Components of a data platform

BUILDING DATA ENGINEERING PIPELINES IN PYTHON

Oliver Willekens
Data Engineer at Data Minded

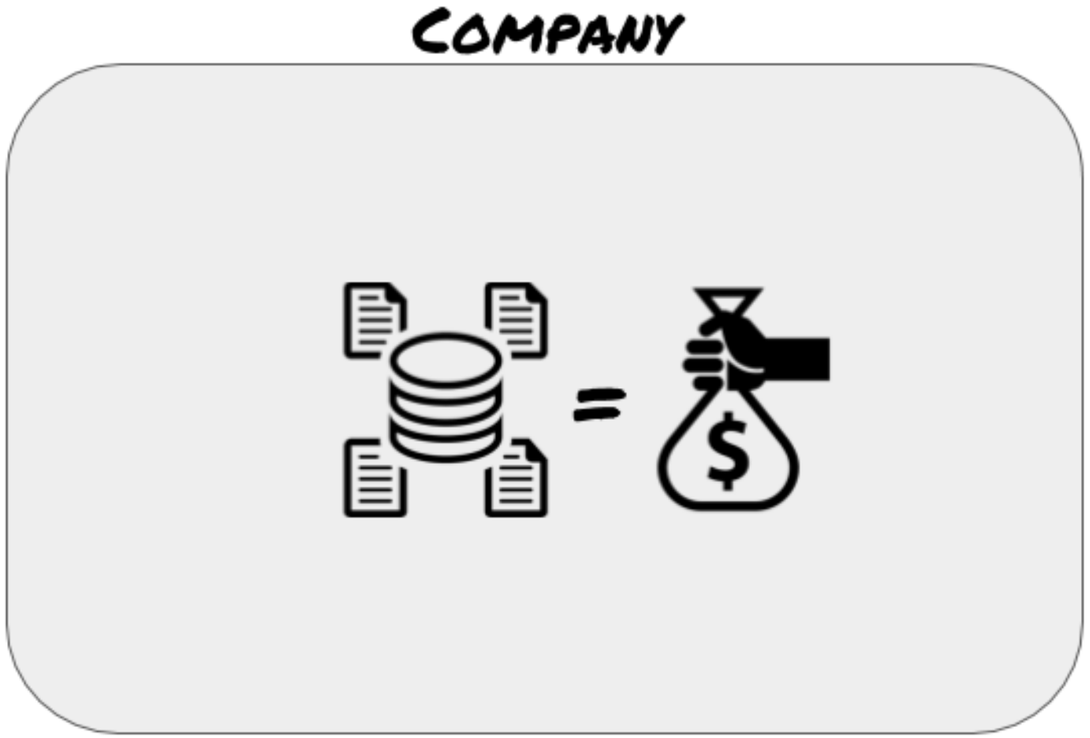
datacamp

Course contents

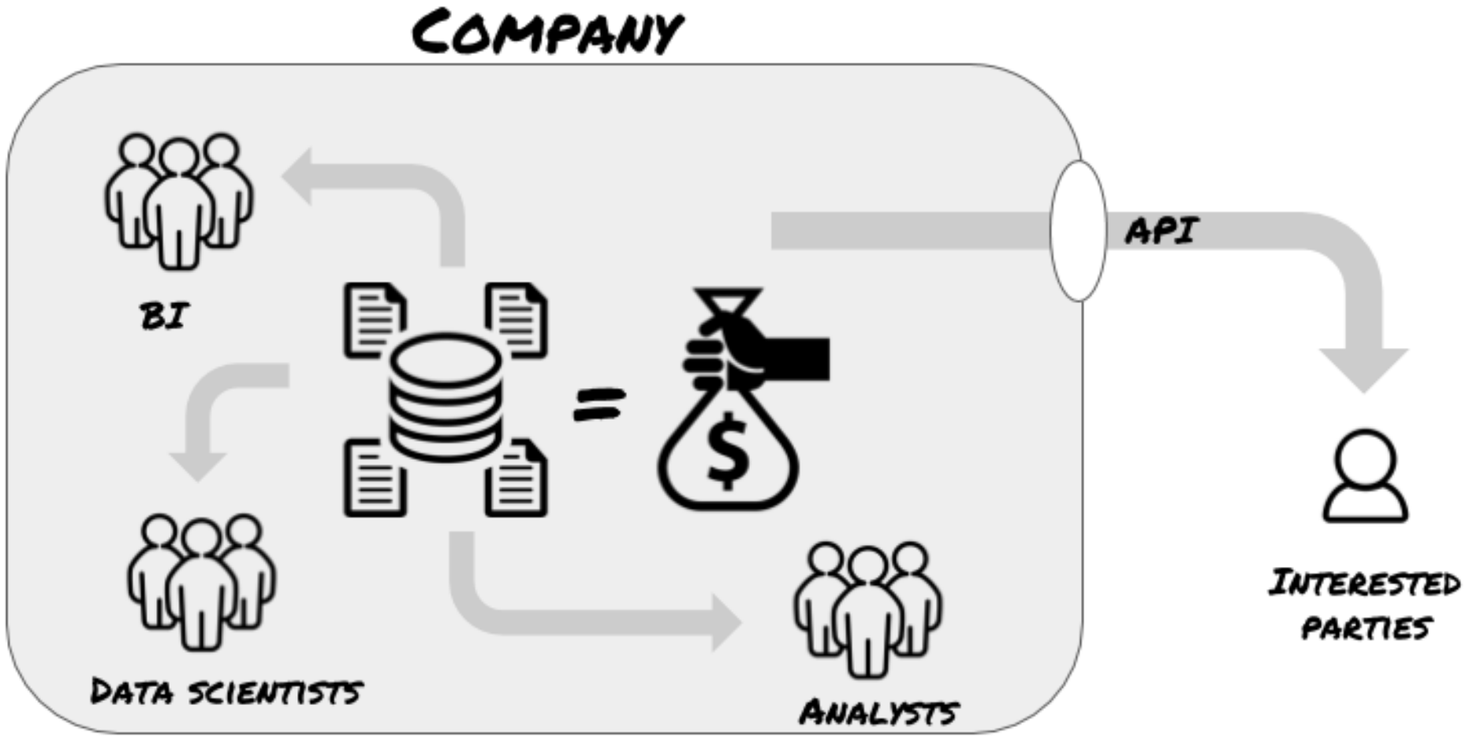
- ingest data using Singer
- apply common data cleaning operations
- gain insights by combining data with PySpark
- test your code automatically
- deploy Spark transformation pipelines

=> intro to data engineering pipelines

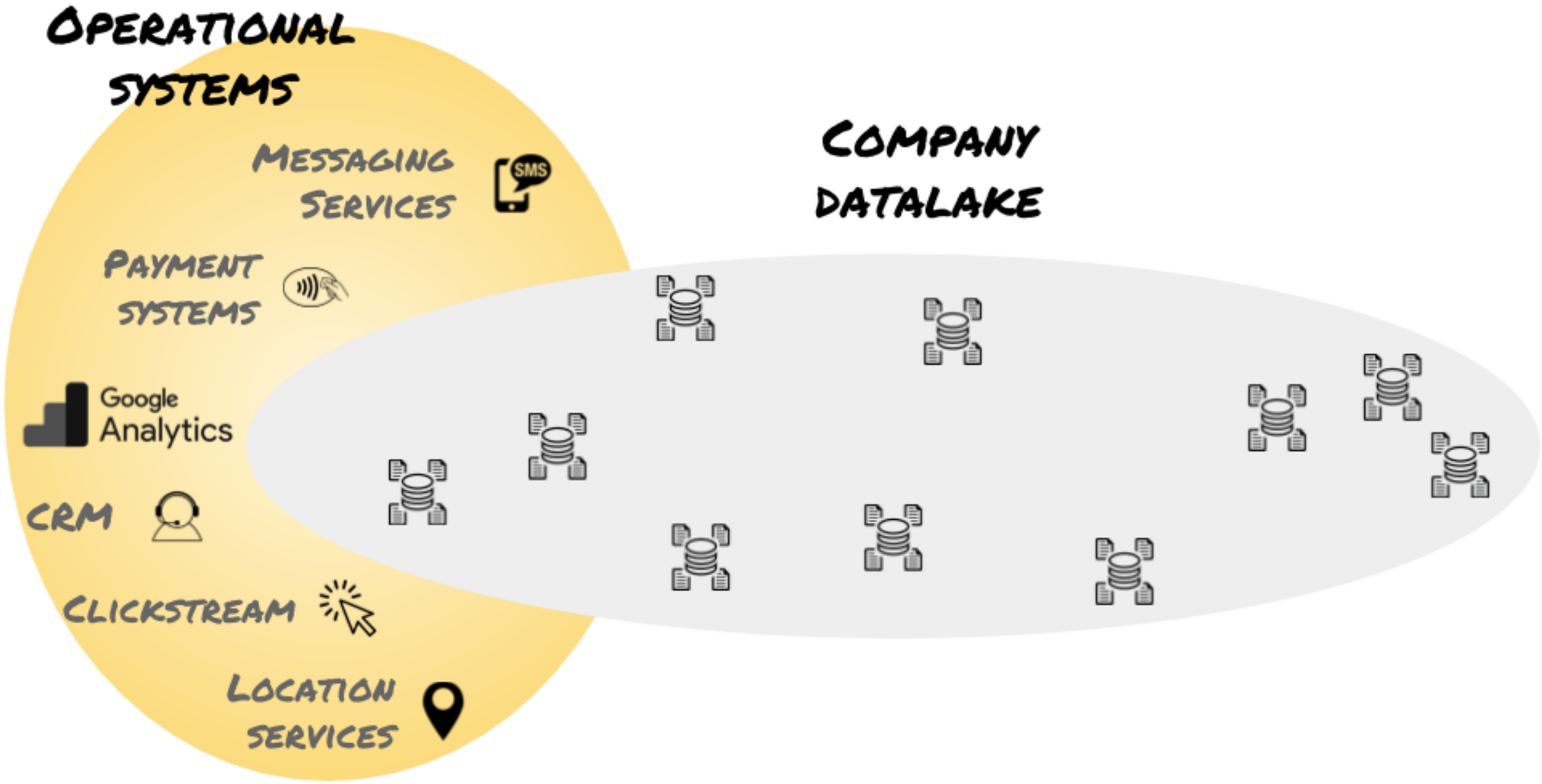
Data is valuable



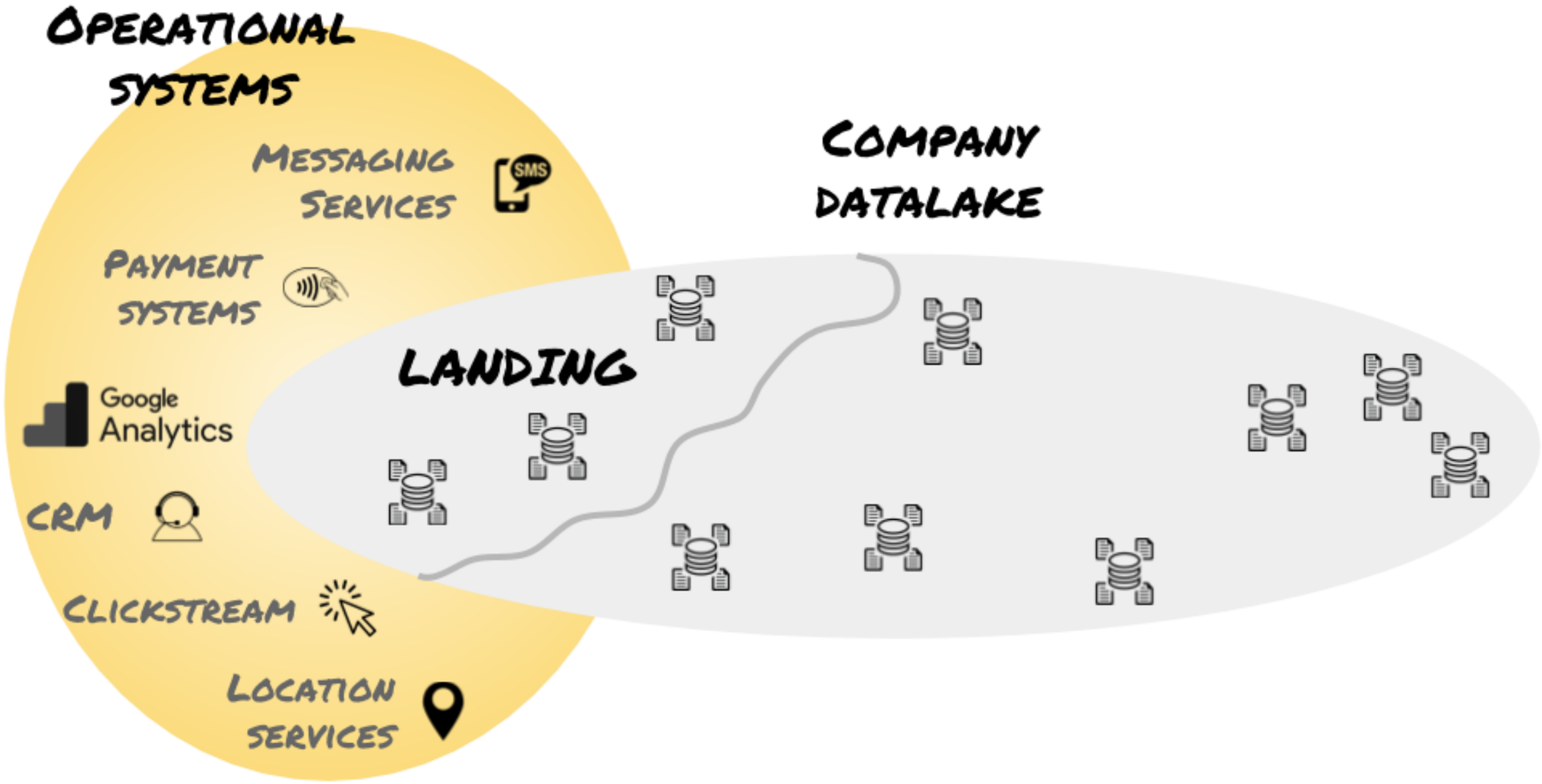
Democratizing data increases insights



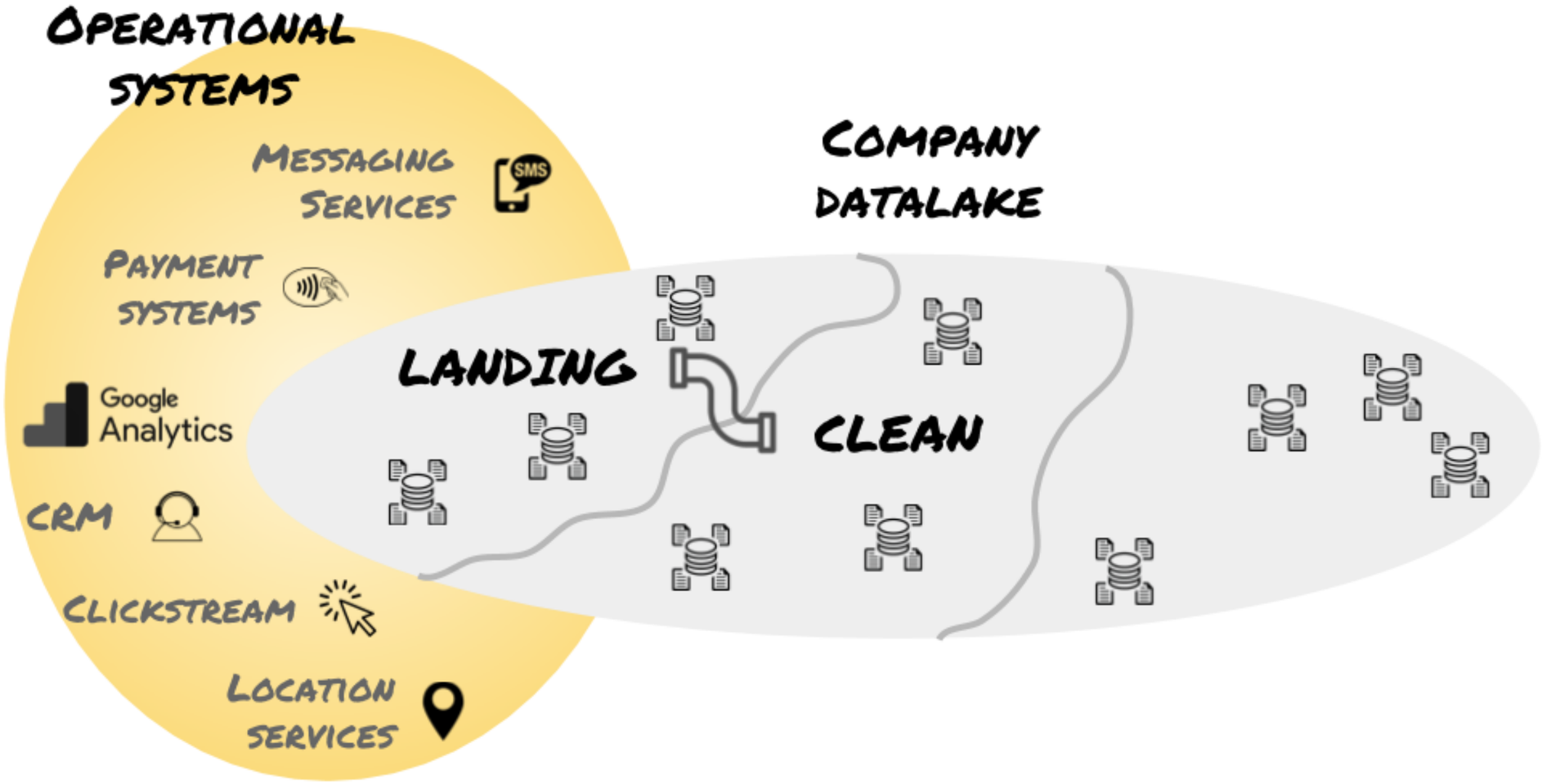
Genesis of the data



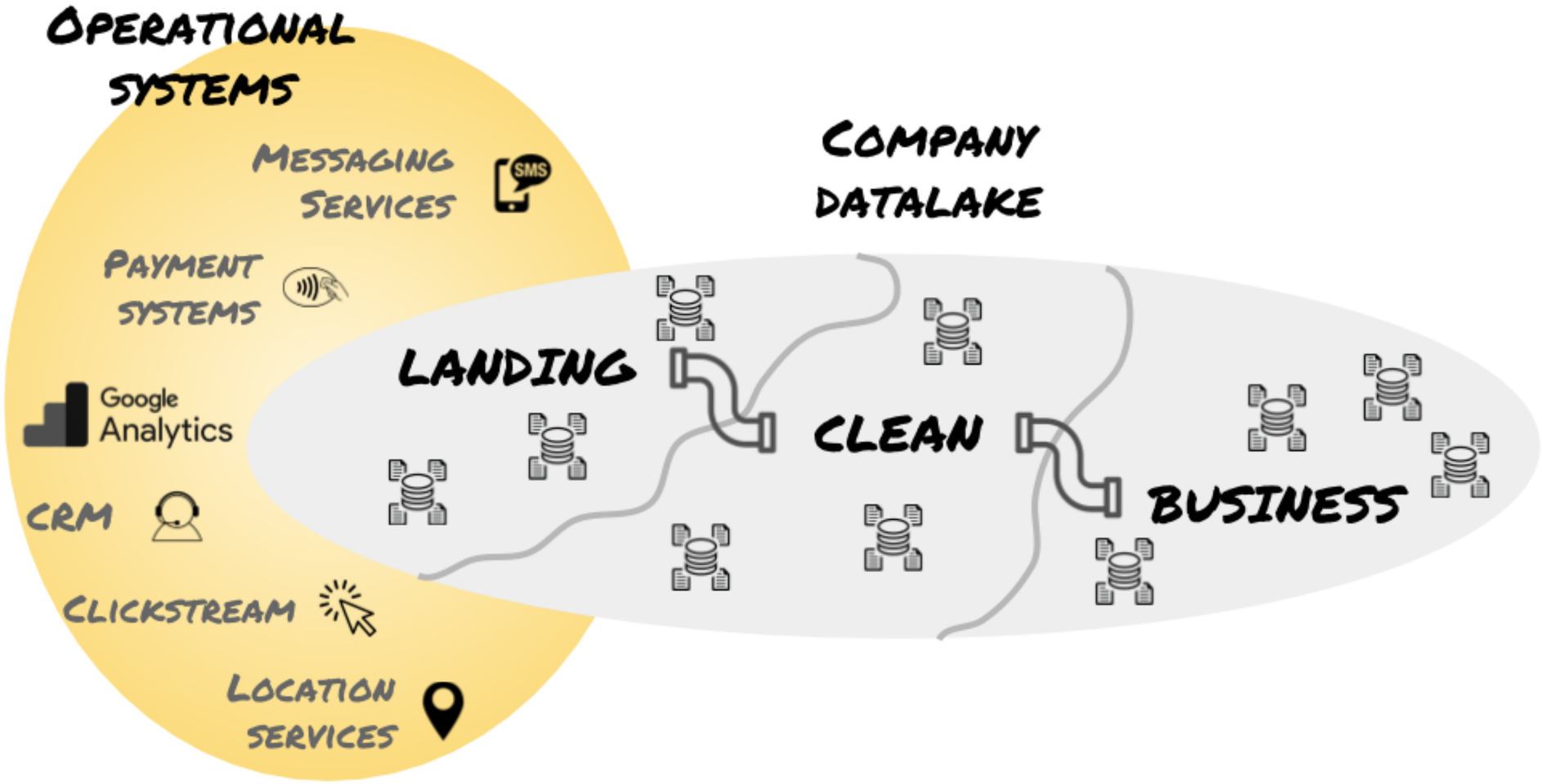
Operational data is stored in the landing zone



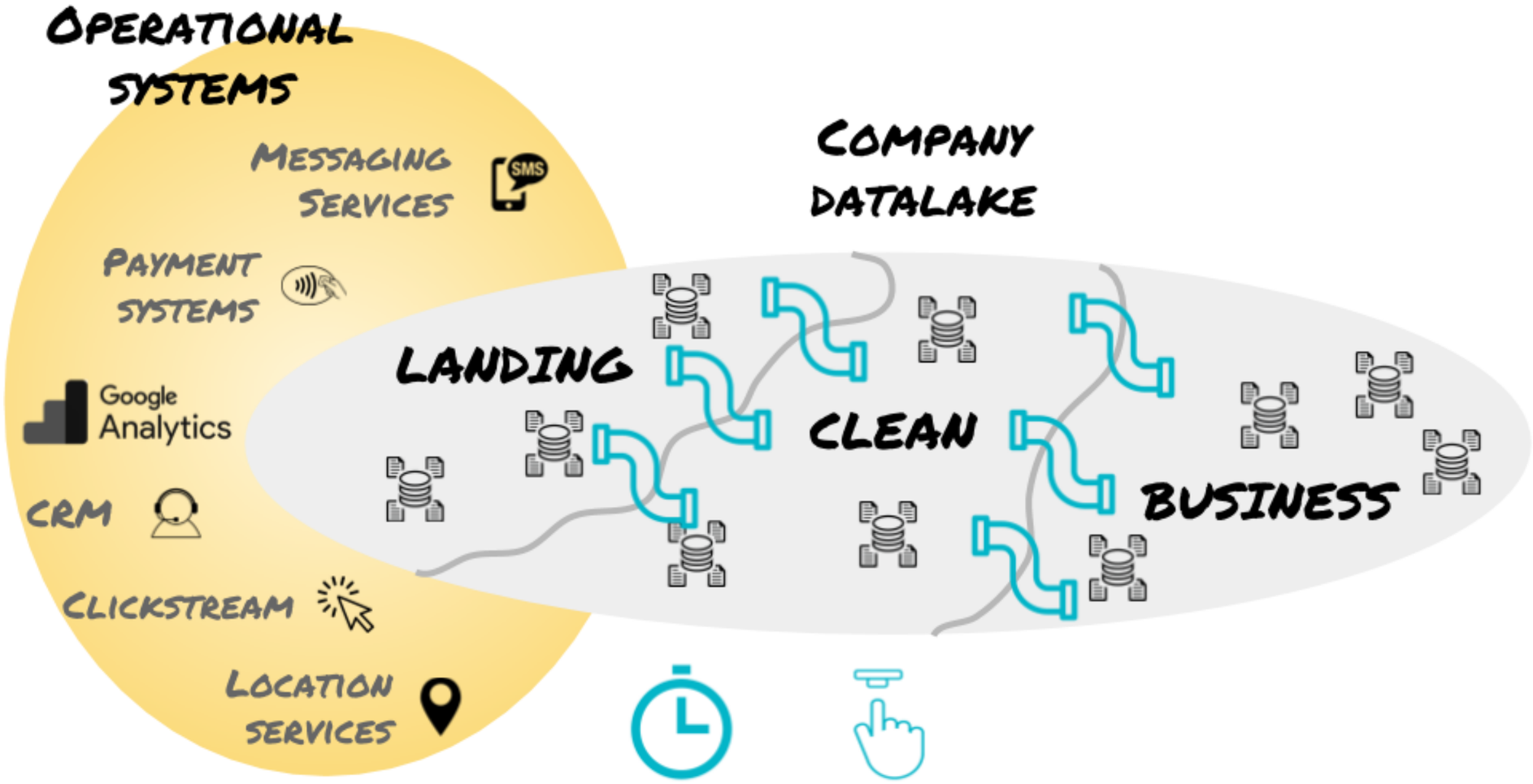
Cleaned data prevents rework



The business layer provides most insights



Pipelines move data from one zone to another



Let's reason!

BUILDING DATA ENGINEERING PIPELINES IN PYTHON

Introduction to data ingestion with Singer

BUILDING DATA ENGINEERING PIPELINES IN PYTHON

Oliver Willekens
Data Engineer at Data Minded

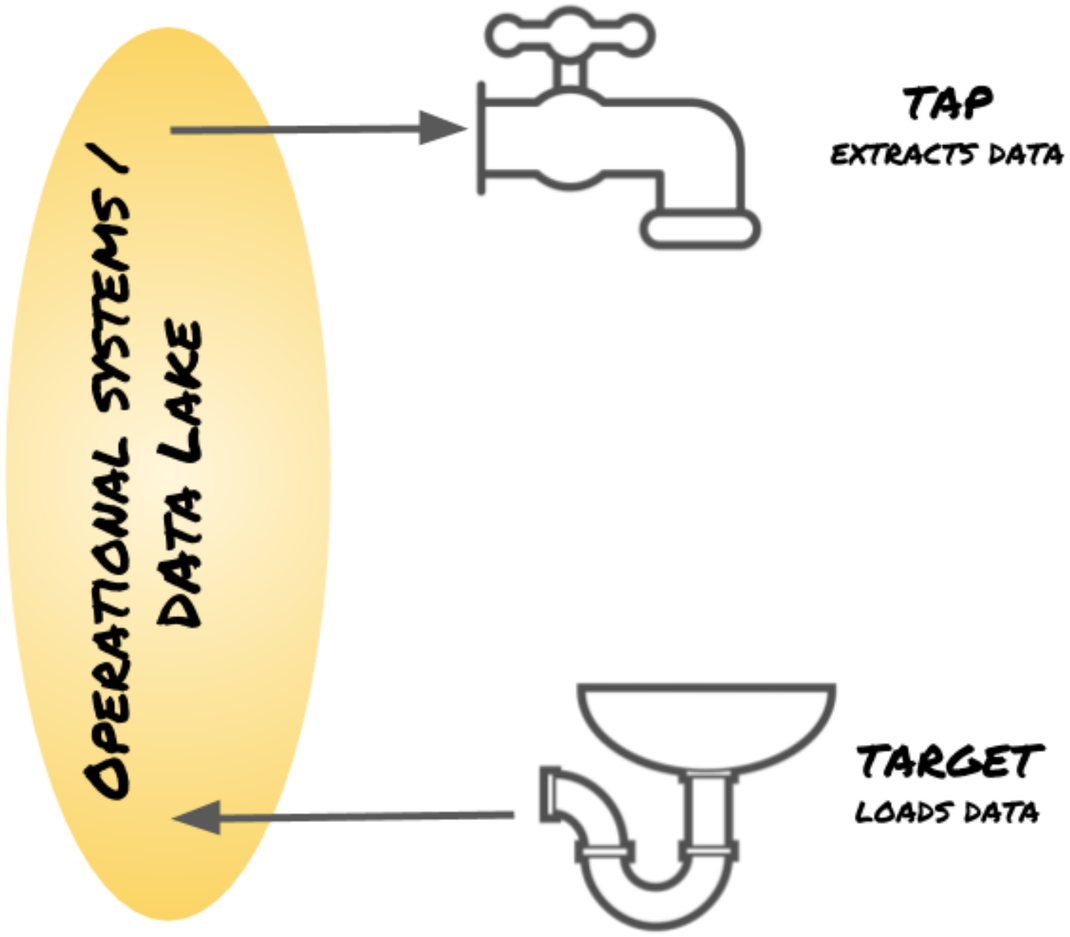
datacamp

Singer's core concepts

Aim: "The open-source standard for writing scripts that move data"

Singer is a *specification*

- data exchange format: *JSON*
- extract and load with *taps* and *targets*
  - => language independent

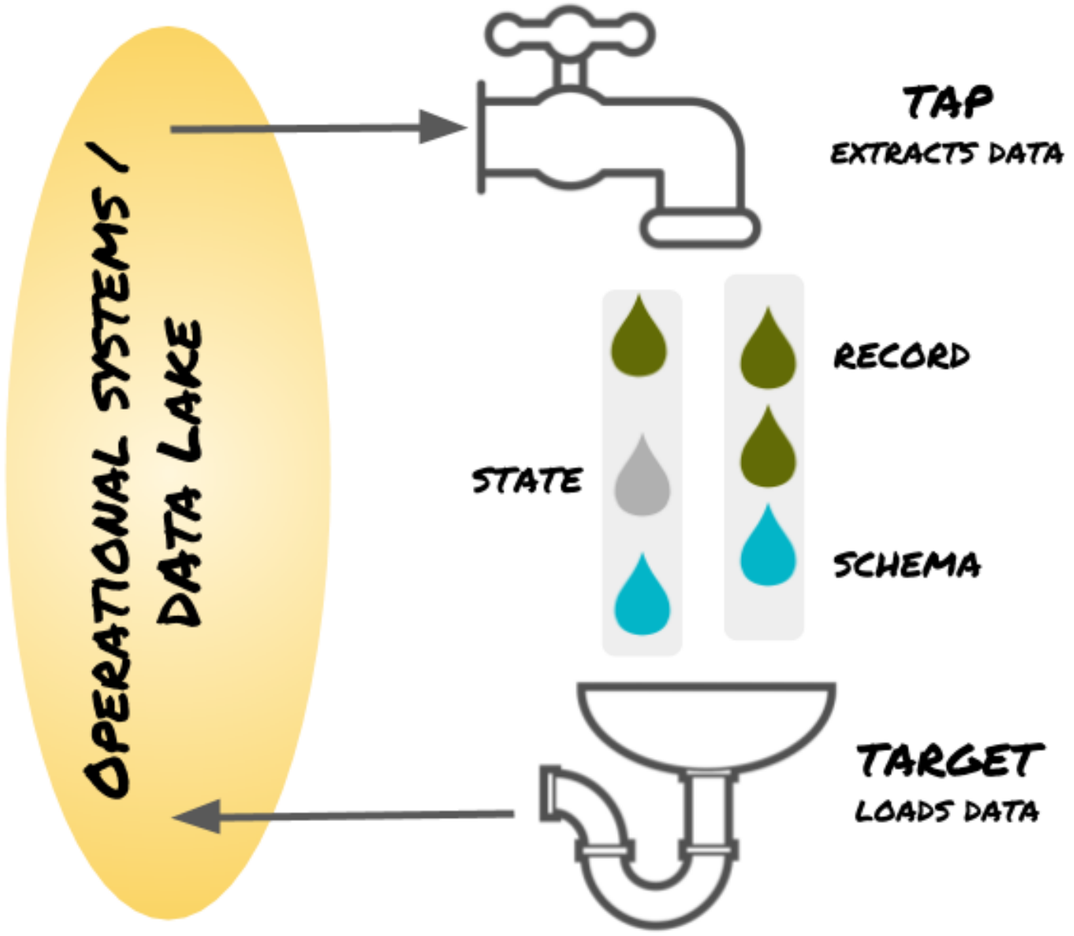


Singer's core concepts

Aim: "The open-source standard for writing scripts that move data"

Singer is a *specification*

- data exchange format: *JSON*
- extract and load with *taps* and *targets*
  - => language independent
- communicate over *streams*:
  - schema (metadata)
  - state (process metadata)
  - record (data)

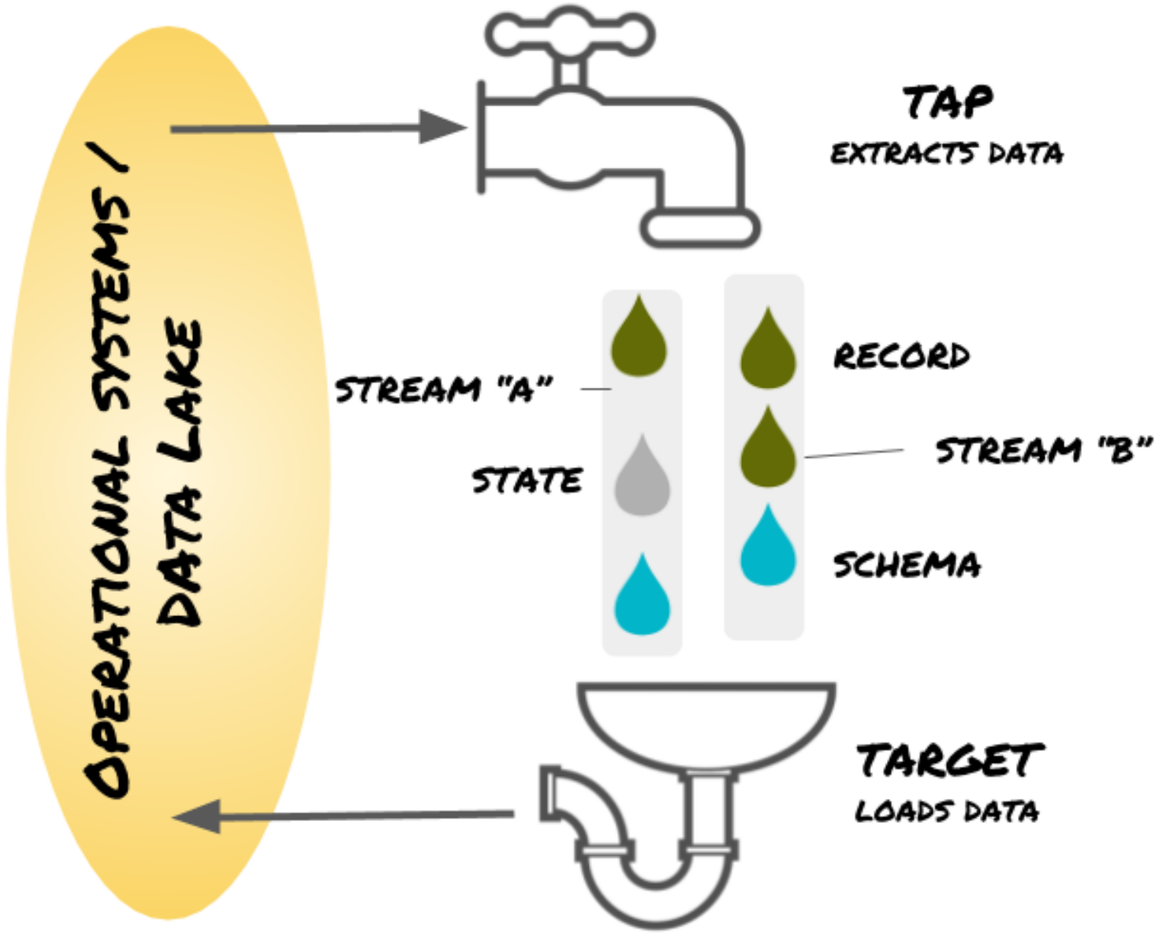


Singer's core concepts

Aim: "The open-source standard for writing scripts that move data"

Singer is a *specification*

- data exchange format: *JSON*
- extract and load with *taps* and *targets*
  - => language independent
- communicate over *streams*:
  - schema (metadata)
  - state (process metadata)
  - record (data)



Describing the data through its schema

```
columns = ("id", "name", "age", "has_children")
users = {(1, "Adrian", 32, False),
         (2, "Ruanne", 28, False),
         (3, "Hillary", 29, True)}
json_schema = {
    "properties": {"age": {"maximum": 130,
                           "minimum": 1,
                           "type": "integer"},
                   "has_children": {"type": "boolean"},
                   "id": {"type": "integer"},
                   "name": {"type": "string"}},
    "$id": "http://yourdomain.com/schemas/my_user_schema.json",
    "$schema": "http://json-schema.org/draft-07/schema#"}
```

Describing the data through its schema

```
import singer
singer.write_schema(schema=json_schema,
                    stream_name='DC_employees',
                    key_properties=["id"])
```

```
{"type": "SCHEMA", "stream": "DC_employees", "schema": {"properties":
{"age": {"maximum": 130, "minimum": 1, "type": "integer"}, "has_children":
{"type": "boolean"}, "id": {"type": "integer"}, "name": {"type": "string"}},
"$id": "http://yourdomain.com/schemas/my_user_schema.json",
"$schema": "http://json-schema.org/draft-07/schema#"}, "key_properties": ["id"]}
```

Serializing JSON

```
import json


json.dumps(json_schema["properties"]["age"])
```

```
'{"maximum": 130, "minimum": 1, "type": "integer"}'
```

```
with open("foo.json", mode="w") as fh:
    json.dump(obj=json_schema, fp=fh)  # writes the json-serialized object
                                       # to the open file handle
```

Let's practice!

BUILDING DATA ENGINEERING PIPELINES IN PYTHON

Running an ingestion pipeline with Singer

BUILDING DATA ENGINEERING PIPELINES IN PYTHON

Oliver Willekens
Data Engineer at Data Minded

datacamp

Streaming record messages

```
columns = ("id", "name", "age", "has_children")
users = {(1, "Adrian", 32, False),
         (2, "Ruanne", 28, False),
         (3, "Hillary", 29, True)}
```

```
singer.write_record(stream_name="DC_employees",
                    record=dict(zip(columns, users.pop())))
```

```
{"type": "RECORD", "stream": "DC_employees", "record": {"id": 1, "name": "Adrian", "age": 32, "has_children": false}}
```

```
fixed_dict = {"type": "RECORD", "stream": "DC_employees"}
record_msg = {**fixed_dict, "record": dict(zip(columns, users.pop()))}
print(json.dumps(record_msg))
```

Chaining taps and targets

```
# Module: my_tap.py
import singer


singer.write_schema(stream_name="foo", schema=…)
singer.write_records(stream_name="foo", records=…)
```

Ingestion pipeline: **Pipe** the tap's output into a Singer target, using the `|` symbol (Linux & MacOS)

```
python my_tap.py | target-csv
python my_tap.py | target-csv --config userconfig.cfg
my-packaged-tap | target-csv --config userconfig.cfg
```

Modular ingestion pipelines

```
my-packaged-tap | target-csv
my-packaged-tap | target-google-sheets
my-packaged-tap | target-postgresql --config conf.json
```

```
tap-custom-google-scraper | target-postgresql --config headlines.json
```

Keeping track with state messages

Keeping track with state messages

id	name	last_updated_on
1	Adrian	2019-06-14T14:00:04.000+02:00
2	Ruanne	2019-06-16T18:33:21.000+02:00
3	Hillary	2019-06-14T10:05:12.000+02:00

```
singer.write_state(value={"max-last-updated-on": some_variable})
```

Run this `tap-mydelta` on 2019-06-14 at 12:00:00.000+02:00 (2nd row wasn't yet present then):

```
{"type": "STATE", "value": {"max-last-updated-on": "2019-06-14T10:05:12.000+02:00"}}
```

Let's practice!

BUILDING DATA ENGINEERING PIPELINES IN PYTHON

Basic introduction to PySpark

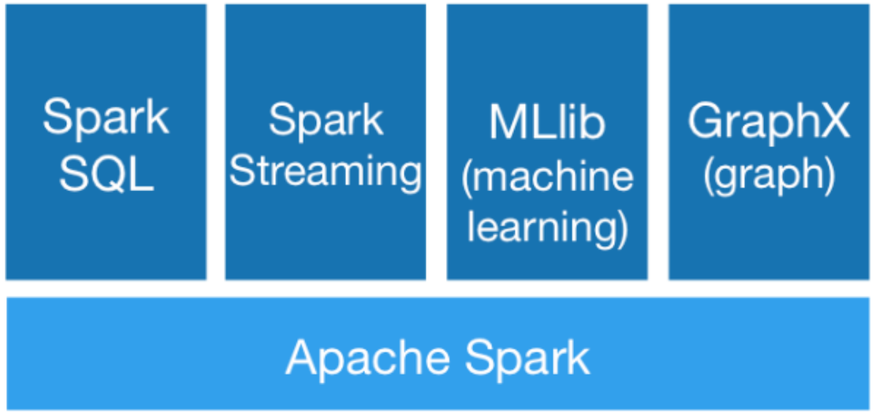
BUILDING DATA ENGINEERING PIPELINES IN PYTHON

Oliver Willekens
Data Engineer at Data Minded

datacamp

What is Spark?

- A fast and general engine for large-scale data processing
- 4 libraries built on top of Spark core:



- API in several languages
  - Java, Scala, Python ("PySpark"), R

When to use Spark

Spark is used for:

- Data processing at scale
- Interactive analytics
- Machine learning

Spark is **not** used for:

- When you have only little data
- When you have only simple operations

Business case: finding the perfect diaper

Find the perfect diaper based on:

- qua**l**itative attributes e.g. comfort
- qua**nt**itative attributes e.g. price

Scraped data available:

- *prices.csv*: pricing details per model per store
- *ratings.csv*: user ratings per model

Starting the Spark analytics engine

```
from pyspark.sql import SparkSession


spark = SparkSession.builder.getOrCreate()
```

Reading a CSV file

```
prices = spark.read.csv("mnt/data_lake/landing/prices.csv")
prices.show()
```

```
+---------+-----------+------------+-----+--------+--------+----------+
|      _c0|        _c1|          _c2|  _c4|     _c5|     _c6|       _c7|
+---------+-----------+------------+-----+--------+--------+----------+
|    store|countrycode|       brand|price|currency|quantity|      date|
|     Aldi|         BE|Diapers-R-Us|  6.8|     EUR|      40|2019-02-03|
|Carrefour|         FR|     Nappy-k|  5.7|     EUR|      30|2019-02-06|
|    Tesco|        IRL|     Pampers|  6.3|     EUR|      35|2019-02-07|
|       DM|         DE|     Huggies|  6.8|     EUR|      40|2019-02-01|
+---------+-----------+------------+-----+--------+--------+----------+
```

Reading a CSV file with headers

```
prices = spark.read.options(header="true").csv("mnt/data_lake/landing/prices.csv")
prices.show()
```

```
+---------+-----------+------------+-----+--------+--------+----------+
|    store|countrycode|       brand|price|currency|quantity|      date|
+---------+-----------+------------+-----+--------+--------+----------+
|     Aldi|         BE|Diapers-R-Us|  6.8|     EUR|      40|2019-02-03|
|Carrefour|         FR|     Nappy-k|  5.7|     EUR|      30|2019-02-06|
|    Tesco|        IRL|     Pampers|  6.3|     EUR|      35|2019-02-07|
|       DM|         DE|     Huggies|  6.8|     EUR|      40|2019-02-01|
+---------+-----------+------------+-----+--------+--------+----------+
```

Automatically inferred data types

```
from pprint import pprint
pprint(prices.dtypes)
```

```
[('store', 'string'),
('countrycode', 'string'),
('brand', 'string'),
('price', 'string'),
('currency', 'string'),
('quantity', 'string'),
('date', 'string')]
```

Enforcing a schema

```
schema = StructType([StructField("store", StringType(), nullable=False),
                     StructField("countrycode", StringType(), nullable=False),
                     StructField("brand", StringType(), nullable=False),
                     StructField("price", FloatType(), nullable=False),
                     StructField("currency", StringType(), nullable=True),
                     StructField("quantity", IntegerType(), nullable=True),
                     StructField("date", DateType(), nullable=False)])

prices = spark.read.options(header="true").schema(schema).csv("mnt/data_lake/landing/prices.csv")
print(prices.dtypes)
```

```
[('store', 'string'), ('countrycode', 'string'), ('brand', 'string'),
('price', 'float'), ('currency', 'string'), ('quantity', 'int'), ('date', 'date')]
```

Let's practice!

BUILDING DATA ENGINEERING PIPELINES IN PYTHON

Cleaning data

BUILDING DATA ENGINEERING PIPELINES IN PYTHON

Oliver Willekens
Data Engineer at Data Minded

datacamp

Reasons to clean data

Most data sources are not ready for analytics. This could be due to:

- Incorrect data types
- Invalid rows
- Incomplete rows
- Badly chosen placeholders

Can we automate data cleaning?

Data cleaning depends on the context

- Can our system cope with data that is 95% clean and 95% complete?
- What are the implicit standards in the company?
  - regional datetimes vs. UTC
  - column naming conventions
  - ...
- What are the low-level details of the systems?
  - representation of unknown / incomplete data
  - ranges for numerical values
  - meaning of fields

Selecting data types

Data type	Value type in Python
`ByteType`	Good for numbers that are within the range of -128 to 127.
`ShortType`	Good for numbers that are within the range of -32768 to 32767.
`IntegerType`	Good for numbers that are within the range of-2147483648 to 2147483647.
`FloatType`	float
`StringType`	string
`BooleanType`	bool
`DateType`	datetime.date

Badly formatted source data

```
cat bad_data.csv  # prints the entire file on stdout
```

```
store,countrycode,brand,price,currency,quantity,date
Aldi,BE,Diapers-R-Us,6.8,EUR,40,2019-02-03
  ----------------------------------
Kruidvat,NL,Nappy-k,5.6,EUR,40,2019-02-15
DM,AT,Huggies,7.2,EUR,40,2019-02-01
```

Spark's default handling of bad source data

```
prices = spark.read.options(header="true").csv('landing/prices.csv')
prices.show()
```

```
+------------------+----------+-----------+-----+--------+--------+----------+
|             store|countrycode|      brand|price|currency|quantity|      date|
+------------------+----------+-----------+-----+--------+--------+----------+
|              Aldi|        BE|Diapers-R-Us|  6.8|     EUR|      40|2019-02-03|
|----------------...|      null|       null| null|    null|    null|      null|
|          Kruidvat|        NL|    Nappy-k|  5.6|     EUR|      40|2019-02-15|
|                DM|        AT|    Huggies|  7.2|     EUR|      40|2019-02-01|
+------------------+----------+-----------+-----+--------+--------+----------+
```

Handle invalid rows

```
prices = (spark
          .read
          .options(header="true", mode="DROPMALFORMED")
          .csv('landing/prices.csv'))
```

```
+--------+-----------+------------+-----+--------+--------+----------+
|   store|countrycode|       brand|price|currency|quantity|      date|
+--------+-----------+------------+-----+--------+--------+----------+
|    Aldi|         BE|Diapers-R-Us|  6.8|     EUR|      40|2019-02-03|
|Kruidvat|         NL|     Nappy-k|  5.6|     EUR|      40|2019-02-15|
|      DM|         AT|     Huggies|  7.2|     EUR|      40|2019-02-01|
+--------+-----------+------------+-----+--------+--------+----------+
```

The significance of null

```
store,countrycode,brand,price,currency,quantity,date
Aldi,BE,Diapers-R-Us,6.8,EUR,40,2019-02-03
Kruidvat,,Nappy-k,5.6,EUR,,2019-02-15
```

```
prices = (spark.read.options(header="true")
          .schema(schema)
          .csv('/landing/prices_with_incomplete_rows.csv'))
prices.show()
```

```
+--------+-----------+------------+-----+--------+--------+----------+
|   store|countrycode|       brand|price|currency|quantity|      date|
+--------+-----------+------------+-----+--------+--------+----------+
|    Aldi|         BE|Diapers-R-Us|  6.8|     EUR|      40|2019-02-03|
|Kruidvat|       null|     Nappy-k|  5.6|     EUR|    null|2019-02-15|
+--------+-----------+------------+-----+--------+--------+----------+
```

Supplying default values for missing data

```
prices.fillna(25, subset=['quantity']).show()
```

```
+--------+-----------+------------+-----+--------+--------+----------+
|   store|countrycode|       brand|price|currency|quantity|      date|
+--------+-----------+------------+-----+--------+--------+----------+
|    Aldi|         BE|Diapers-R-Us|  6.8|     EUR|      40|2019-02-03|
|Kruidvat|       null|     Nappy-k|  5.6|     EUR|      25|2019-02-15|
+--------+-----------+------------+-----+--------+--------+----------+
```

Badly chosen placeholders

Example: contracts of employees

```
employees = spark.read.options(header="true").schema(schema).csv('employees.csv')
```

```
+-------------+----------+----------+----------+
|employee_name|department|start_date|  end_date|
+-------------+----------+----------+----------+
|          Bob| marketing|2012-06-01|2016-05-02|
|        Alice|        IT|2018-04-03|9999-12-31|
+-------------+----------+----------+----------+
```

Conditionally replace values

```
from pyspark.sql.functions import col, when
from datetime import date, timedelta
one_year_from_now = date.today().replace(year=date.today().year + 1)
better_frame = employees.withColumn("end_date",
    when(col("end_date") > one_year_from_now, None).otherwise(col("end_date")))
better_frame.show()
```

```
+-------------+----------+----------+----------+
|employee_name|department|start_date|  end_date|
+-------------+----------+----------+----------+
|          Bob| marketing|2012-06-01|2016-05-02|
|        Alice|        IT|2018-04-03|      null|
+-------------+----------+----------+----------+
```

Let's practice!

BUILDING DATA ENGINEERING PIPELINES IN PYTHON

Transforming data with Spark

BUILDING DATA ENGINEERING PIPELINES IN PYTHON

Oliver Willekens
Data Engineer at Data Minded

datacamp

Why do we need to transform data?

Process:

1. Collect data
2. “Massage” data: involves cleaning and business logic
3. **Derive insights**

Example:

1. Collect data from *booking.com* and *hotels.com*.
2. Standardize hotel names, normalizing review scores.
3. Join datasets, filter on location and rank results.

Common data transformations

1. **Filtering rows**

```
country | purchase_order
________|_______________
India   | 87254800912
Ukraine | 32498562223
```

European purchases?

```
country | purchase_order
________|_______________
Ukraine | 32498562223
```

Common data transformations

1. **Filtering rows**
2. **Selecting and renaming columns**

```
country | purchase_order | store_keep
________|________________|___________
Ukraine | 32498562223    | Oksana D.
Spain   | 74398221190    | Pedro R.
```

->

```
country_of_purchase | purchase_order
____________________|________________
Ukraine             | 32498562223
Spain               | 74398221190
```

Common data transformations

1. **Filtering rows**
2. **Selecting and renaming columns**
3. **Grouping and aggregation**

```
country | purchase_order | price          country | total_revenue
________|________________|______         ________|______________
Ukraine | 32498562223    | $12     =>    Ukraine | $12
Spain   | 74398221190    | $54           Spain   | $80
Spain   | 49876776100    | $26
```

Common data transformations

1. **Filtering rows**
2. **Selecting and renaming columns**
3. **Grouping and aggregation**
4. **Joining multiple datasets**

```
country | purchase_order | price        purchase_order | category
________|________________|______      _________________|_________
Ukraine | 32498562223    | $12      +   32498562223    | food
Spain   | 74398221190    | $54          49876776100    | electronics
Spain   | 49876776100    | $26          74398221190    | clothing
```

Common data transformations

1. Filtering rows
2. Selecting and renaming columns
3. Grouping and aggregation
4. Joining multiple datasets
5. Ordering results

```
country | purchase_order | price          country | purchase_order | price
________|________________|______         ________|________________|______
Spain   | 74398221190    | $26     =>    Ukraine | 32498562223    | $12
Ukraine | 32498562223    | $12           Spain   | 74398221190    | $26
Spain   | 49876776100    | $54           Spain   | 49876776100    | $54
```

Recall the prices dataset

```
prices = spark.read.options(header="true").schema(schema).csv('landing/prices.csv')
```

```
+---------+-----------+------------+-----+--------+--------+----------+
|    store|countrycode|       brand|price|currency|quantity|      date|
+---------+-----------+------------+-----+--------+--------+----------+
|     Aldi|         BE|Diapers-R-Us|  6.8|     EUR|      40|2019-02-03|
| Kruidvat|         BE|     Nappy-k|  4.8|     EUR|      30|2019-01-28|
|Carrefour|         FR|     Nappy-k|  5.7|     EUR|      30|2019-02-06|
|    Tesco|        IRL|     Pampers|  6.3|     EUR|      35|2019-02-07|
|       DM|         DE|     Huggies|  6.8|     EUR|      40|2019-02-01|
+---------+-----------+------------+-----+--------+--------+----------+
```

Filtering and ordering rows

```
prices_in_belgium = prices.filter(col('countrycode') == 'BE').orderBy(col('date'))
```

```
+-------+-----------+------------+-----+--------+--------+----------+
|   store|countrycode|       brand|price|currency|quantity|      date|
+-------+-----------+------------+-----+--------+--------+----------+
|Kruidvat|         BE|     Nappy-k|  4.8|     EUR|      30|2019-01-28|
|    Aldi|         BE|Diapers-R-Us|  6.8|     EUR|      40|2019-02-03|
+-------+-----------+------------+-----+--------+--------+----------+
```

- Function `col` creates Column objects
- Method `orderBy` sorts values by a certain column.

Selecting and renaming columns

```
prices.select(



)
```

Selecting and renaming columns

```
prices.select(
    col("store"),
    col("brand")
)
```

Selecting and renaming columns

```
prices.select(
    col("store"),
    col("brand").alias("brandname")
)
```

```
+---------+-------------+
|    store|    brandname|
+---------+-------------+
|     Aldi|Diapers-R-Us|
| Kruidvat|      Nappy-k|
|Carrefour|      Nappy-k|
| Kruidvat|      Nappy-k|
|    Tesco|      Pampers|
|       DM|      Huggies|
|       DM|      Huggies|
+---------+-------------+
```

Reducing duplicate values

```
prices.select(
    col("store"),
    col("brand").alias("brandname")
).distinct()
```

```
+---------+-----------+
|    store|  brandname|
+---------+-----------+
|       DM|     Huggies|
| Kruidvat|     Nappy-k|
|Carrefour|     Nappy-k|
|     Aldi|Diapers-R-Us|
|    Tesco|     Pampers|
+---------+-----------+
```

Grouping and aggregating with mean()

```
(prices
    .groupBy(col('brand'))
    .mean('price')
).show()
```

```
+------------+------------------+
|       brand|        avg(price)|
+------------+------------------+
|Diapers-R-Us| 6.800000190734863|
|     Pampers| 6.300000190734863|
|     Huggies|               7.0|
|     Nappy-k|5.3666666348775225|
+------------+------------------+
```

Grouping and aggregating with agg()

```
(prices
    .groupBy(col('brand'))
    .agg(
        avg('price').alias('average_price'),
        count('brand').alias('number_of_items')
        )
).show()
```

```
+-----------+-----------------+---------------+
|      brand|    average_price|number_of_items|
+-----------+-----------------+---------------+
|Diapers-R-Us| 6.800000190734863|              1|
|    Pampers| 6.300000190734863|              1|
|    Huggies|              7.0|              2|
|    Nappy-k|5.3666666348775225|              3|
+-----------+-----------------+---------------+
```

Joining related data

```
+--------+-----------+-----------+-------+-----+--------+--------+----------+
|   store|countrycode|      brand|  model|price|currency|quantity|      date|
+--------+-----------+-----------+-------+-----+--------+--------+----------+
|    Aldi|         BE|Diapers-R-Us|6months|  6.8|     EUR|      40|2019-02-03|
|Kruidvat|         BE|    Nappy-k|2months|  4.8|     EUR|      30|2019-01-28|
|Carrefour|         FR|    Nappy-k|2months|  5.7|     EUR|      30|2019-02-06|
|   Tesco|        IRL|    Pampers|3months|  6.3|     EUR|      35|2019-02-07|
|      DM|         DE|    Huggies|newborn|  6.8|     EUR|      40|2019-02-01|
+--------+-----------+-----------+-------+-----+--------+--------+----------+
+-----------+-------+---------------+-------+
|      brand|  model|absorption_rate|comfort|
+-----------+-------+---------------+-------+
|Diapers-R-Us|6months|              2|      3|
|    Nappy-k|2months|              3|      4|
|    Pampers|3months|              4|      4|
|    Huggies|newborn|              3|      5|
+-----------+-------+---------------+-------+
```

Executing a join with 2 foreign keys

```
ratings_with_prices = ratings.join(prices, ["brand", "model"])
```

```
+------------+-------+---------------+------+---------+-----------+-----+--------+--------+--------
|       brand|  model|absorption_rate|comfort|    store|countrycode|price|currency|quantity|      da
+------------+-------+---------------+------+---------+-----------+-----+--------+--------+--------
|Diapers-R-Us|6months|              2|      3|     Aldi|         BE|  6.8|     EUR|      40|2019-02-
|     Nappy-k|2months|              3|      4| Kruidvat|         BE|  4.8|     EUR|      30|2019-01-
|     Nappy-k|2months|              3|      4|Carrefour|         FR|  5.7|     EUR|      30|2019-02-
|     Pampers|3months|              4|      4|    Tesco|        IRL|  6.3|     EUR|      35|2019-02-
|     Huggies|newborn|              3|      5|       DM|         DE|  6.8|     EUR|      40|2019-02-
+------------+-------+---------------+------+---------+-----------+-----+--------+--------+--------
```

Let's practice!

BUILDING DATA ENGINEERING PIPELINES IN PYTHON

Packaging your application

BUILDING DATA ENGINEERING PIPELINES IN PYTHON

Oliver Willekens
Data Engineer at Data Minded

datacamp

Running your pipeline locally

Running a Python program:

```
python hello_world.py  # script does something
```

Running a PySpark program *locally* is no different:

```
python my_pyspark_data_pipeline.py  # script starts at least a SparkSession
```

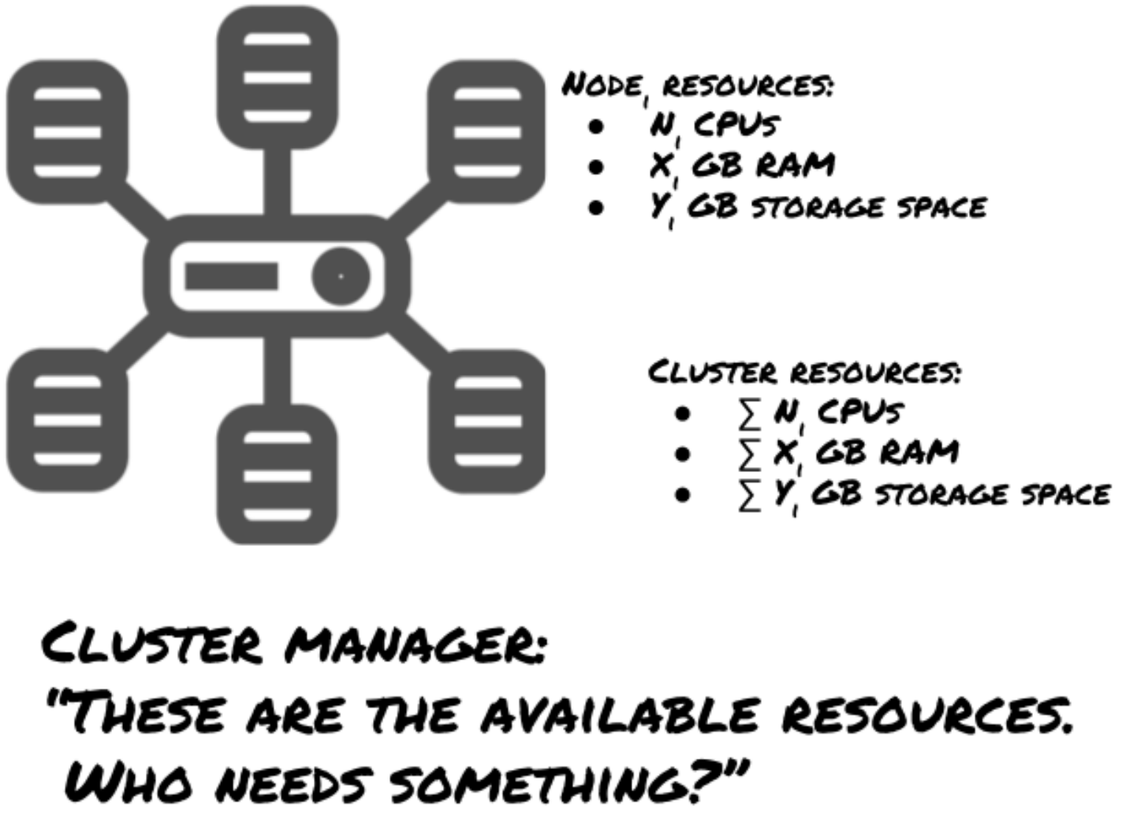
Conditions:

- local installation of Spark
- access to referenced resources
- classpath is properly configured

Using the “spark-submit” helper program

`spark-submit` comes with any Spark installation

1. sets up launch environment for use with the *cluster manager* and the selected *deploy mode*
2. invokes main class/app/module/function

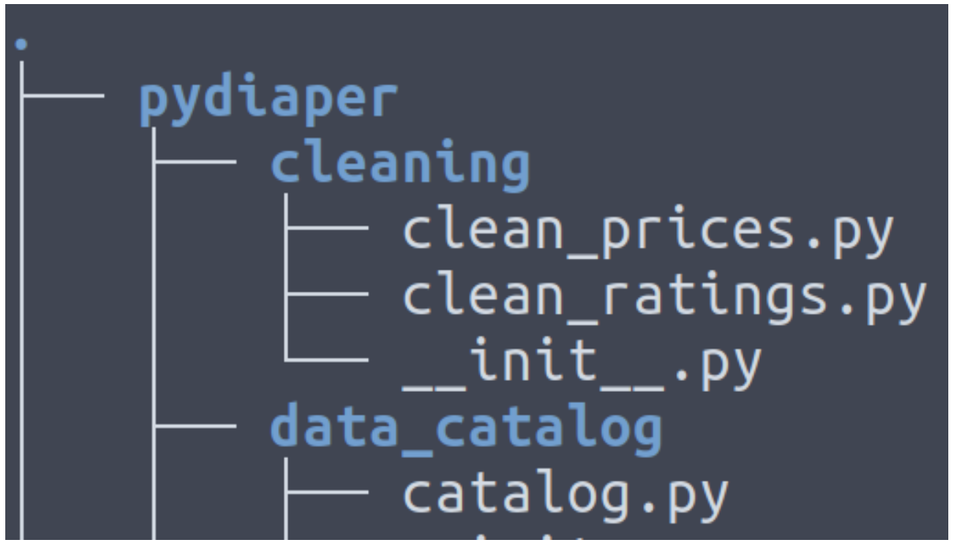


Basic arguments of “spark-submit”

```
spark-submit \
  --master "local[*]" \
  --py-files PY_FILES \
  MAIN_PYTHON_FILE \
  app_arguments
```

```
On your path, if Spark is installed
URL of the cluster manager
Comma-separated list of zip, egg or py
Path to the module to be run
Optional arguments parsed by main script
```

Collecting all dependencies in one archive




```
zip \
  --recurse-paths \
  dependencies.zip \
  pydiaper
```

```
spark-submit \
  --py-files dependencies.zip \
  pydiaper/cleaning/clean_prices.py
```

Let's practice!

BUILDING DATA ENGINEERING PIPELINES IN PYTHON

On the importance of tests

BUILDING DATA ENGINEERING PIPELINES IN PYTHON

Oliver Willekens
Data Engineer at Data Minded

datacamp

Software tends to change

Common reasons for change:

- new functionality desired
- bugs need to get squashed
- performance needs to be improved

Core functionality rarely evolves

How to ensure stability in light of changes?

Rationale behind testing

- **improves chance of code being correct in the *future***
  - prevent introducing breaking changes

- **raises *confidence* (not a guarantee) that code is correct *now***
  - assert actuals match expectations

- **most up-to-date documentation**
  - form of documentation that is always in sync with what's running

The test pyramid: where to invest your efforts

Testing takes time

- thinking what to test
- writing tests
- running tests

Testing has a high return on investment

- when targeted at the correct layer
- when testing the non-trivial parts, e.g.
  distance between 2 coordinates ?
  uppercasing a first name

© Martin Fowler “TestPyramid”

The test pyramid: where to invest your efforts

Testing takes time

- thinking what to test
- writing tests
- running tests

Testing has a high return on investment

- when targeted at the correct layer
- when testing the non-trivial parts, e.g.
  distance between 2 coordinates ?
  uppercasing a first name

© Martin Fowler “TestPyramid”

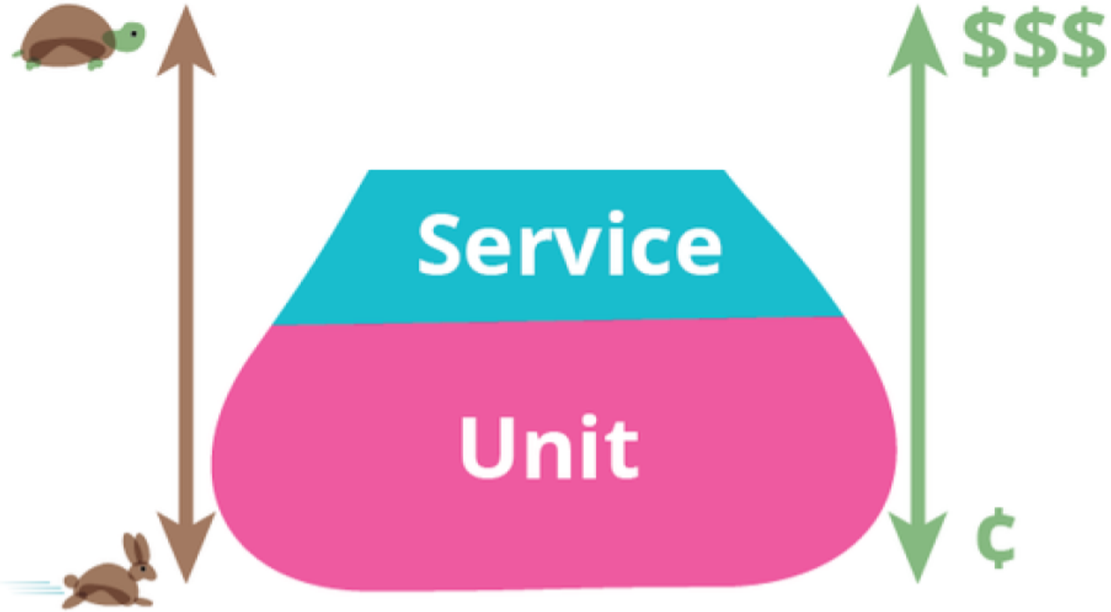
The test pyramid: where to invest your efforts

Testing takes time

- thinking what to test
- writing tests
- running tests

Testing has a high return on investment

- when targeted at the correct layer
- when testing the non-trivial parts, e.g.
  distance between 2 coordinates ?
  uppercasing a first name



© Martin Fowler “TestPyramid”

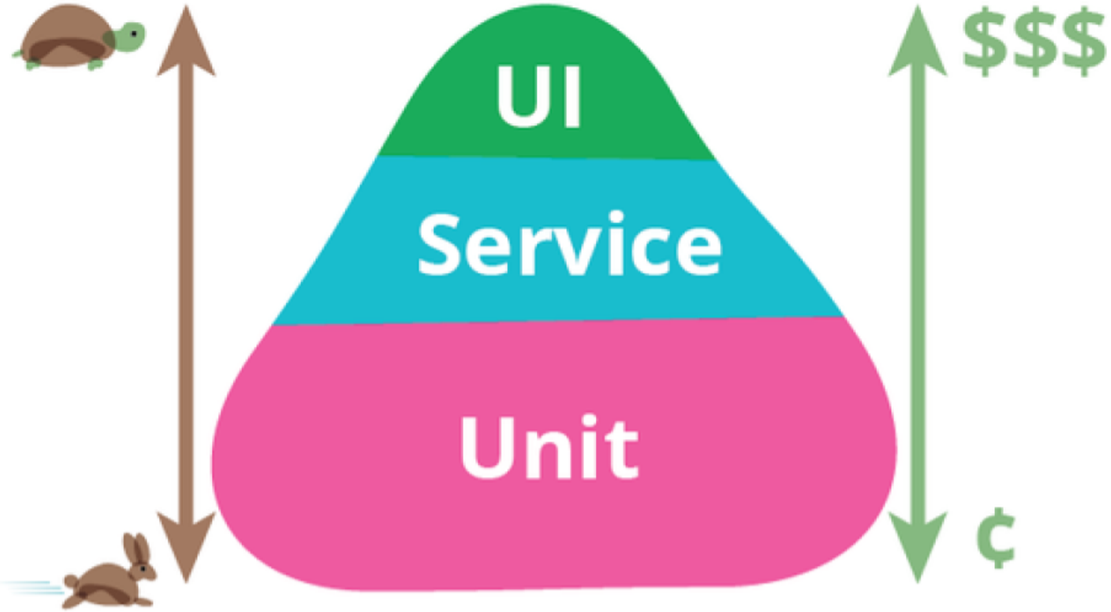
The test pyramid: where to invest your efforts

Testing takes time

- thinking what to test
- writing tests
- running tests

Testing has a high return on investment

- when targeted at the correct layer
- when testing the non-trivial parts, e.g.
  distance between 2 coordinates ?
  uppercasing a first name



© Martin Fowler “TestPyramid”

Let's have this sink in!

BUILDING DATA ENGINEERING PIPELINES IN PYTHON

Writing unit tests for PySpark

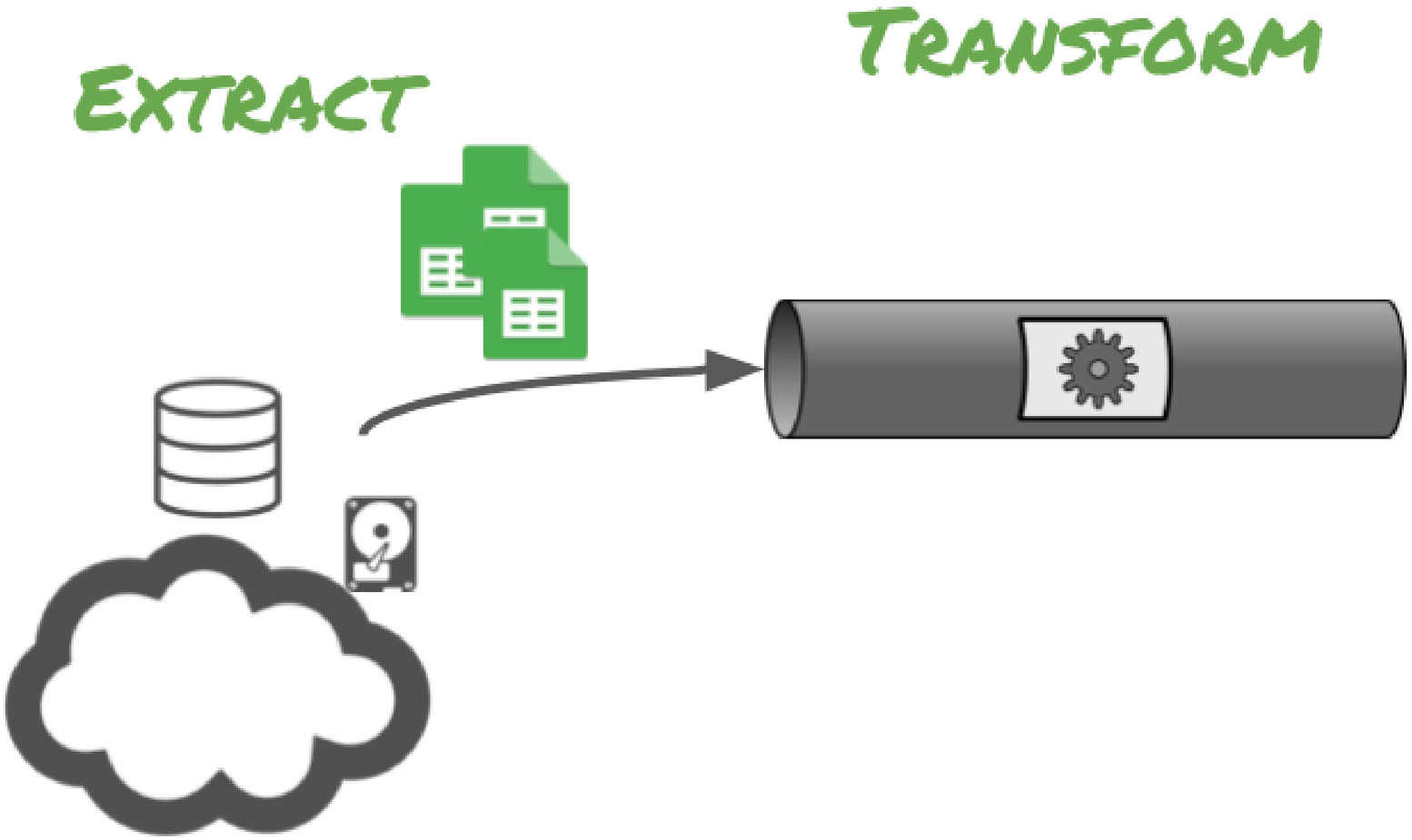
BUILDING DATA ENGINEERING PIPELINES IN PYTHON

Oliver Willekens
Data Engineer at Data Minded

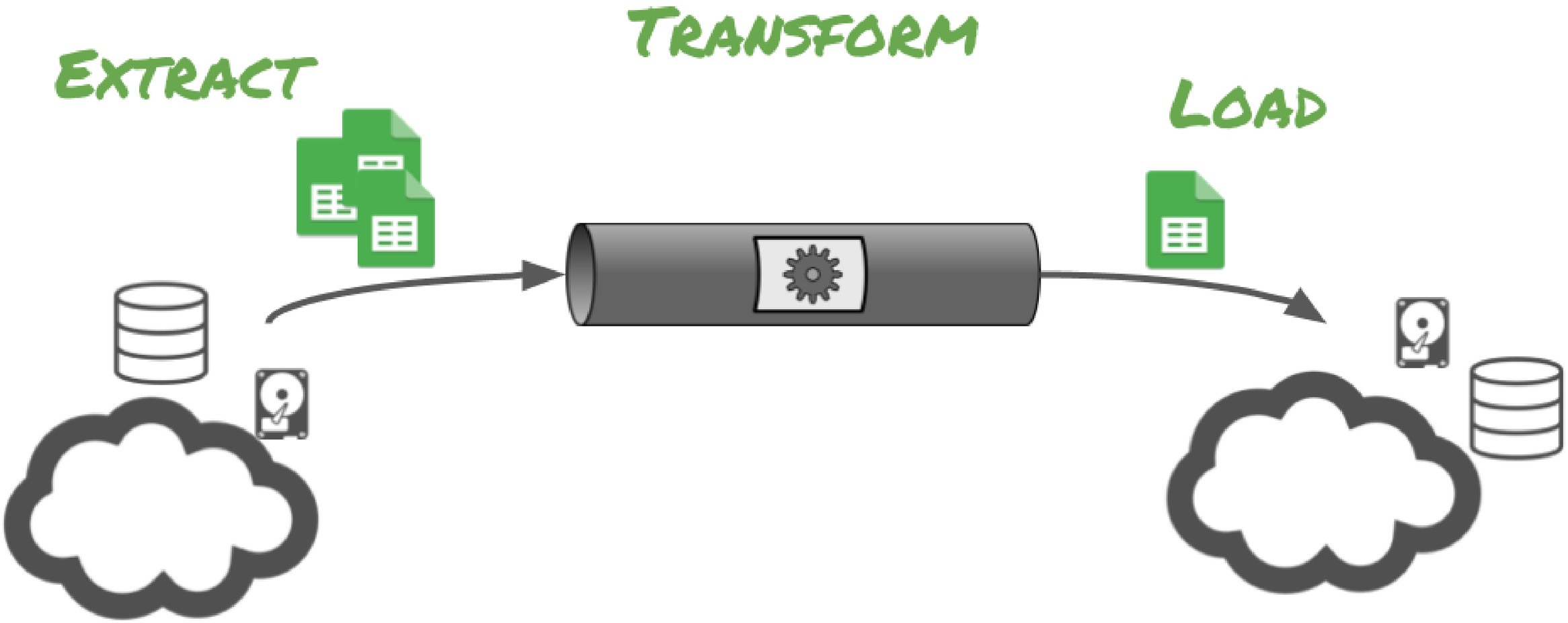
datacamp

Our earlier Spark application is an ETL pipeline

Our earlier Spark application is an ETL pipeline



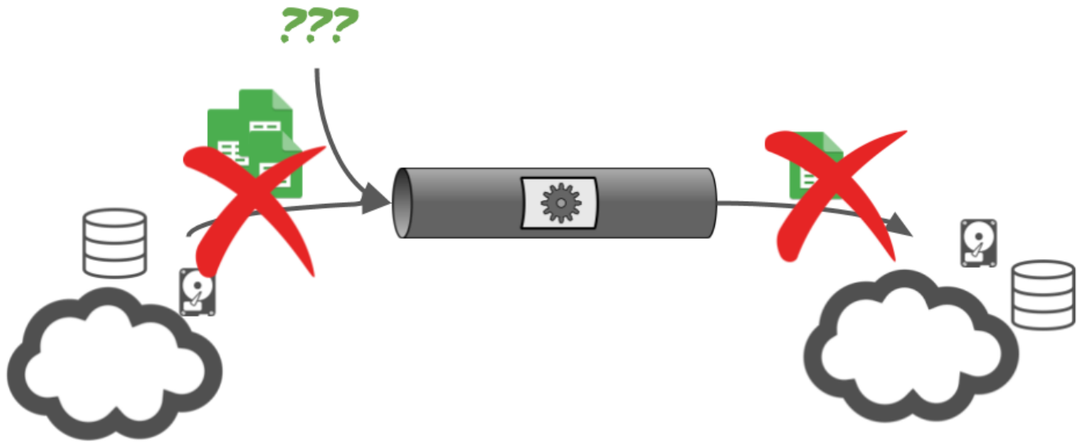
Our earlier Spark application is an ETL pipeline



Separate transform from extract and load

```
prices_with_ratings = spark.read.csv(…)  # extract
exchange_rates = spark.read.csv(…)  # extract

unit_prices_with_ratings = (prices_with_ratings
                            .join(…) # transform
                            .withColumn(…))  # transform
```



Solution: construct DataFrames in-memory

```
# Extract the data
df = spark.read.csv(path_to_file)
```

- depends on input/output (network access, filesystem permissions, ...)

- unclear how big the data is

- unclear what data goes in

```
from pyspark.sql import Row
purchase = Row("price",
               "quantity",
               "product")
record = purchase(12.99, 1, "cake")
df = spark.createDataFrame((record,))
```

+ inputs are clear

+ data is close to where it is being used ("code-proximity")

Create small, reusable and well-named functions

```
unit_prices_with_ratings = (prices_with_ratings
                            .join(exchange_rates, ["currency", "date"])
                            .withColumn("unit_price_in_euro",
                                        col("price") / col("quantity")
                                        * col("exchange_rate_to_euro"))
```

```
def link_with_exchange_rates(prices, rates):
    return prices.join(rates, ["currency", "date"])

def calculate_unit_price_in_euro(df):
    return df.withColumn(
        "unit_price_in_euro",
        col("price") / col("quantity") * col("exchange_rate_to_euro"))
```

Create small, reusable and well-named functions

```
def link_with_exchange_rates(prices, rates):
    return prices.join(rates, ["currency", "date"])

def calculate_unit_price_in_euro(df):
    return df.withColumn(
        "unit_price_in_euro",
        col("price") / col("quantity") * col("exchange_rate_to_euro"))
```

```
unit_prices_with_ratings = (
  calculate_unit_price_in_euro(
    link_with_exchange_rates(prices, exchange_rates)
  )
)
```

Testing a single unit

```
def test_calculate_unit_price_in_euro():
    record = dict(price=10,
                  quantity=5,
                  exchange_rate_to_euro=2.)
    df = spark.createDataFrame([Row(**record)])
```

Testing a single unit

```
def test_calculate_unit_price_in_euro():
    record = dict(price=10,
                  quantity=5,
                  exchange_rate_to_euro=2.)
    df = spark.createDataFrame([Row(**record)])
    result = calculate_unit_price_in_euro(df)
```

Testing a single unit

```
def test_calculate_unit_price_in_euro():
    record = dict(price=10,
                  quantity=5,
                  exchange_rate_to_euro=2.)
    df = spark.createDataFrame([Row(**record)])
    result = calculate_unit_price_in_euro(df)

    expected_record = Row(**record, unit_price_in_euro=4.)
    expected = spark.createDataFrame([expected_record])
```

Testing a single unit

```
def test_calculate_unit_price_in_euro():
    record = dict(price=10,
                  quantity=5,
                  exchange_rate_to_euro=2.)
    df = spark.createDataFrame([Row(**record)])
    result = calculate_unit_price_in_euro(df)

    expected_record = Row(**record, unit_price_in_euro=4.)
    expected = spark.createDataFrame([expected_record])

    assertDataFrameEqual(result, expected)
```

Take home messages

1. Interacting with external data sources is costly
2. Creating in-memory DataFrames makes testing easier
   - the data is in plain sight,
   - focus is on just a small number of examples.
3. Creating small and well-named functions leads to more reusability and easier testing.

Let's practice!

BUILDING DATA ENGINEERING PIPELINES IN PYTHON

Continuous testing

BUILDING DATA ENGINEERING PIPELINES IN PYTHON

Oliver Willekens
Data Engineer at Data Minded

Running a test suite

Execute tests in Python, with one of:

in stdlib	3rd party
unittest	pytest
doctest	nose

Core task: **assert** or raise

Examples:

```
assert computed == expected
with pytest.raises(ValueError):  # pytest specific
```

Manually triggering tests

In a Unix shell:

```
cd ~/workspace/my_good_python_project
pytest .
# Lots of output…
== 19 passed, 2 warnings in 36.80 seconds ==
```

```
cd ~/workspace/my_bad_python_project
pytest .
# Lots of output…
== 3 failed, 1 passed in 6.72 seconds ==
```

Note: Spark increases time to run unit tests.

Automating tests

Problem:

- forget to run unit tests when making changes

Solution:

- Automation

How:

- Git -> configure hooks
- Configure CI/CD pipeline to run tests automatically

CI/CD

Continuous Integration:

- get code changes integrated with the master branch regularly.

Continuous Delivery:

- Create “artifacts” (deliverables like documentation, but also programs) that can be deployed into production without breaking things.

Configuring a CI/CD tool

CircleCI looks for *.circleci/config.yml*.

Example:

```
jobs:
  test:
    docker:
      - image: circleci/python:3.6.4
    steps:
      - checkout
      - run: pip install -r requirements.txt
      - run: pytest .
```

Often:

1. checkout code
2. install test & build requirements
3. run tests
4. package/build the software artefacts
5. deploy the artefacts (update docs / install app / ...)

Let's practice!

BUILDING DATA ENGINEERING PIPELINES IN PYTHON

Modern day workflow management

BUILDING DATA ENGINEERING PIPELINES IN PYTHON

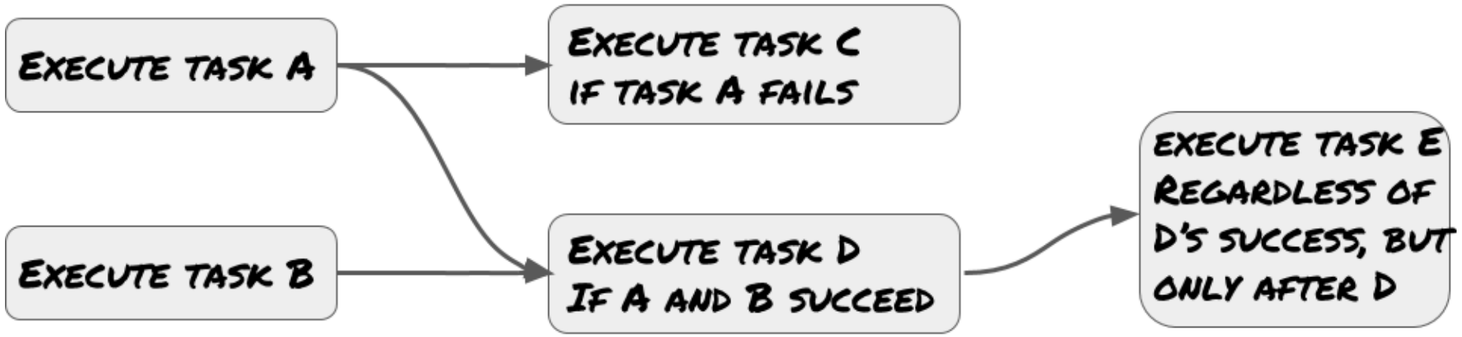
Oliver Willekens
Data Engineer at Data Minded

datacamp

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
```

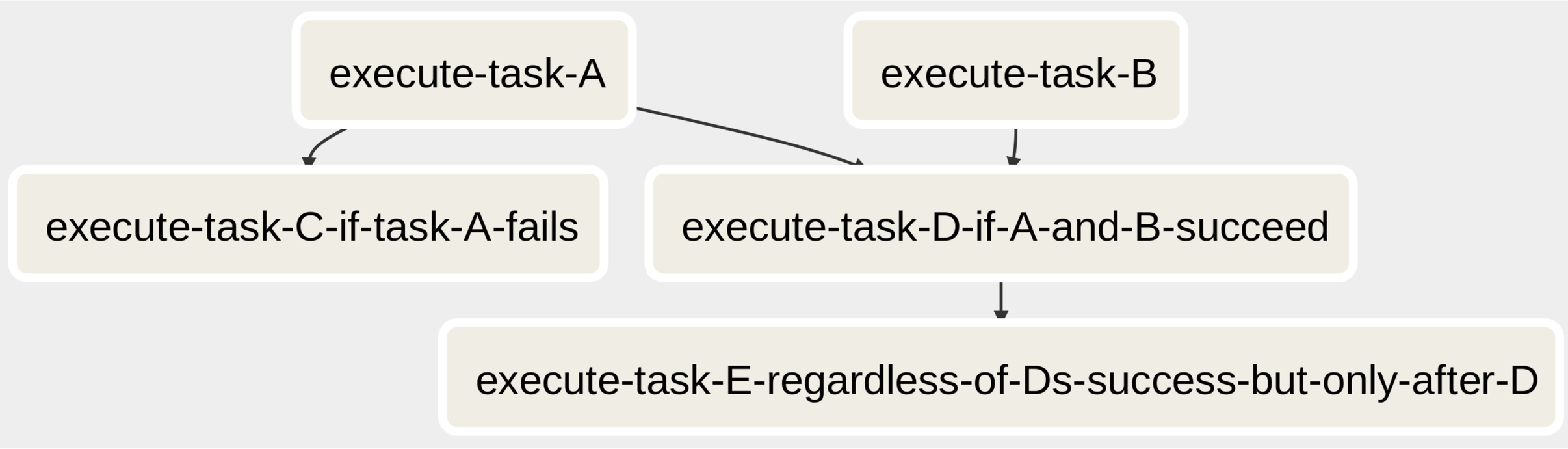
Cron is a dinosaur.

Modern workflow managers:

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

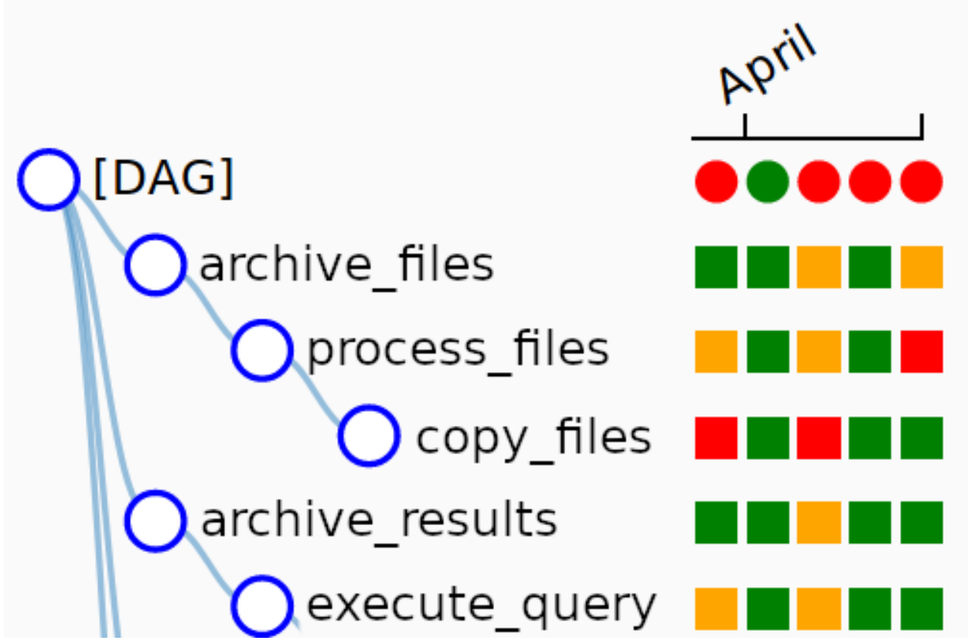
Apache Airflow fulfills modern engineering needs

1. Create and visualize complex workflows,



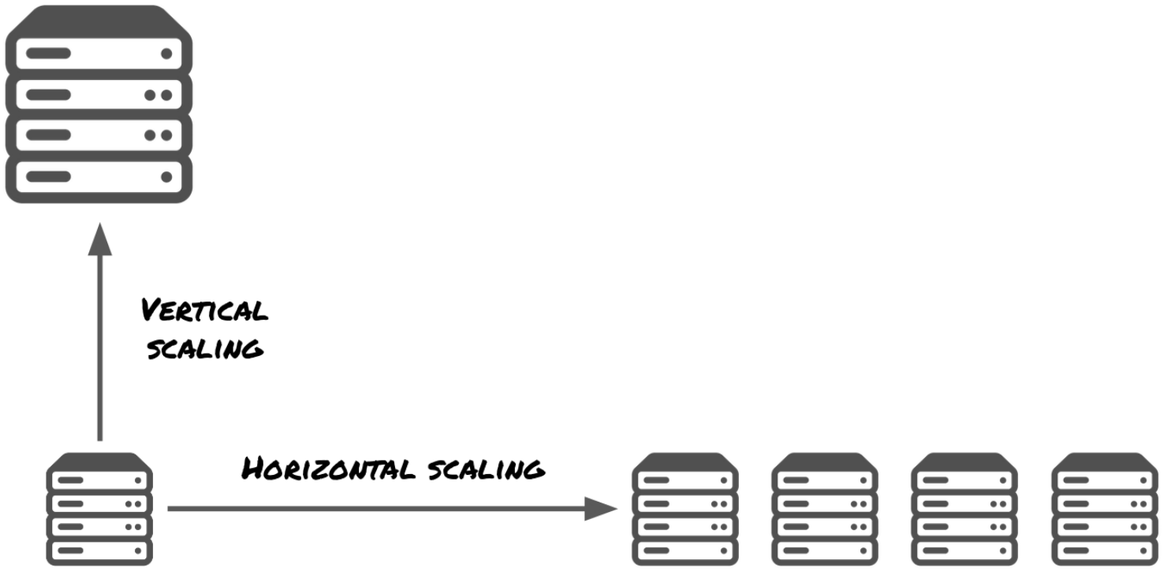
Apache Airflow fulfills modern engineering needs

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

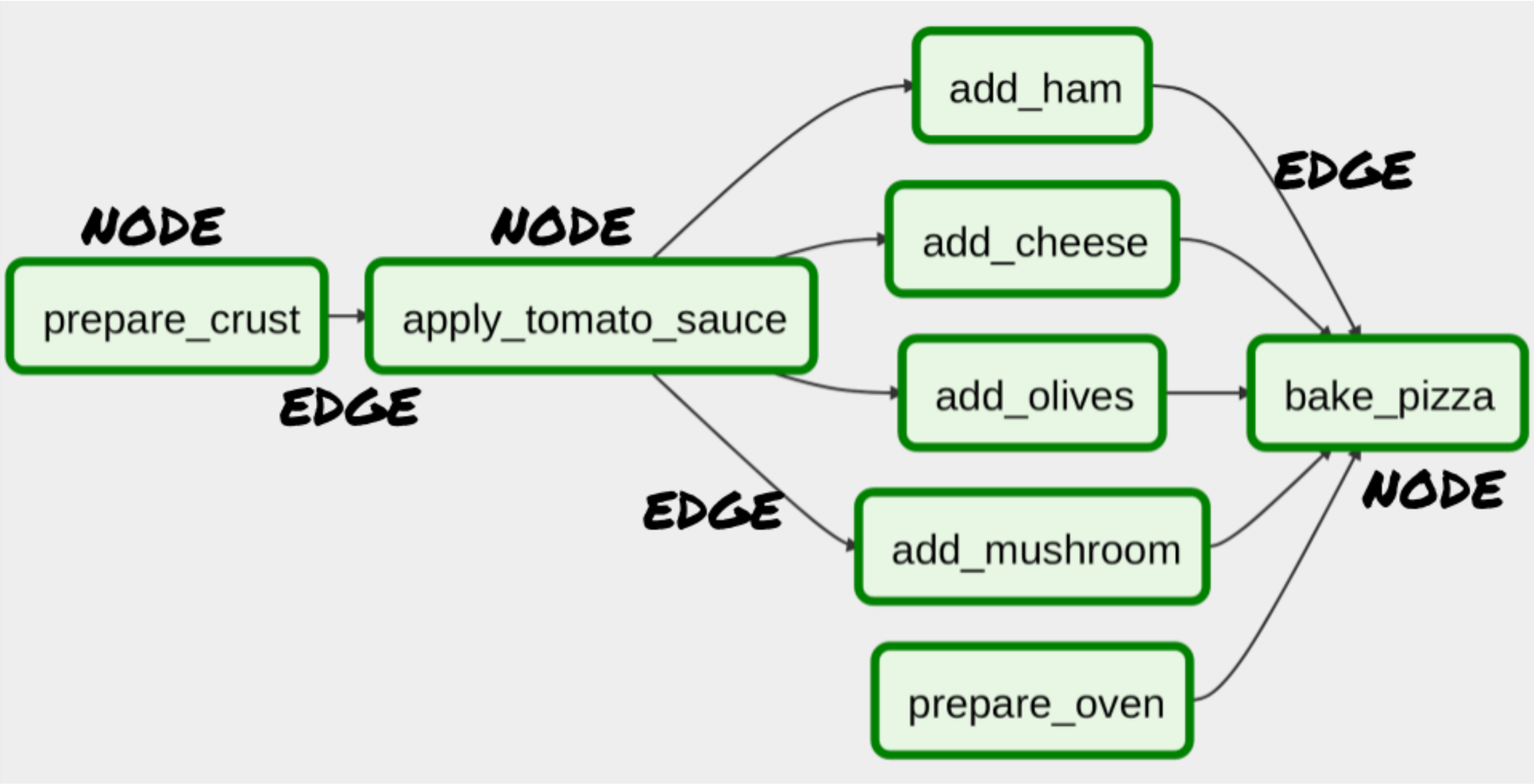


Apache Airflow fulfills modern engineering needs

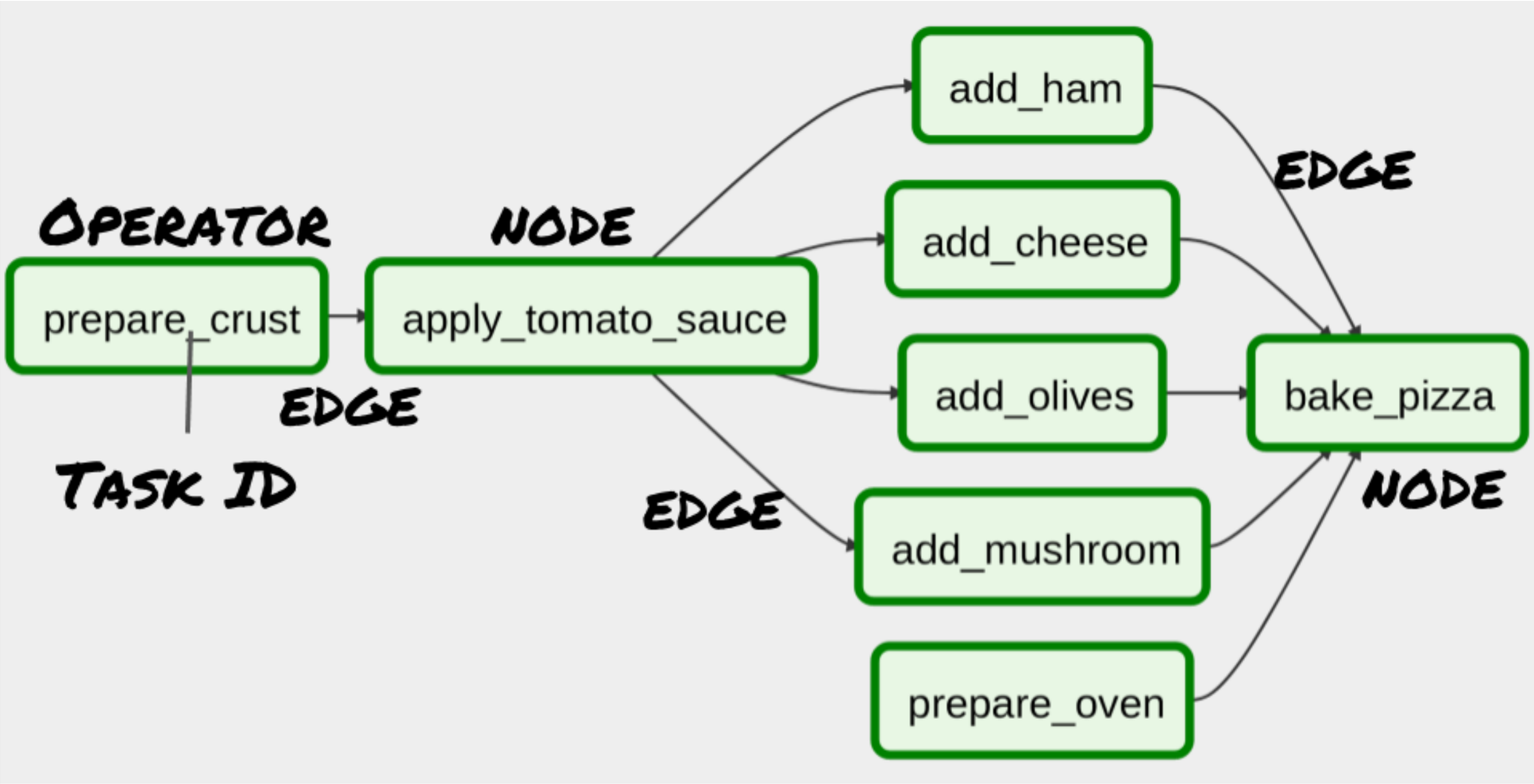
1. Create and visualize complex workflows,
2. Monitor and log workflows,
3. Scales horizontally.



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)
)
```

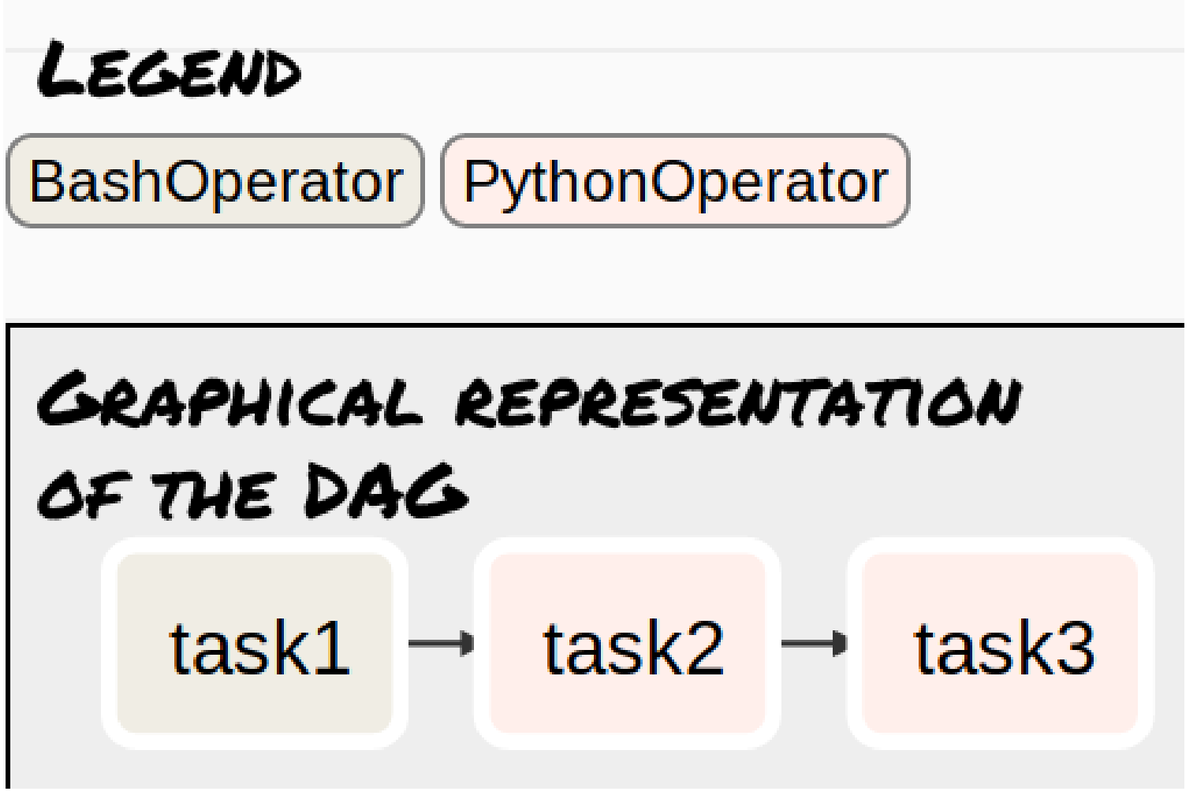
Classes of operators

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

datacamp

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

datacamp

Installing and configuring Airflow

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

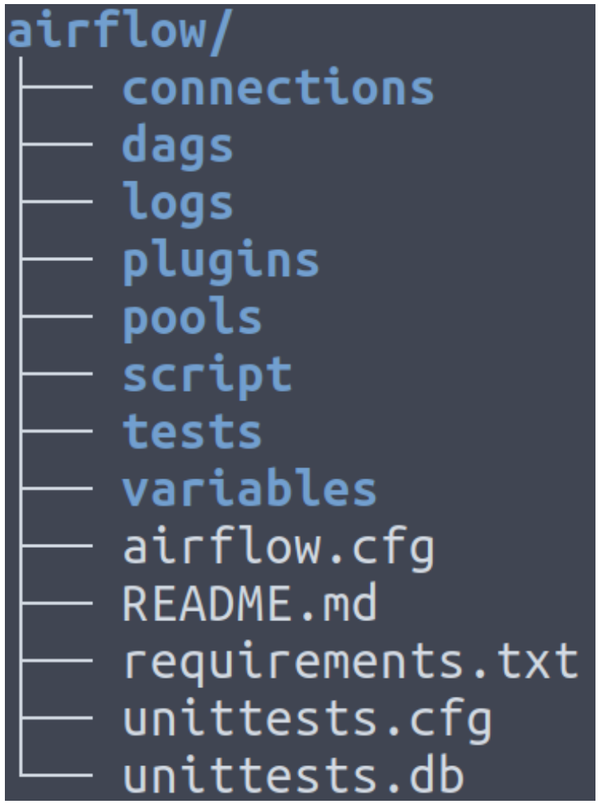
```
airflow/
├── logs
├── airflow.cfg
├── airflow.db
└── unittests.cfg
```

```
[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.

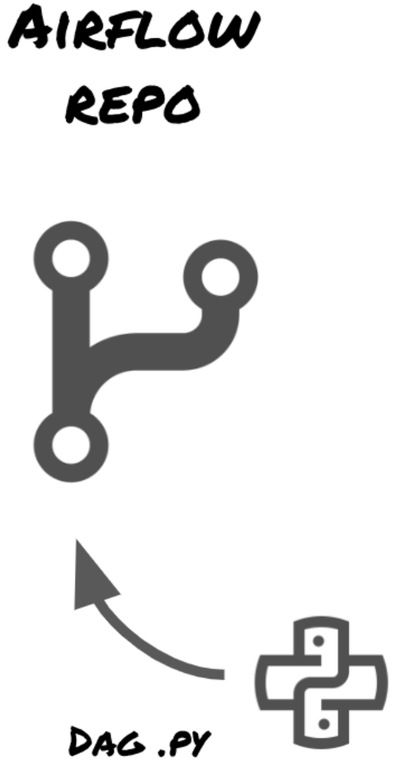


Example Airflow deployment test

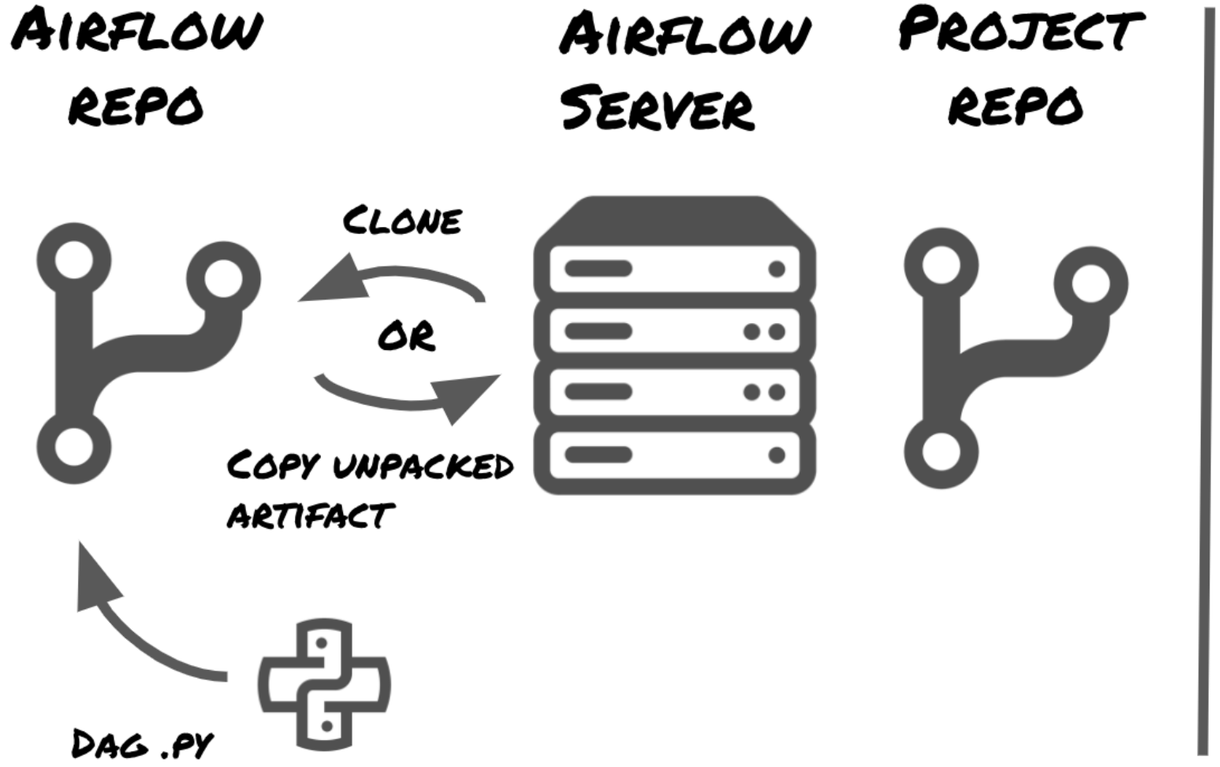
```
from airflow.models import DagBag

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
```

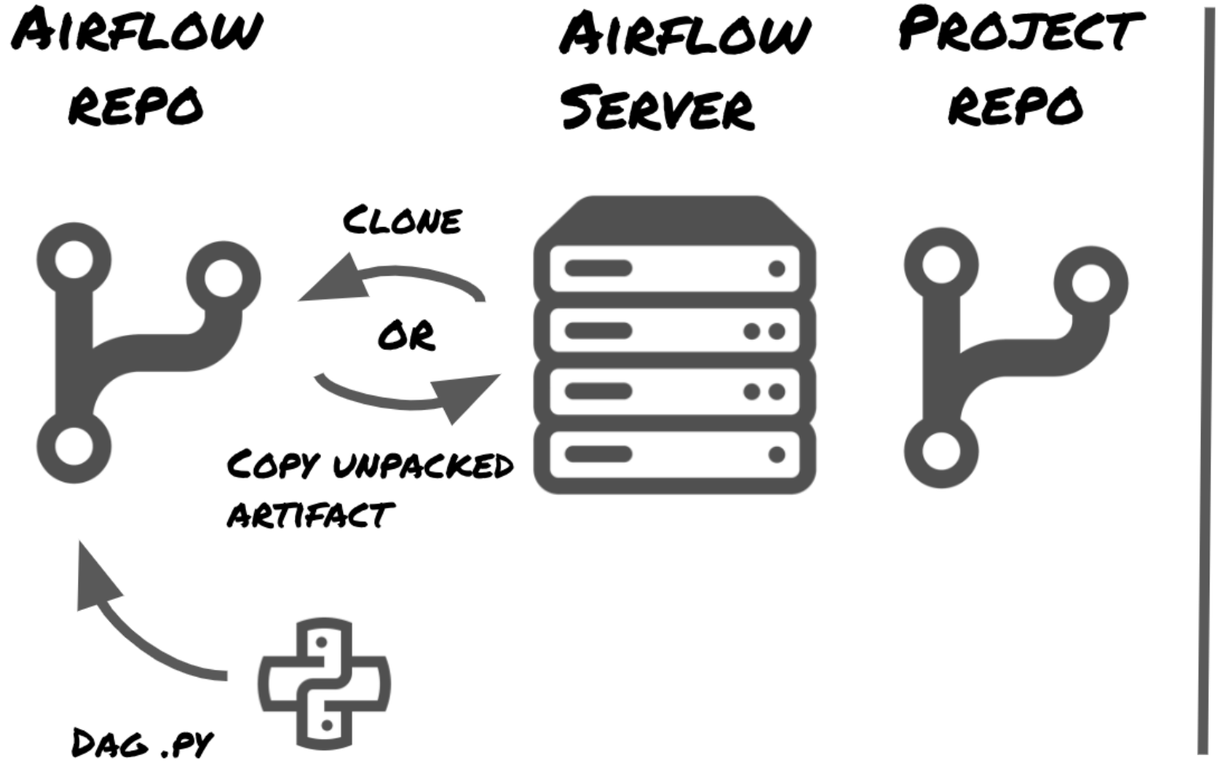
Transferring DAGs and plugins

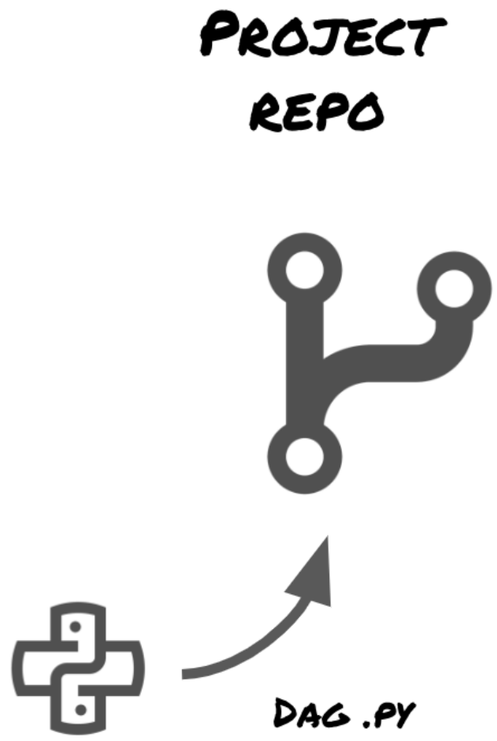


Transferring DAGs and plugins

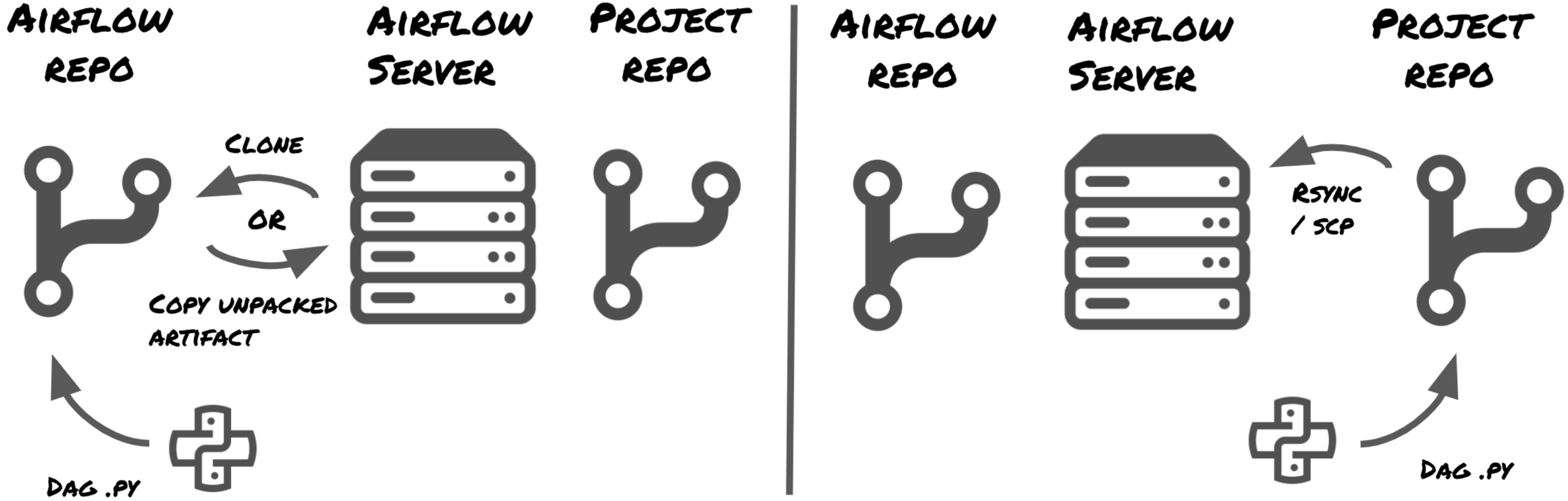


Transferring DAGs and plugins





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

Congratulations!
????

BUILDING DATA ENGINEERING PIPELINES IN PYTHON