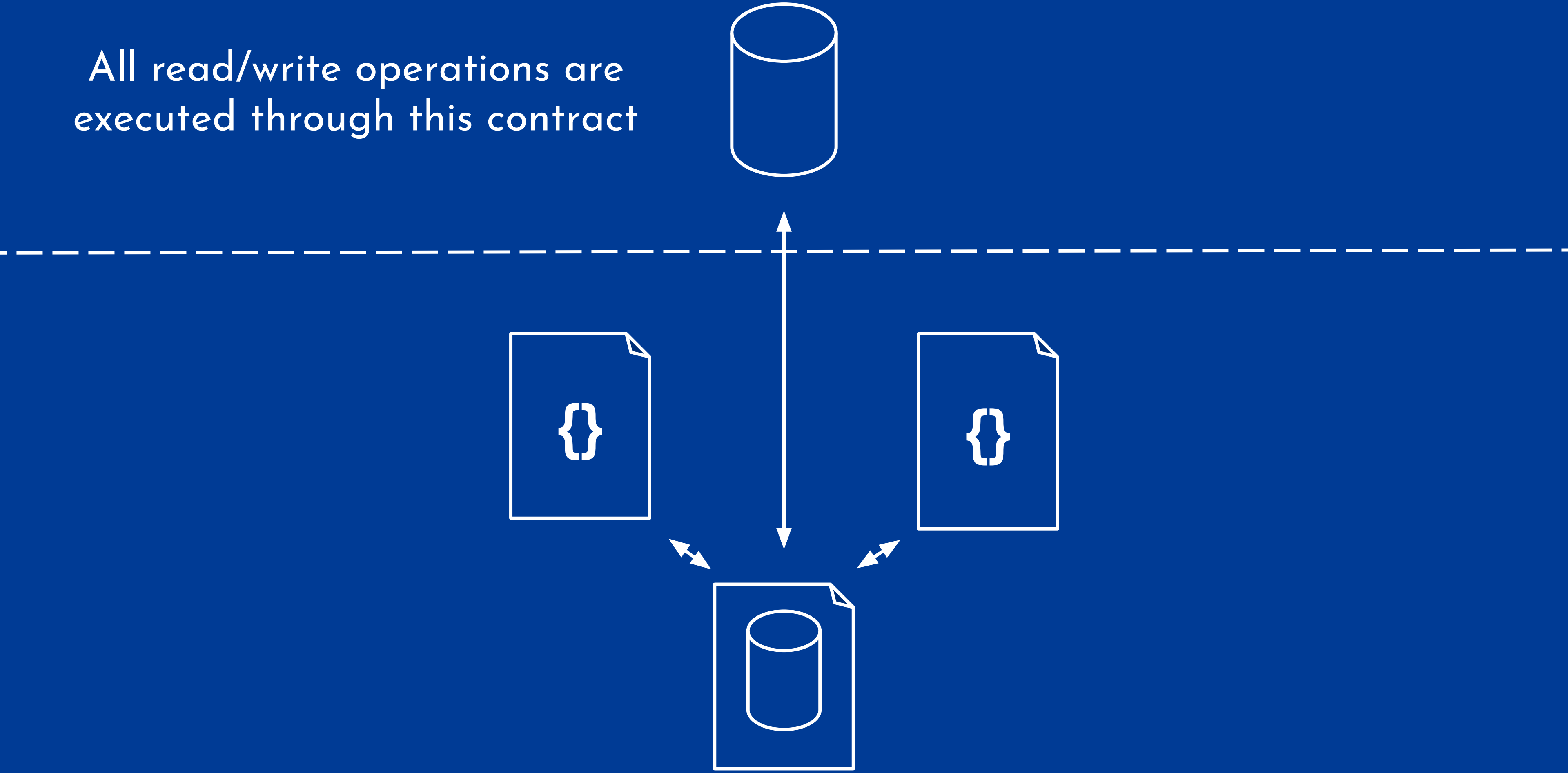
The structure of this talk:

- 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()),
    ])

    format = "orc"
    partition_by = ["date"]
```

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

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

```python
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)
```

```python
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)
```

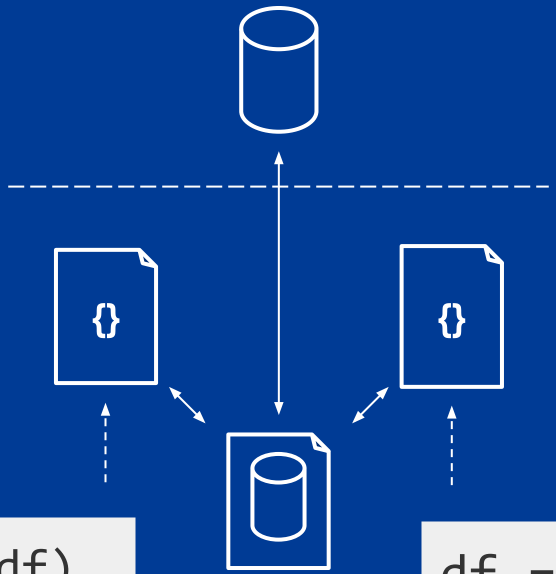
Or any other arbitrary methods, both custom and predefined

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

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

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



```
hotels.write(df)
```

```
df = hotels.load()
```

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

All of these changes might sound somewhat interesting.

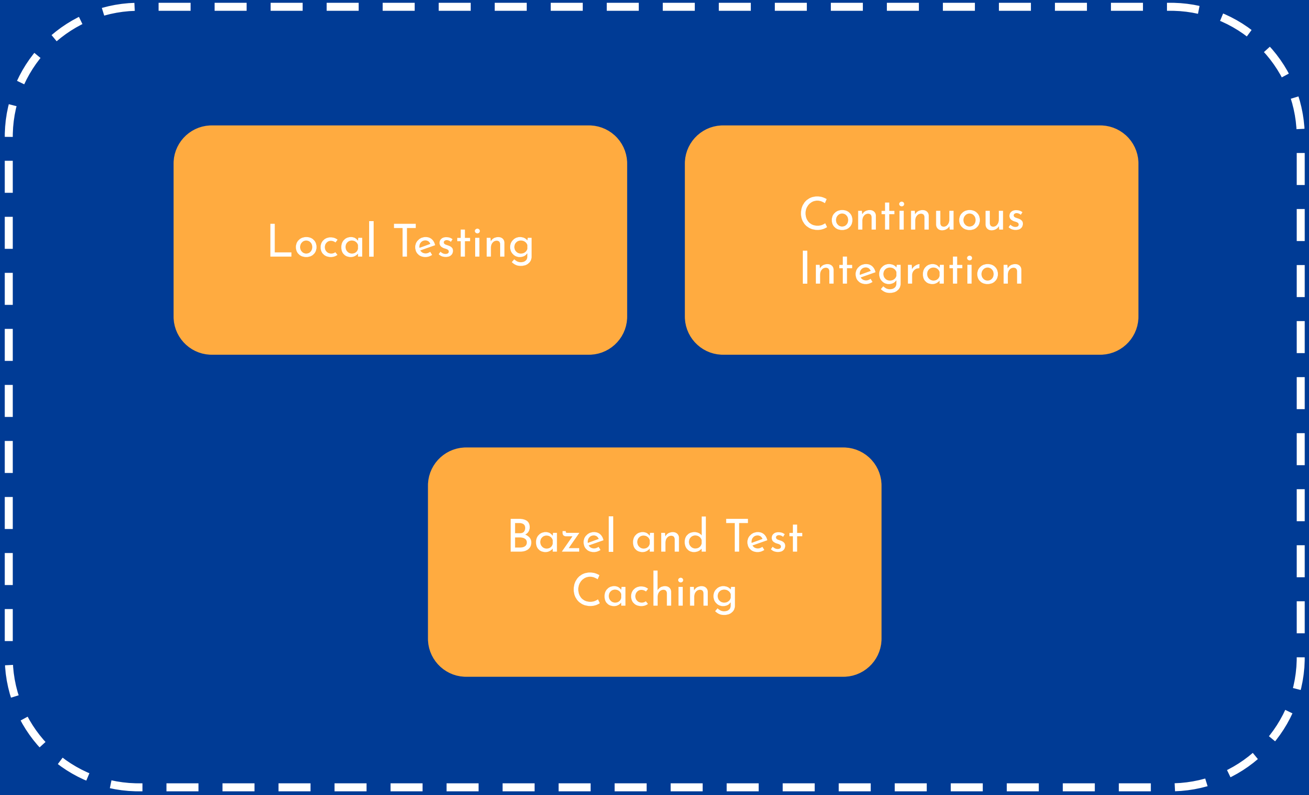
However, the real benefits come with…

The Monorepo

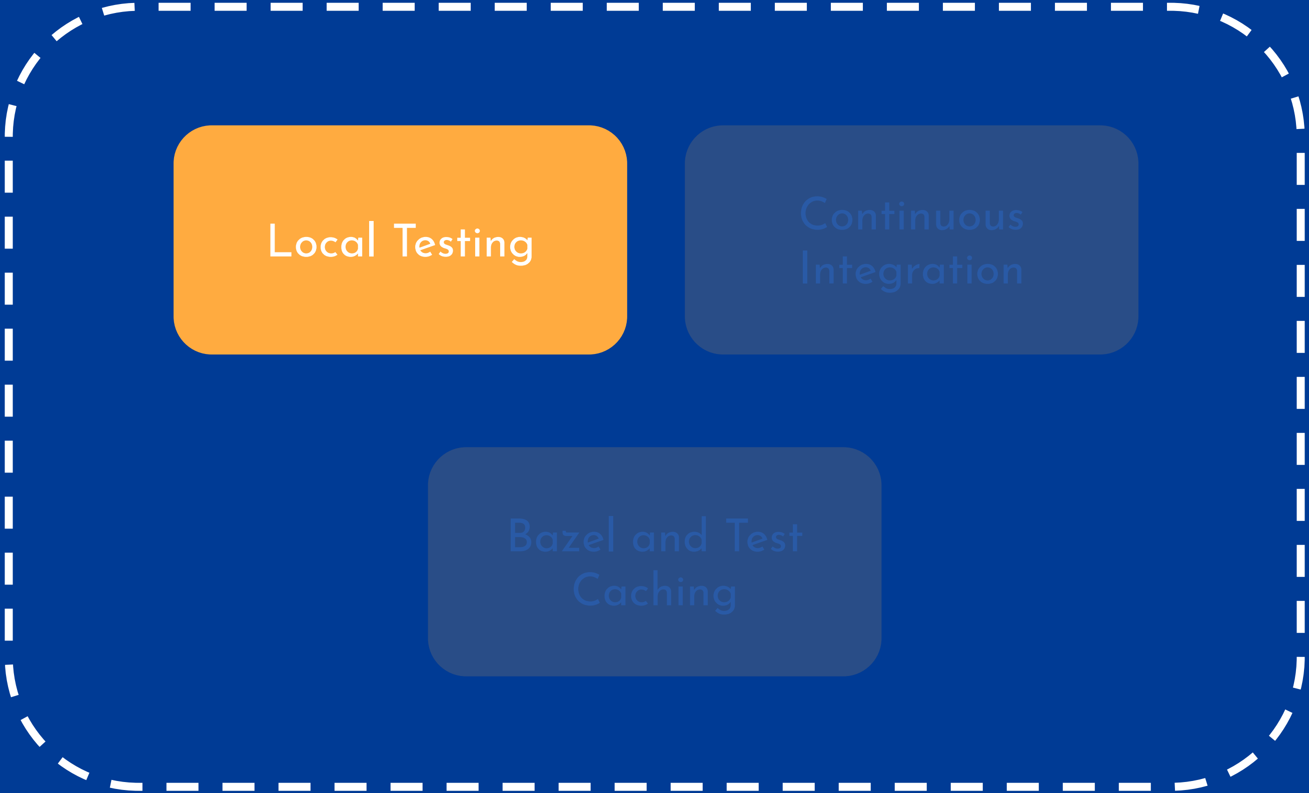
Monorepo vs. Polyrepo Debate

Monorepo vs. Polyrepo Debate

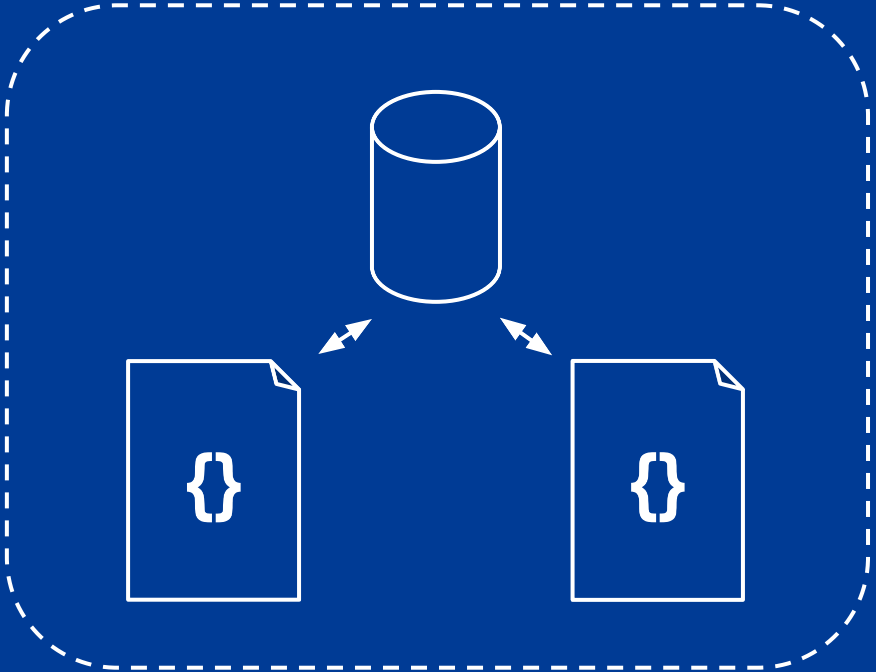
For now...



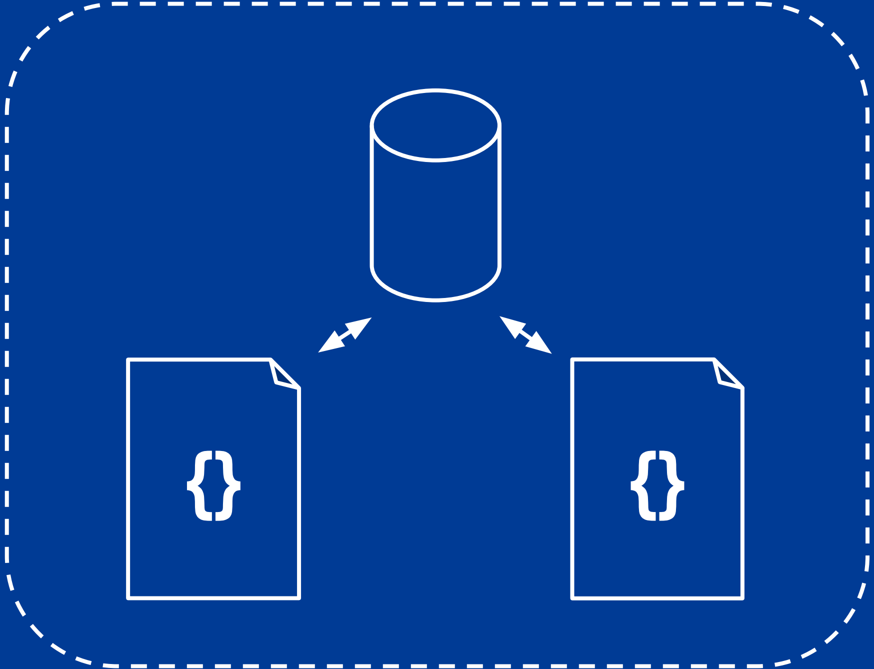
The Monorepo



The Monorepo



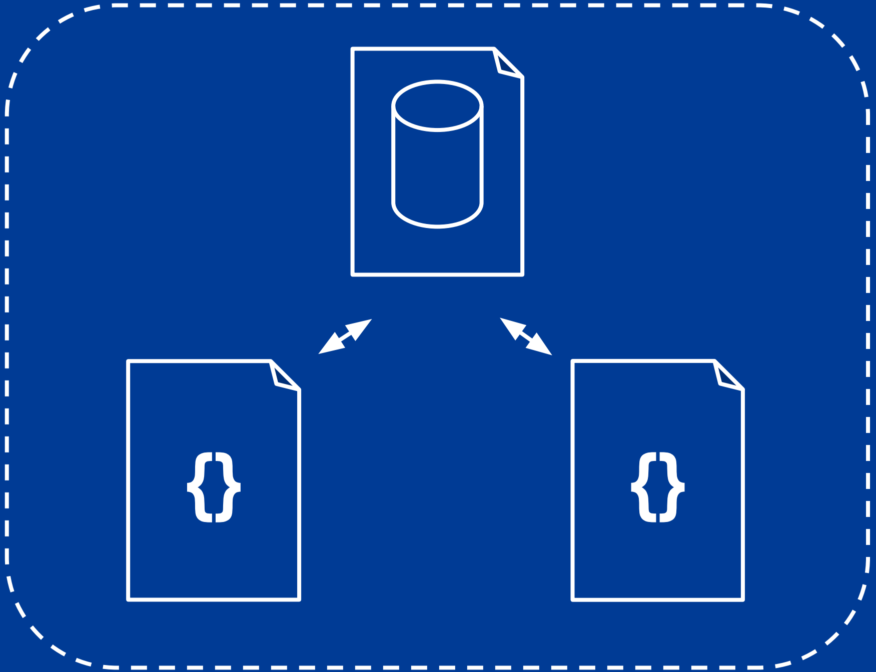
Validate code/data contract...



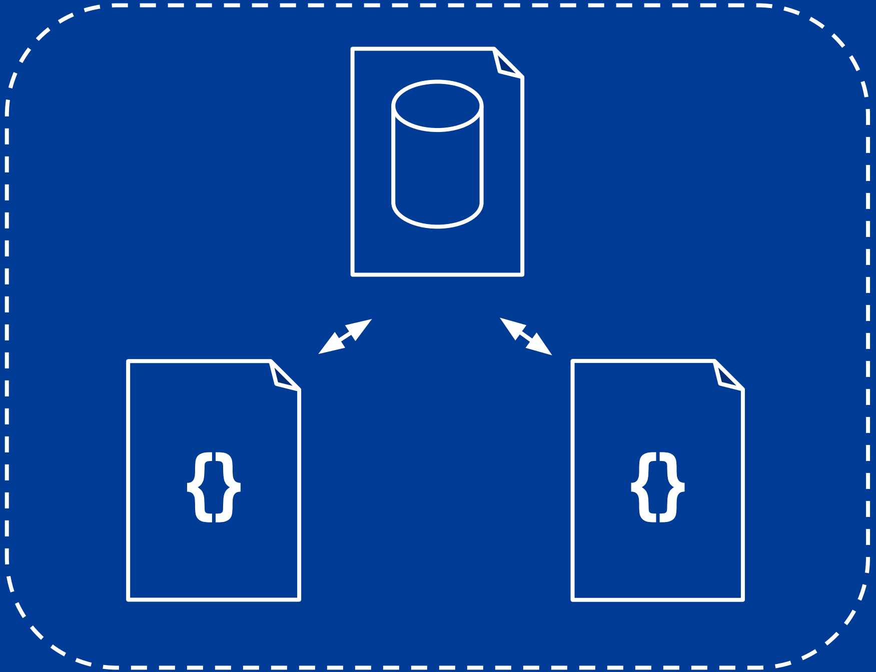
Validate code/data contract by testing in prod

$\downarrow$

Surprise ‘ColumnNotFound’ exception after 45min



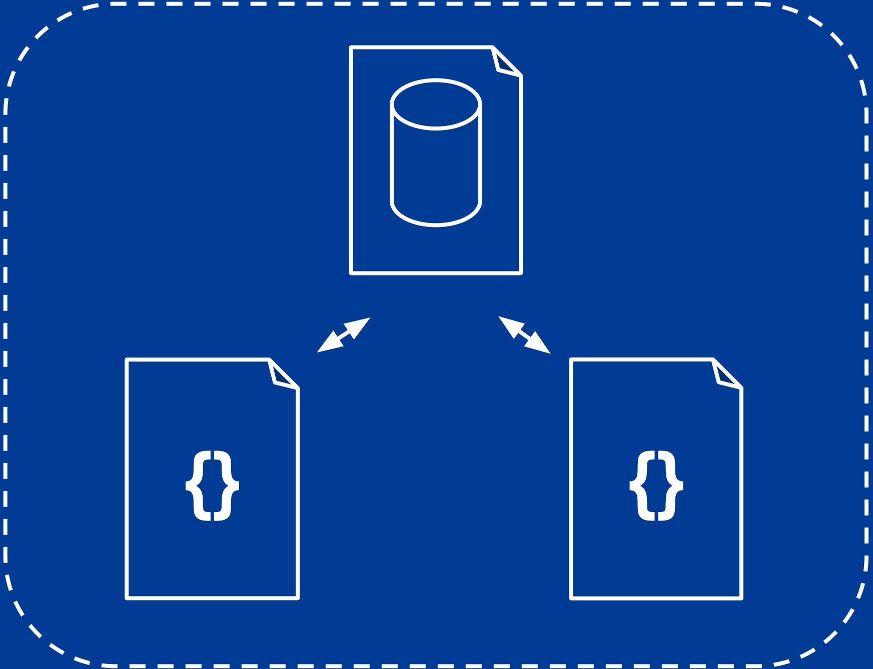
Validate code/data contract *locally*



Validate code/data contract *locally*

↓

No surprise exceptions in prod



Validate code/data contract *locally*

$\downarrow$

No surprise exceptions in prod

$\searrow$

(Faster development cycle)

de/data contract

xceptions 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 Spa
mock
Setup /
teardown
Mocking
interfaces
Handled

```python
from tables.hive import hotels

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

```python
from tables.hive import hotels


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

```python
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

```python
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

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

def process_hotel_pipeline():
    hotel_df = hotels.load()
    a
    a
```

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

```python
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
    hotel_df = hotels.load()
    available_rooms_df = get_available_rooms(hotel_df)
    available_rooms.write(available_rooms_df)
```

This function accesses a `num_rooms` column…

```
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 p
```

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_payments import available_rooms

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

Surprisingly powerful test!

```
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_payments import available_rooms

def process_hotel_pipeline():
    h
    a
    available_rooms.write(available_rooms_df)
```

Surprisingly powerful test!

Fails if table has no `num_rooms` column

```
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

def process_hotel_pipeline():
    h
    a
    a
```

Surprisingly powerful test!

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()
```

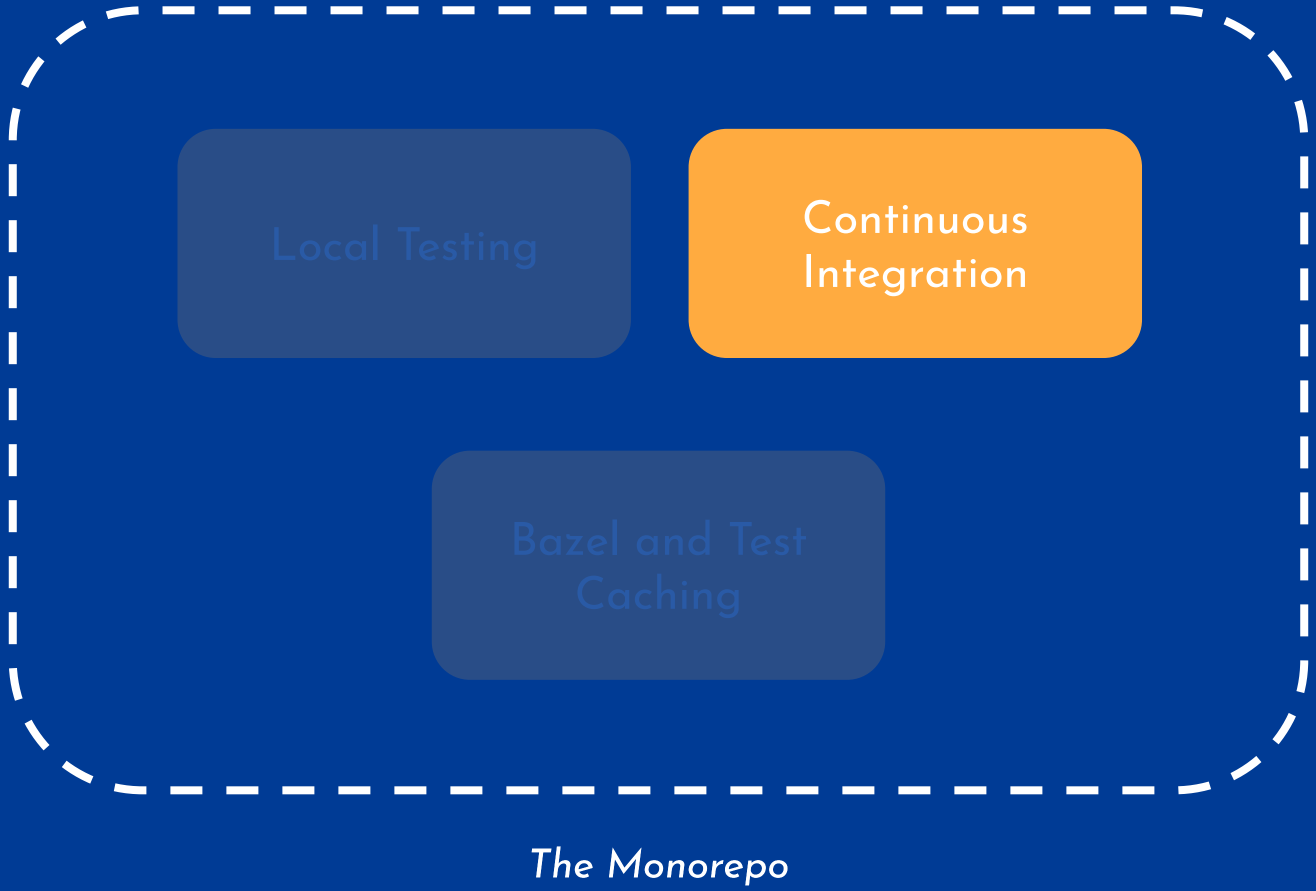
```python
from tables.hive import hotels
from pyspark.sql import Row

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

```python
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)
    ...
```

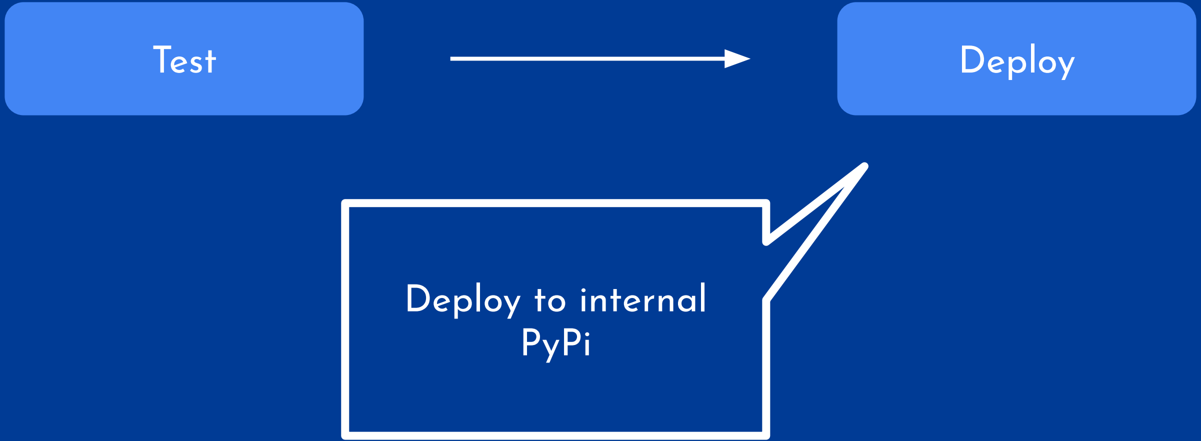
- write custom data to table
- don't need to write every column

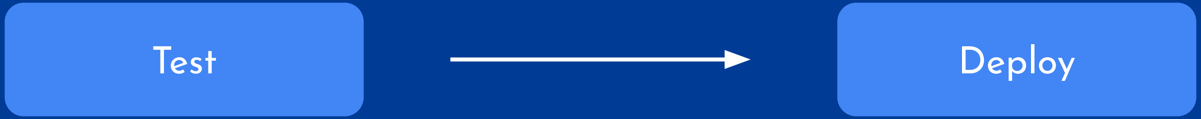

Local Testing
Continuous Integration
Bazel and Test Caching
The Monorepo

Test

- Unit tests
- Integration tests
- Coverage check

GitLab

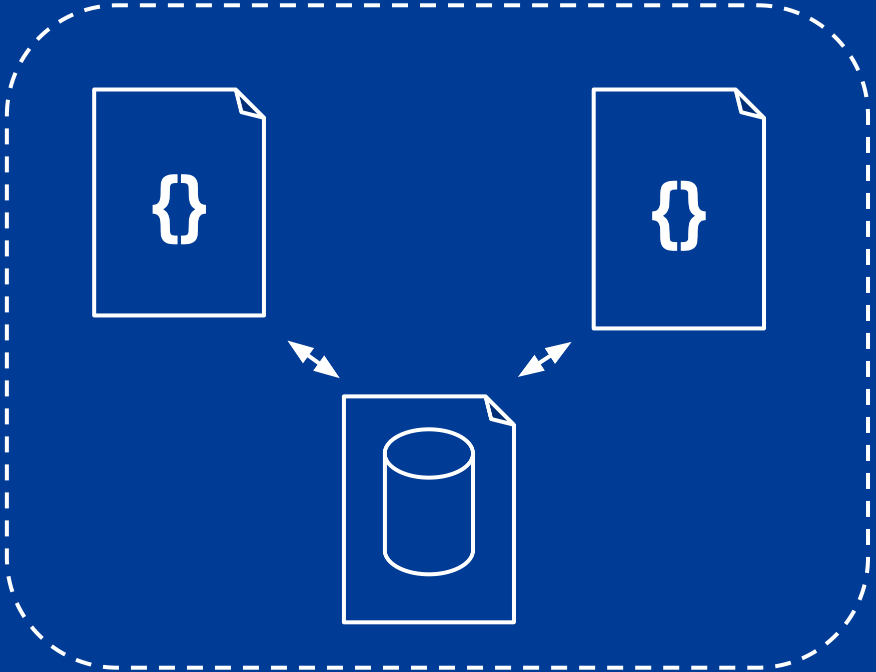

Test
Deploy
Deploy to internal PyPi



all tests must pass before merge

GitLab

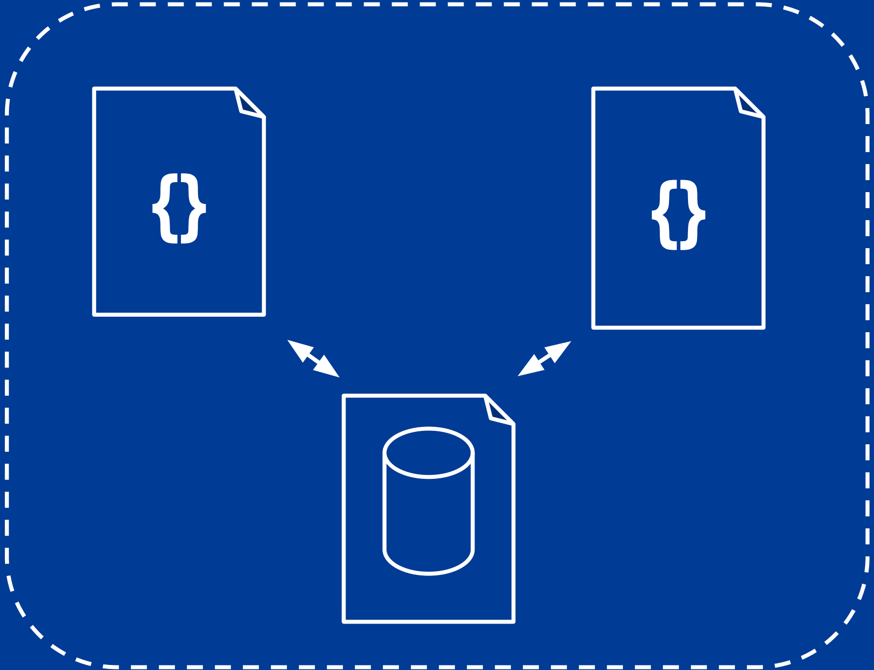
100% (line) test coverage



100% (line) test coverage

$\downarrow$

Code/data contract validated



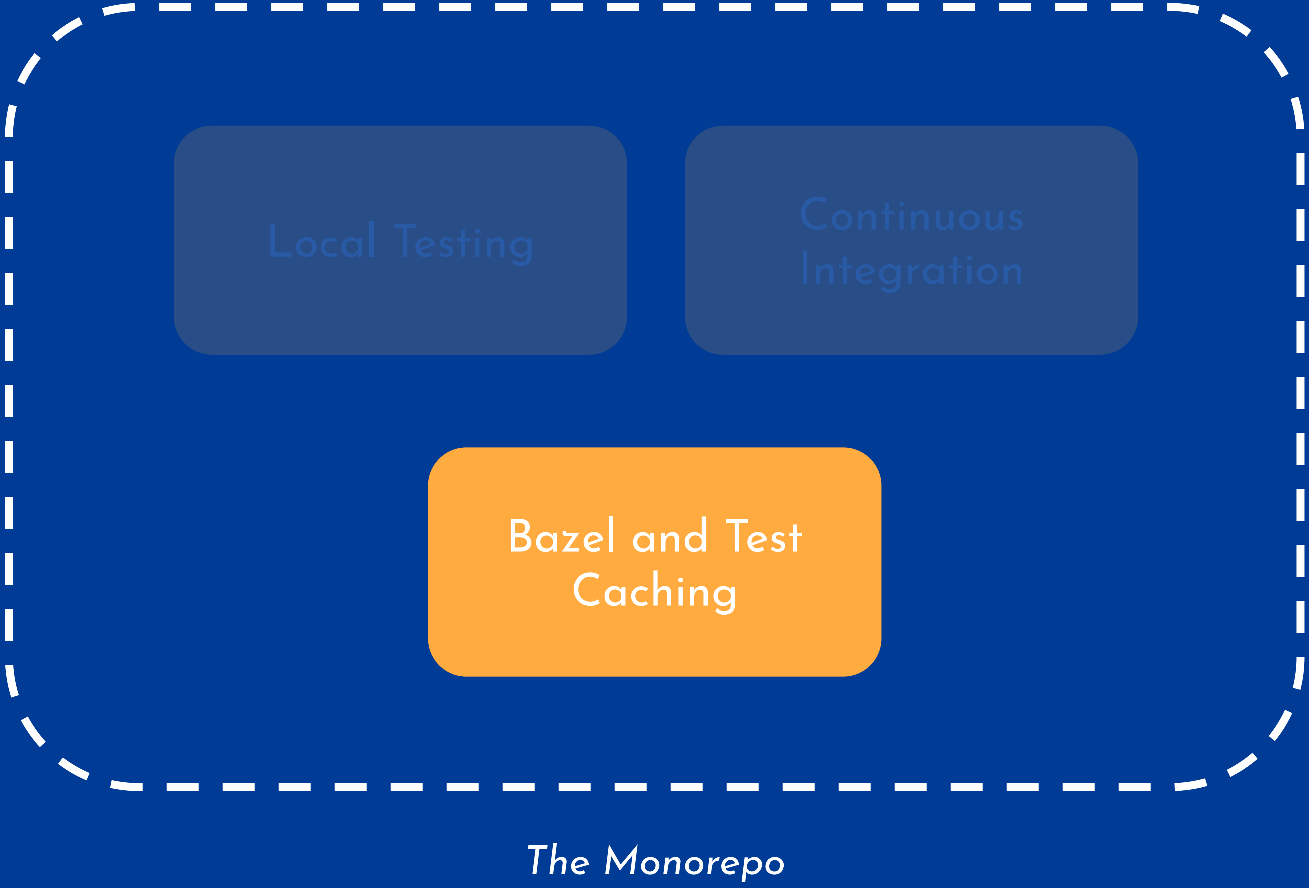
100% (line) test coverage

↓

Code/data contract validated

↓

Friday afternoon deployments

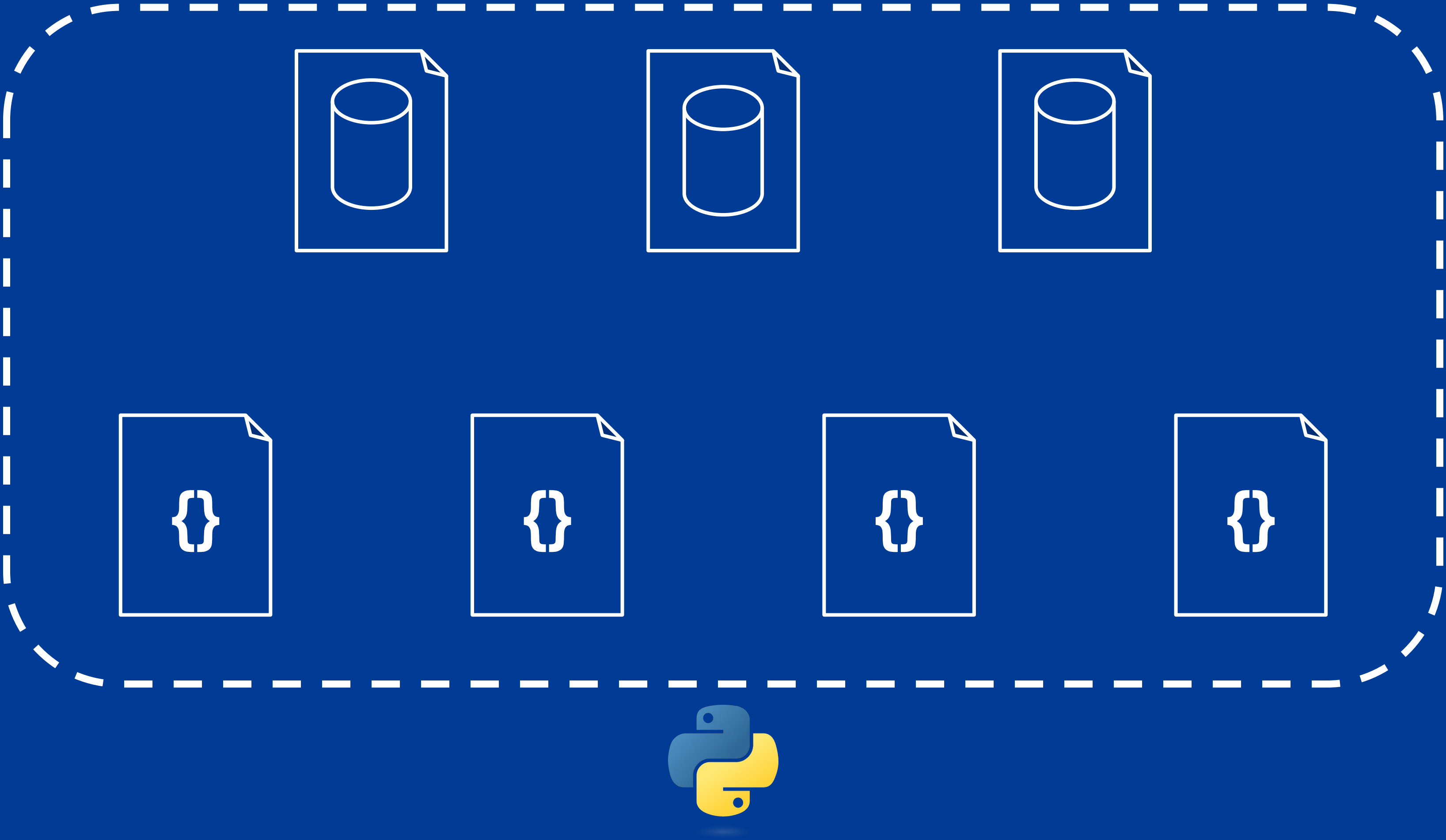

Local Testing
Continuous Integration
Bazel and Test Caching
The Monorepo

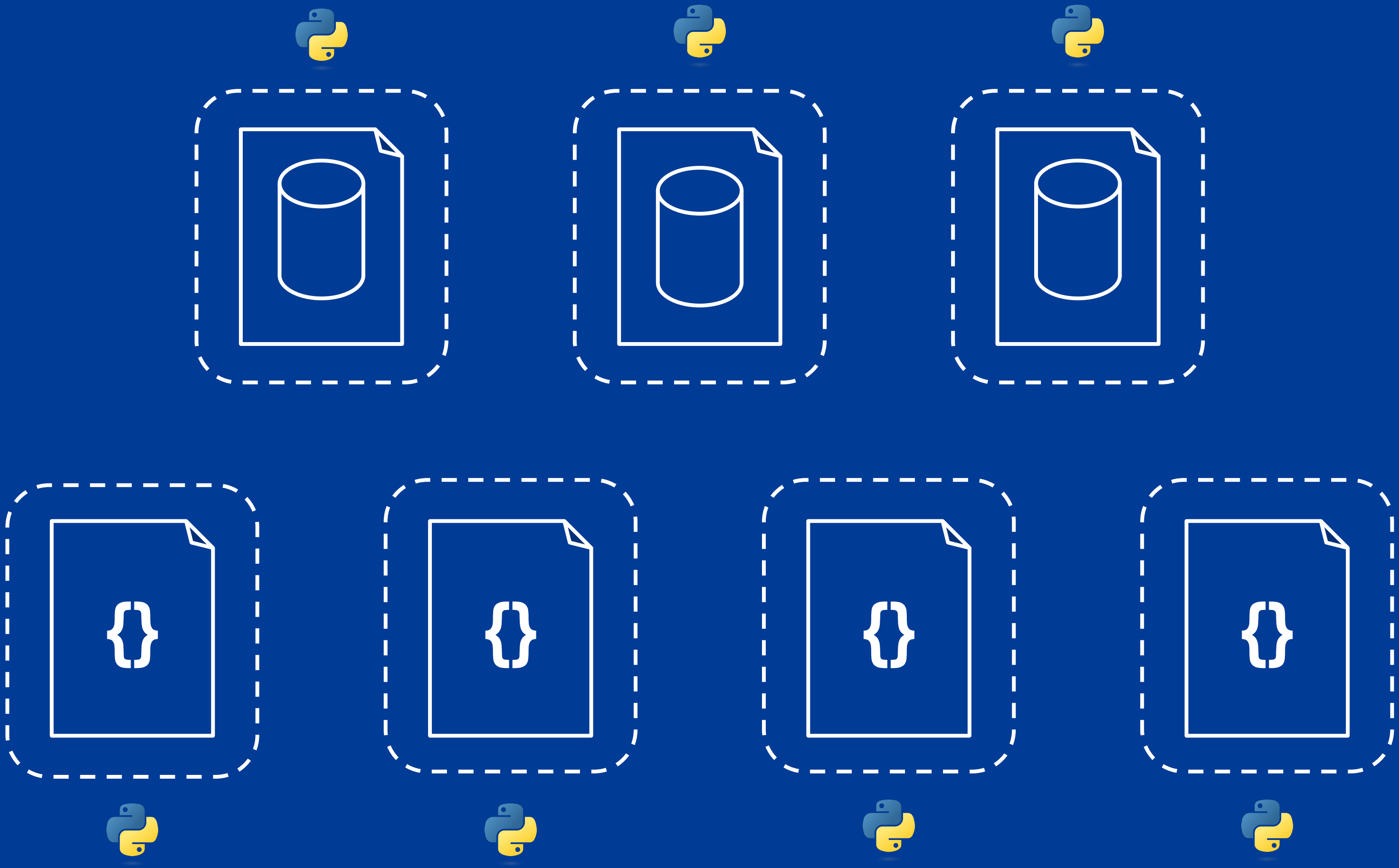
Test
Deploy

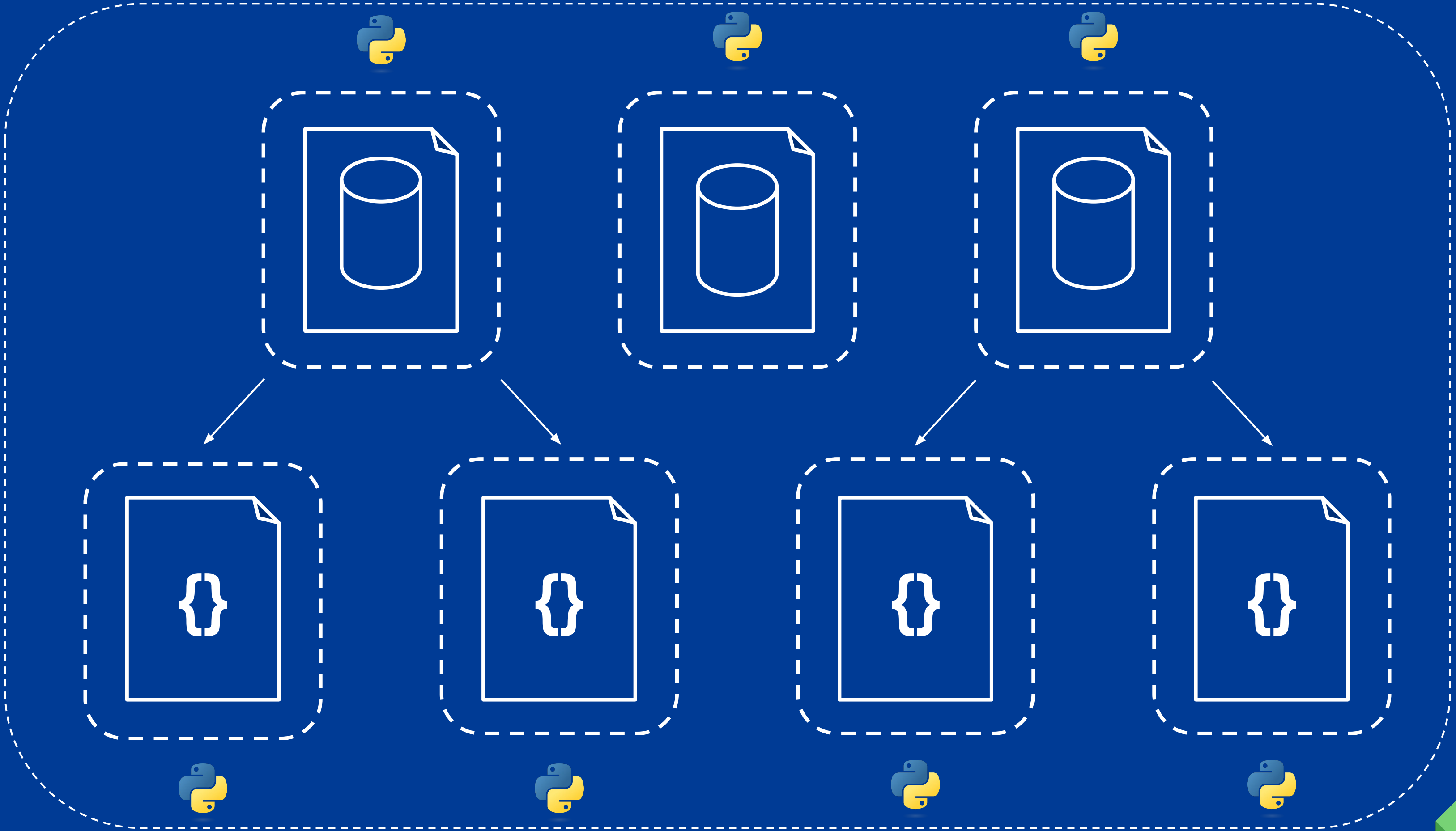
- 1000s of tests, big and small
- Can take >1hr on one machine

GitLab

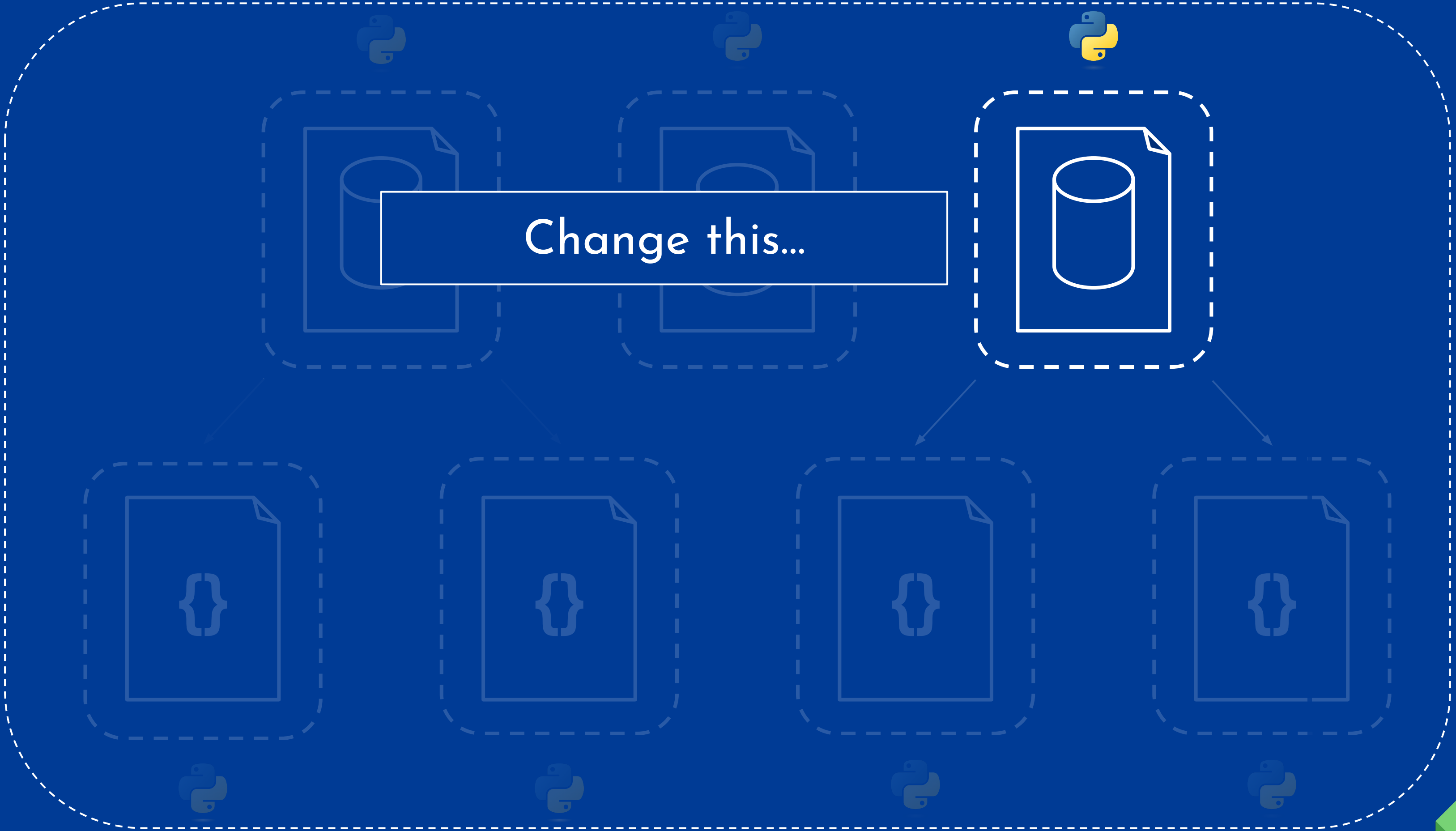
Bazel builds





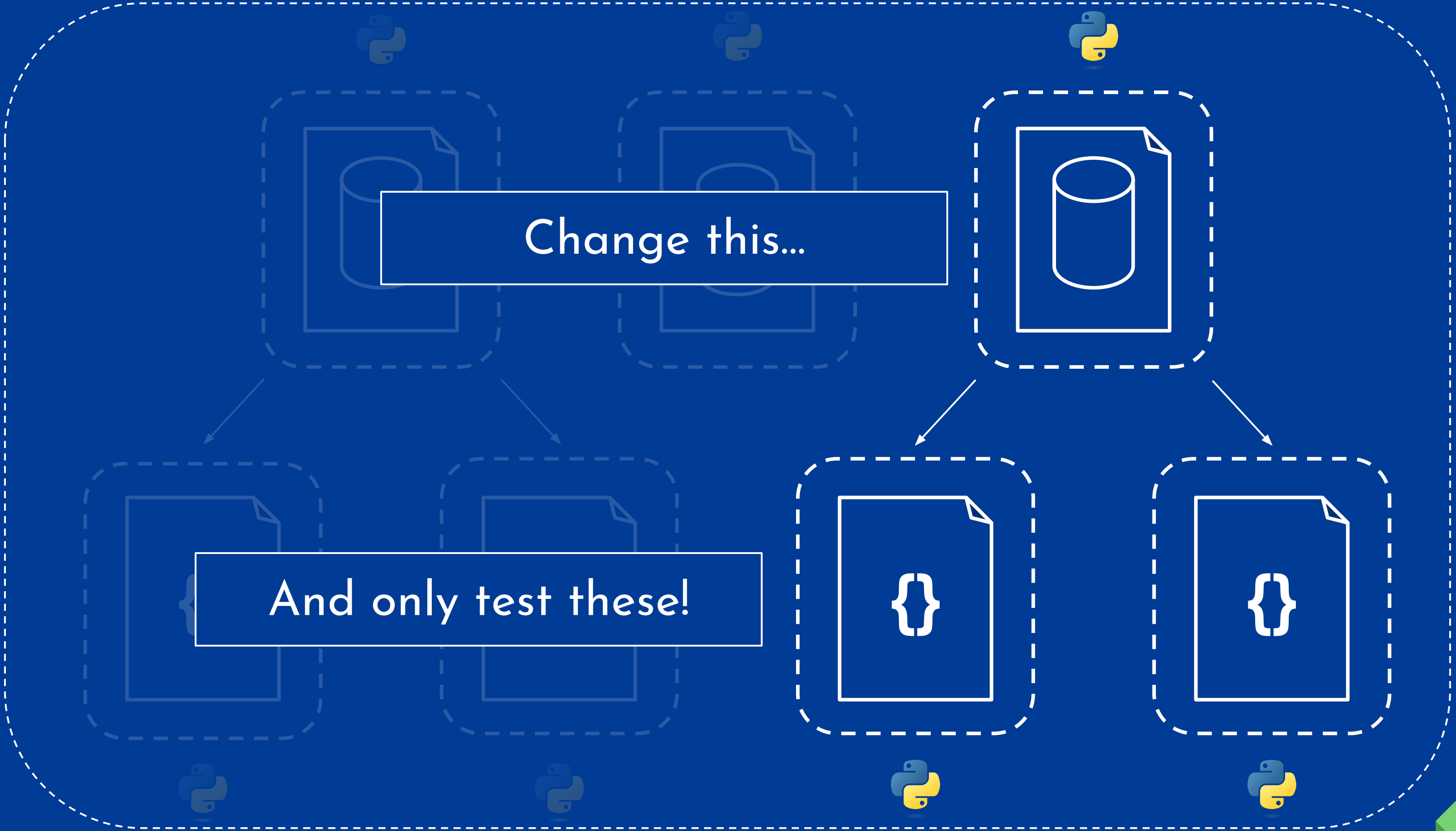


Bazel



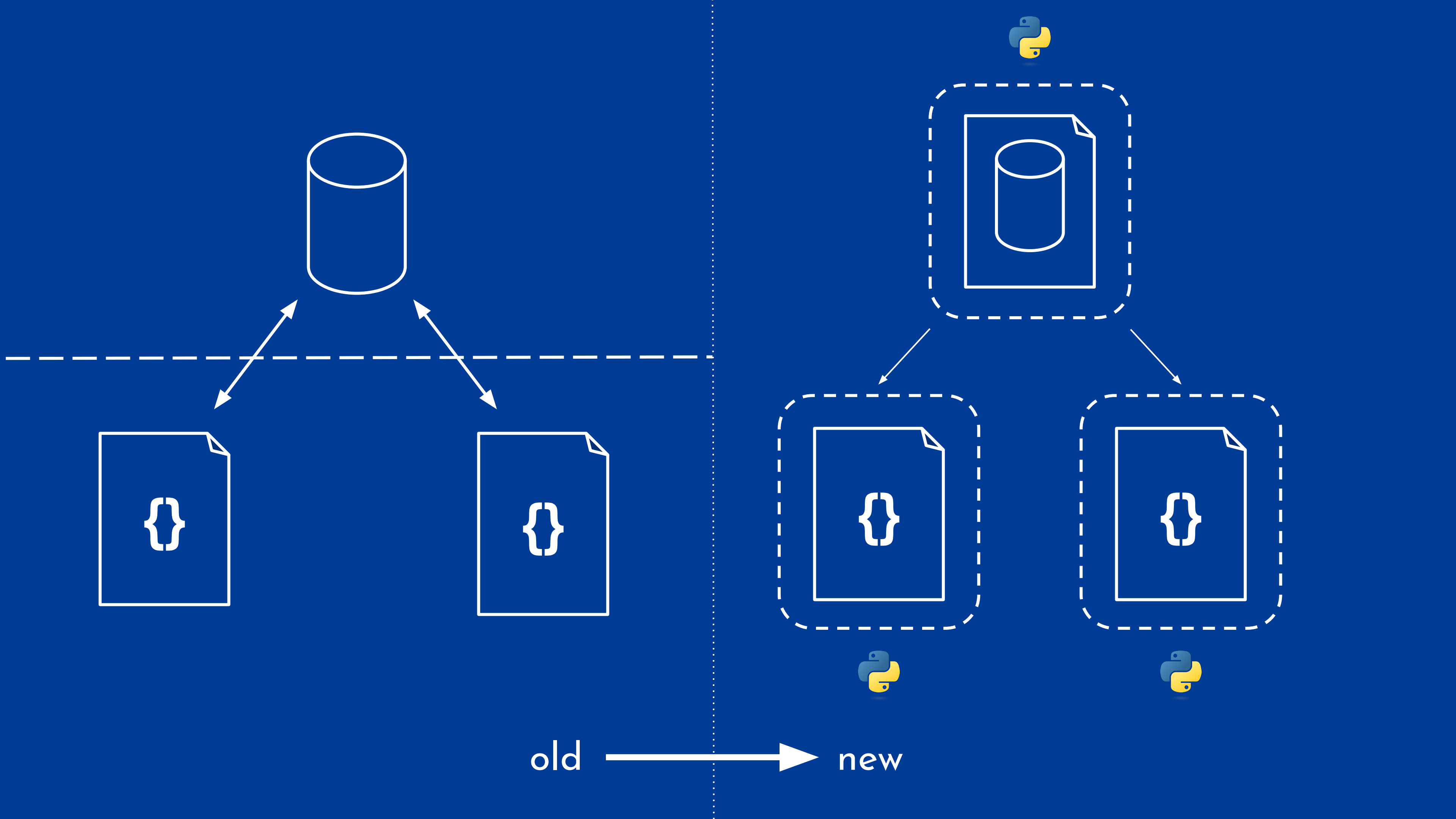
Change this...

Bazel


Change this...

And only test these!

Bazel

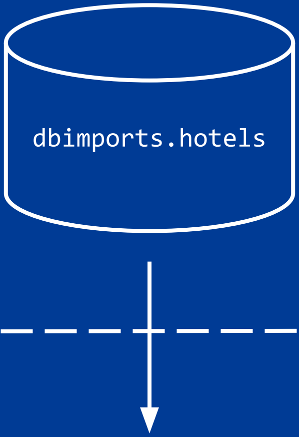

old
new

Community

Began as part of a single product team

Dogfooding!

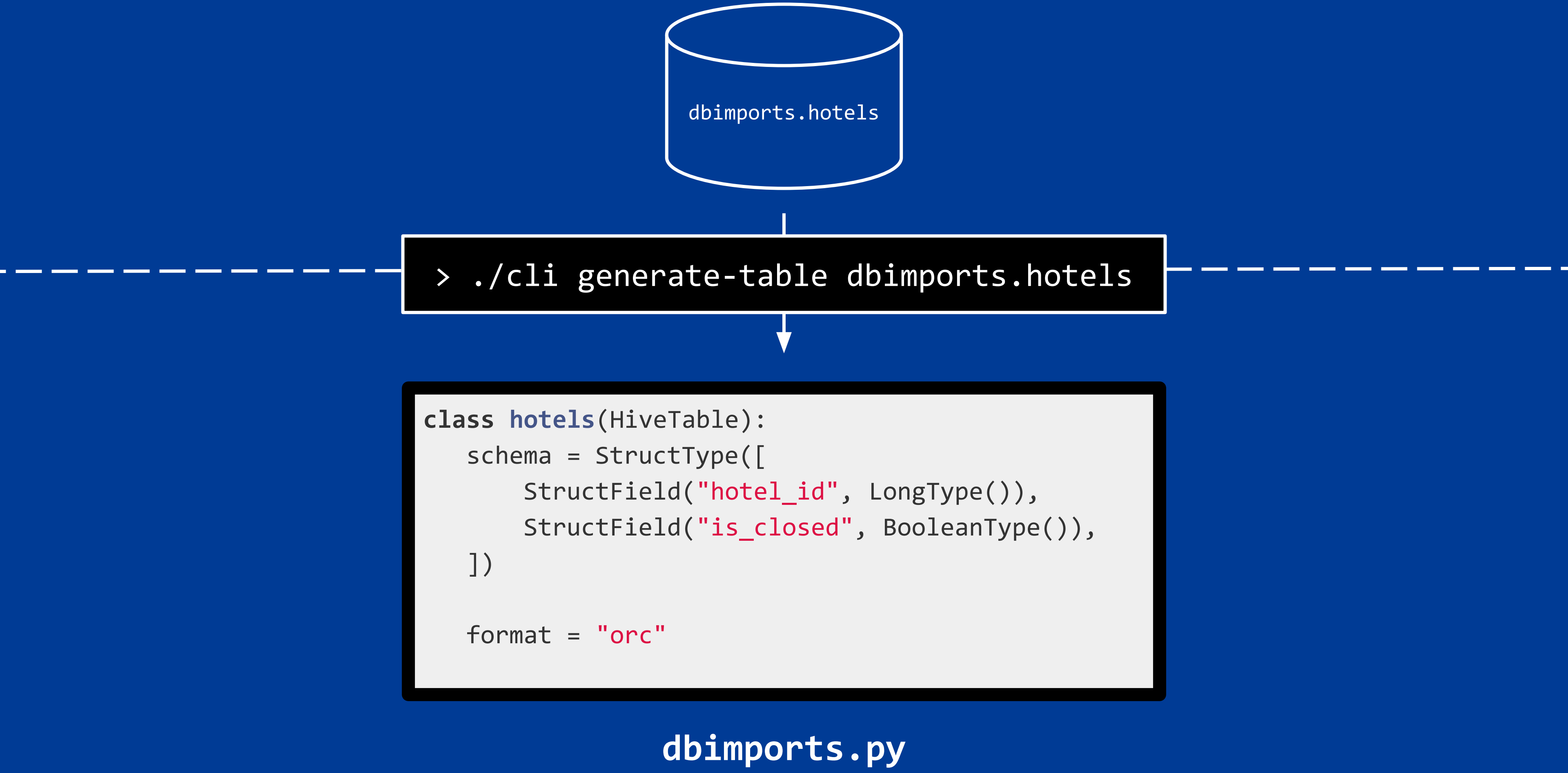
Began as part of a single product team



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

    format = "orc"
```

dbimports.py

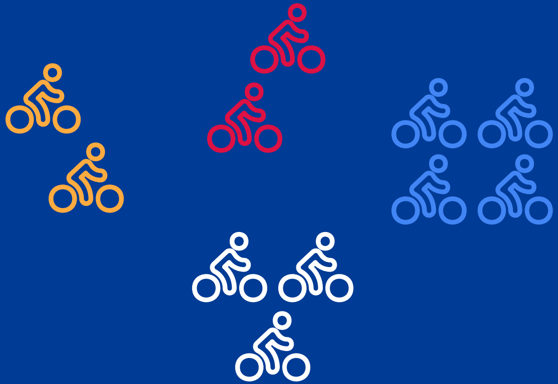


```
> ./cli generate-table dbimports.hotels
```

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

   format = "orc"
```

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

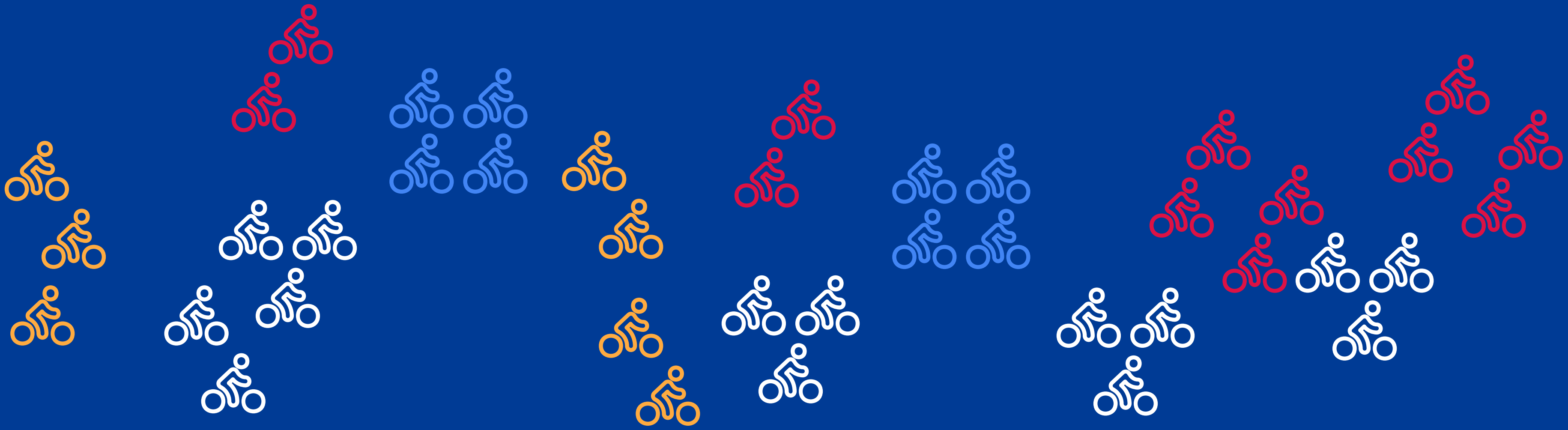


Grew organically across teams

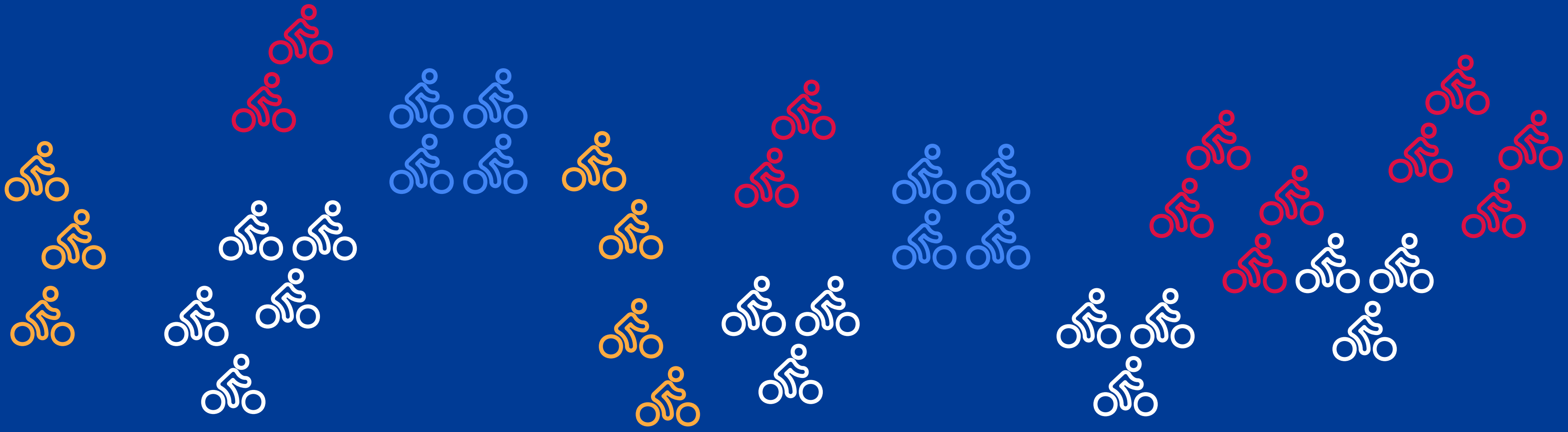
Testing

Testing

docker



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?