Copyright Notice

DeepLearning.AI

Orchestration, Monitoring, and Automating your Data Pipelines

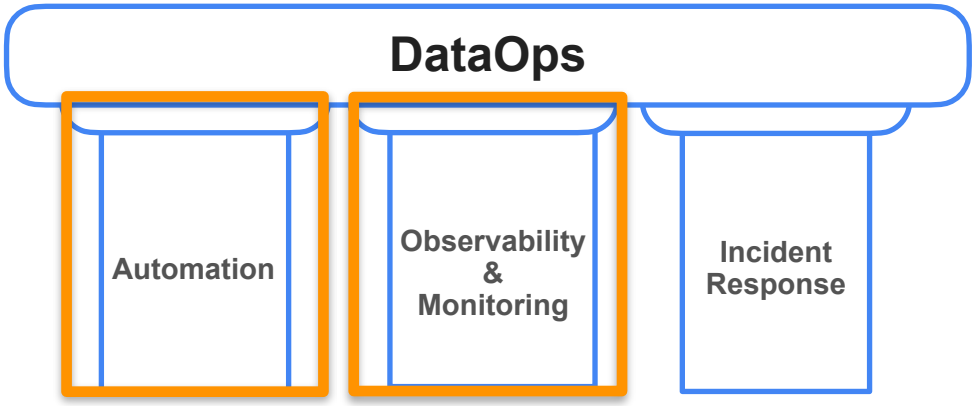
Week 4

DeepLearning.AI

Orchestration, Monitoring, and Automating your Data Pipelines

Week 4 Overview

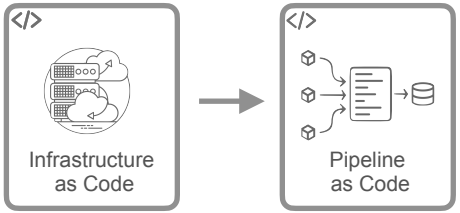
The Three Pillars of DataOps



HashiCorp Terraform

great expectations

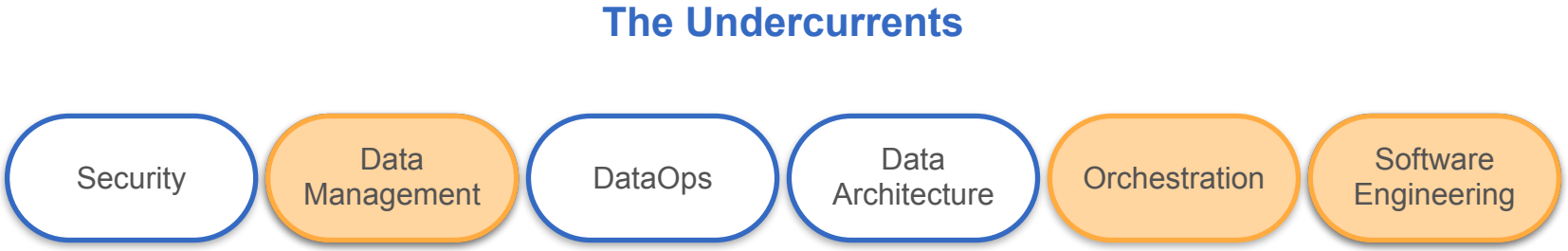
Amazon CloudWatch



- Automate individual tasks in your data pipeline
- Build in data quality tests and monitoring

Apache Airflow

Orchestration



Week 4 Plan

1. Evolution of Orchestration Tools

Week 4 Plan

1. Evolution of Orchestration Tools

2. Basic Details of Orchestration

Week 4 Plan

1. Evolution of Orchestration Tools

2. Basic Details of Orchestration

3. Details of Airflow

DeepLearning.AI

Orchestration, Monitoring, and Automating Data Pipelines

Before Orchestration

Cron

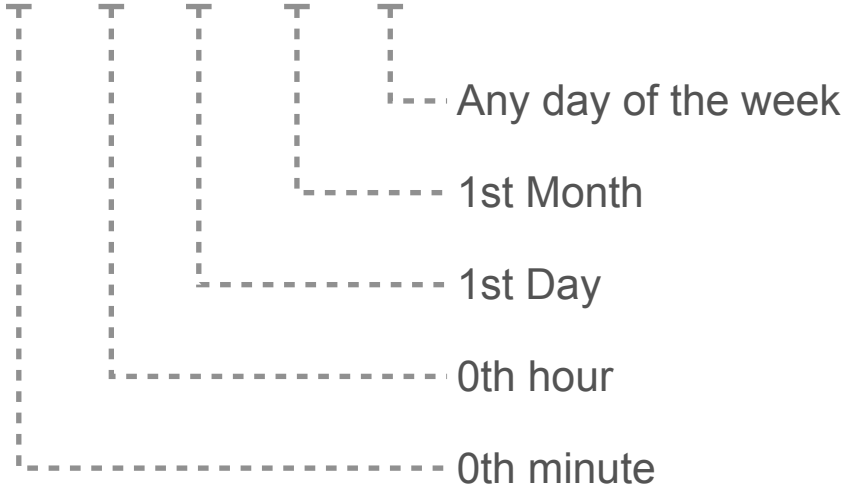
- A command line utility introduced in the 1970s
- Used to execute a particular command at a specified date and time.

Cron Job


No restriction
* * * * * command to be executed
Week (0 = Sun … 6 = Saturday)
Month (1 - 12)
Day (1 - 31)
Hour (0 - 23)
Minute (0 - 59)

Cron Job

```
0 0 1 1 *  echo "Happy New Year"
```



Every year at midnight on January 1st

```
01/01/2006 0:00:00 Happy New Year
01/01/2007 0:00:00 Happy New Year
01/01/2008 0:00:00 Happy New Year
01/01/2009 0:00:00 Happy New Year
```

Scheduling Data Pipeline with Cron

Every night at midnight

```
00*** python ingest_from_rest_api.py
```

Scheduling Data Pipeline with Cron

Every night at midnight

```
00*** python ingest_from_rest_api.py
```

1 AM every night

```
01*** python transform_api_data.py
```

Scheduling Data Pipeline with Cron

Every night at midnight

```
00*** python ingest_from_rest_api.py
```

1 AM every night

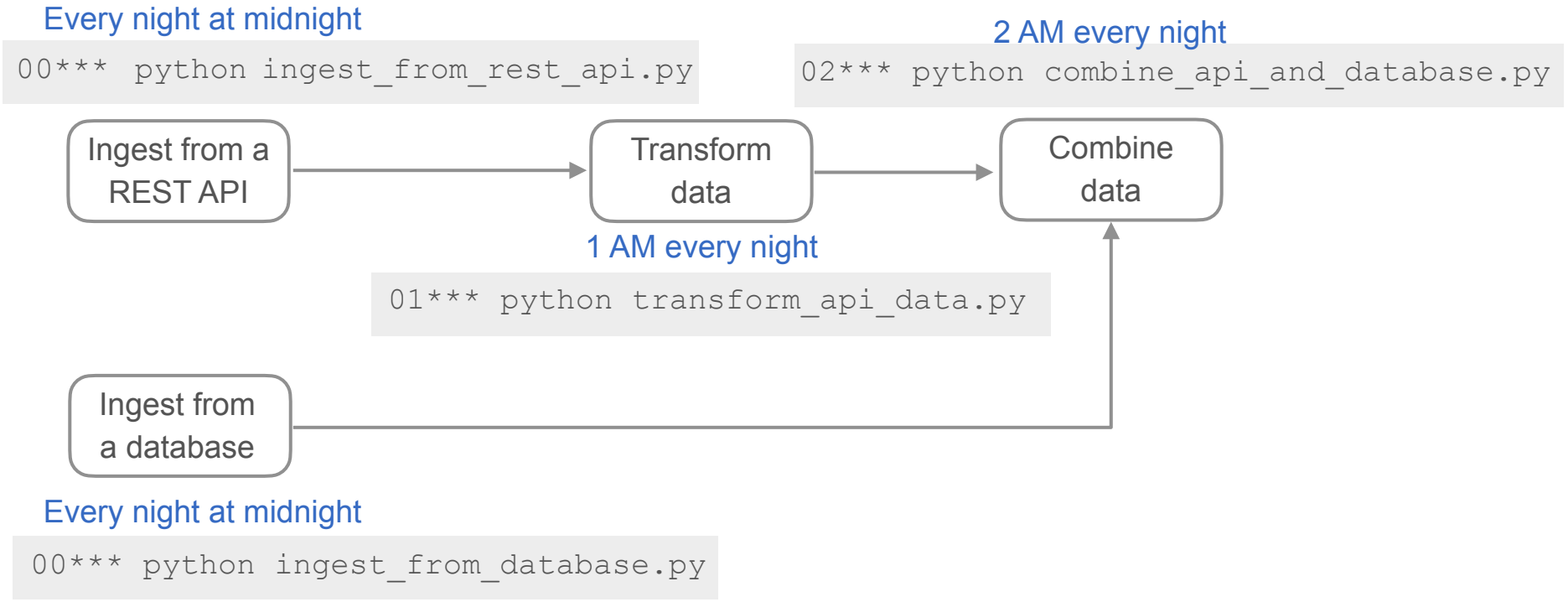
```
01*** python transform_api_data.py
```

Ingest from a database

Every night at midnight

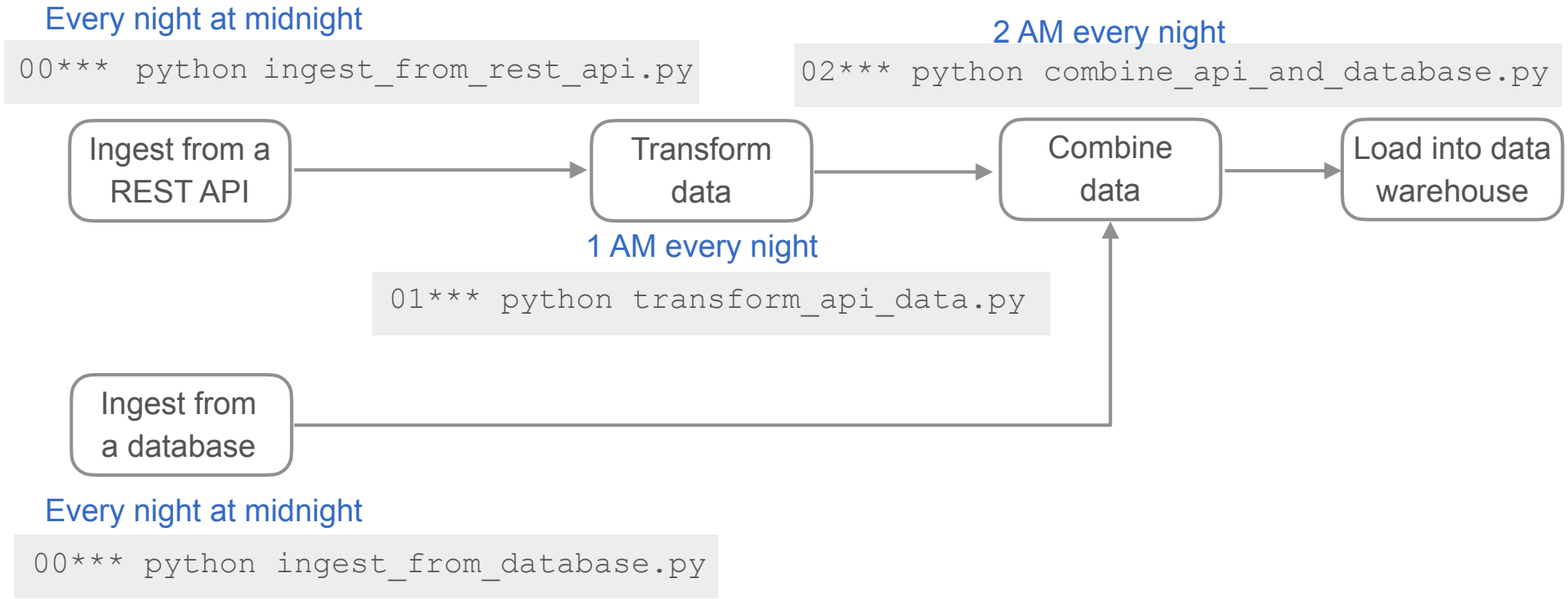
```
00*** python ingest_from_database.py
```

Scheduling Data Pipeline with Cron



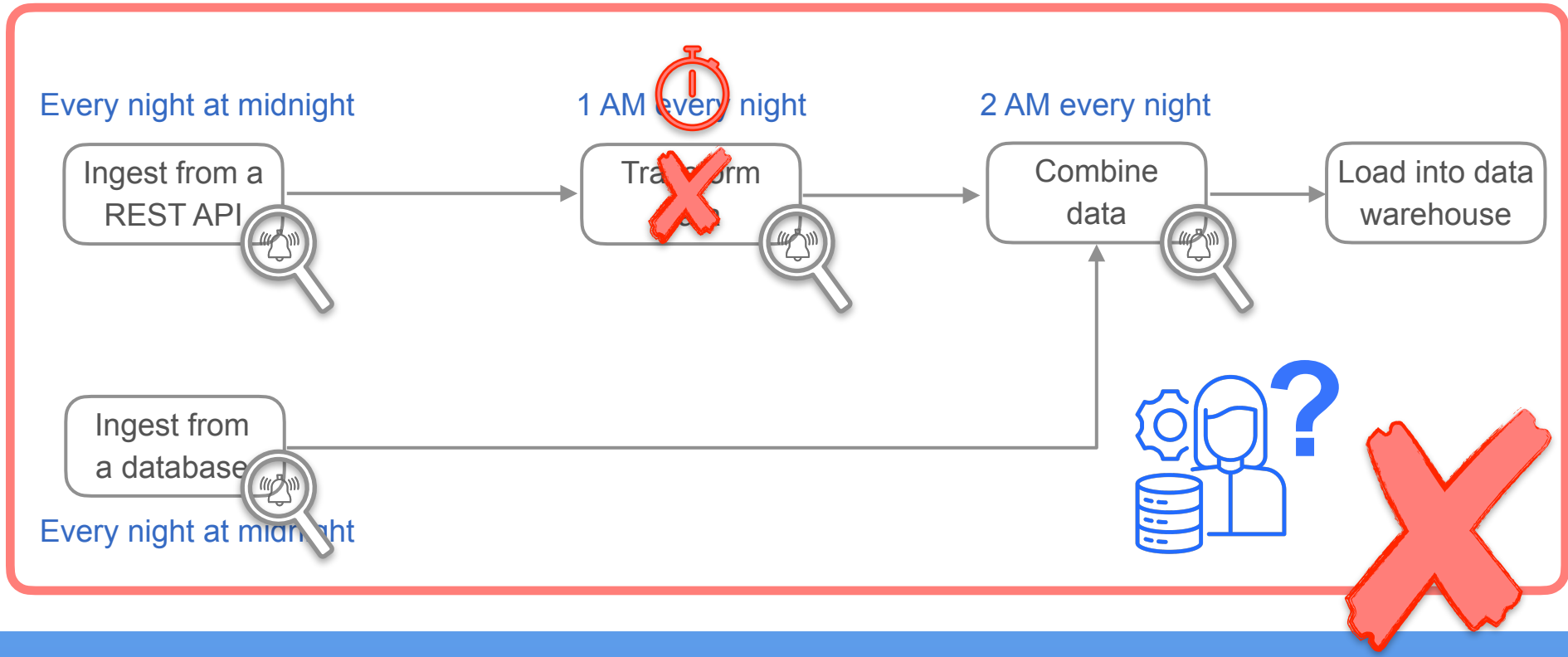
Scheduling Data Pipeline with Cron

Pure Scheduling Approach



Scheduling Data Pipeline with Cron

Pure Scheduling Approach



When To Use Cron?

- To schedule simple and repetitive tasks:

  Example: Regular data downloads

- In the prototyping phase

  Example: Testing aspects of your data pipeline

DeepLearning.AI

Orchestration, Monitoring, and Automating Data Pipelines

Evolution of Orchestration Tools

Orchestration Tools

late 2000s

Dataswarm

facebook

Orchestration Tools

late 2000s

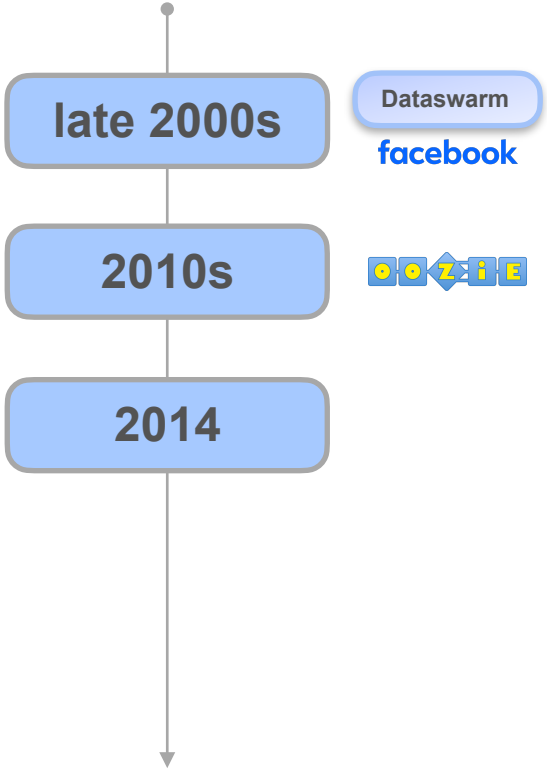
Dataswarm

facebook

2010s

- Designed to work within a Hadoop cluster
- Difficult to use in a more heterogeneous environment

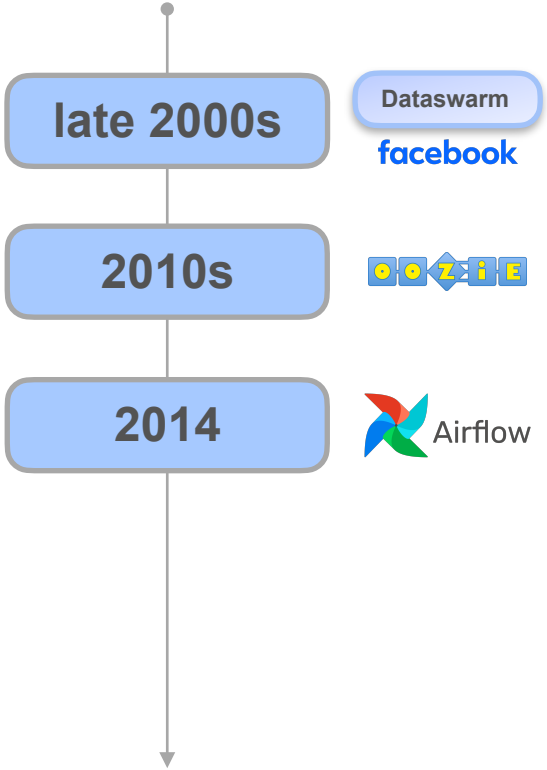
Orchestration Tools



airbnb

Airflow

Orchestration Tools



Orchestration Tools

late 2000s — Dataswarm, facebook

2010s — Oozie

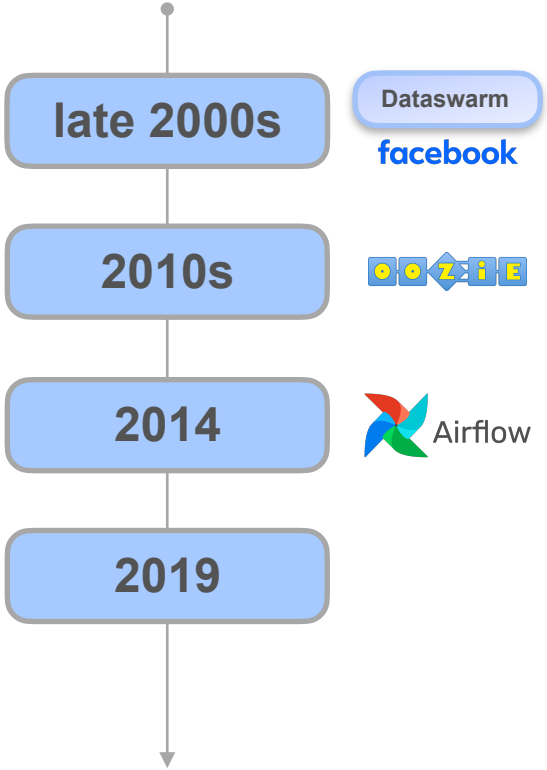
2014 — Airflow

Maxime Beauchemin

airbnb

Open source project

Orchestration Tools



Advantages and Challenges of Airflow

Advantages	Challenges
• Airflow is written in Python • The Airflow open source project is very active • Airflow is available as a managed service	• Scalability challenge • Ensuring data integrity • No support for streaming pipelines

Other Open-Source Orchestration Tools

Other Open-Source Orchestration Tools

More scalable orchestration solutions than Airflow

Built-in capabilities for data quality testing

DeepLearning.AI

Orchestration, Monitoring, and Automating your Data Pipelines

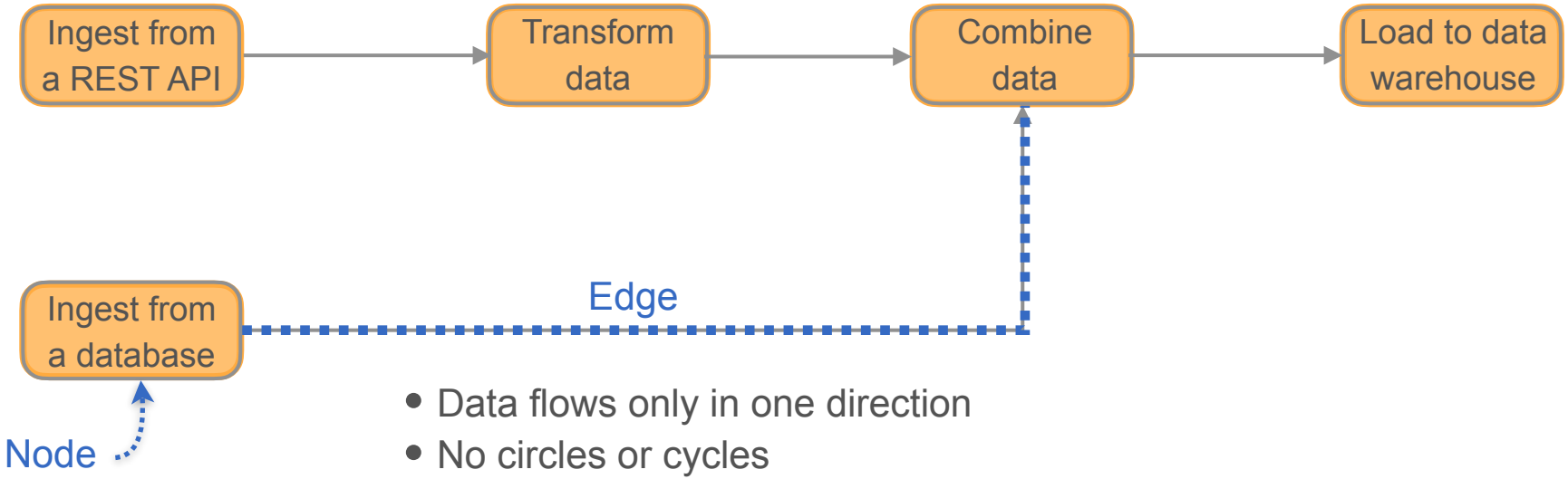
Orchestration Basics

Orchestration

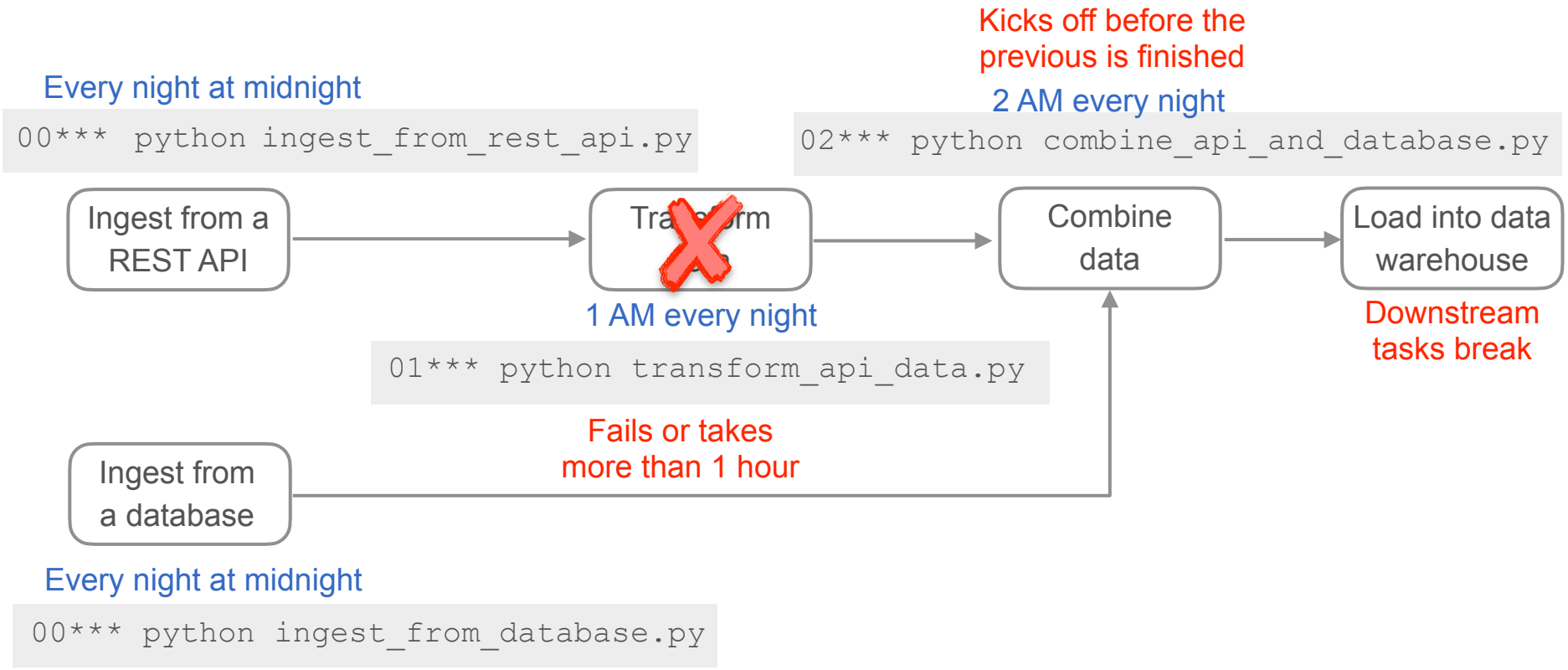
PROS	CONS
• Set up dependencies • Monitor tasks • Get alerts • Create fallback plans	• More operational overhead than simple Cron scheduling

Orchestration

Directed Acyclic Graph (DAG)

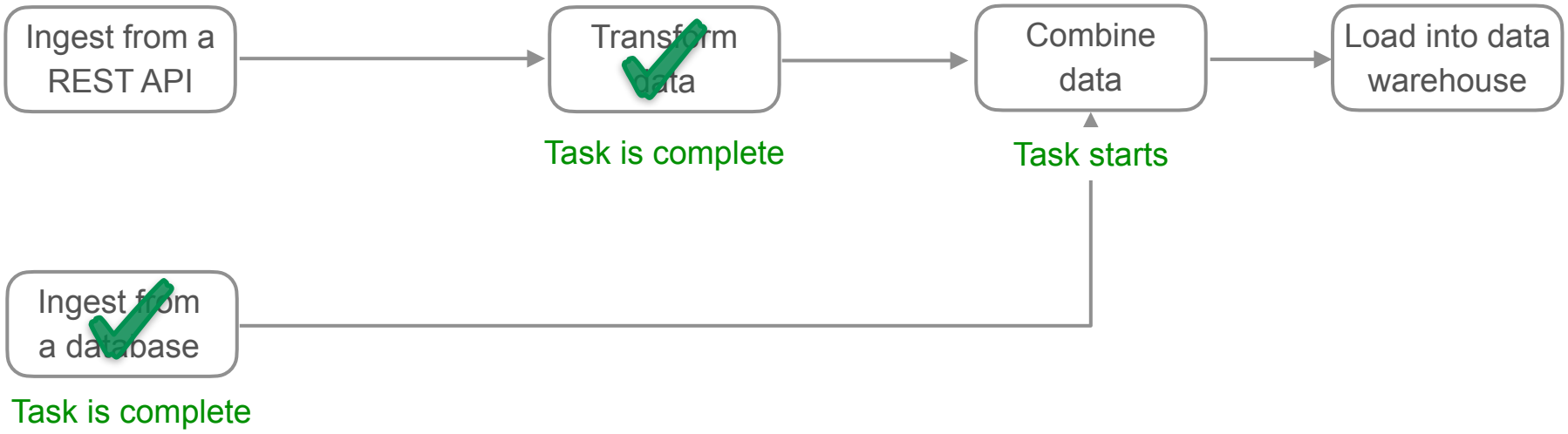


Scheduling Tasks with Cron



Orchestration

You could build in dependencies between tasks.



Orchestration in Airflow

```
with DAG(dag_id="dag_etl_example", start_date=datetime( year: 2024,  month: 3,  day: 23), schedule='@weekly'):

    task_ingest_api = PythonOperator(task_id='ingest_from_API', python_callable=ingest_from_rest_api)
    task_ingest_database = PythonOperator(task_id='ingest_from_database', python_callable=ingest_from_database)
    task_transform_api = PythonOperator(task_id = 'transform_data', python_callable=transform_api_data)
    task_combine_data = PythonOperator(task_id = 'combine_data', python_callable=combine_api_and_database)
    task_load_warehouse = PythonOperator(task_id = 'load_warehouse', python_callable=load_to_warehouse)

    [task_ingest_api >> task_transform_api, task_ingest_database] >> task_combine_data >> task_load_warehouse
```

Tasks

Dependencies

Orchestration in Airflow

You can set up the conditions on which the DAG should run:

time-based conditions

event-based conditions

Orchestration in Airflow

Trigger the DAG based on a schedule

```
with DAG(dag_id="my_first_dag",
         start_date=datetime( year: 2024,  month: 3,  day: 13),
         description="First DAG",
         tags=["data_engineering_team"],
         schedule='@daily',
         catchup=False):
```

Orchestration in Airflow

Trigger the DAG based on an event

Example: when a dataset has been updated by another DAG

```
dataset = Dataset("s3://dataset-bucket/example.csv")

with DAG(dag_id="dag_etl_example",
         start_date=datetime( year: 2024,  month: 3,  day: 23),
         schedule=[dataset],
         catchup= False):
```

Orchestration in Airflow

Trigger a portion of the DAG based on an event

Example: presence of a file in an S3 bucket

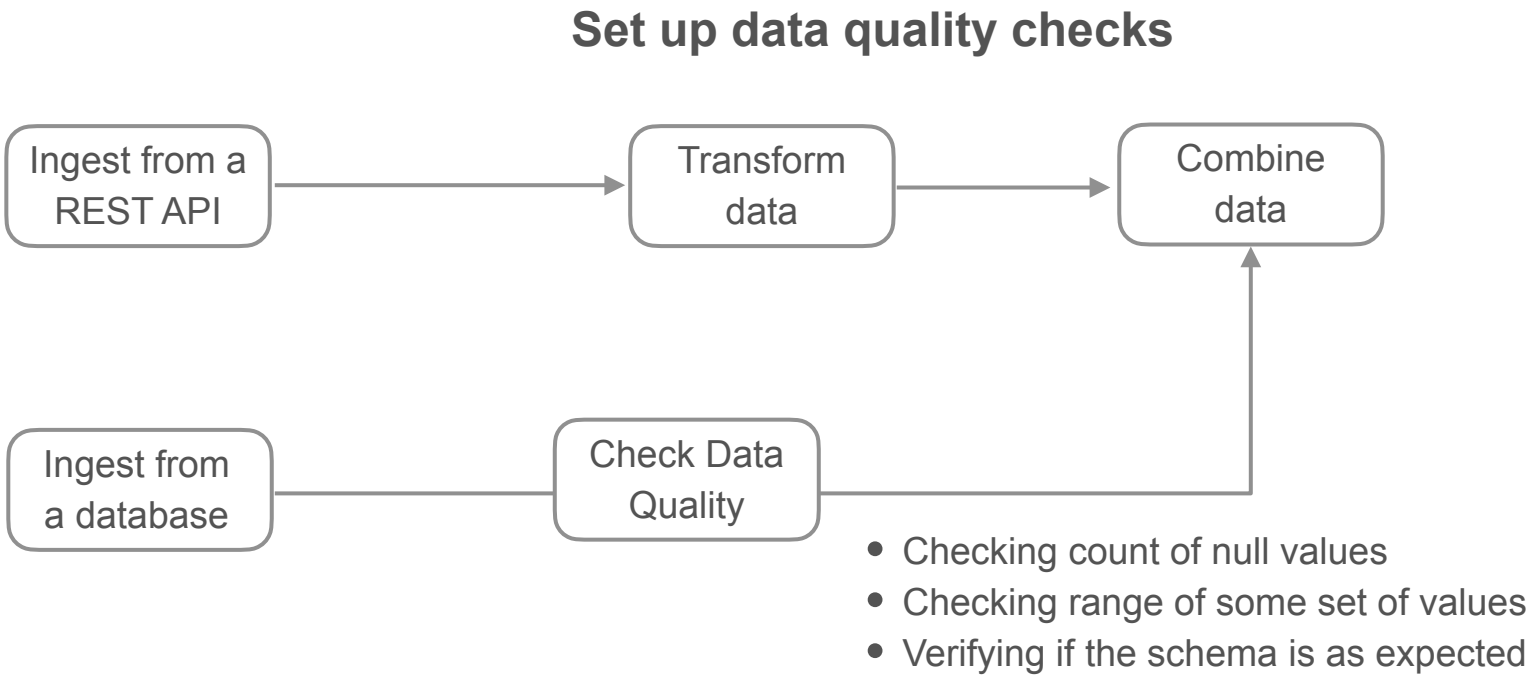
```
s3_sensor = S3KeySensor(task_id='s3_file_check',
                        bucket_key='my_file.csv',
                        bucket_name='my_bucket_name',
                        aws_conn_id='my_aws_connection')
```

Orchestration in Airflow

Set up monitoring, logging and alerts

```
t1 = PythonOperator(
    task_id="ingest_data",
    python_callable=extract_data,
    email=['someone@deeplearning.ai'],
    email_on_failure=True,
    email_on_retry=True
)
```

Orchestration in Airflow

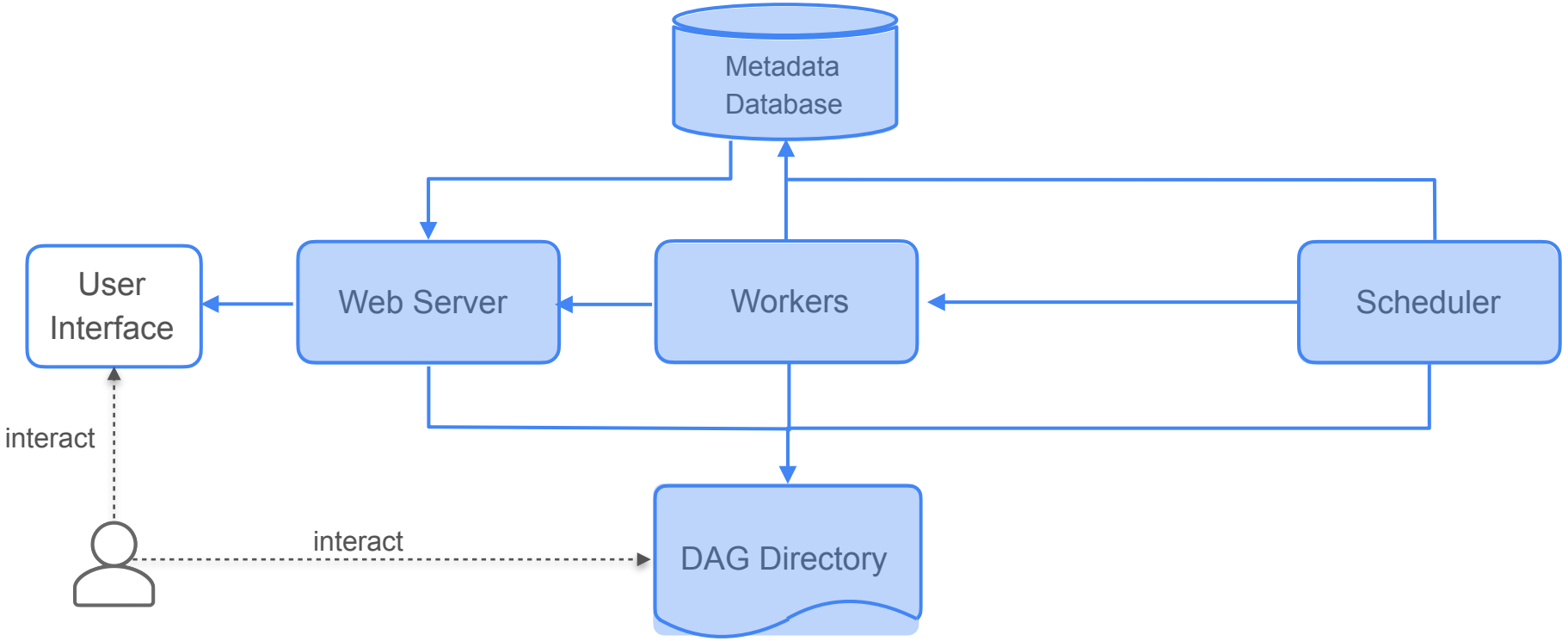


DeepLearning.AI

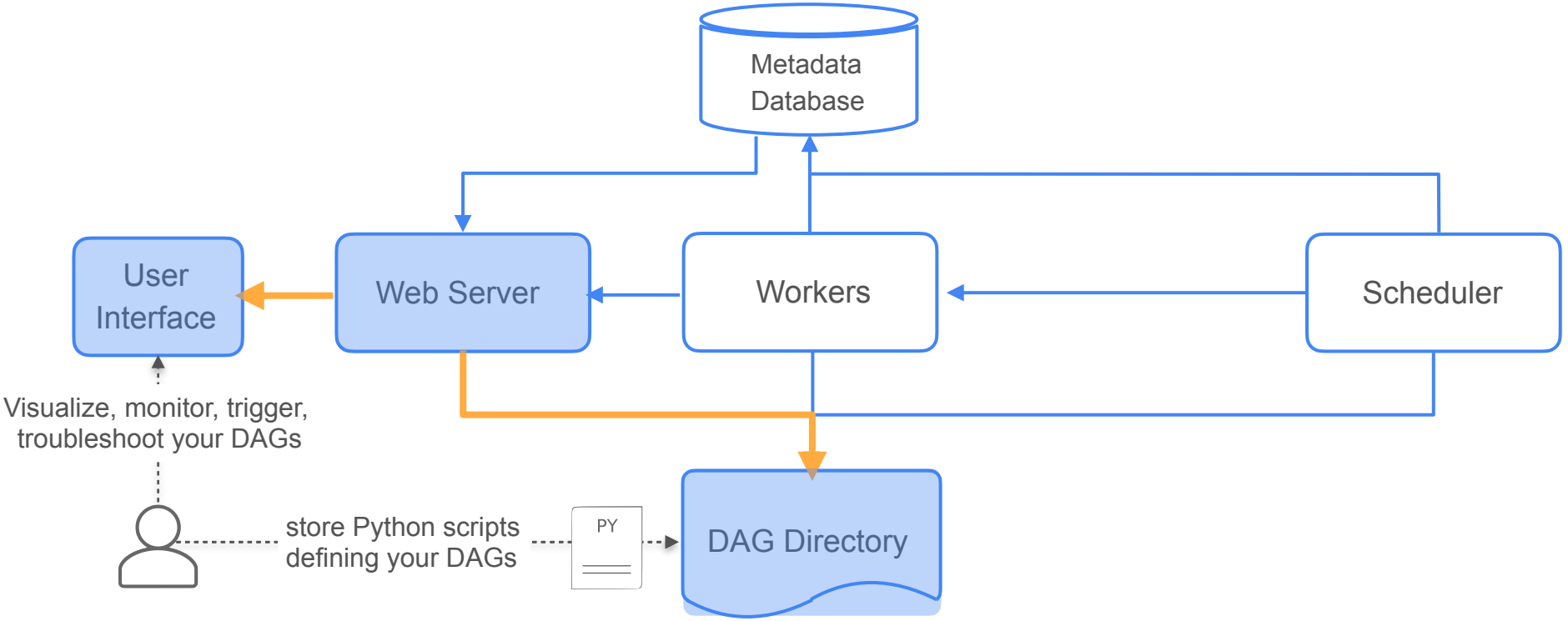
Orchestration, Monitoring, and Automating your Data Pipelines

Airflow - Core Components

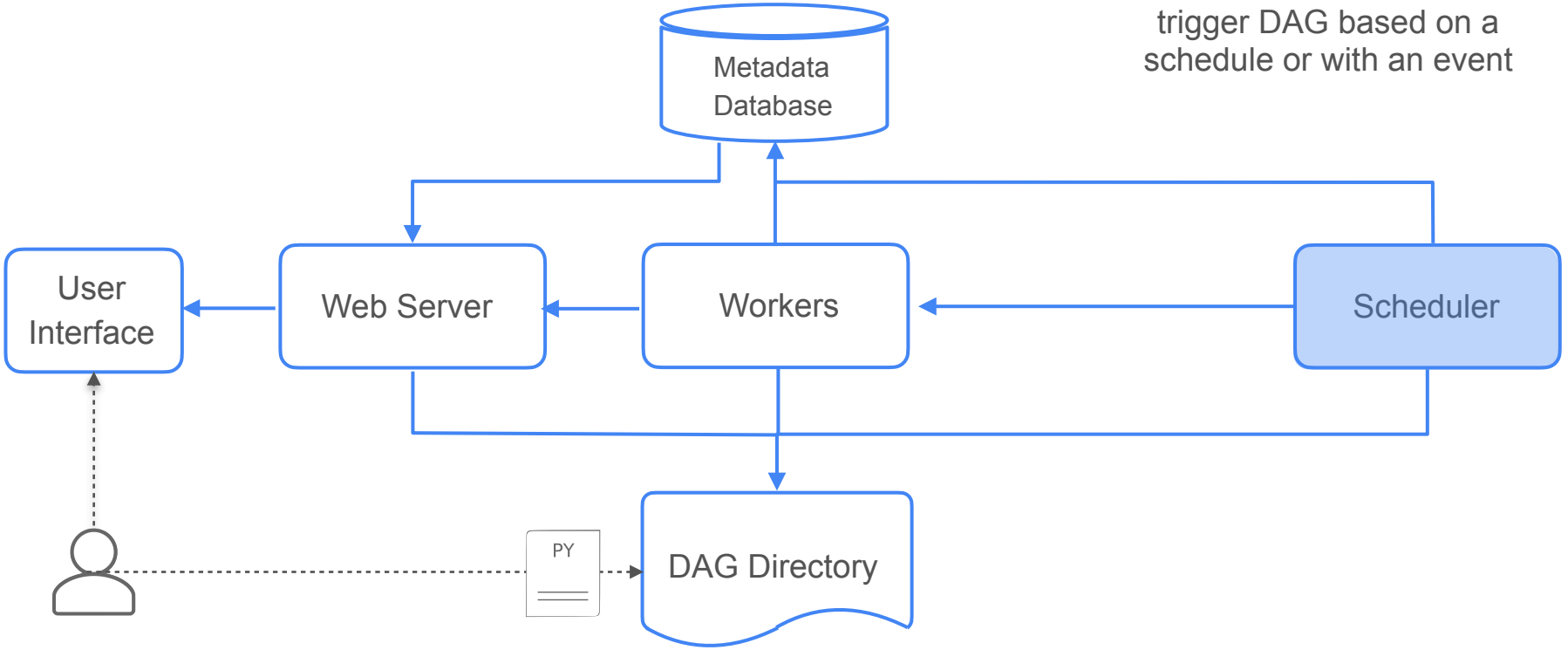
Airflow Components



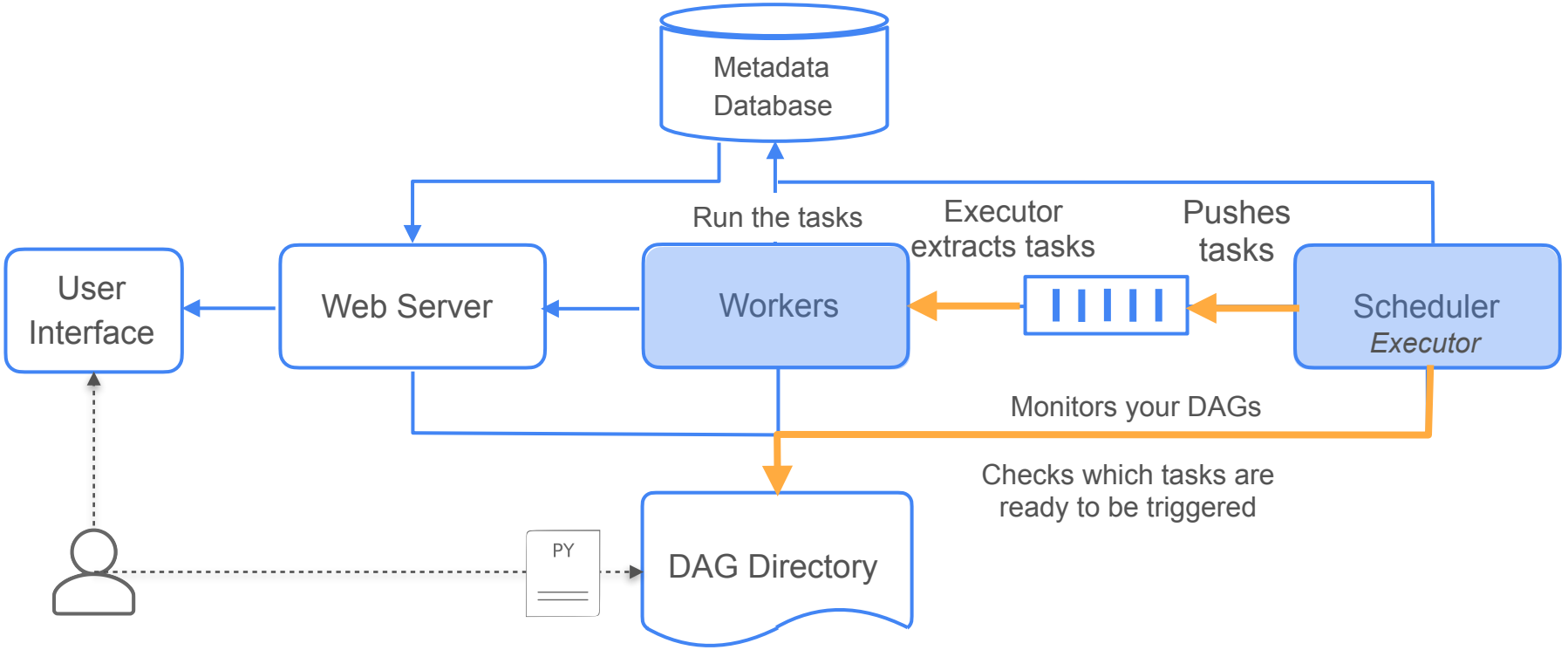
Airflow Components



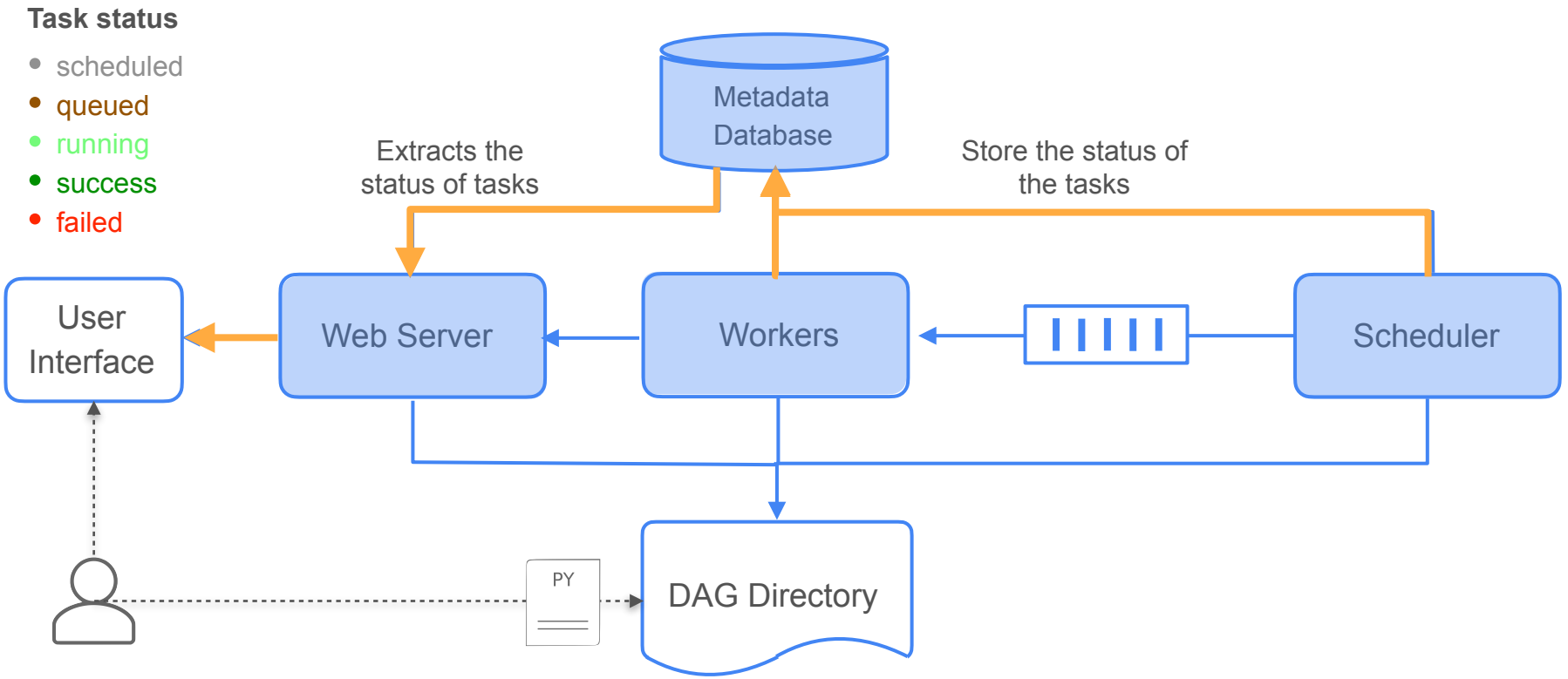
Airflow Components



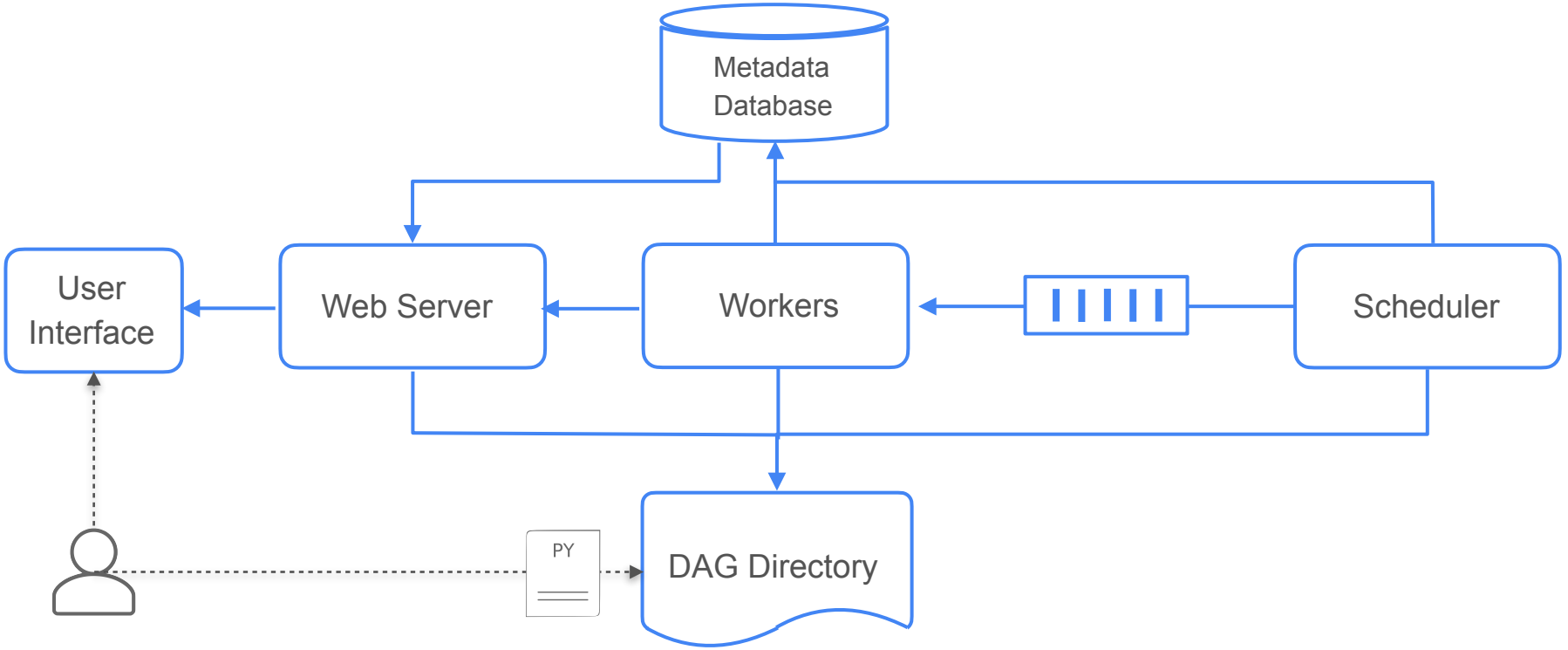
Airflow Components



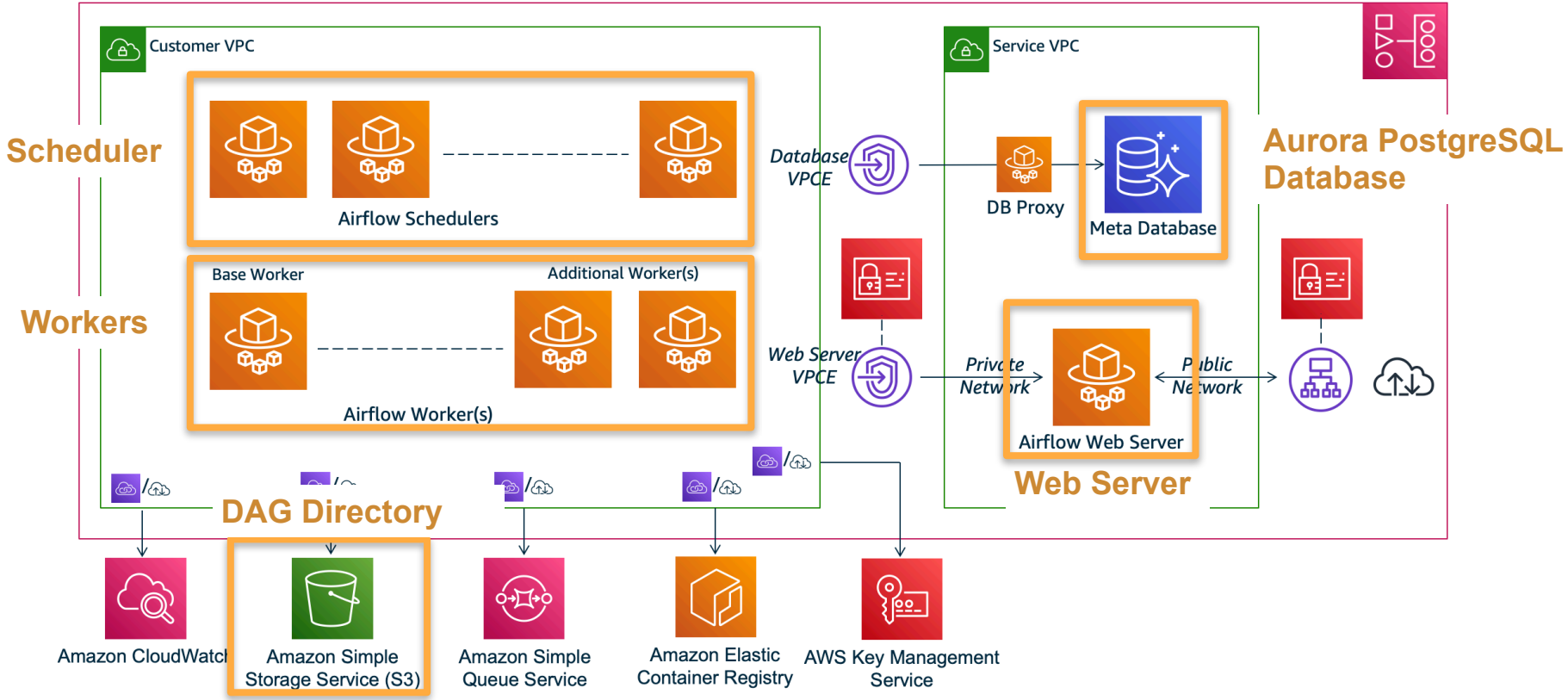
Airflow Components



Airflow Components



Amazon Managed Workflows for Apache Airflow (MWAA)



Airflow Components

In the labs of this week,

1. set up the Airflow environment
2. use Cloud9 to define your DAGs as Python scripts
3. upload the DAG scripts to the S3 bucket
4. open the Airflow UI

DeepLearning.AI

Orchestration, Monitoring, and Automating your Data Pipelines

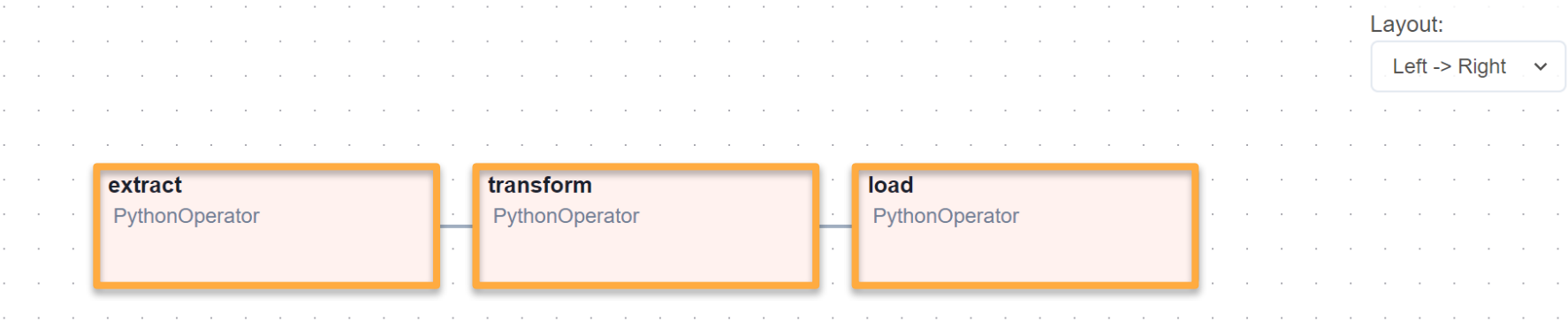
Airflow - The Airflow UI

DeepLearning.AI

Orchestration, Monitoring, and Automating your Data Pipelines

Airflow - Creating a DAG

DAG Example


Layout:
Left -> Right
extract
PythonOperator
transform
PythonOperator
load
PythonOperator

Airflow Operators

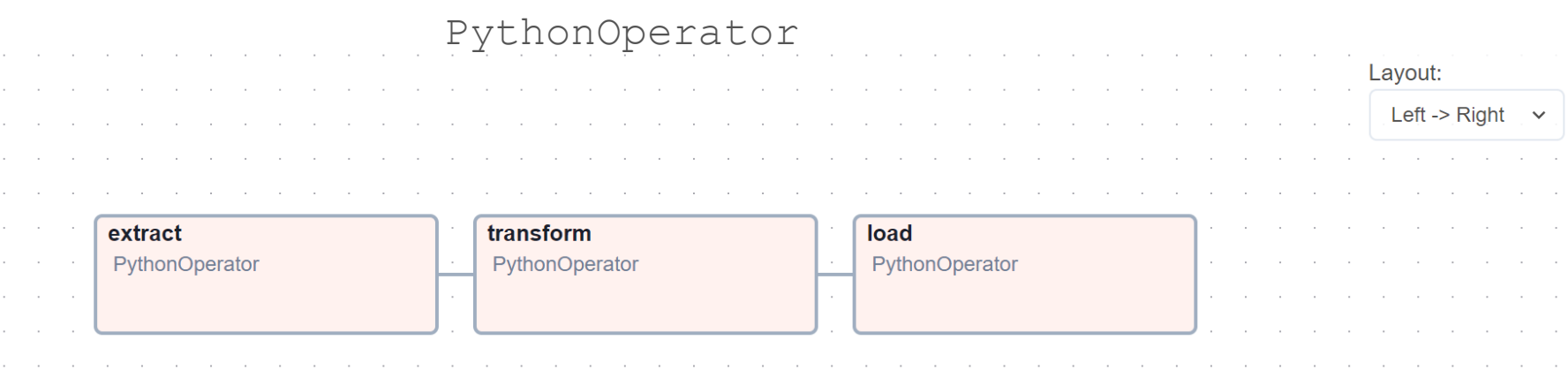
- Operators are Python classes used to encapsulate the logic of the tasks.
- You can import Operators into your code.
- Each task is an instance of an Airflow Operator.

Airflow Operators

Types of Operators:

- `PythonOperator`: execute a python script
- `BashOperator`: execute bash commands
- `EmptyOperator`: organize the DAGs
- `EmailOperator`: send you notification via an email
- `Sensor`: special type of Operator used to make your DAGs event-driven

Airflow Operators

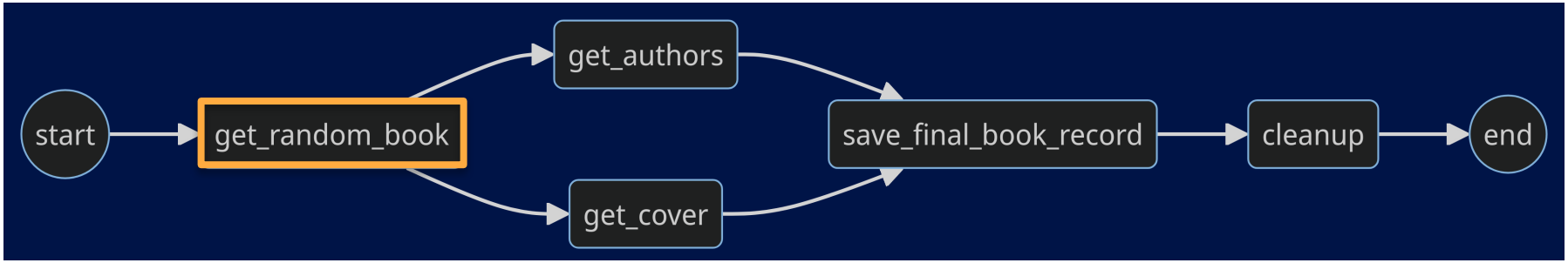


DeepLearning.AI

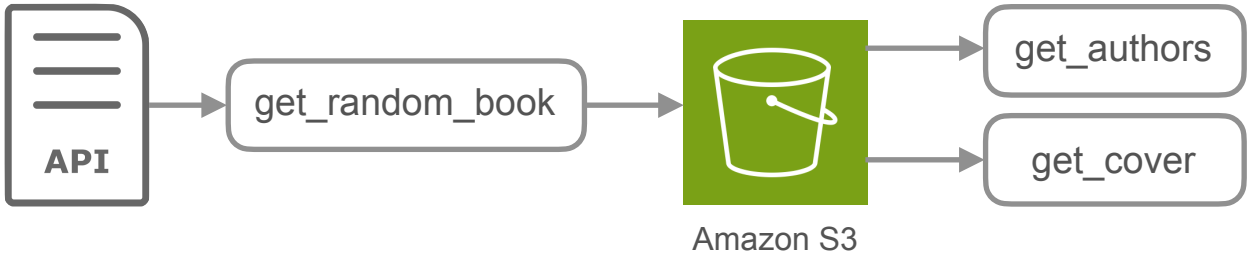
Orchestration, Monitoring, and Automating your Data Pipelines

Airflow - XCom and Variables

Lab 1



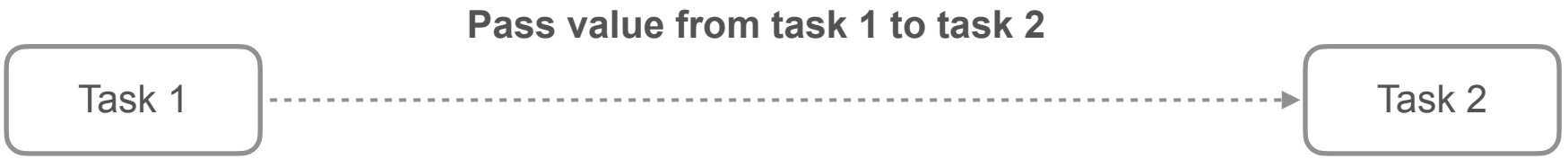
Pass data from one task to another using an intermediate storage



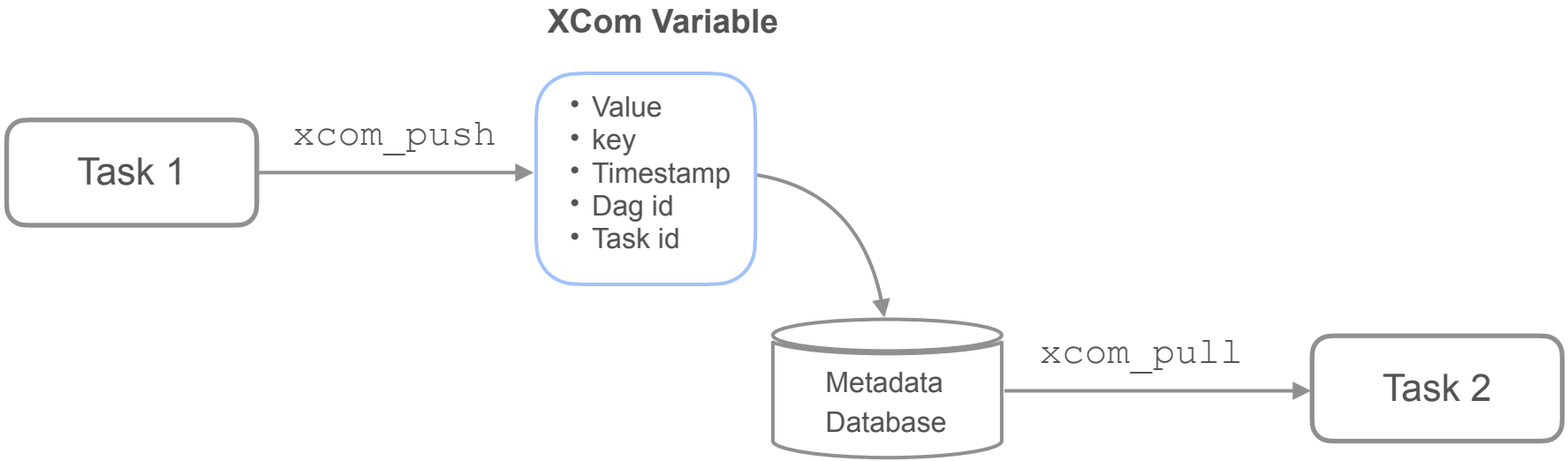
XCom

- **XCom** is short for cross-communication.
- Designed to pass small amounts of data: metadata, dates, single value metrics, simple computations.

XCom



XCom


XCom Variable
Task 1
xcom_push
• Value
• key
• Timestamp
• Dag id
• Task id
Metadata
Database
xcom_pull
Task 2

User-Created Variable

```
def extract_from_api(**context):
    import requests
    response = requests.get("https://jobicy.com/api/v2/remote-jobs",
                            params={"count": 40,
                                    "geo": "usa",
                                    "industry": "engineering",
                                    "tag": "data engineer"}).json()
    count = 0
    for job in response['jobs']:
        if job['jobLevel'] == 'Senior':
            count += 1
    ratio_senior_jobs = count / len(response['jobs'])
    context['ti'].xcom_push(key='ratio_senior_jobs', value=ratio_senior_jobs)
```

User-Created Variable

Instead of including hard-coded values:

- Create variables in the Airflow UI.
- Create environmental variables.

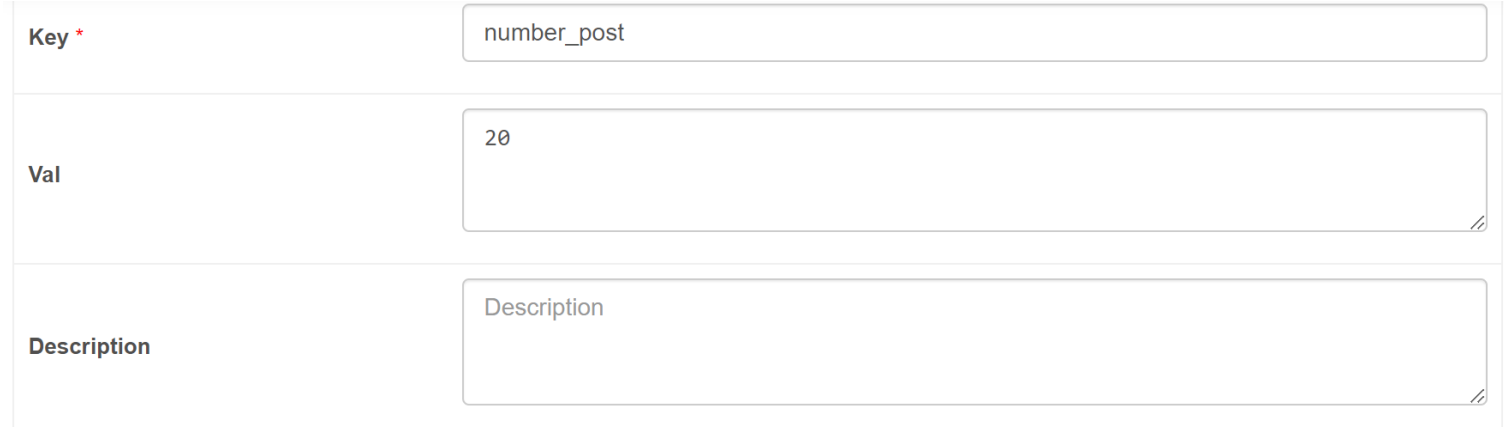
User-Created Variables

```
def extract_from_api(**context):
    import requests
    response = requests.get("https://jobicy.com/api/v2/remote-jobs",
                            params={"count": 40,
                                    "geo": "usa",
                                    "industry": "engineering",
                                    "tag": "data engineer"}).json()
    count = 0
    for job in response['jobs']:
        if job['jobLevel'] == 'Senior':
            count += 1
    ratio_senior_jobs = count / len(response['jobs'])
    context['ti'].xcom_push(key='ratio_senior_jobs', value=ratio_senior_jobs)
```

User-Created Variable

Key *	locations
Val	{"geo":["usa", "canada", "france", "australia"]}
Description	Description

User-Created Variable



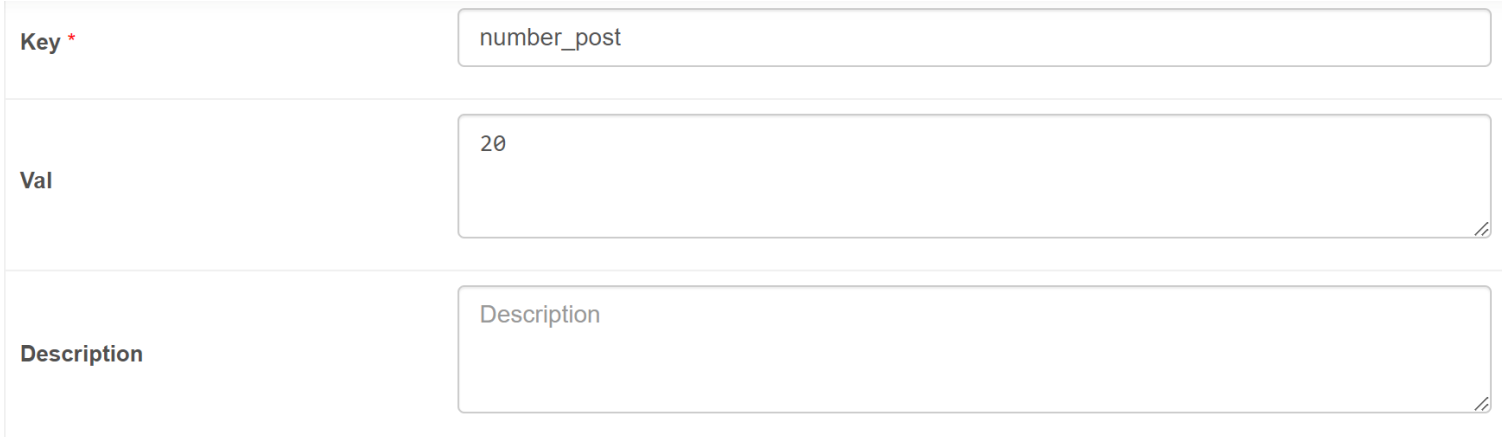
User-Created Variable

```
from airflow.models import Variable
```

```
Variable.get(key='number_post')
```

```
Variable.get(key='locations',deserialize_json=True)['geo']
```

User-Created Variable



DeepLearning.AI

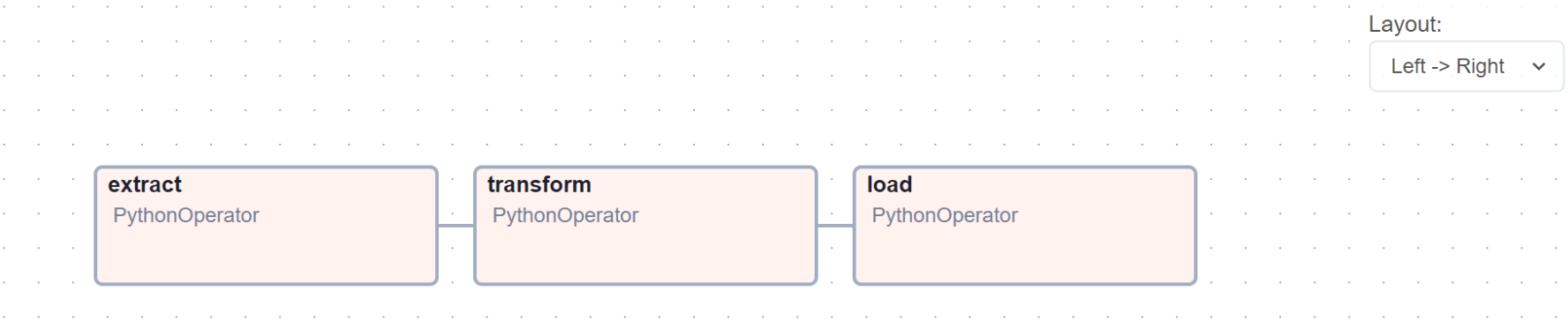
Orchestration, Monitoring, and Automating your Data Pipelines

Airflow - TaskFlow API

TaskFlow API

Traditional Paradigm	TaskFlow API
• Instantiate a DAG object to define the DAG • Use PythonOperator to define tasks	• DAGs are easier to write and more concise • Use decorators to create the DAG and its tasks

TaskFlow API



DAG Definition

Traditional Approach

```python
from airflow import DAG
from datetime import datetime

# context manager
with DAG(
    dag_id = "my_first_dag",
    description = "ETL pipeline",
    tags = ["data_engineering_team"],
    schedule = "@daily",
    start_date = datetime(2024, 12, 1),
    catchup = False):
        # define tasks here

        # define dependencies here
```

TaskFlow API

```python
from datetime import datetime
from airflow.decorators import dag, task


@dag(
    description = "ETL pipeline",
    tags = ["data_engineering_team"],
    schedule = "@daily",
    start_date = datetime(2024, 12, 1),
    catchup = False)
def my_first_dag():
    # define tasks here

    # define dependencies here

my_first_dag()
```

DAG Definition

Traditional Approach

```python
from airflow import DAG
from datetime import datetime

# context manager
with DAG(
    dag_id = "my_first_dag",
    description = "ETL pipeline",
    tags = ["data_engineering_team"],
    schedule = "@daily",
    start_date = datetime(2024, 12, 1),
    catchup = False):
        # define tasks here

        # define dependencies here
```

TaskFlow API

```python
from datetime import datetime
from airflow.decorators import dag, task


@dag(
    description = "ETL pipeline",
    tags = ["data_engineering_team"],
    schedule = "@daily",
    start_date = datetime(2024, 12, 1),
    catchup = False)
def my_first_dag():
    # define tasks here

    # define dependencies here

my_first_dag()
```

Implicitly call the DAG constructor

dag_id

Task Definition

Traditional Approach

Keep track:

- the task_id,
- the name of the Python function,
- the name of the task variable.

```python
from airflow import DAG
from datetime import datetime
from airflow.operators.python import PythonOperator

def extract_data():
    #code for extracting data
    print("Done with the extraction task")

def transform_data():
    #code for transforming data
    print("Done with the transformation task")

def load_data():
    #code for loading data
    print("Done with the loading task")

# context manager
with DAG(
    dag_id = "my_first_dag",
    description = "ETL pipeline",
    tags = ["data_engineering_team"],
    schedule = "@daily",
    start_date = datetime(2024, 12, 1),
    catchup = False):
        # define tasks here
        task_1 = PythonOperator(task_id="extract", python_callable=extract_data)
        task_2 = PythonOperator(task_id="transform", python_callable=transform_data)
        task_3 = PythonOperator(task_id="load", python_callable=load_data)

        # define dependencies here
        task_1 >> task_2 >> task_3
```

DAG Definition

TaskFlow API

- Keep track of fewer names
- Use the @task decorator

```python
from datetime import datetime
from airflow.decorators import dag, task


@dag(
    description = "ETL pipeline",
    tags = ["data_engineering_team"],
    schedule = "@daily",
    start_date = datetime(2024, 12, 1),
    catchup = False)
def my_first_dag():
    # define tasks here
    @task
    def extract_data():
        # code for extracting data
        print("Done with the extraction task")

    @task
    def transform_data():
        # code for transforming data
        print("Done with the transformation task")

    @task
    def load_data():
        #code for loading data
        print("Done with the loading task")

    # define dependencies here


my_first_dag()
```

Implicitly call PythonOperator

task_id

DAG Dependencies

TaskFlow API

Use the bit-shift operator: >>

```python
from datetime import datetime
from airflow.decorators import dag, task


@dag(
    description = "ETL pipeline",
    tags = ["data_engineering_team"],
    schedule = "@daily",
    start_date = datetime(2024, 12, 1),
    catchup = False)
def my_first_dag():
    # define tasks here
    @task
    def extract_data():
        # code for extracting data
        print("Done with the extraction task")

    @task
    def transform_data():
        # code for transforming data
        print("Done with the transformation task")

    @task
    def load_data():
        #code for loading data
        print("Done with the loading task")

    # define dependencies here
    extract_data() >> transform_data() >> load_data()

my_first_dag()
```

TaskFlow API

Traditional Approach

```python
from airflow import DAG
from datetime import datetime
from airflow.operators.python import PythonOperator

def extract_data():
    #code for extracting data
    print("Done with the extraction task")

def transform_data():
    #code for transforming data
    print("Done with the transformation task")

def load_data():
    #code for loading data
    print("Done with the loading task")

# context manager
with DAG(
    dag_id = "my_first_dag",
    description = "ETL pipeline",
    tags = ["data_engineering_team"],
    schedule = "@daily",
    start_date = datetime(2024, 12, 1),
    catchup = False):
        # define tasks here
        task_1 = PythonOperator(task_id="extract", python_callable=extract_data)
        task_2 = PythonOperator(task_id="transform", python_callable=transform_data)
        task_3 = PythonOperator(task_id="load", python_callable=load_data)

        # define dependencies here
        task_1 >> task_2 >> task_3
```

TaskFlow API

```python
from datetime import datetime
from airflow.decorators import dag, task


@dag(
    description = "ETL pipeline",
    tags = ["data_engineering_team"],
    schedule = "@daily",
    start_date = datetime(2024, 12, 1),
    catchup = False)
def my_first_dag():
    # define tasks here
    @task
    def extract_data():
        # code for extracting data
        print("Done with the extraction task")

    @task
    def transform_data():
        # code for transforming data
        print("Done with the transformation task")

    @task
    def load_data():
        #code for loading data
        print("Done with the loading task")

    # define dependencies here
    extract_data() >> transform_data() >> load_data()

my_first_dag()
```

XCom

Traditional Approach

```
def extract_from_api(**context):
    #code that connect API
    ratio_senior_jobs = #code
    context['ti'].xcom_push(key='ratio_senior_jobs', value=ratio_senior_jobs)

def print_data(**context):
    print(context['ti'].xcom_pull(key = 'ratio_senior_jobs', task_ids='extract_from_API'))
```

XCom

TaskFlow API

```python
from datetime import datetime
from airflow.decorators import dag, task
from airflow.models import Variable

@dag(
    start_date=datetime(2024, 3, 13),
    description="First DAG",
    tags=["data_engineering_team"],
    schedule='@daily',
    catchup=False)
def example_xcom_taskapi():
    # define tasks here
    @task
    def extract_from_api():
        #code that connect API
        ratio_senior_jobs = #code
        return ratio_senior_jobs

    @task
    def print_data(geo_ratios: dict):
        print(geo_ratios)

    # define dependencies here


example_xcom_taskapi()
```

XCom

TaskFlow API

print_data(extract_from_api())

```python
from datetime import datetime
from airflow.decorators import dag, task
from airflow.models import Variable

@dag(
    start_date=datetime(2024, 3, 13),
    description="First DAG",
    tags=["data_engineering_team"],
    schedule='@daily',
    catchup=False)
def example_xcom_taskapi():
    # define tasks here
    @task
    def extract_from_api():
        #code that connect API
        ratio_senior_jobs = #code
        return ratio_senior_jobs

    @task
    def print_data(geo_ratios: dict):
        print(geo_ratios)

    # define dependencies here
    data = extract_from_api()
    print_data(data)


example_xcom_taskapi()
```

XCom

TaskFlow API

```
# define tasks here
@task
def extract_from_api():
    #code that connect API
    ratio_senior_jobs = #code
    return ratio_senior_jobs

@task
def print_data(**context):
    print(context['ti'].xcom_pull(task_ids='extract_from_api'))

# define dependencies here
extract_from_api() >> print_data()
```

DeepLearning.AI

Orchestration, Monitoring, and Automating your Data Pipelines

Orchestration on AWS

More control

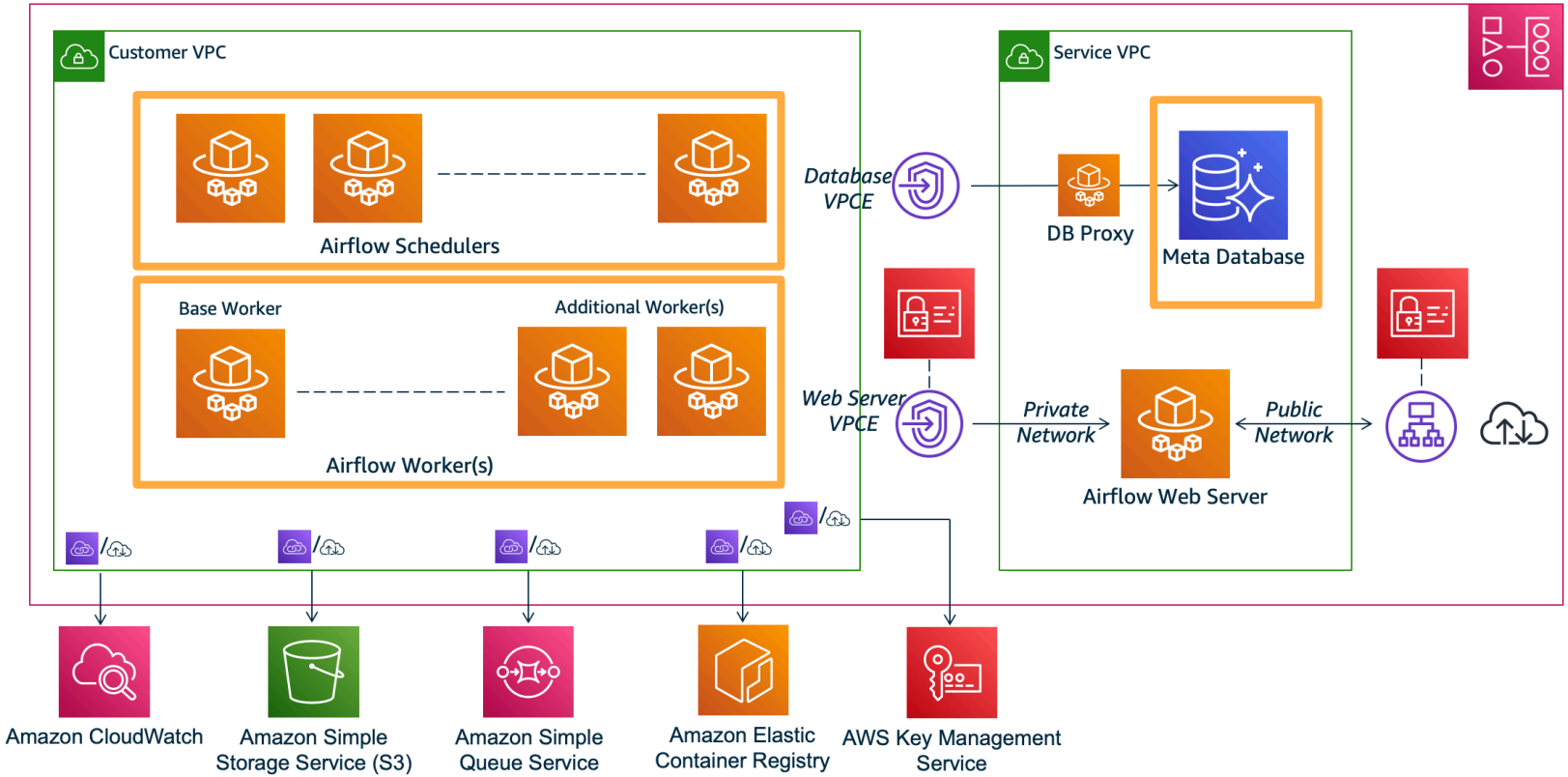
- Run the open-source version of Airflow on an Amazon EC2 instance or in a container.
- You have full control over the configuration and scaling of your Airflow environment.
- You need to manage all of the underlying infrastructure and integrations yourself.

Amazon Managed Workflows
for Apache Airflow
(Amazon MWAA)

More convenience

- MWAA runs Airflow for you and handles tasks like the provisioning and scaling of the underlying infrastructure.

Amazon MWAA Architecture



Apache Airflow

More control

- Run the open-source version of Airflow on an Amazon EC2 instance or in a container.
- You have full control over the configuration and scaling of your Airflow environment.
- You need to manage all of the underlying infrastructure and integrations yourself.

Amazon Managed Workflows for Apache Airflow (Amazon MWAA)

More convenience

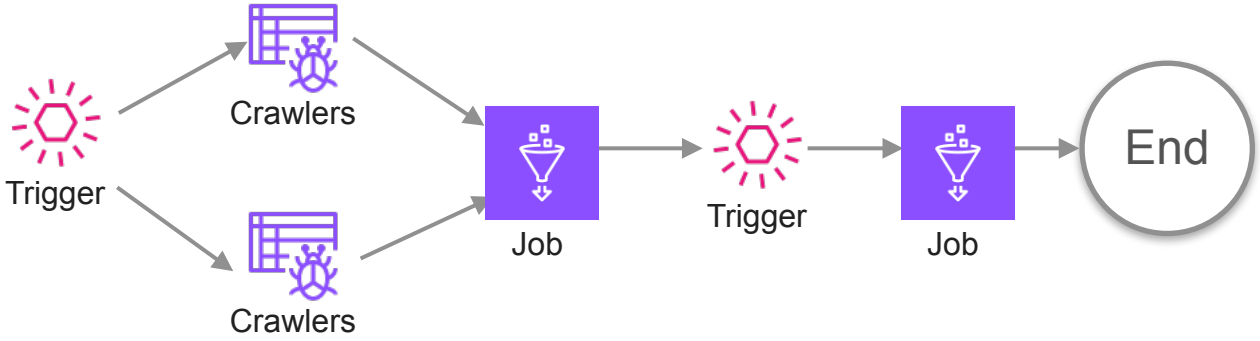
- MWAA runs Airflow for you and handles tasks like the provisioning and scaling of the underlying infrastructure.
- It integrates with other AWS services.

Amazon CloudWatch

Amazon Simple Storage Service (Amazon S3)

AWS Key Management Service (AWS KMS)

AWS Glue Workflows: Allows you to create, run, and monitor complex ETL workflow



Trigger types:

- A schedule
- On-demand
- Event from Amazon EventBridge

AWS Step Functions

Allow you to orchestrate multiple AWS services and states into workflows called state machines

Lambda function Glue job ECS Task

- States can be tasks which do work
- States can make decisions based on their input, perform actions from those inputs, and pass output to other states

- Offers a lot of flexibility
- Good for complex workflows.

AWS Step Functions

- Provide a serverless option with extensive native AWS service integration
- Ideal for AWS-centric workflows

Glue Workflows

- Specifically designed for ETL processes
- Allows you to orchestrate Glue jobs, crawlers, and triggers in a serverless environment

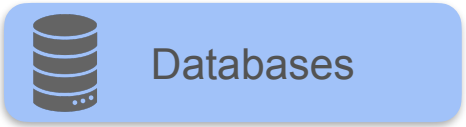
- What are your requirements?
- What are you trying to optimize for?

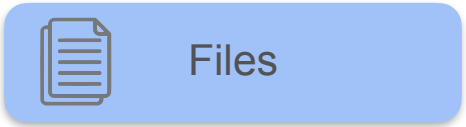
DeepLearning.AI

Source Systems, Data Ingestion, and Pipelines

Course Summary

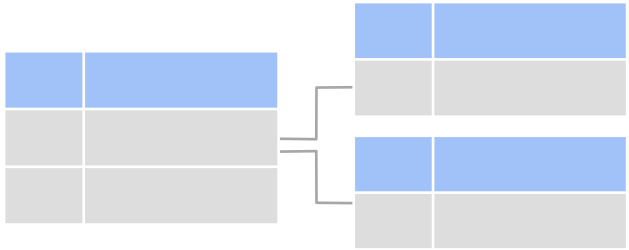
Week 1





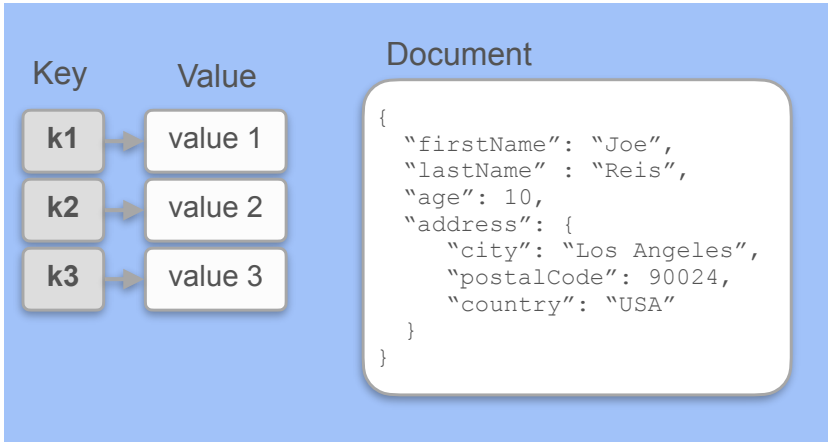
Week 1

Relational databases

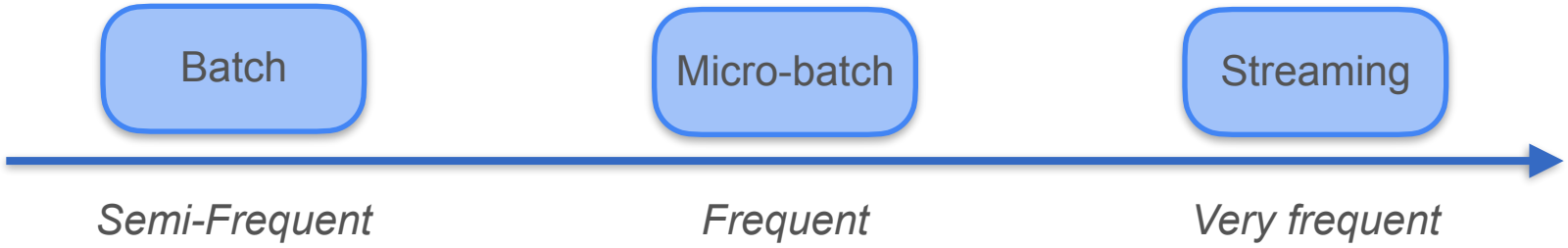


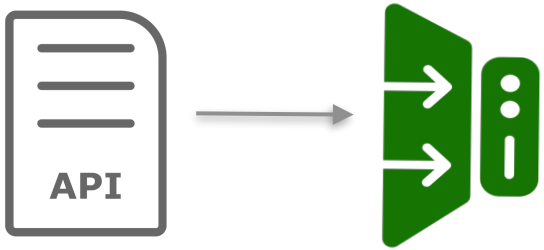
Object Storage

NoSQL databases



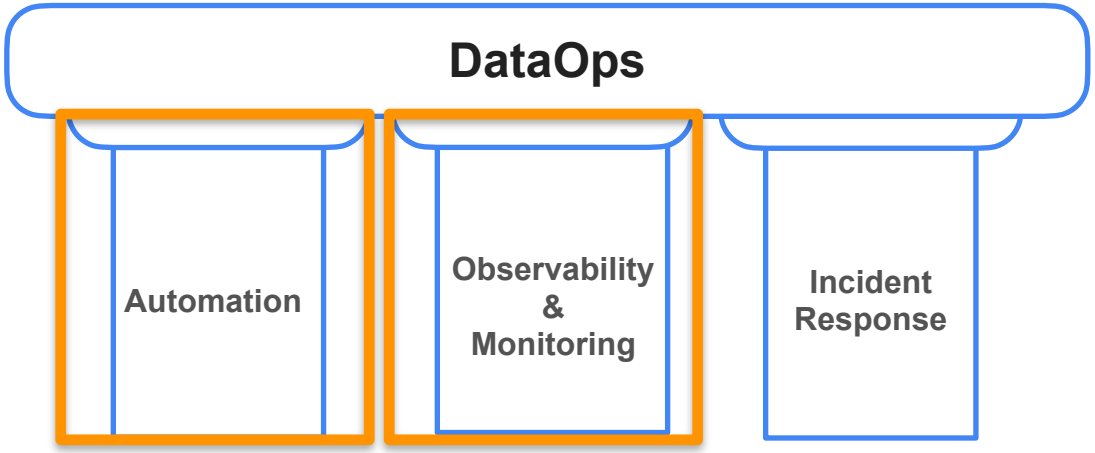
Week 2





Amazon Kinesis
Data Streams

Week 3

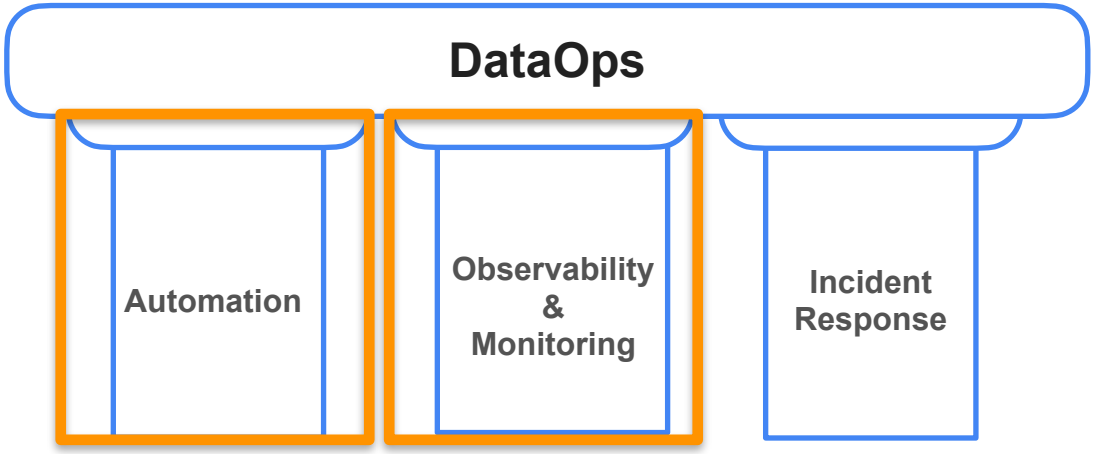


HashiCorp Terraform

great expectations

Amazon CloudWatch

Week 4



Apache Airflow