Ensuring your DAGs work before going to production

Airflow Summit 2020

Bas Harenslak

GO DATA DRIVEN

About me

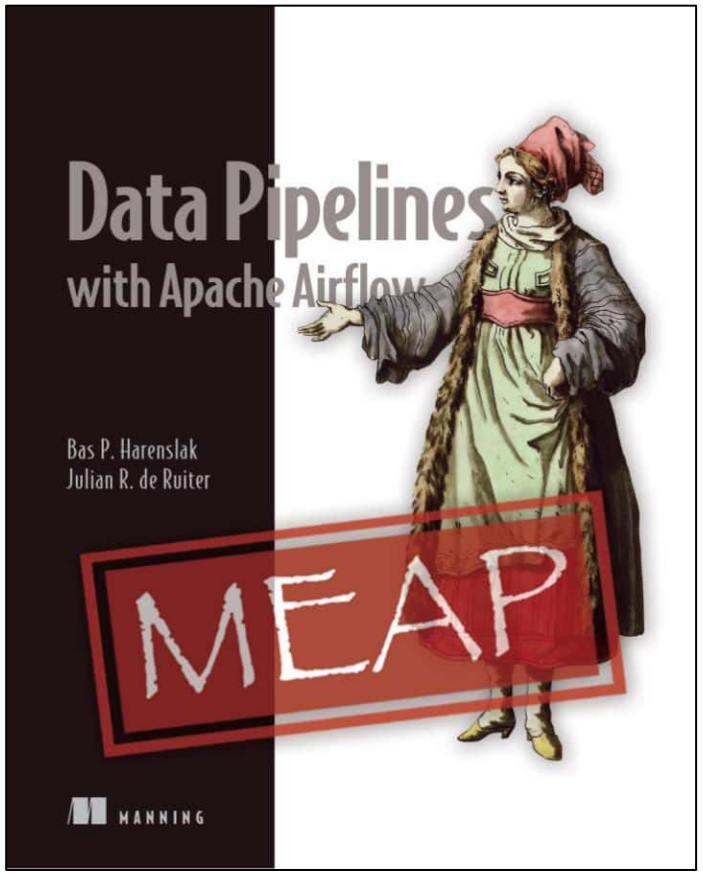
- Background in Software Engineering (Haagse Hogeschool) and Computer Science (University of Leiden)

- Data Engineer at GoDataDriven last 4,5 years

GO DATA DRIVEN

About me

- Apache Airflow committer
- (Co-)author of Manning's
  "Data Pipelines with Apache Airflow"
- First 10 chapters available in preview, new chapters added on a regular basis
- https://www.manning.com/books/data-pipelines-with-apache-airflow



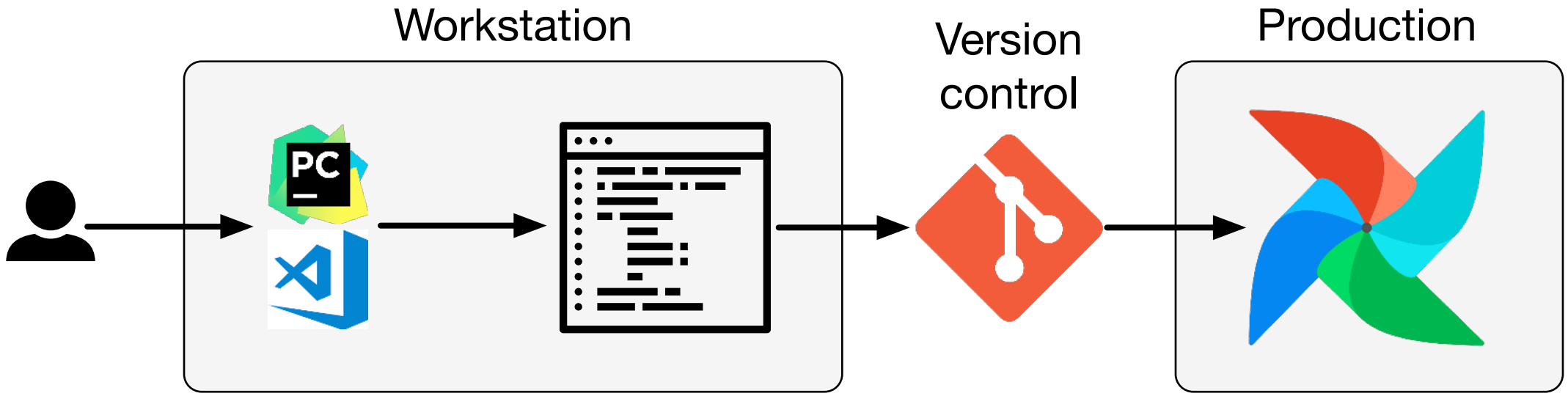
GO DATA DRIVEN

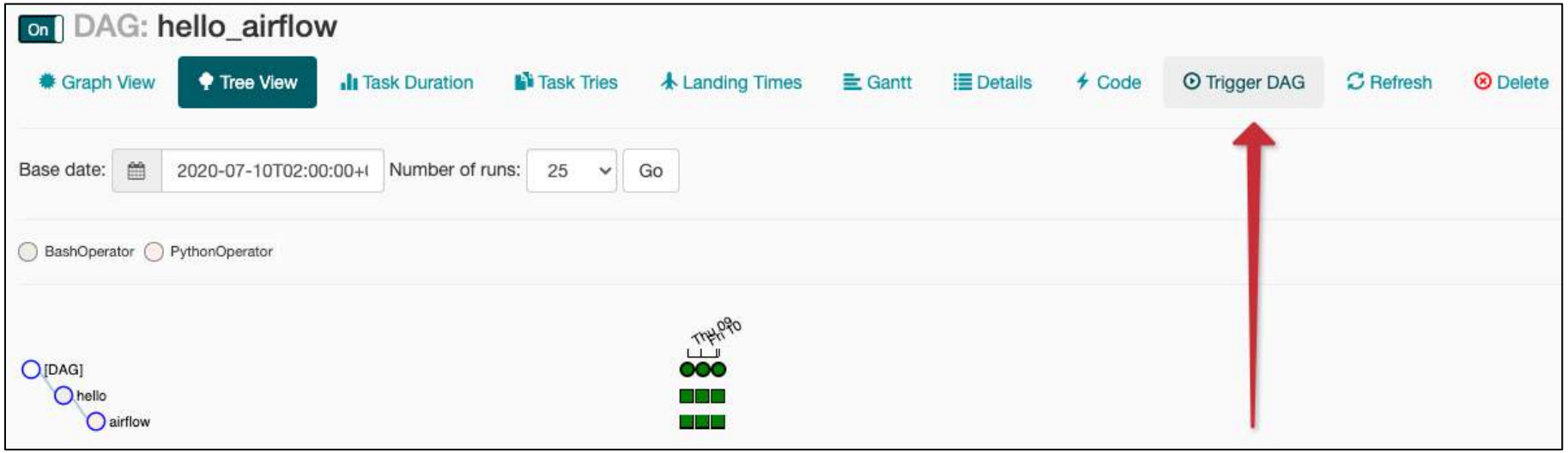
Ensuring your DAGs work before going to production

1. Unit testing
2. Testing in Python with pytest
3. Mocking
4. Using fake external systems
5. Debugging

GO DATA DRIVEN

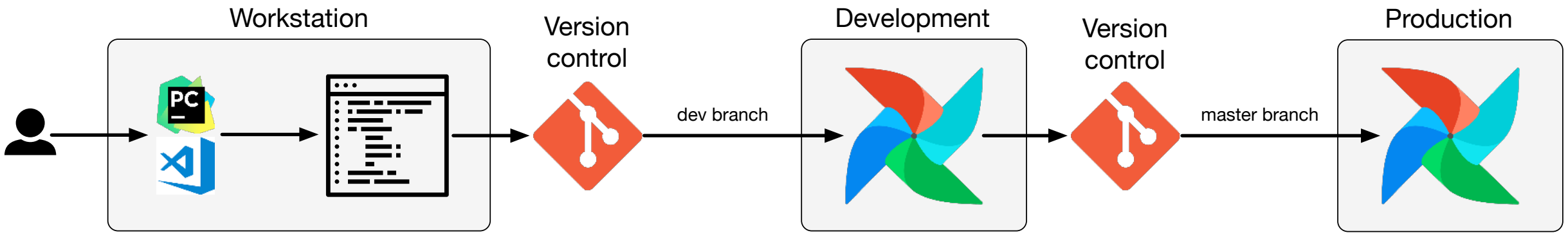
Typical very first deployment and test





GO DATA DRIVEN

Second deployment



Development:

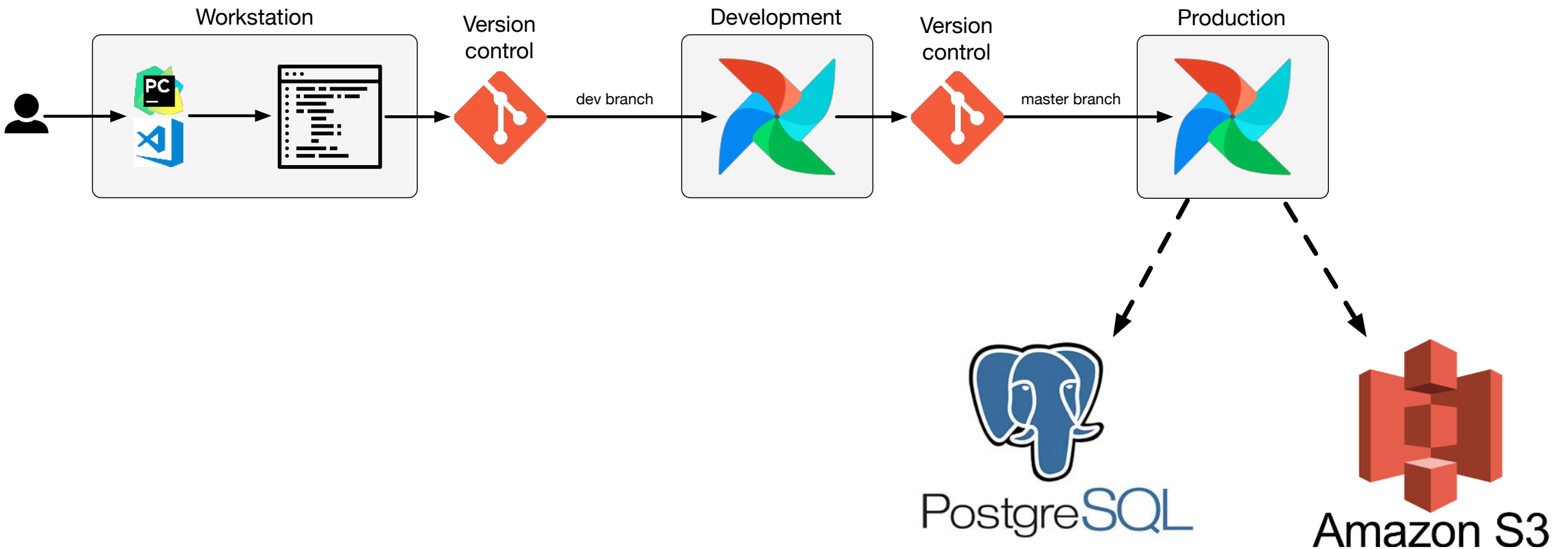
Production:

GO DATA DRIVEN

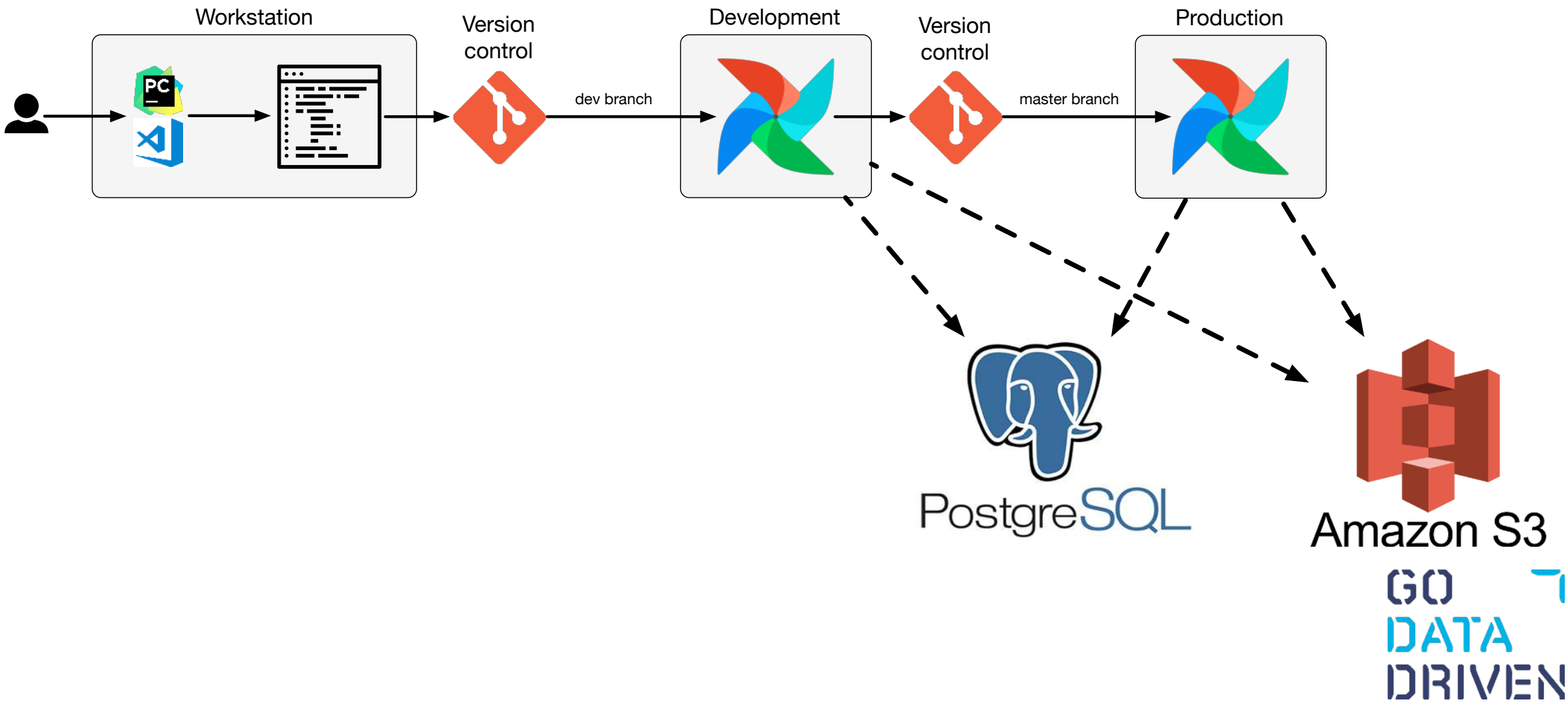
“Testing Airflow is hard”

GO DATA DRIVEN

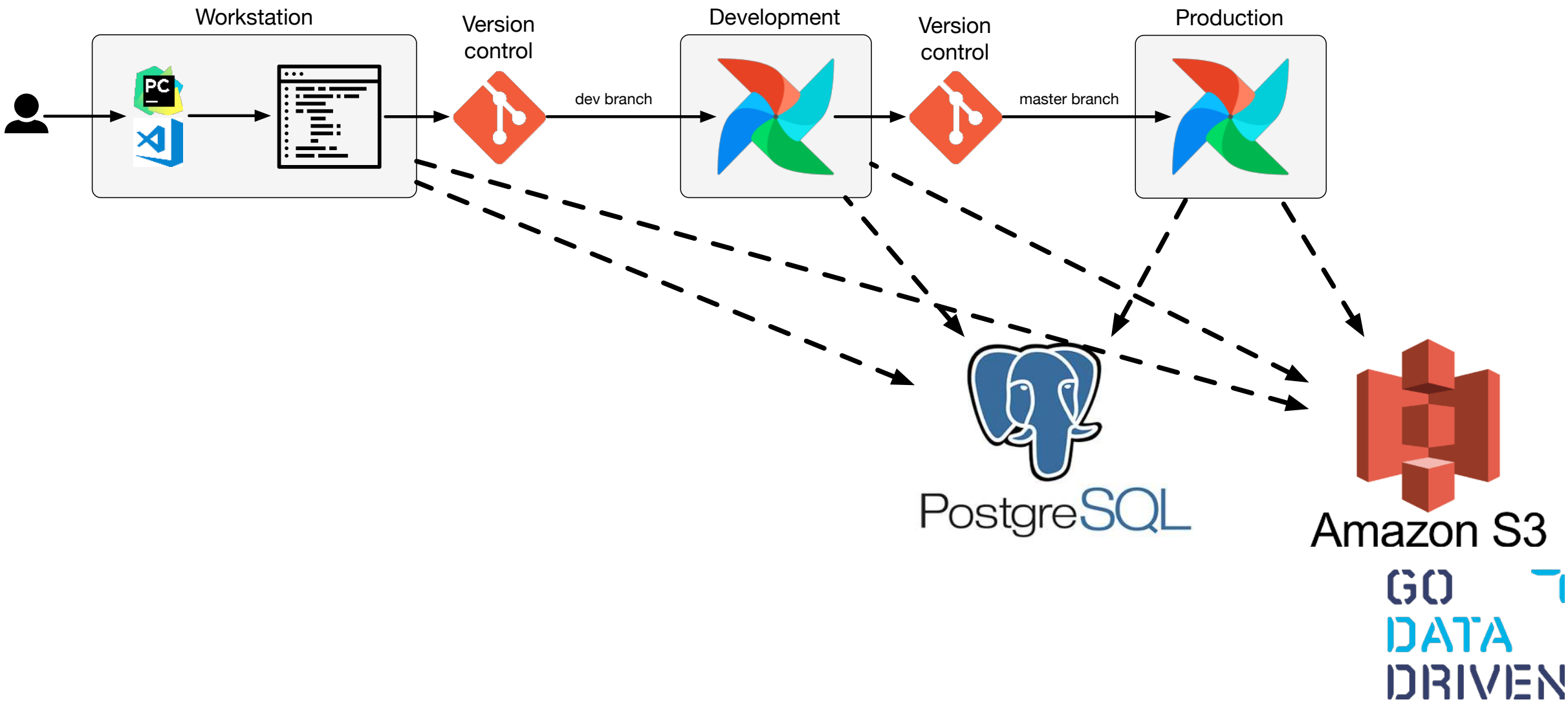
Testing Airflow is hard



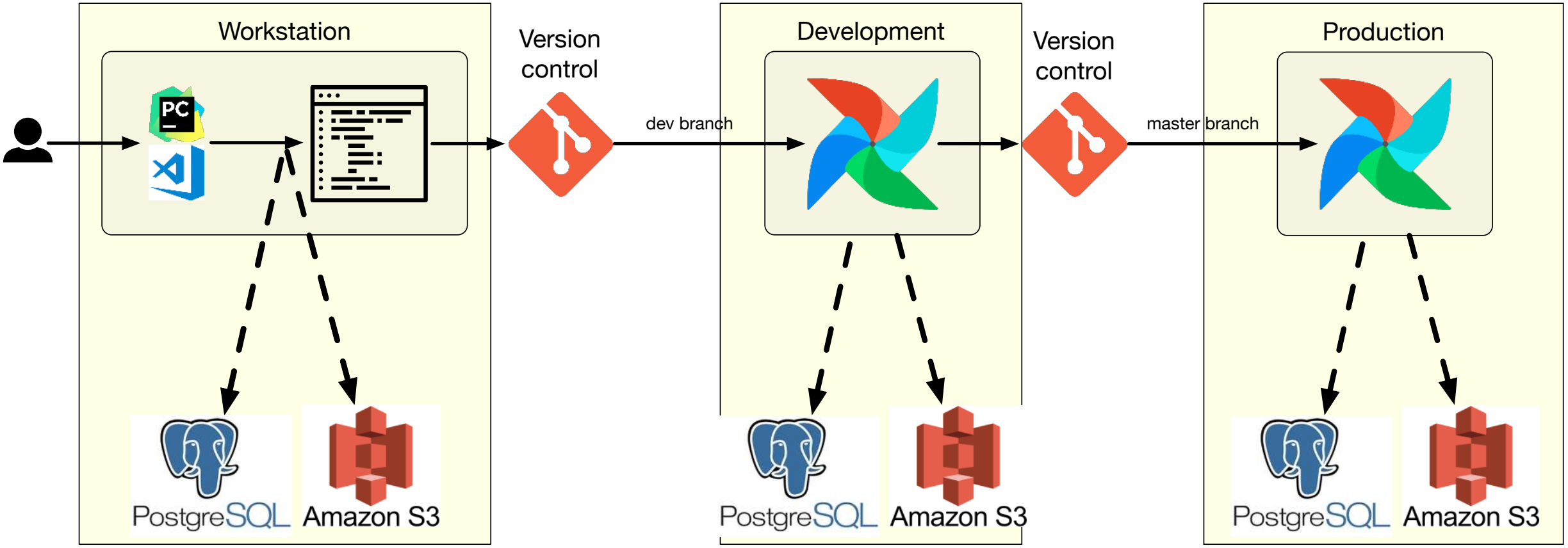
Testing Airflow is hard



Testing Airflow is hard



Testing Airflow is hard



GO DATA DRIVEN

Unit testing

GO DATA DRIVEN

What is a unit test?

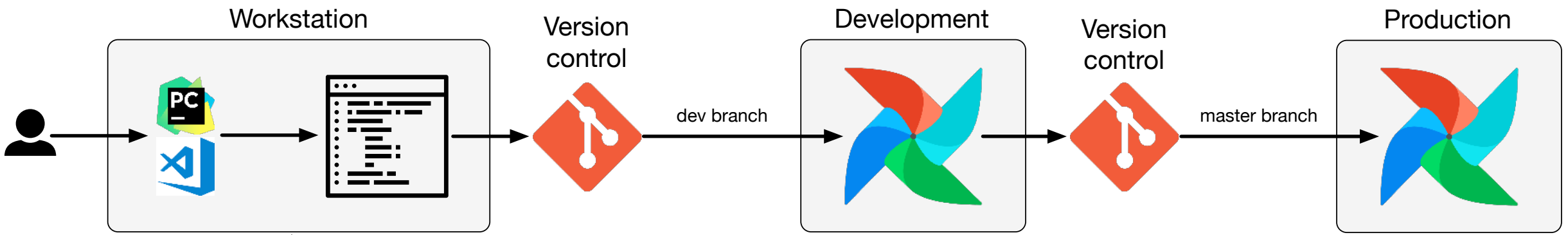
```
def count_lines_in_postgres_table(table_name):
    # SELECT COUNT(*) FROM table_name...
```

```
result = count_lines_in_postgres_table("foobar")
```

```
assert result == 5
```

GO DATA DRIVEN

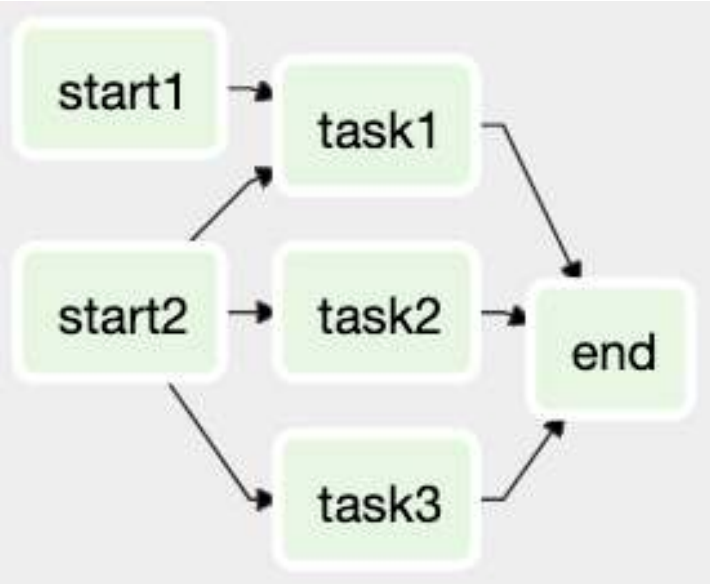
What is a unit test?




```
def count_lines_in_postgres_table(table_name):
    # SELECT COUNT(*) FROM table_name...

result = count_lines_in_postgres_table("foobar")

assert result == 5
```

GO DATA DRIVEN

Unit testing and Airflow



- A DAG consists of one or more tasks, represented as “Operators”

- We can test Operators as a whole
- Or callable functions (e.g. when using PythonOperator)
- Or underlying code (e.g. custom hooks/operators)

- Does *not* validate “integration” of multiple tasks

GO DATA DRIVEN

So how do I unit test an operator?

- “It depends”

- Example test for PythonOperator:

```
def test_python_operator():
    test = PythonOperator(task_id="test", python_callable=lambda: "testme")
    result = test.execute(context={})
    assert result == "testme"
```

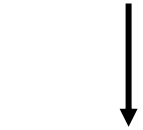
GO DATA DRIVEN

So how do I unit test an operator? (2)

- “It depends”

- Example test for BashOperator:

“it depends”



```
def test_bash_operator():
    test = BashOperator(task_id="test", bash_command="echo testme", xcom_push=True)
    result = test.execute(context={})
    assert result == "testme"
```

GO DATA DRIVEN

So how do I unit test an operator? (3)

- It depends:

```
class BaseOperator():
    # …

    def execute(self, context):
        raise NotImplementedError()
```

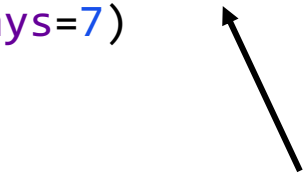
- Note `context` is a required argument, hence the "`context={}`".

GO DATA DRIVEN

How do I test with context?

- If you need one single, fixed, thing from the context:

```
def next_week(**context):
    return context["execution_date"] + datetime.timedelta(days=7)


def test_python_operator():
    test = PythonOperator(task_id="test", python_callable=next_week, provide_context=True)
    testdate = datetime.datetime(2020, 1, 1)
    result = test.execute(context={"execution_date": testdate})
    assert result == testdate + datetime.timedelta(days=7)
```



provide the context yourself

GO DATA DRIVEN

Testing with pytest

GO DATA DRIVEN

pytest

- Default testing package in Python in unittest

- Why 3rd party library pytest?
  - Because nice features (e.g. fixtures)

- https://pytest.org
- pip install pytest

GO DATA DRIVEN

pytest vs unittest example

pytest

```python
@pytest.fixture
def a():
    return 1


@pytest.fixture
def b():
    return 2


def test_sum(a, b):
    assert sum([a, b]) == 3
```

unittest

```python
class TestWithUnittest(unittest.TestCase):
    def setUp(self):
        self.a = 1
        self.b = 2

    def test_sum(self):
        self.assertEqual(sum([self.a, self.b]), 3)
```

GO DATA DRIVEN

pytest vs unittest example (2)

```python
@pytest.fixture
def a():
    return 1

@pytest.fixture
def b():
    return 2

@pytest.fixture
def c():
    return 3

def test_sum_ab(a, b):
    assert sum([a, b]) == 3

def test_sum_ac(a, c):
    assert sum([a, c]) == 4
```

```python
class TestWithUnittest(unittest.TestCase):
    def setUp(self):
        self.a = 1
        self.b = 2
        self.c = 3

    def test_sum_ab(self):
        self.assertEqual(sum([self.a, self.b]), 3)

    def test_sum_ac(self):
        self.assertEqual(sum([self.a, self.c]), 4)
```

GO DATA DRIVEN

pytest fixture scopes

- Scope of fixtures can be:
  - Function (default)
  - Class
  - Module
  - Package
  - Session

- Especially useful to define when (not) to re-use variables

GO DATA DRIVEN

pytest fixture scopes example

```python
@pytest.fixture(scope="module")
def a():
    return [1]


@pytest.fixture
def c():
    return [2]


class TestBla:
    def test_something(self, a):
        a.append(1)
        assert sum(a) == 2

    def test_something_ac(self, a, c):
        assert sum(a + c) == 3
```

Initialized ***once*** per module!

failure

GO DATA DRIVEN

pytest fixture scopes

- Module scope is especially useful for “expensive” things
  - e.g. database clients

GO DATA DRIVEN

pytest built-in fixtures

- capfd
- capfdbinary
- caplog
- capsys
- capsysbinary
- cache
- doctest_namespace
- monkeypatch
- pytestconfig
- record_property
- record_testsuite_property
- recwarn
- request
- testdir
- tmp_path
- tmp_path_factory
- tmpdir
- tmpdir_factory

https://docs.pytest.org/en/stable/fixture.html#fixtures

GO DATA DRIVEN

pytest tmpdir fixture

```python
def test_writing_to_disk(tmpdir):
    tmpfile = tmpdir.join("hello.txt")

    task = BashOperator(task_id="test", bash_command=f"echo 'hello' > {tmpfile}")
    task.execute(context={})

    assert len(tmpdir.listdir()) == 1
    assert tmpfile.read().replace("\n", "") == "hello"
```

Define tmp file path

Run BashOperator

Write “hello” to tmpfile

Assertions

GO DATA DRIVEN

Enough pytest – show me Airflow!

- One useful pytest plugin: pytest-helpers-namespace
  - `pip install pytest-helpers-namespace`

```
import datetime

import pytest
from airflow.models import DAG

pytest_plugins = ["helpers_namespace"]
```

- If you need the “full” context, you will need a DAG:

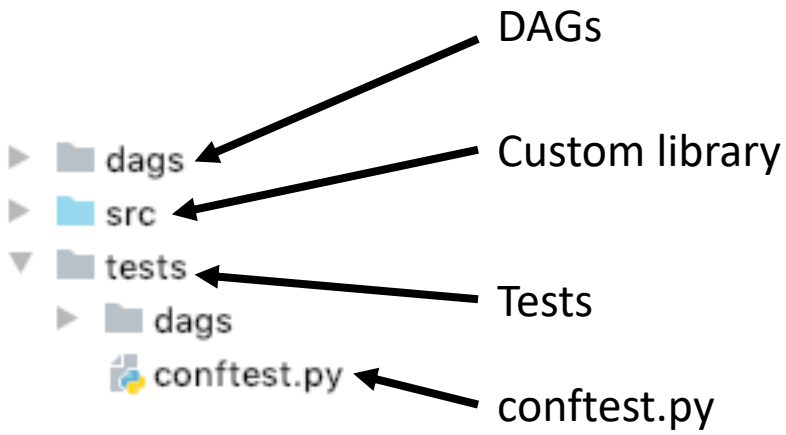
```
@pytest.fixture
def test_dag():
    """Airflow DAG for testing."""
    return DAG(
        "test_dag",
        start_date=datetime.datetime(2020, 1, 1),
        schedule_interval=datetime.timedelta(days=1),
    )
```

```
@pytest.helpers.register
def run_task(task, dag):
    """Run an Airflow task."""
    dag.clear()
    task.run(start_date=dag.start_date, end_date=dag.start_date)
```

GO DATA DRIVEN

Place this script in a conftest.py

- Pytest will auto-discover anything in /tests prefixed with `test_/Test`
- And automagically read everything defined in `conftest.py`
- ”per-directory configuration”



GO DATA DRIVEN

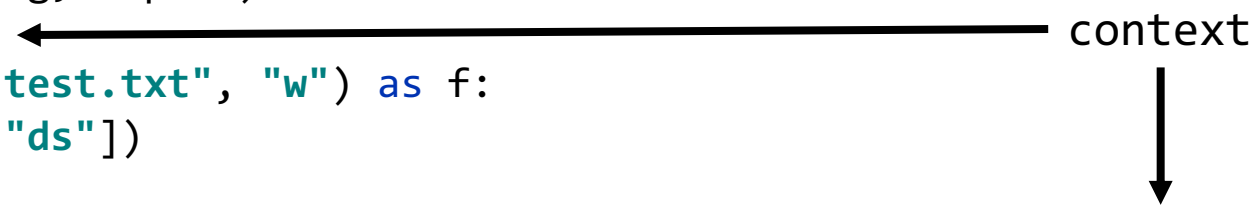
Running a task with DAG

```
def test_bash_operator_tmpdir(test_dag, tmpdir):
    tmpfile = tmpdir.join("hello.txt")

    task = BashOperator(task_id="test", bash_command=f"echo 'hello' > {tmpfile}", dag=test_dag)
    pytest.helpers.run_task(task=task, dag=test_dag)

    assert len(tmpdir.listdir()) == 1
    assert tmpfile.read().replace("\n", "") == "hello"
```

GO DATA DRIVEN

Running a task with full context

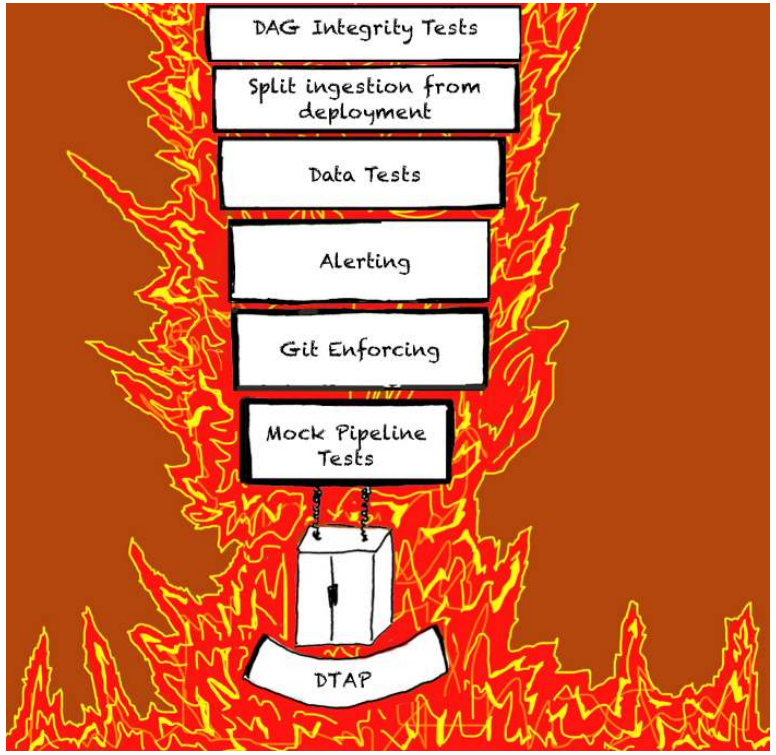
```
def test_full_context(test_dag, tmpdir):
    def do_magic(**context):
        with open(tmpdir / "test.txt", "w") as f:
            f.write(context["ds"])

    task = PythonOperator(task_id="test", python_callable=do_magic, provide_context=True, dag=test_dag)
    pytest.helpers.run_task(task=task, dag=test_dag)

    with open(tmpdir / "test.txt", "r") as f:
        assert f.readlines()[0] == test_dag.start_date.strftime("%Y-%m-%d")
```



GO DATA DRIVEN

The “DAG integrity test”

- Asserts the validity of DAG objects, i.e. “can Python instantiate this DAG”
- Plus, test for cycles
- https://medium.com/wbaa/datas-inferno-7-circles-of-data-testing-hell-with-airflow-cef4adff58d8



GO DATA DRIVEN

“Hello world” of checking DAG correctness

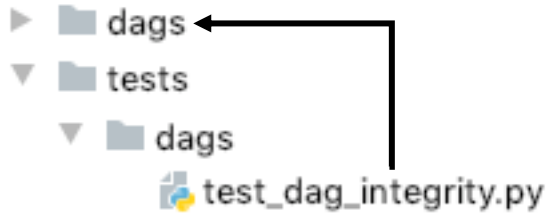
- `python your_dag.py`

- Does it take long?
- Does it produce many logs?
- Does it make connections to external systems?
- Etc…

- *Anything in the global scope must be avoided!*

GO DATA DRIVEN

The “DAG integrity test”




```python
import glob
from os import path

import pytest
from airflow import models as airflow_models

DAG_PATHS = glob.glob(path.join(path.dirname(__file__), "..", "..", "dags", "*.py"))


@pytest.mark.parametrize("dag_path", DAG_PATHS)
def test_dag_integrity(dag_path):
    """Import DAG files and check for a valid DAG instance."""
    dag_name = path.basename(dag_path)
    module = _import_file(dag_name, dag_path)

    # Validate if there is at least 1 DAG object in the file
    dag_objects = [var for var in vars(module).values() if isinstance(var, airflow_models.DAG)]
    assert dag_objects

    # For every DAG object, test for cycles
    for dag in dag_objects:
        dag.test_cycle()


def _import_file(module_name, module_path):
    import importlib.util

    spec = importlib.util.spec_from_file_location(module_name, str(module_path))
    module = importlib.util.module_from_spec(spec)
    spec.loader.exec_module(module)
    return module
```

For every DAG file

Assert if valid DAG object

Test for cycles

GO DATA DRIVEN

The “DAG integrity test”

```
task1 = DummyOperator(task_id="task1", dag=dag)
task2 = DummyOperator(task_id="task2", dag=dag)

task1 >> task2 >> task1
```

```
airflow.exceptions.AirflowDagCycleException: Cycle detected in DAG. Faulty task: task2 to task1
```

GO DATA DRIVEN

Mocking

GO DATA DRIVEN

Mocking – what and why?

- Provide “canned” responses, typically to fake calls to external systems
- For example – to test a function processing an API call, provide it a fake result to validate the processing part.
- Within Airflow – we can use mocking to e.g. provide a Connection object, so we don’t need a metastore for testing.
- Sometimes requires digging in internal code.

GO DATA DRIVEN

Mocking example

- Let’s consider an external API (something not maintained by ourselves):

- https://api.sunrise-sunset.org/json?lat=52.370216&lng=4.895168&formatted=0
- Returns sunrise and sunset times of the given location (Amsterdam city centre):


```
{
  "results": {
    "sunrise": "2020-07-12T03:34:14+00:00",
    "sunset": "2020-07-12T19:58:09+00:00",
    "solar_noon": "2020-07-12T11:46:12+00:00",
    "day_length": 59035,
    "civil_twilight_begin": "2020-07-12T02:47:34+00:00",
    "civil_twilight_end": "2020-07-12T20:44:49+00:00",
    "nautical_twilight_begin": "2020-07-12T01:37:06+00:00",
    "nautical_twilight_end": "2020-07-12T21:55:17+00:00",
    "astronomical_twilight_begin": "1970-01-01T00:00:01+00:00",
    "astronomical_twilight_end": "1970-01-01T00:00:01+00:00"
  },
  "status": "OK"
}
```


Sunrise/sunset times on the date of making this call

GO DATA DRIVEN

Mocking example

Test definition

Mock BaseHook.get_connection()

Fake Connection response

```python
from datetime import datetime

import pytest
from airflow.hooks.base_hook import BaseHook
from airflow.models import Connection
from airflow.operators.http_operator import SimpleHttpOperator


def test_simple_http_operator(test_dag, mocker):
    mocker.patch.object(
        BaseHook,
        "get_connection",
        return_value=Connection(schema="https", host="api.sunrise-sunset.org")
    )

    def _check_light(sunset_sunrise_response):
        results = sunset_sunrise_response.json()["results"]
        sunrise = datetime.strptime(results["sunrise"][:-6], "%Y-%m-%dT%H:%M:%S")
        sunset = datetime.strptime(results["sunset"][:-6], "%Y-%m-%dT%H:%M:%S")

        if sunrise < datetime.utcnow() < sunset:
            print("It is light!")
        else:
            print("It is dark!")

        return True

    is_it_light = SimpleHttpOperator(
        task_id="is_it_light",
        http_conn_id="my_http_conn",
        endpoint="json",
        method="GET",
        data={"lat": "52.370216", "lng": "4.895168", "formatted": "0"},
        response_check=_check_light,
        dag=test_dag,
    )

    pytest.helpers.run_task(task=is_it_light, dag=test_dag)
```

GO DATA DRIVEN

Mocking and pytest

- Unittest comes with a built-in mocking package (`from unittest import mock`)
- While you can use this in pytest
- A convenient plugin is pytest-mock, which makes it available as a fixture
- `pip install pytest-mock`

GO DATA DRIVEN

How to mock the ”unmockable”?

- Some systems cannot be mocked
- Cloud services for example don’t always provide a mocking library

- Three options:
  1. Mock and assert calls to the external systems
  2. Run a Dockerized version of the external system (e.g. PostgreSQL)
  3. Run a development version of the external system (e.g. Google Cloud Storage)

GO DATA DRIVEN

Mock and assert calls to external systems

```python
def test_simple_http_operator_no_external_call(test_dag, mocker):
    mocker.patch.object(
        BaseHook, "get_connection", return_value=Connection(schema="https", host="api.sunrise-sunset.org")
    )
    mock_run = mocker.patch.object(HttpHook, "run")

    is_it_light = SimpleHttpOperator(
        task_id="is_it_light",
        http_conn_id="my_http_conn",
        endpoint="json",
        method="GET",
        data={"lat": "52.370216", "lng": "4.895168", "date": "{{ ds }}", "formatted": "0"},
        dag=test_dag,
    )

    pytest.helpers.run_task(task=is_it_light, dag=test_dag)
    mock_run.assert_called_once()
    assert mock_run.call_args_list[0][0][1] == {
        "lat": "52.370216",
        "lng": "4.895168",
        "date": test_dag.start_date.strftime("%Y-%m-%d"),
        "formatted": "0",
    }
```

GO DATA DRIVEN

Faking external systems

Testing with Docker

- We can run external systems for the duration of our tests in Docker
- Note: a Docker image of your desired system must exist (which is not the case for e.g. Google Cloud Storage)

- One option: create a Docker Compose for your tests
- Alternative option: use `pytest_docker_tools` plugin

GO DATA DRIVEN

pytest_docker_tools

- `pip install pytest_docker_tools`
- Small wrapper around your Docker client (so Docker must be installed!)
- Provide access to Docker SDK with a set of pytest fixtures

GO DATA DRIVEN

pytest_docker_tools

- Say we implemented a “PostgresToLocalOperator”, which runs a PostgreSQL query and writes the result to a local file.

- We need:
  - PostgreSQL database
  - Some location to write to locally
  - Data in the database to download

GO DATA DRIVEN

pytest_docker_tools – create database

```
from pytest_docker_tools import container, fetch


postgres_image = fetch(repository="postgres:11.1-alpine")

postgres = container(
    image="{postgres_image.id}",
    environment={
        "POSTGRES_USER": "secretuser",
        "POSTGRES_PASSWORD": "secretpassword",
    },
    ports={"5432/tcp": None},
    volumes={
        path.join(path.dirname(__file__), "postgres-init.sql"): {
            "bind": "/docker-entrypoint-initdb.d/postgres-init.sql"
        }
    },
)
```

Defines Docker image

Defines Docker container (very similar to docker run …)

GO DATA DRIVEN

pytest_docker_tools – initialize database

postgres-init.sql:

```
SET search_path TO public;
CREATE TABLE dummy (
    id integer,
    name character varying(255)
);
INSERT INTO dummy (id,name) VALUES (1, 'dummy1');
INSERT INTO dummy (id,name) VALUES (2, 'dummy2');
INSERT INTO dummy (id,name) VALUES (3, 'dummy3');
```

GO DATA DRIVEN

pytest_docker_tools – the actual test

Mock Postgres connection

Reference to Postgres container

Call the operator

Check results

```python
def test_postgres_to_local_operator(test_dag, mocker, tmpdir, postgres):
    mocker.patch.object(
        PostgresHook,
        "get_connection",
        return_value=Connection(
            host="localhost",
            conn_type="postgres",
            login=postgres_credentials.username,
            password=postgres_credentials.password,
            port=postgres.ports["5432/tcp"][0],
        ),
    )

    output_path = str(tmpdir / "pg_dump")
    task = PostgresToLocalOperator(
        task_id="test",
        postgres_conn_id="postgres",
        pg_query="SELECT * FROM dummy",
        local_path=output_path,
        dag=test_dag,
    )
    pytest.helpers.run_task(task=task, dag=test_dag)

    # Assert if output file exists
    output_file = Path(output_path)
    assert output_file.is_file()

    # Assert file contents, should be the same as in postgres-init.sql
    expected = [
        {"id": 1, "name": "dummy1"},
        {"id": 2, "name": "dummy2"},
        {"id": 3, "name": "dummy3"},
    ]
    with open(output_file, "r") as f:
        assert json.load(f) == expected
```

GO DATA DRIVEN

Debugging

GO DATA DRIVEN

airflow test

- `airflow test [dag id] [task id] [execution date]`
- E.g. “airflow test mydag mytask 2020-01-01”

- Running a single task from the command line, for a given execution date

- No state recorded in metastore

“airflow task test” in Airflow 2.0

GO DATA DRIVEN

Debugging Airflow code

- Set breakpoints in your IDE locally:

```
task = PostgresToLocalOperator(
    task_id="test",
    postgres_conn_id="postgres",
    pg_query="SELECT * FROM dummy",
    local_path=output_path,
    dag=test_dag,
)
pytest.helpers.run_task(task=task, dag=test_dag)

# Assert if output file exists
output_file = Path(output_path)
assert output_file.is_file()
```

GO DATA DRIVEN

Debugging in production

- Try to avoid
- But if you must;
- (i)Python DeBugger: (i)pdb

```
import ipdb
ipdb.set_trace()
```

GO DATA DRIVEN

Debugging in production (2)

Python Debugger Cheatsheet

Getting started

start pdb from within a script:
import pdb;pdb.set_trace()
start pdb from the commandline:
python -m pdb <file.py>

Basics

h(elp)	print available commands
h(elp) ***command***	print help about *command*
q(uit)	quit debugger

Examine

p(rint) ***expr***	print the value of *expr*
pp ***expr***	pretty-print the value of *expr*
w(here)	print current position (including stack trace)
l(ist)	list 11 lines of code around the current line
l(ist) ***first*****,** ***last***	list from *first* to *last* line number
a(rgs)	print the args of the current function

Movement

<ENTER>	repeat the last command
n(ext)	execute the current statement (step over)
s(tep)	execute and step into function
r(eturn)	continue execution until the current function returns
c(ontinue)	continue execution until a breakpoint is encountered
u(p)	move one level up in the stack trace
d(own)	move one level down in the stack trace

Breakpoints

b(reak)	show all breakpoints
b(reak) ***lineno***	set a breakpoint at *lineno*
b(reak) ***func***	set a breakpoint at the first line of a *func*

Manipulation

!*stmt* treat *stmt* as a Python statement instead of a pdb command

GO DATA DRIVEN

airflow test

- airflow test [dag_id] [task_id] [execution_date]

“airflow tasks test” in Airflow 2.0

GO DATA DRIVEN

DebugExecutor

- https://airflow.apache.org/docs/stable/executor/debug.html

- Add to your DAG script:

```
if __name__ == "__main__":
    dag.clear(reset_dag_runs=True)
    dag.run()
```

- export `AIRFLOW__CORE__EXECUTOR=DebugExecutor`
- And run with python your_dag.py
- ***Warning***: will run ALL runs from configured start_date!

GO DATA DRIVEN

Coming in Airflow 2.0: airflow dags test

- Complete CLI was rewritten in Airflow 2.0
- Can run a DAG using the DebugExecutor *without* editing the DAG

```
airflow dags test dag_id execution_date
```

- Executes all tasks in the order they're defined in

GO DATA DRIVEN

Final words

- Airflow operators can be tested (run) by simply calling them
- If you want the context, you need a DAG (or explicitly provide it yourself)
- Mocking is useful for e.g. avoiding metastore calls
- Docker can be useful for running systems temporarily during a test
- “airflow test” can be helpful for debugging

GO DATA DRIVEN

Final words (2)

- Talk is recorded
- Code is available at https://github.com/godatadriven/airflow-testing-examples
- @basph on Airflow Slack

GO DATA DRIVEN