# Modern day workflow management

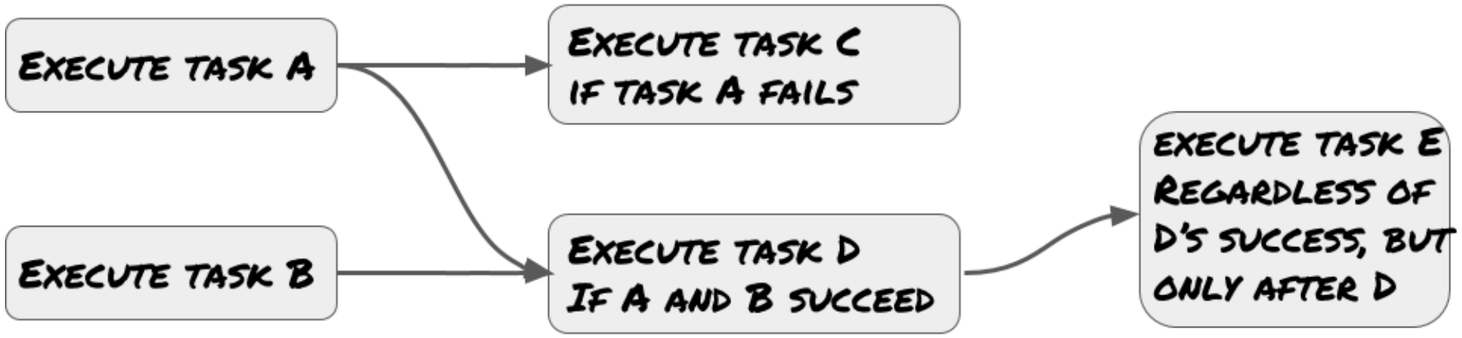
BUILDING DATA ENGINEERING PIPELINES IN PYTHON

**Oliver Willekens**
Data Engineer at Data Minded

# What is a workflow?

A workflow:

- Sequence of tasks




- *Scheduled* at a time or *triggered* by an event
- Orchestrate data processing pipelines

# Scheduling with cron

Cron reads “crontab” files:

- tabulate tasks to be executed at certain times
- one task per line

```
*/15 9-17 * * 1-3,5 log_my_activity
```

# Scheduling with cron

```
*/15 9-17 * * 1-3,5 log_my_activity

____
```

# Scheduling with cron

The Airflow task:

- An instance of an Operator class
  - Inherits from `BaseOperator` -> Must implement `execute()` method.
- Performs a specific action (delegation):
  - `BashOperator` -> run bash command/script
  - `PythonOperator` -> run Python script
  - `SparkSubmitOperator` -> submit a Spark job with a cluster

# Scheduling with cron

```
*/15 9-17 * * 1-3,5 log_my_activity
          _
```

# Scheduling with cron

```
*/15 9-17 * * 1-3,5 log_my_activity
            _
```

# Scheduling with cron

```
*/15 9-17 * * 1-3,5 log_my_activity
             _____
```

# Scheduling with cron

```
*/15 9-17 * * 1-3,5 log_my_activity
```

# Scheduling with cron

```
*/15 9-17 * * 1-3,5 log_my_activity

# Minutes   Hours   Days   Months   Days of the week   Command
*/15        9-17    *      *        1-3,5              log_my_activity
```

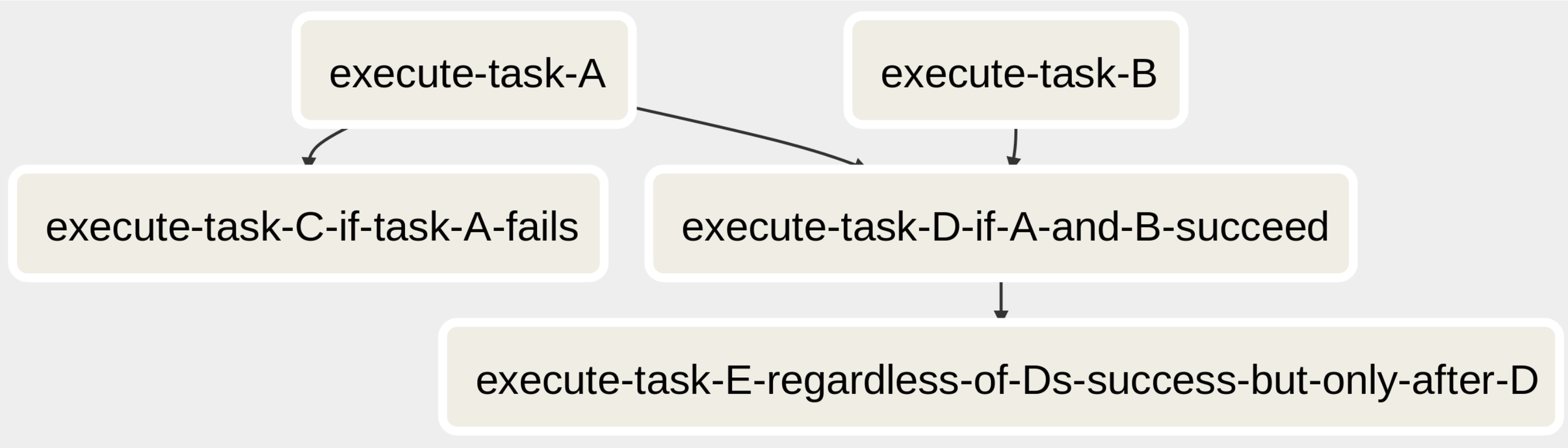
Cron is a dinosaur.

Modern workflow managers:

- Luigi (Spotify, 2011, Python-based)
- Azkaban (LinkedIn, 2009, Java-based)
- Airflow (Airbnb, 2015, Python-based)

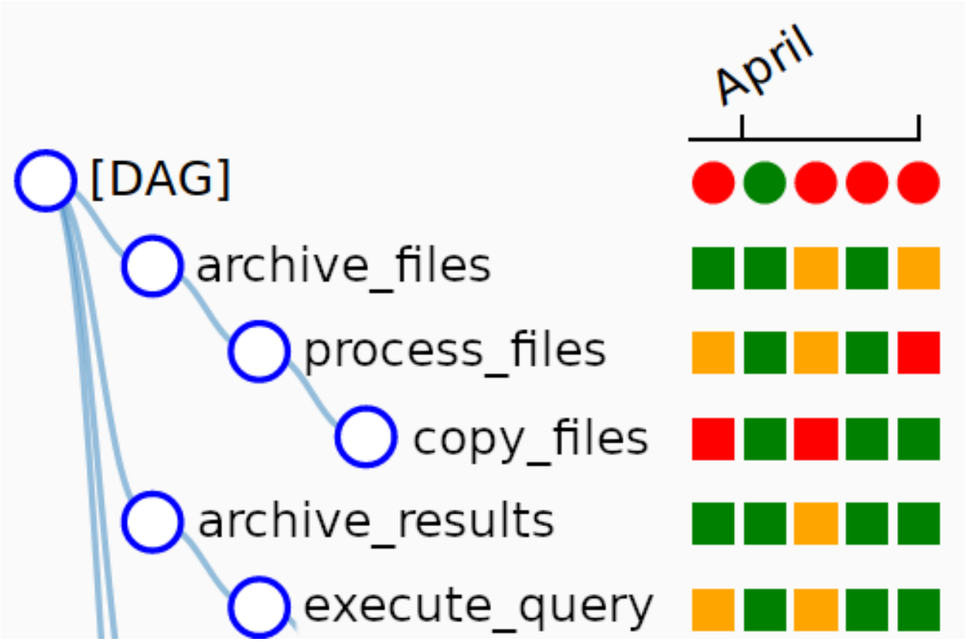
# Apache Airflow fulfills modern engineering needs

1. Create and visualize complex workflows,



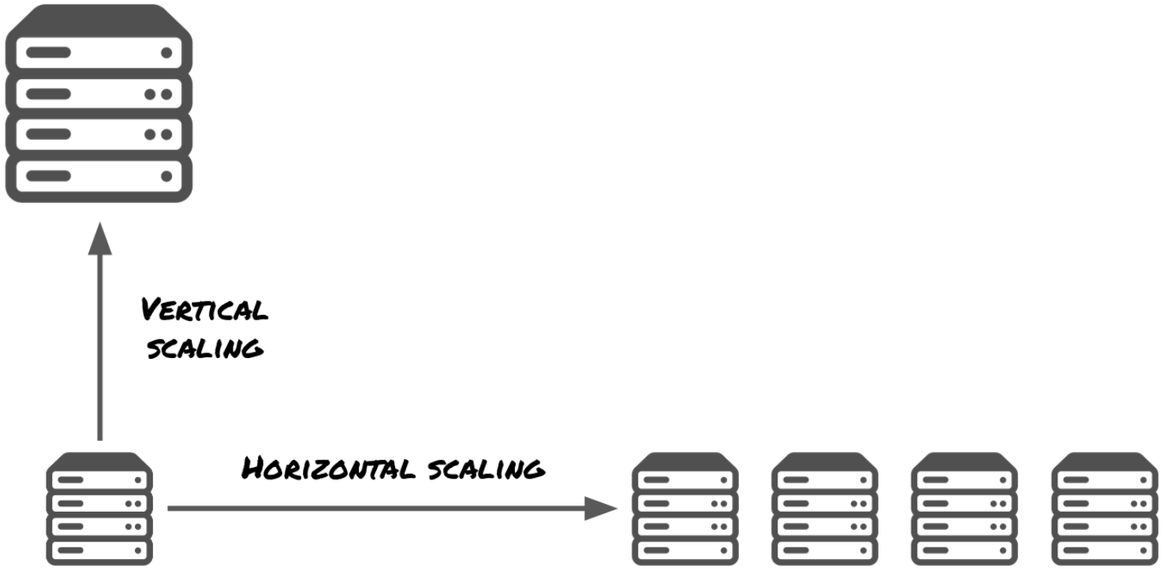
# Apache Airflow fulfills modern engineering needs

1. Create and visualize complex workflows,
2. Monitor and log workflows,

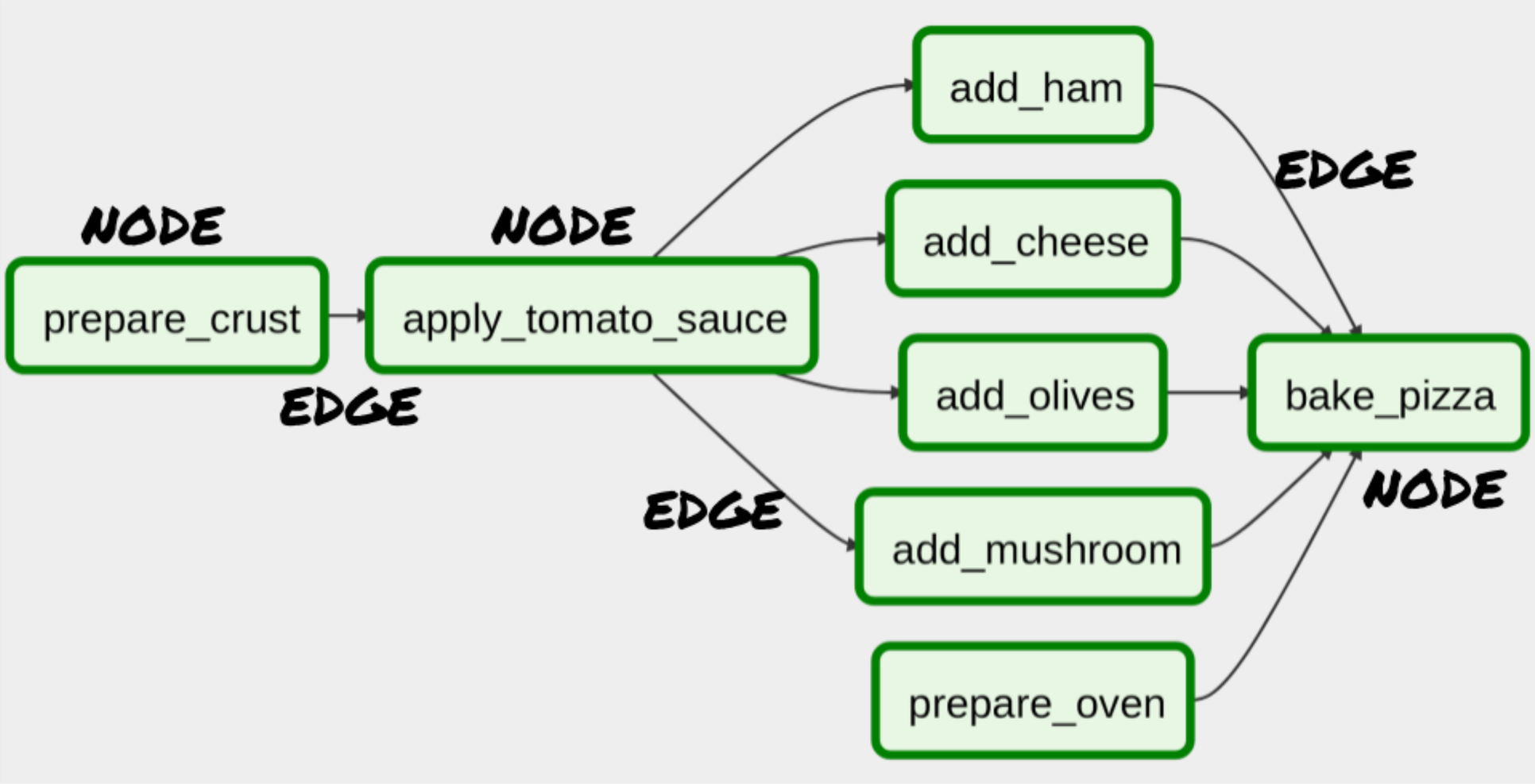


# Apache Airflow fulfills modern engineering needs

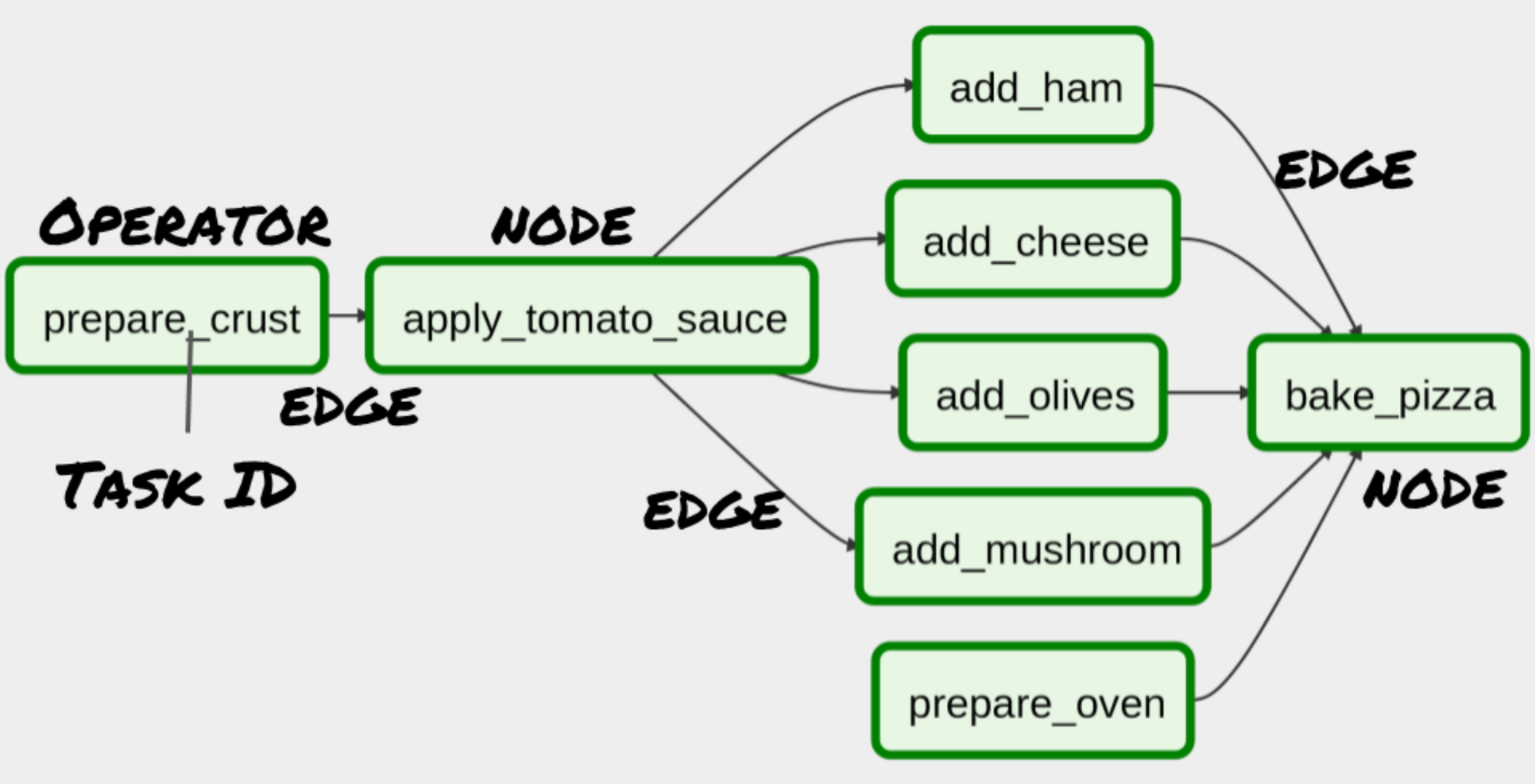
1. Create and visualize complex workflows,
2. Monitor and log workflows,
3. Scales horizontally.



# The Directed Acyclic Graph (DAG)



# The Directed Acyclic Graph (DAG)



# The Directed Acyclic Graph in code

```
from airflow import DAG

my_dag = DAG(
  dag_id="publish_logs",
  schedule_interval="* * * * *",
  start_date=datetime(2010, 1, 1)
)
```

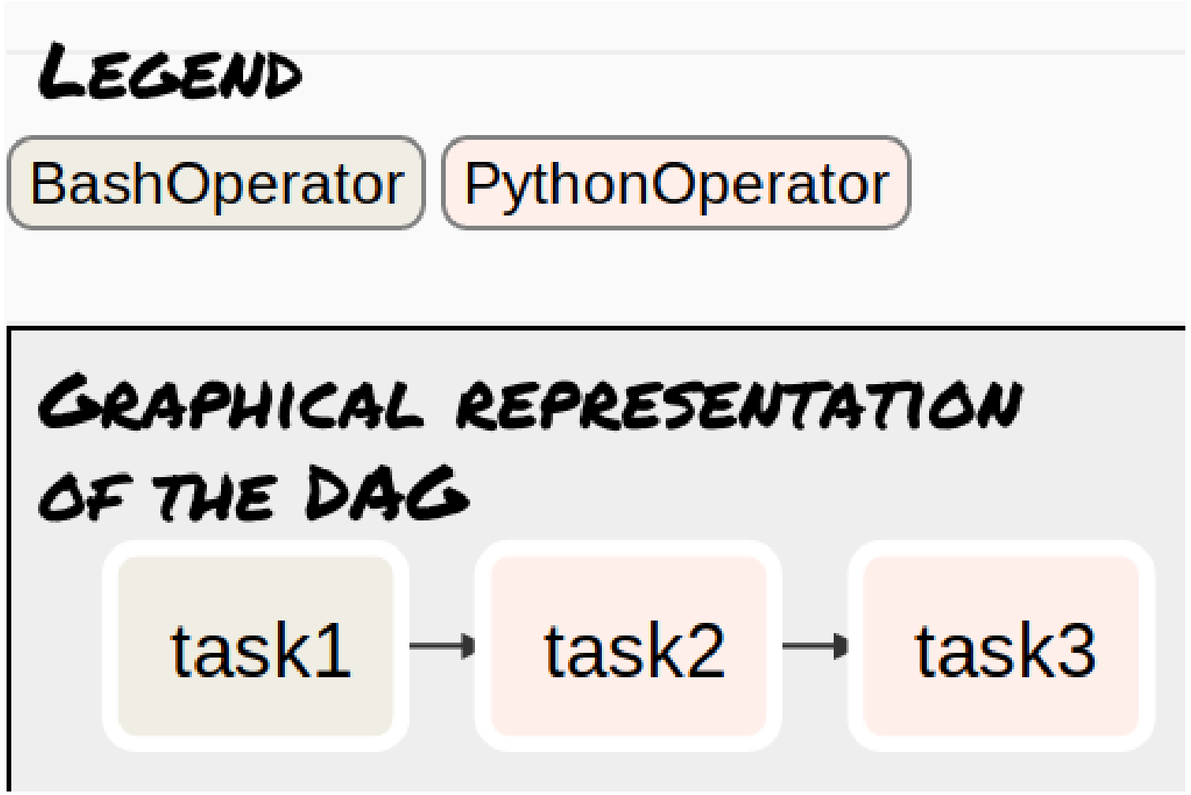
# Classes of operators

The Airflow task:

- An instance of an Operator class
  - Inherits from `BaseOperator` -> Must implement `execute()` method.
- Performs a specific action (delegation):
  - `BashOperator` -> run bash command/script
  - `PythonOperator` -> run Python script
  - `SparkSubmitOperator` -> submit a Spark job with a cluster

# Expressing dependencies between operators

```
dag = DAG(…)
task1 = BashOperator(…)
task2 = PythonOperator(…)
task3 = PythonOperator(…)

task1.set_downstream(task2)
task3.set_upstream(task2)

# equivalent, but shorter:
# task1 >> task2
# task3 << task2

# Even clearer:
# task1 >> task2 >> task3
```



# Let's practice!

BUILDING DATA ENGINEERING PIPELINES IN PYTHON

# Building a data pipeline with Airflow

BUILDING DATA ENGINEERING PIPELINES IN PYTHON

**Oliver Willekens**
Data Engineer at Data Minded

# Airflow's BashOperator

- Executes bash commands
- Airflow adds logging, retry options and metrics over running this yourself.

```
from airflow.operators.bash_operator import BashOperator

bash_task = BashOperator(
        task_id='greet_world',
        dag=dag,
        bash_command='echo "Hello, world!"'
)
```

# Airflow's PythonOperator

- Executes Python callables

```
from airflow.operators.python_operator import PythonOperator
from my_library import my_magic_function

python_task = PythonOperator(
    dag=dag,
    task_id='perform_magic',
    python_callable=my_magic_function,
    op_kwargs={"snowflake": "*", "amount": 42}
)
```

# Running PySpark from Airflow

- BashOperator:

```
spark_master = (
  "spark://"
  "spark_standalone_cluster_ip"
  ":7077")

command = (
  "spark-submit "
  "--master {master} "
  "--py-files package1.zip "
  "/path/to/app.py"
).format(master=spark_master)

BashOperator(bash_command=command, …)
```

- SSHOperator

```
from airflow.contrib.operators\
  .ssh_operator import SSHOperator

task = SSHOperator(
    task_id='ssh_spark_submit',
    dag=dag,
    command=command,
    ssh_conn_id='spark_master_ssh'
)
```

# Running PySpark from Airflow

- SparkSubmitOperator

```
from airflow.contrib.operators\
  .spark_submit_operator \
  import SparkSubmitOperator

spark_task = SparkSubmitOperator(
    task_id='spark_submit_id',
    dag=dag,
    application="/path/to/app.py",
    py_files="package1.zip",
    conn_id='spark_default'
)
```

- SSHOperator

```
from airflow.contrib.operators\
  .ssh_operator import SSHOperator

task = SSHOperator(
    task_id='ssh_spark_submit',
    dag=dag,
    command=command,
    ssh_conn_id='spark_master_ssh'
)
```

# Let's practice!

BUILDING DATA ENGINEERING PIPELINES IN PYTHON

# Deploying Airflow

## BUILDING DATA ENGINEERING PIPELINES IN PYTHON

**Oliver Willekens**
Data Engineer at Data Minded

# Installing and configuring Airflow

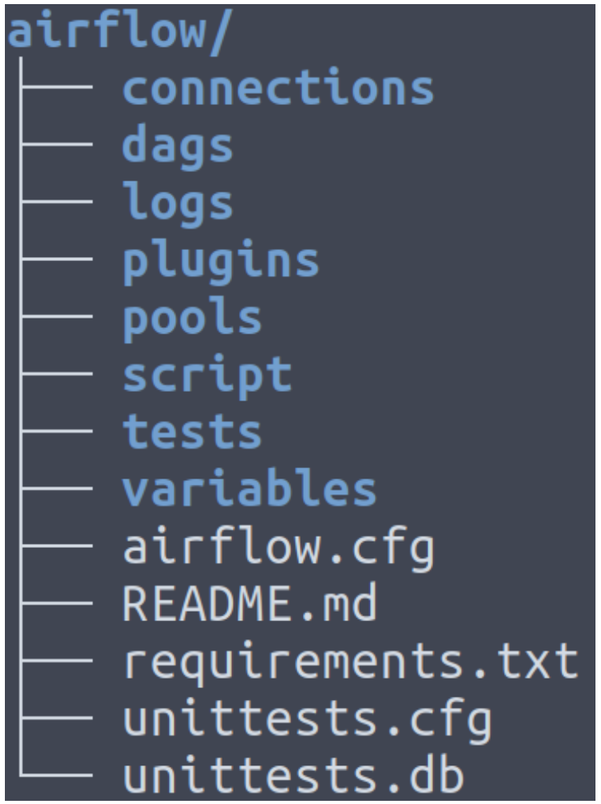
```
export AIRFLOW_HOME=~/airflow
pip install apache-airflow
airflow initdb
```

```
airflow/
├── logs
├── airflow.cfg
├── airflow.db
└── unittests.cfg
```

```
[core]
# lots of other configuration settings
# …

# The executor class that airflow should u
# Choices include SequentialExecutor,
# LocalExecutor, CeleryExecutor, DaskExecu
# KubernetesExecutor
executor = SequentialExecutor
```

# Setting up for production

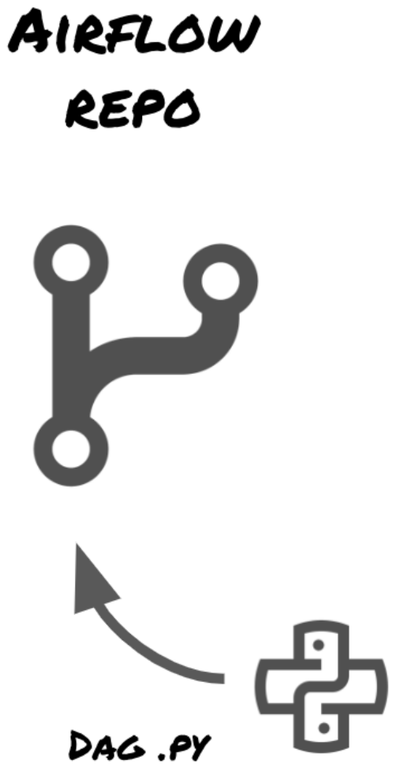
- *dags*: place to store the dags (configurable)
- *tests*: unit test the possible deployment, possibly ensure consistency across DAGs
- *plugins*: store custom operators and hooks
- *connections*, *pools*, *variables*: provide a location for various configuration files you can import into Airflow.



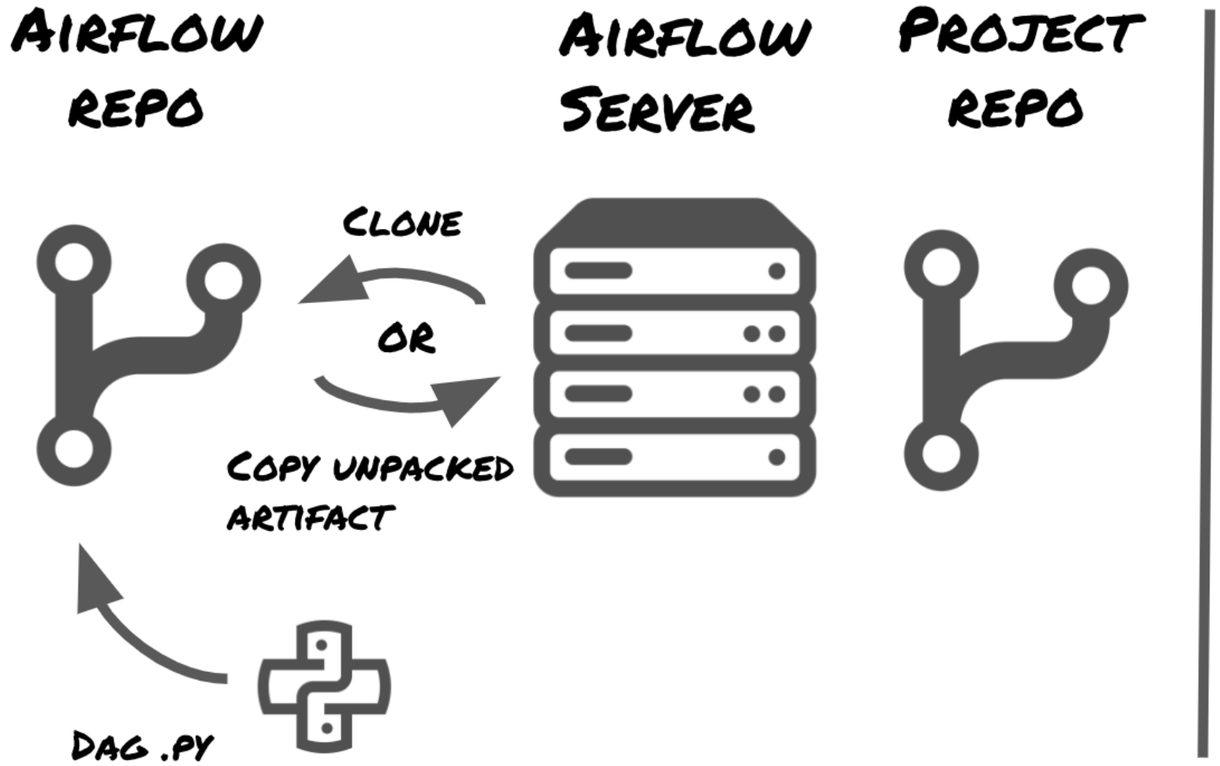
# Example Airflow deployment test

```
from airflow.models import DagBag

def test_dagbag_import():
    """Verify that Airflow will be able to import all DAGs in the repository."""
    dagbag = DagBag()

    number_of_failures = len(dagbag.import_errors)
    assert number_of_failures == 0, \
        "There should be no DAG failures. Got: %s" % dagbag.import_errors
```

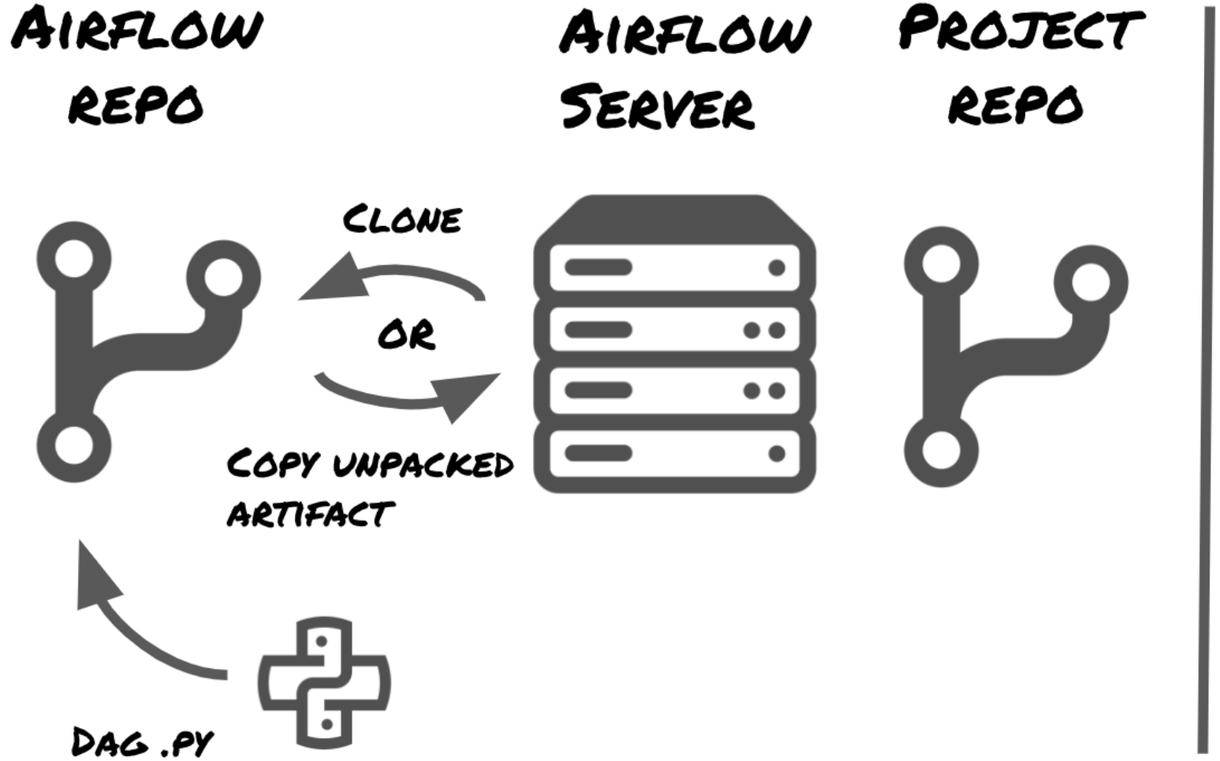
# Transferring DAGs and plugins

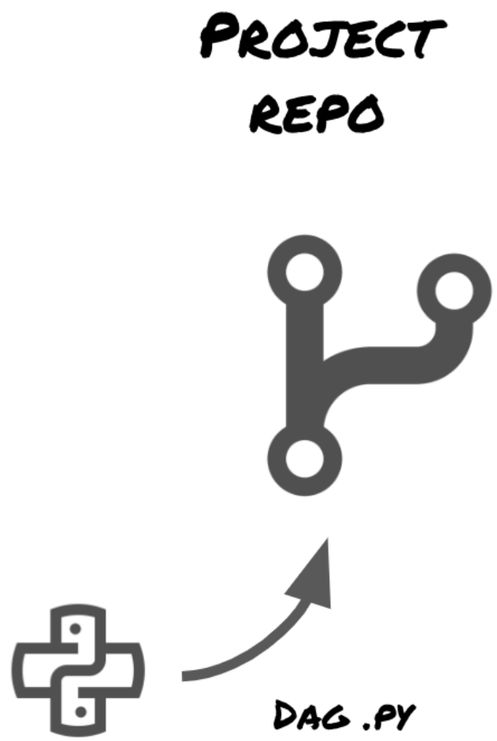


# Transferring DAGs and plugins

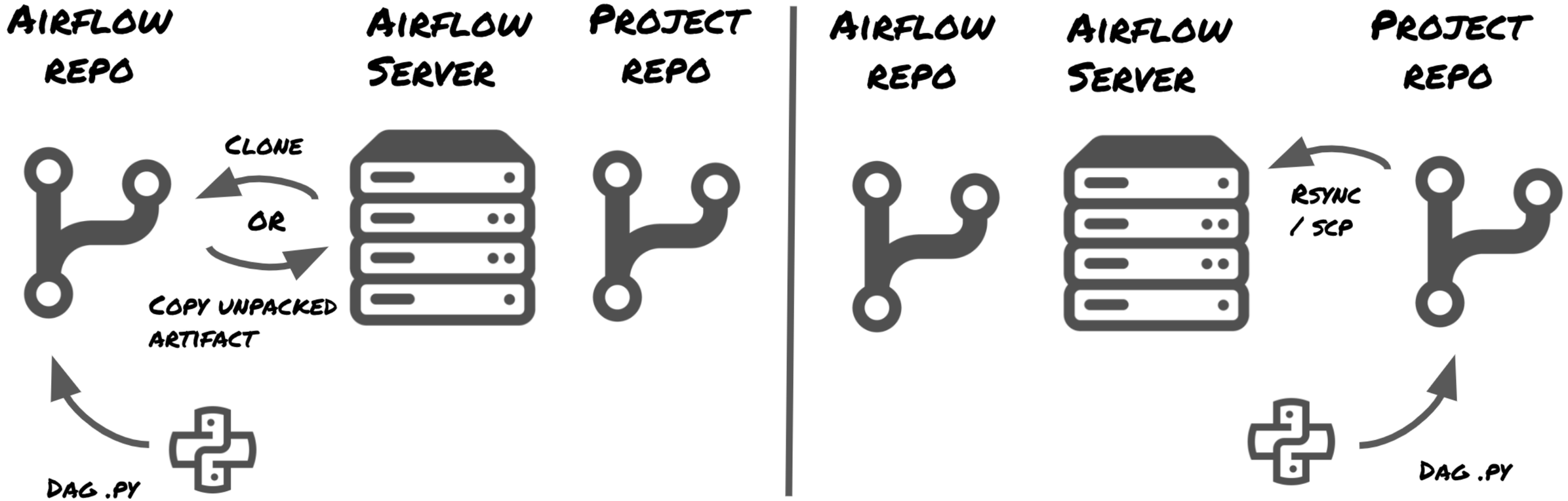


# Transferring DAGs and plugins



# Transferring DAGs and plugins



# Let's practice!

BUILDING DATA ENGINEERING PIPELINES IN PYTHON

# Final thoughts

BUILDING DATA ENGINEERING PIPELINES IN PYTHON

**Oliver Willekens**
Data Engineer at Data Minded

# What you learned

- Define purpose of components of data platforms
- Write an ingestion pipeline using Singer
- Create and deploy pipelines for big data in Spark
- Configure automated testing using CircleCI
- Manage and deploy a full data pipeline with Airflow

# Additional resources

## External resources

- Singer: **https://www.singer.io/**
- Apache Spark: **https://spark.apache.org/**
- Pytest: **https://pytest.org/en/latest/**
- Flake8: **http://flake8.pycqa.org/en/latest/**
- Circle CI: - **https://circleci.com/**
- Apache Airflow: **https://airflow.apache.org/**

## DataCamp courses

- Software engineering: **https://www.datacamp.com/courses/software-engineering-for-data-scientists-in-python**
- Spark: **https://www.datacamp.com/courses/cleaning-data-with-apache-spark-in-python** (and other courses)
- Unit testing: link yet to be revealed

# Congratulations! ????

BUILDING DATA ENGINEERING PIPELINES IN PYTHON