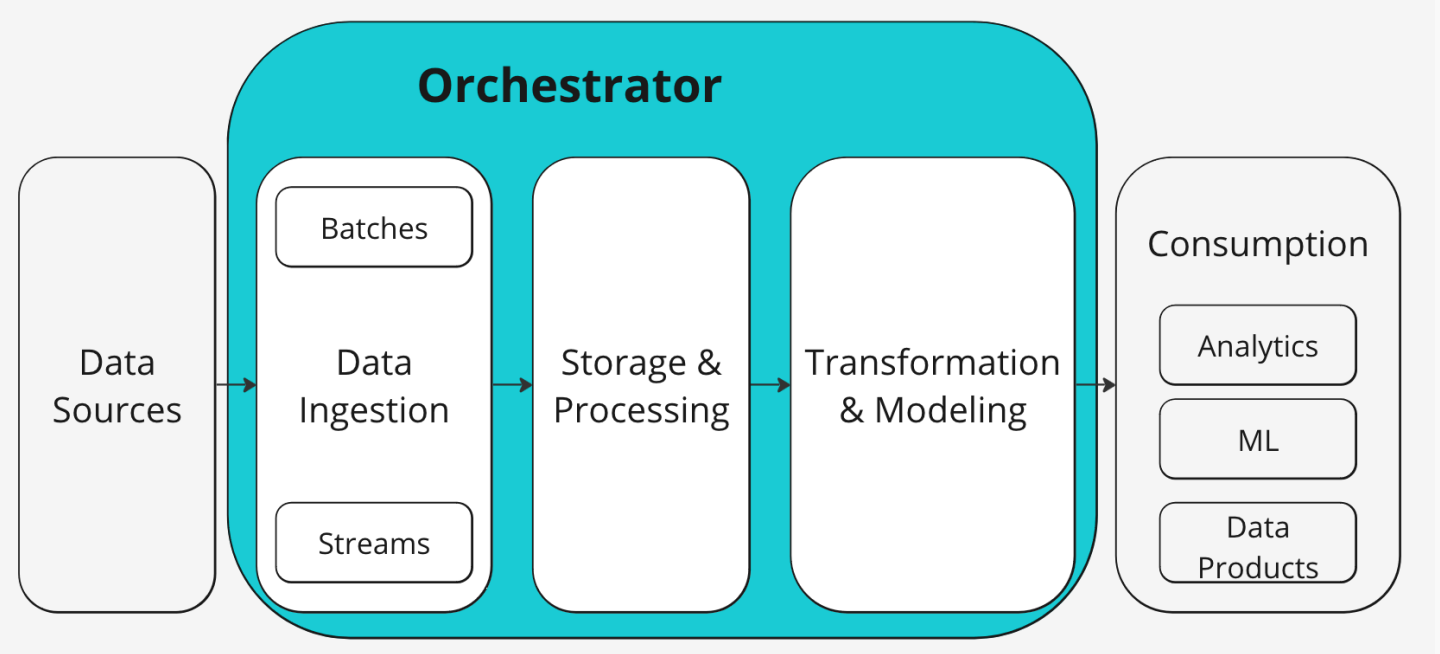
Esoon Ko
IT Högskolan

Airflow and Intro to Orchestration

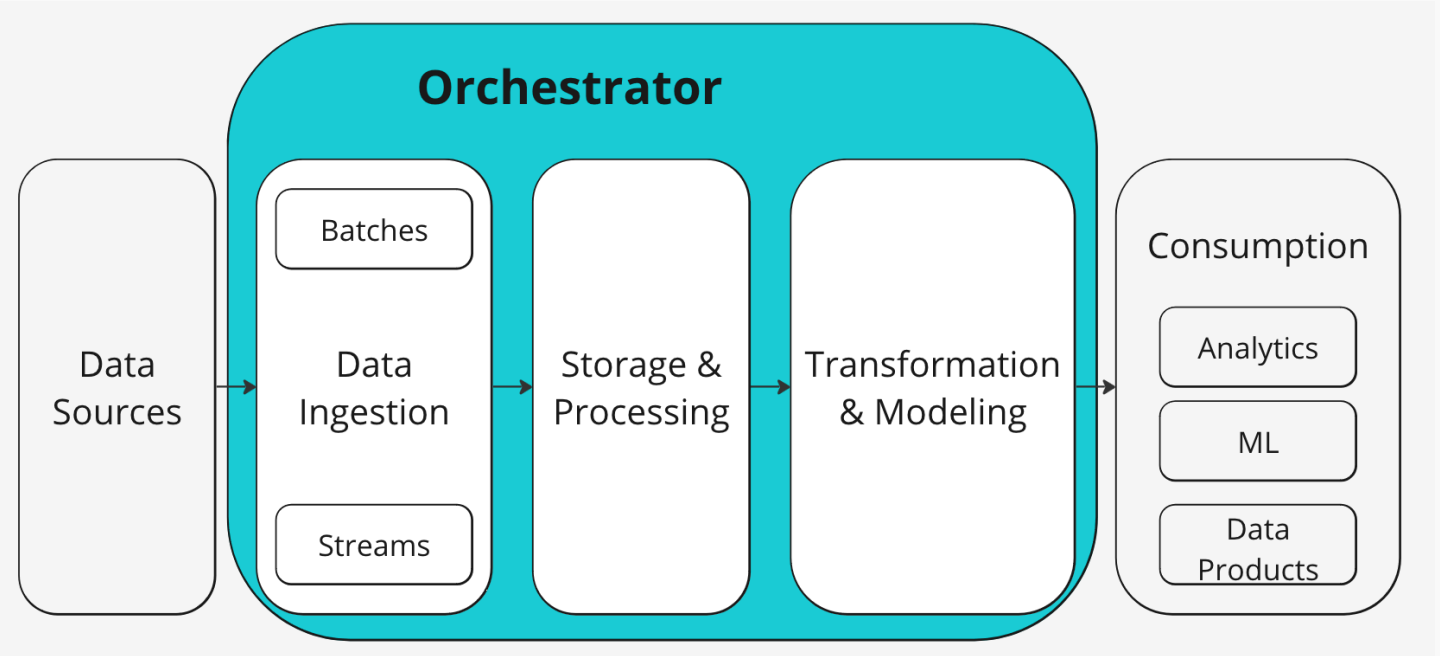
Orchestration

Automated process of managing, coordinating, and optimizing data pipelines and workflows. It ensures that data flows efficiently and accurately from one system to another, maintaining quality and integrity.



Why **Orchestration?**

- **Efficiency**: Reduces manual intervention and automates repetitive tasks.
- **Scalability**: Handles large volumes of data and complex workflows.
- **Reliability**: Ensures consistent and error-free data processing.
- **Agility**: Quickly adapt to changes in data sources, processing needs, or business requirements.



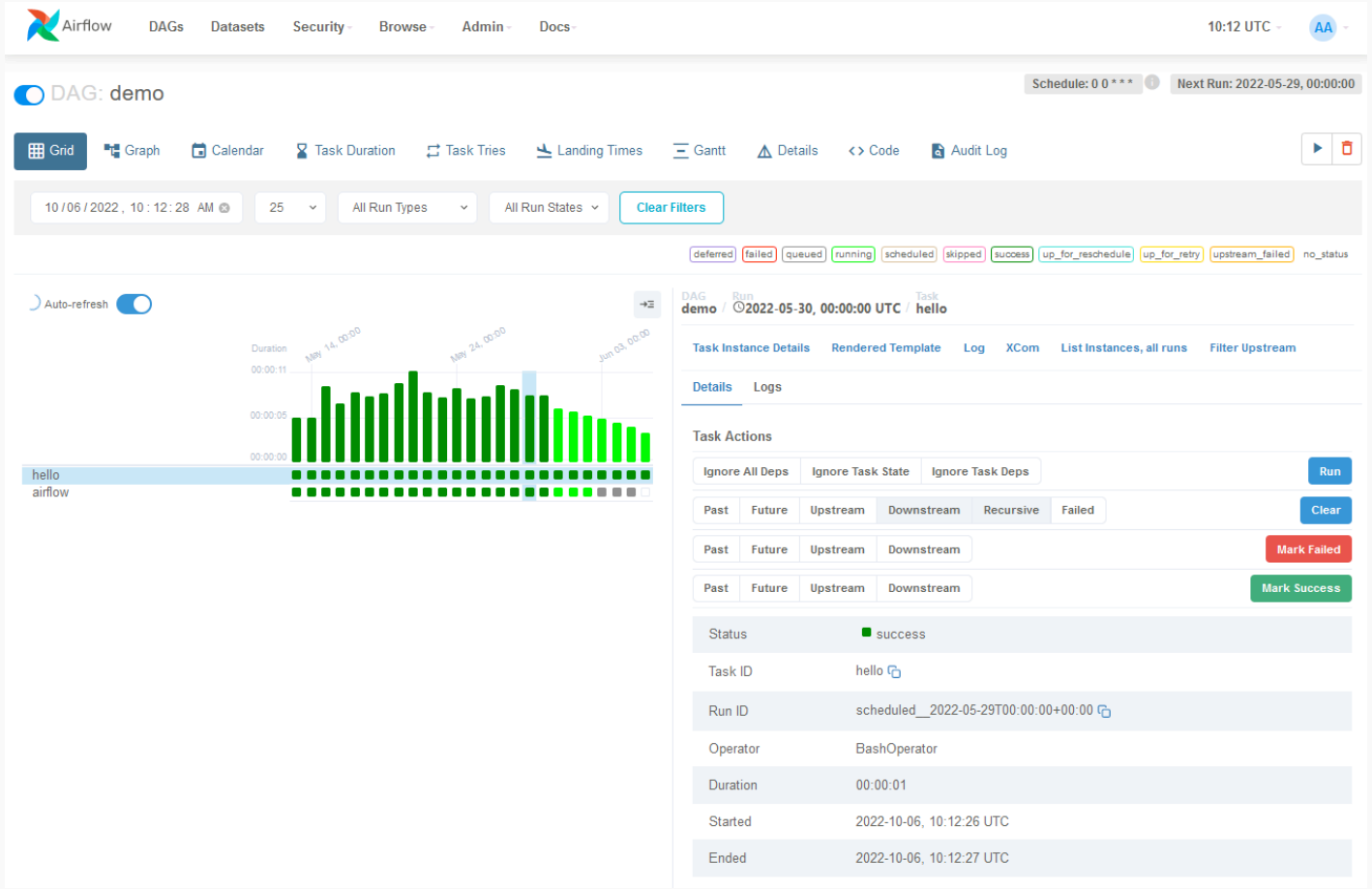
Apache Airflow

Apache Airflow is an open-source platform used to programmatically author, schedule, and monitor workflows.

It allows you to define workflows as code, making it easier to maintain and reproduce complex data pipelines.

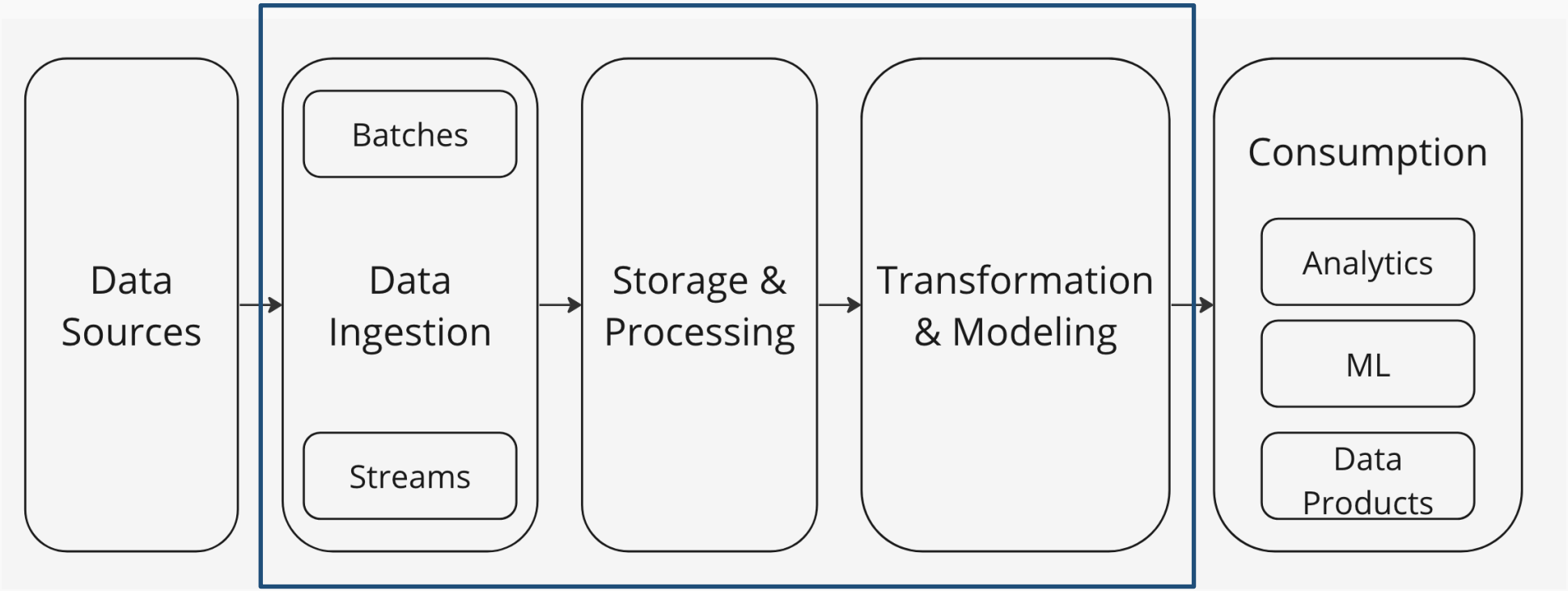
It is written in Python and uses Directed Acyclic Graphs (DAGs) to represent workflows.

Historical fact: Originally made by Airbnb to manage their workflows and was open source, becoming an Apache incubator project.



What is a **workflow??**

A sequence of tasks.
The data lifecycle we drew up can be seen as containing a workflow



So why **Apache Airflow?** (For this course)

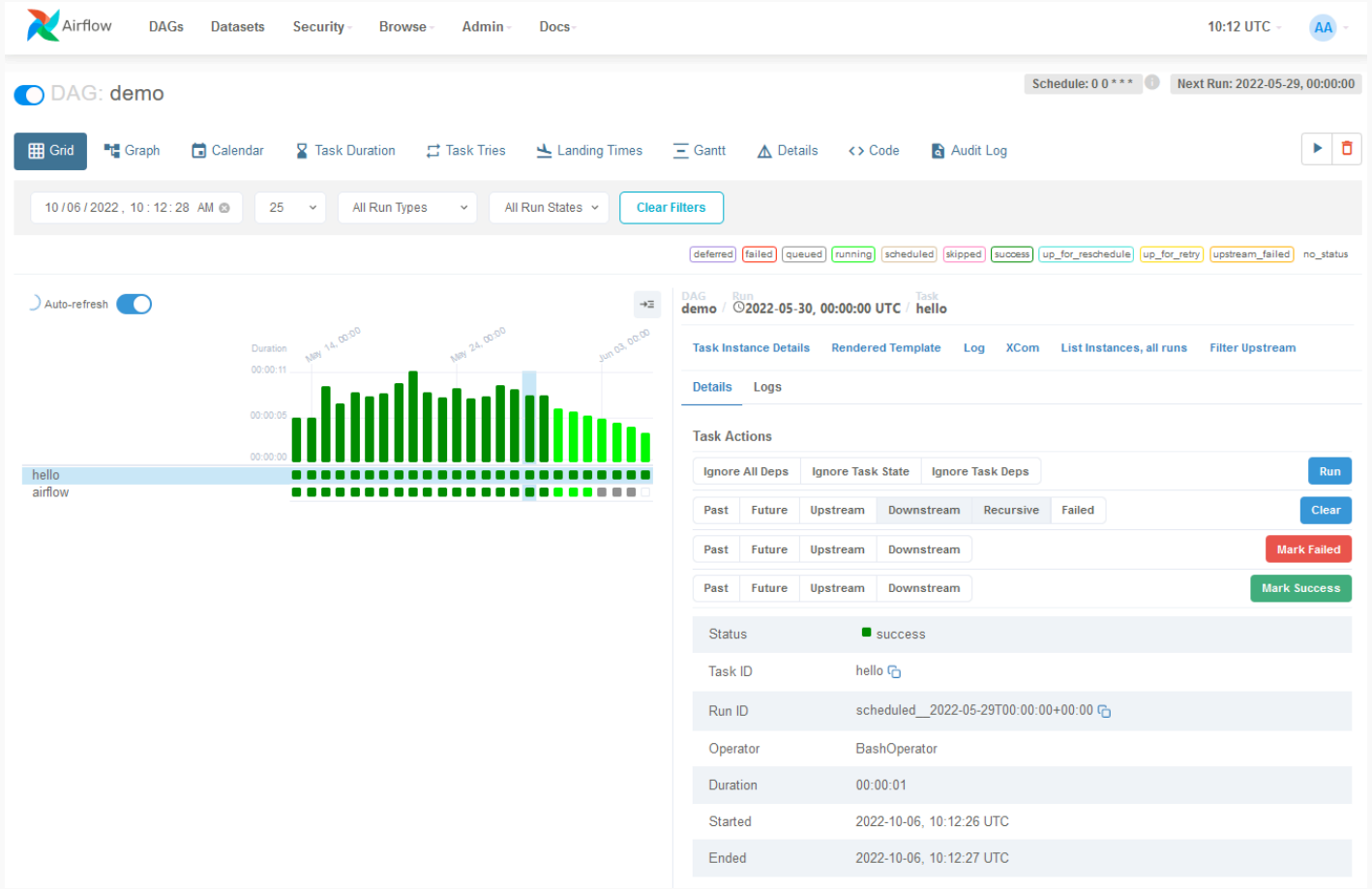
Apache Airflow is not the most scalable and most that use it build implementation on it to avoid writing DAGs directly.

So why are we choosing it for this class?

Benefits:

- Flexibility in defining workflows as code
- Easy to understand and provides a full suite for workflow management

Airflow is an easy tool that helps understans key concepts of data engineering



Lets create Airflow - via docker

Instructions:https://airflow.apache.org/docs/apache-airflow/stable/howto/docker-compose/index.html

In a new directory, run **curl -LfO 'https://airflow.apache.org/docs/apache-airflow/2.9.1/docker-compose.yaml'**

Open the yaml and make the following changes:
- AIRFLOW__CORE__EXECUTOR to localExecutor
- DELETE AIRFLOW__CELERY__RESULT_BACKEND
- DELETE AIRFLOW__CELERY__BROKERW_URL
- DELETE redis
- DELETE celery_worker
- DELETE flower

Run the commands:
mkdir -p ./dags ./logs ./plugins ./config
echo -e "AIRFLOW_UID=$(id -u)" > .env (ONLY IF YOU USE LINUX)

docker compose up airflow-init

docker compose up -d

Go to http://localhost:8080 - login with username: **airflow** and **password: airflow**

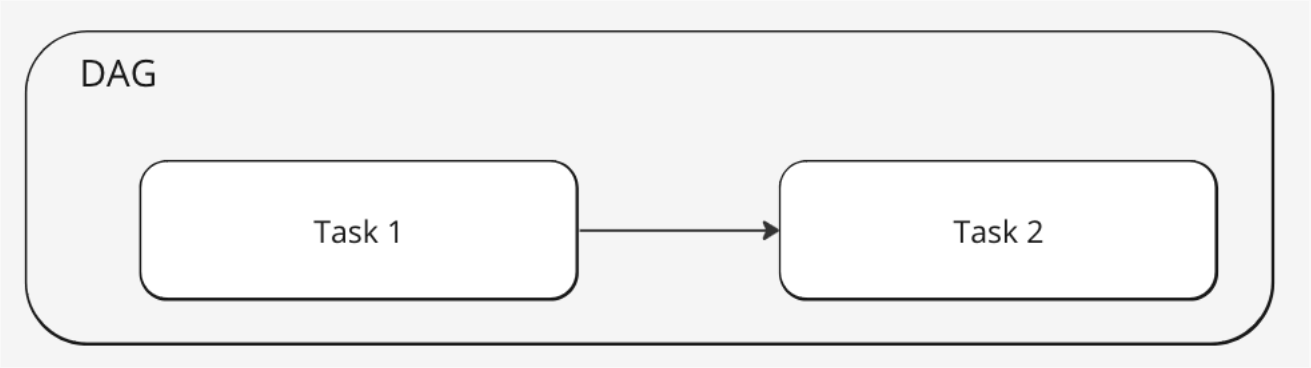
DAGs

Workflows in Airflow are called **DAG**s

DAG stands for **Directed Acyclic Graphs**

DAGs consist of **tasks** and **dependencies** between them. **DAG**s ensure that workflows are executed in a specific order without cycles

Directed - Each DAG points somewhere
Acyclic - DAG only points one direction, not backwards
Graph - the tasks map out in a network of tasks



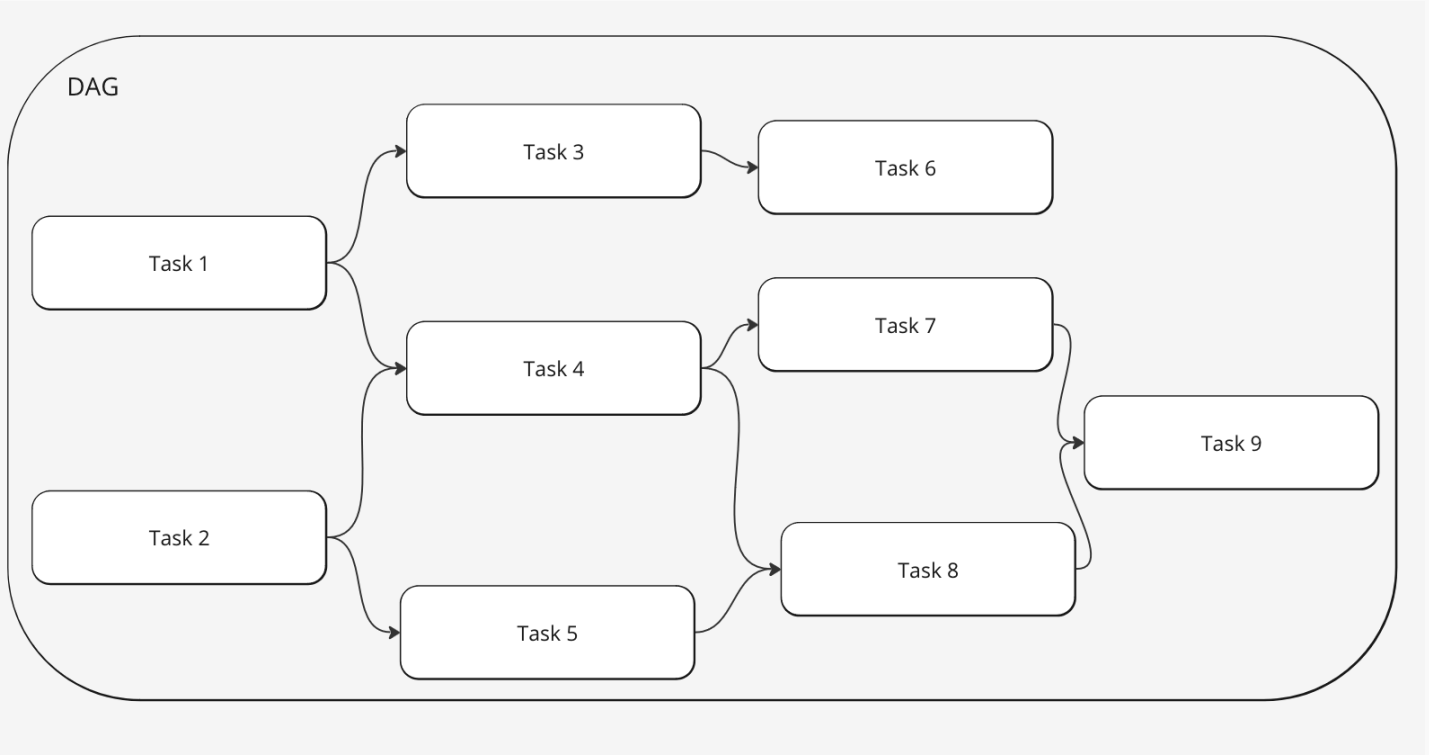
DAGs

Workflows in Airflow are called **DAG**s

DAG stands for **Directed Acyclic Graphs**

DAGs consist of **tasks** and **dependencies** between them. **DAG**s ensure that workflows are executed in a specific order without cycles

Directed - Each DAG points somewhere
Acyclic - DAG only points one direction, not backwards
Graph - the tasks map out in a network of tasks

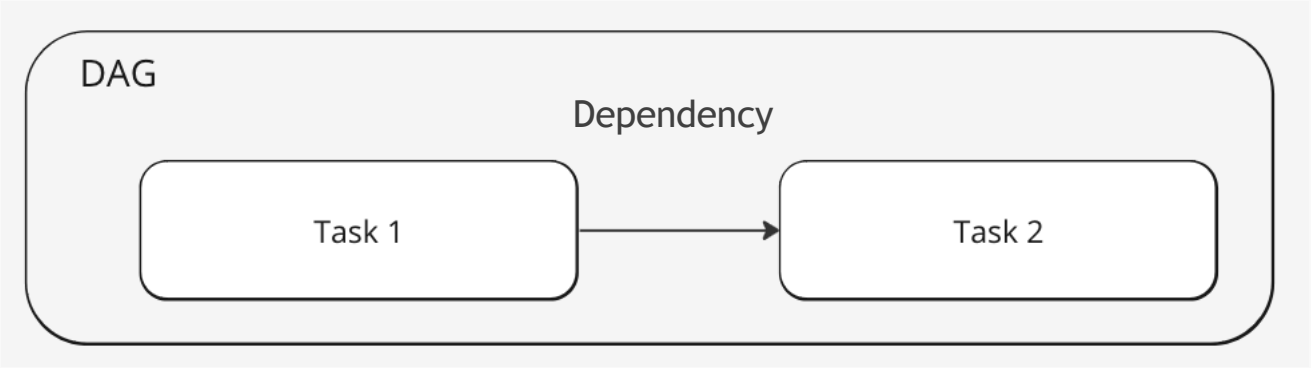


Tasks, Dependencies and Operators

Tasks represent units of work in a **DAG.** It represents a callable or an executable action (e.g., running a Python function, executing a shell command).

Dependencies define the order in which **tasks** should be executed. Task 2 is the **downstream** of task 1. Task 1 is the **upstream** of task 2.

Operators are the building blocks of DAGs in Airflow. They define a single task, which can be anything from running a shell command to executing Python code or interacting with external systems. If we say **DAGs** describe how to run a workflow, then **operators** determines what gets done in a task



Lets create a basic **DAG**

We will create a DAG in BASH with three tasks
(“start”, “bash_task_1”, “end”)
using the BashOperator

```python
from airflow import DAG
from airflow.operators.bash_operator import BashOperator
from datetime import datetime, timedelta

def print_hello():
    print("Hello, Airflow!")

default_args = {
    'owner': 'airflow',
    'start_date': datetime(2024, 6, 1),
    'retries': 5,
    'retry_delay': timedelta(minutes=5)
}

with DAG(
    'bash_operator_dag',
    default_args=default_args,
    schedule_interval='@daily'
    ) as dag:
        start = BashOperator(
            task_id = 'start',
            bash_command = 'echo start'
        )
        bash_task_1 = BashOperator(
              task_id = 'bash_task_1',
              bash_command='echo hello world'
        )
        end = BashOperator(
            task_id = 'end',
            bash_command = 'echo this is the end'
        )

        start >> bash_task_1 >> end
```

Lets create a basic **DAG**

We will create a DAG in Python with three tasks
(“start”, “python_task_1”, “end”)
using the PythonOperator

```python
from airflow import DAG
from airflow.operators.python_operator import PythonOperator
from airflow.operators.dummy_operator import DummyOperator
from datetime import datetime, timedelta

def start():
    print("Start of DAG")

def python_task_1():
    print("Hello, Airflow!")

def end():
    print("End of DAG")

default_args = {
    'owner': 'airflow',
    'start_date': datetime(2024, 6, 1),
    'retries': 5,
    'retry_delay': timedelta(minutes=5)
}

with DAG(
    'python_operator_dag',
    default_args=default_args,
    schedule_interval='@daily'
    ) as dag:
        start = PythonOperator(task_id='start', python_callable=start)
        python_task_1 = PythonOperator(task_id='python_task_1', python_callable=python_task_1)
        end = PythonOperator(task_id='end', python_callable=end)

        start >> python_task_1 >> end
```

Tasks lifecycle

Task lifecycle involves various states that a task instance can go through from the moment it is scheduled until it completes execution.

- None
- Scheduled
- Queued
- Running
- Success
- Failed
- Up for Retry
- Up for Reschedule
- Skipped
- Deferred
- Shutdown
- Removed
- Upstream Failed

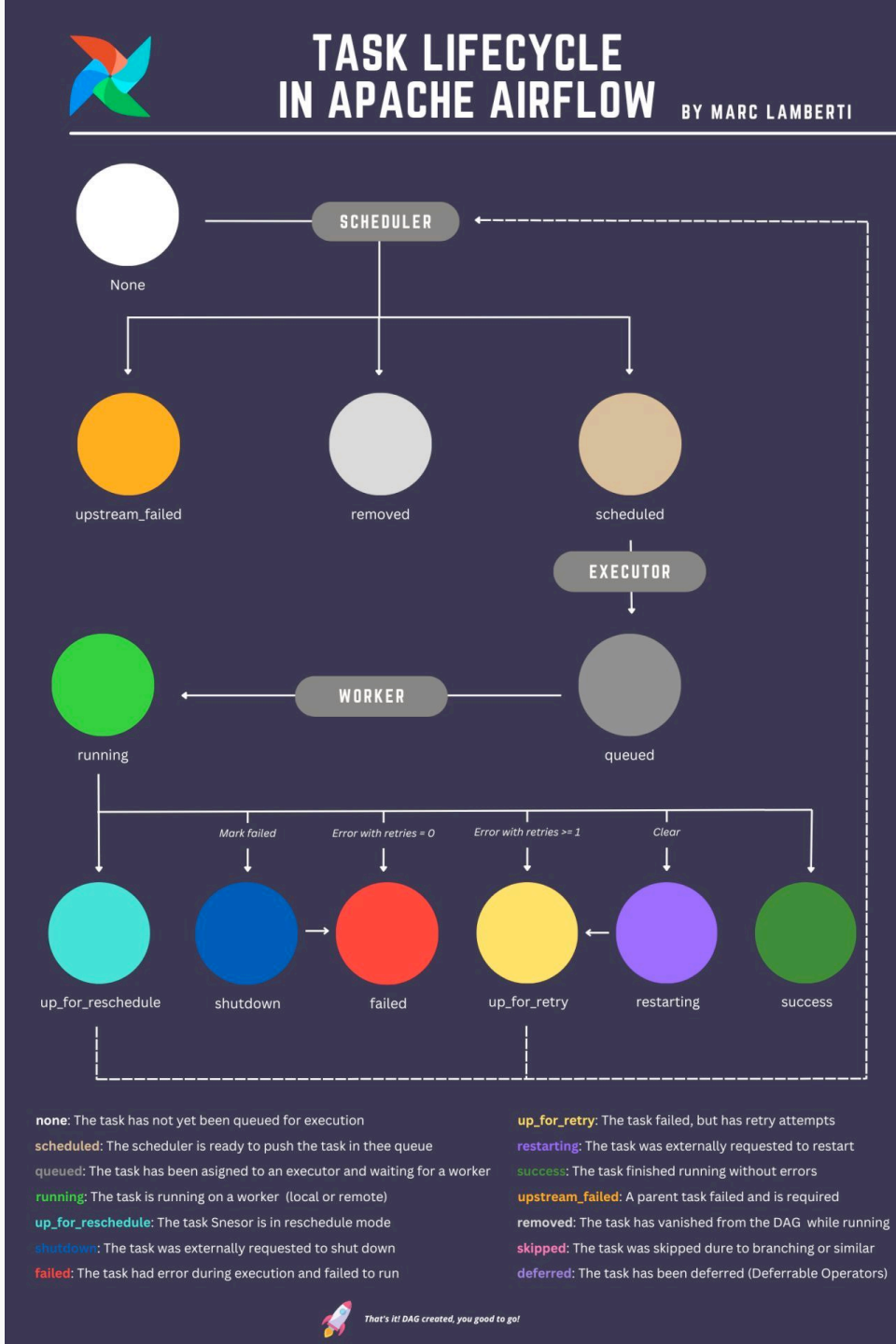


Image by Marc Lamberti

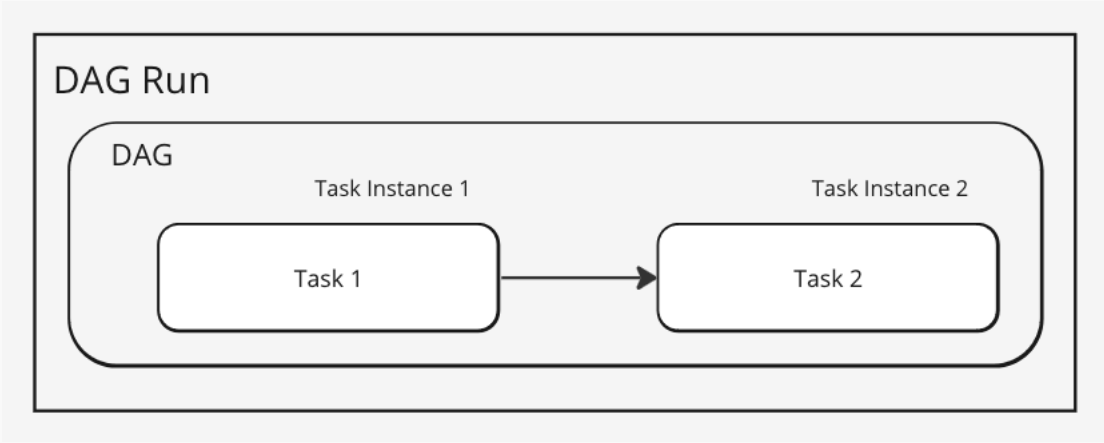
https://www.linkedin.com/posts/marclamberti_dataengineering-dataengineer-airflow-activity-7107743501525639168-TdTR/

Execution Date, Task Instance and DAG Run

Execution Date represents the logical date and time for which a DAG run is scheduled. It is not necessarily the actual time when the tasks run, but rather the point in time the run is supposed to be processing data for.

Task instance is an instance of a task in a DAG for a specific execution date. It represents the state and results of running that task for that particular execution date.

DAG run is an instance of a DAG for a specific execution date. It represents the overall execution of all tasks in the DAG for that particular execution date.



Execution Date: 2024-06-03T02:00:00 @Daily
Should run at this time every day from the execution date

Sets in motion a DAG Run that happens somewhere at or after the execution date, which runs task instances for each task.

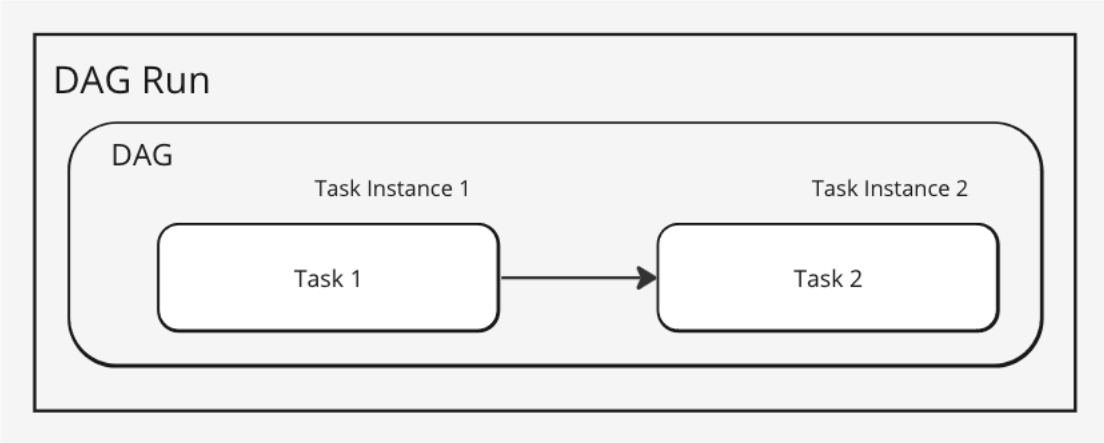
Execution Date, Task Instance and DAG Run

In other words:

Execution Date: Logical point in time for which the DAG and its tasks are meant to run. It helps in defining and organizing scheduled runs.

Task Instance: Specific run of a task within a DAG for a given execution date. It provides details about the execution status of each task.

DAG Run: Overall run of the DAG for a specific execution date, encapsulating the status of all task instances.



Execution Date: 2024-06-03T02:00:00 @Daily

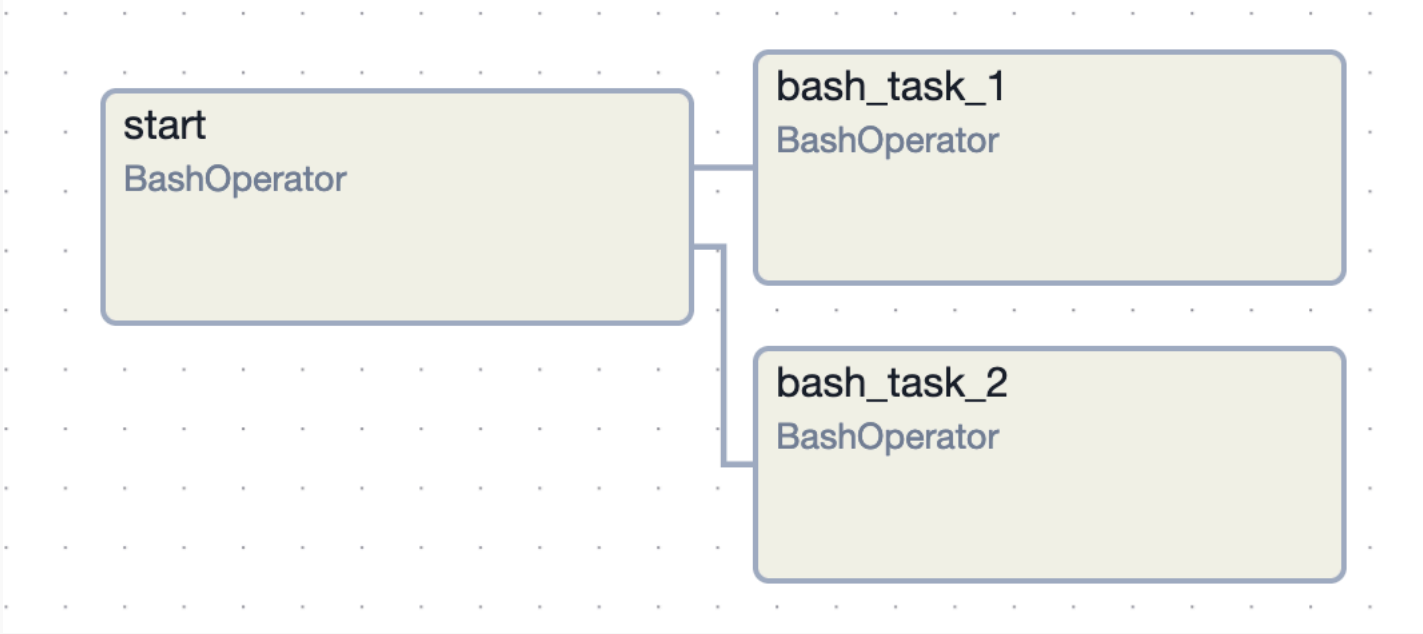
Should run at this time every day from the execution date

Sets in motion a DAG Run that happens somewhere at or after the execution date, which runs task instances for each task.

Playing with **DAG dependencies**

We will create a DAG in Python with four tasks
(“start”, “task_1”, “task_2)
using the any operator

Create two branches so that we get the following graph:



Playing with **DAG dependencies**

We will create a DAG in Python with four tasks
(“start”, “task_1”, “task_2)
using the any operator

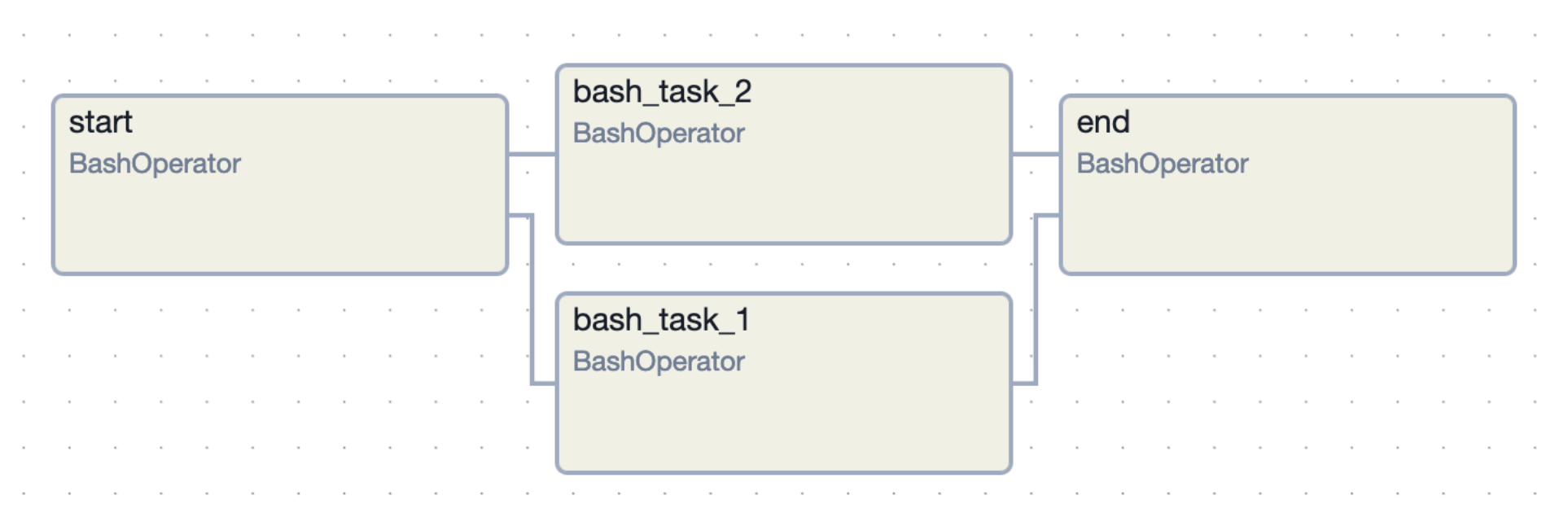
Create two branches so that we get the following graph:

```python
from airflow import DAG
from airflow.operators.bash_operator import BashOperator
from datetime import datetime, timedelta

def print_hello():
    print("Hello, Airflow!")

default_args = {
    'owner': 'airflow',
    'start_date': datetime(2024, 6, 1),
    'retries': 5,
    'retry_delay': timedelta(minutes=5)
}

with DAG(
    'bash_operator_dag',
    default_args=default_args,
    schedule_interval='@daily'
    ) as dag:
        start = BashOperator(
            task_id = 'start',
            bash_command = 'echo start'
        )
        bash_task_1 = BashOperator(
              task_id = 'bash_task_1',
              bash_command='echo hello world'
        )
        bash_task_2 = BashOperator(
              task_id = 'bash_task_2',
              bash_command='echo This is another branch'
        )

        start >> bash_task_1
        start >> bash_task_2
```

Playing with **DAG dependencies**

We will create a DAG in Python with four tasks
(“start”, “task_1”, “task_2”, “end”)
using the any operator

Create two branches so that we get the following graph:



Playing with **DAG dependencies**

We will create a DAG in Python with four tasks
(“start”, “task_1”, “task_2”, “end”)
using the any operator

Create two branches so that we get the following graph:

```python
from airflow import DAG
from airflow.operators.bash_operator import BashOperator
from datetime import datetime, timedelta

def print_hello():
    print("Hello, Airflow!")

default_args = {
    'owner': 'airflow',
    'start_date': datetime(2024, 6, 1),
    'retries': 5,
    'retry_delay': timedelta(minutes=5)
}

with DAG(
    'bash_operator_dag',
    default_args=default_args,
    schedule_interval='@daily'
    ) as dag:
        start = BashOperator(
            task_id = 'start',
            bash_command = 'echo start'
        )
        bash_task_1 = BashOperator(
              task_id = 'bash_task_1',
              bash_command='echo hello world'
        )
        bash_task_2 = BashOperator(
              task_id = 'bash_task_2',
              bash_command='echo This is another branch'
        )
        end = BashOperator(
            task_id = 'end',
            bash_command = 'echo this is the end'
        )

        start >> bash_task_1 >> end
        start >> bash_task_2 >> end
```

Python DAG

We will create a DAG in Python with three tasks
(“start”, “task_1”, “end”)
using the any python operator

We want to have input arguments in python

```python
from airflow import DAG
from airflow.operators.python_operator import PythonOperator
from airflow.operators.dummy_operator import DummyOperator
from datetime import datetime, timedelta

def start():
    print("Start of DAG")

def python_task_1(location, date):
    print(f"Hello, today is {date} and we are at {location}")

def end():
    print("End of DAG")

default_args = {
    'owner': 'airflow',
    'start_date': datetime(2024, 6, 1),
    'retries': 5,
    'retry_delay': timedelta(minutes=5)
}

with DAG(
    'python_operator_dag',
    default_args=default_args,
    schedule_interval='@daily'
    ) as dag:
        start = PythonOperator(task_id='start', python_callable=start)
        python_task_1 = PythonOperator(
            task_id='python_task_1',
            python_callable=python_task_1,
            op_kwargs={'location': 'Stockholm', 'date': datetime.now().date()}
            )
        end = PythonOperator(task_id='end', python_callable=end)

        start >> python_task_1 >> end
```

Exchanging data between tasks

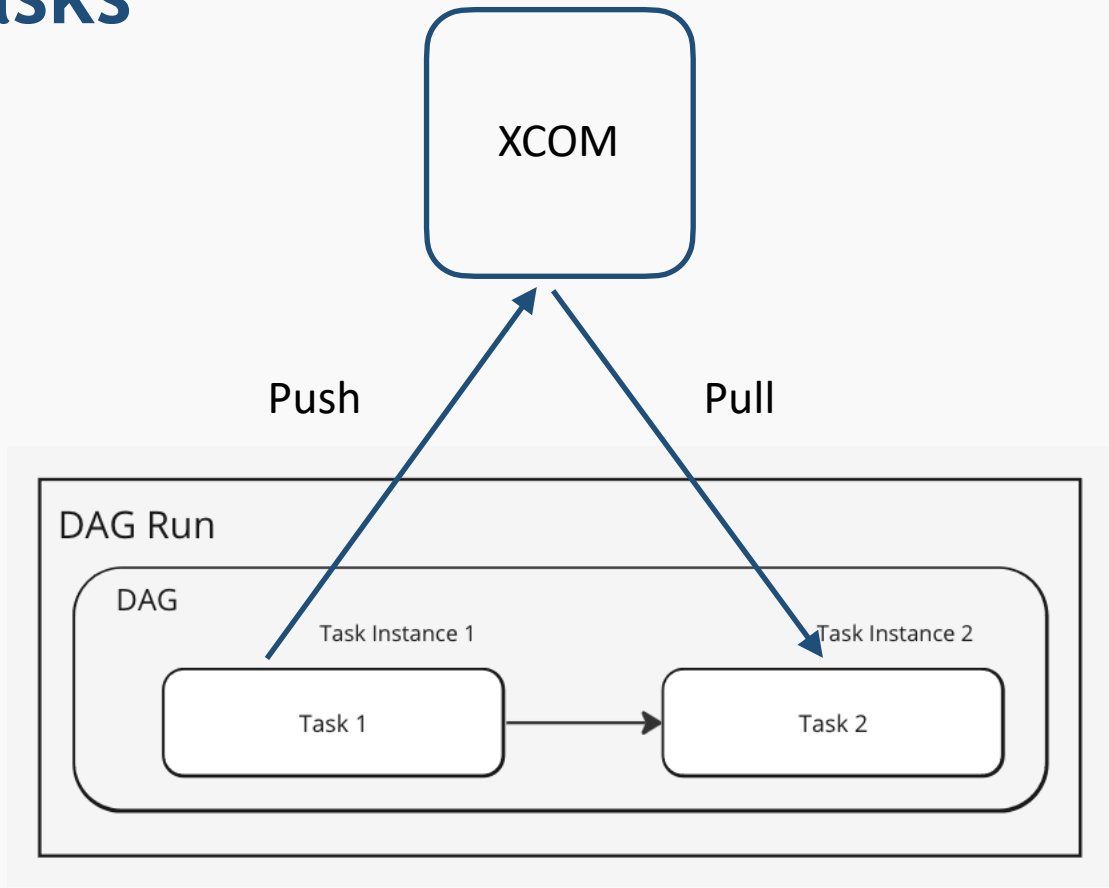
Is it possible to send around data between tasks?

Yes: Through **XCOMS**

XCOMS are a mechanism that allows tasks to exchange small amounts of data, enabling inter-task communication.

Every functions **return value** will be **pushed** by default.
Can be pushed manually.
Pull must be done manually.

XCOM max size is 48kb.



Lets use **XCOMS**

We will create two DAGs in Python: One with one task each.

The first DAG will have a “push_task”, the other DAG will have a “pull_task

```python
from airflow import DAG
from airflow.operators.python_operator import PythonOperator
from datetime import datetime, timedelta

def push_function():
    return 'This is the value being pushed'

default_args = {
    'owner': 'airflow',
    'start_date': datetime(2024, 6, 1),
    'retries': 5,
    'retry_delay': timedelta(minutes=5)
}

with DAG(
    'python_operator_xcom_push_dag',
    default_args=default_args,
    schedule_interval='@daily'
    ) as dag:
        push_task = PythonOperator(
            task_id='push_task',
            python_callable=push_function,
            dag=dag,
        )

        push_task
```

```python
from airflow import DAG
from airflow.operators.python_operator import PythonOperator
from datetime import datetime, timedelta
def pull_function(ti):
    value = ti.xcom_pull(task_ids='push_task')
    print(f"Value from XCom: {value}")

default_args = {
    'owner': 'airflow',
    'start_date': datetime(2024, 6, 1),
    'retries': 5,
    'retry_delay': timedelta(minutes=5)
}

with DAG(
    'python_operator_xcom_pull_dag',
    default_args=default_args,
    schedule_interval='@daily'
    ) as dag:
        pull_task = PythonOperator(
            task_id='pull_task',
            python_callable=pull_function,
            dag=dag,
        )

        pull_task
```

Lets use **XCOMS**

We will create a DAG in Python with a “push_task” and “pull_task” Using Airflow TI(Task instance) and keys

```python
from airflow import DAG
from airflow.operators.python_operator import PythonOperator
from airflow.operators.dummy_operator import DummyOperator
from datetime import datetime, timedelta

def push_function(ti):
    ti.xcom_push(key='python-xcom', value='This is the value being pushed')

def pull_function(ti):
    value = ti.xcom_pull(key='python-xcom', task_ids='push_task')
    print(f"Value from XCom: {value}")

default_args = {
    'owner': 'airflow',
    'start_date': datetime(2024, 6, 1),
    'retries': 5,
    'retry_delay': timedelta(minutes=5)
}

with DAG(
    'python_operator_xcom_dag',
    default_args=default_args,
    schedule_interval='@daily'
    ) as dag:
        push_task = PythonOperator(
            task_id='push_task',
            python_callable=push_function,
            dag=dag,
        )
        pull_task = PythonOperator(
            task_id='pull_task',
            python_callable=pull_function,
            dag=dag,
        )

        push_task >> pull_task
```

Another way of **handling dependencies: Task Flow API**

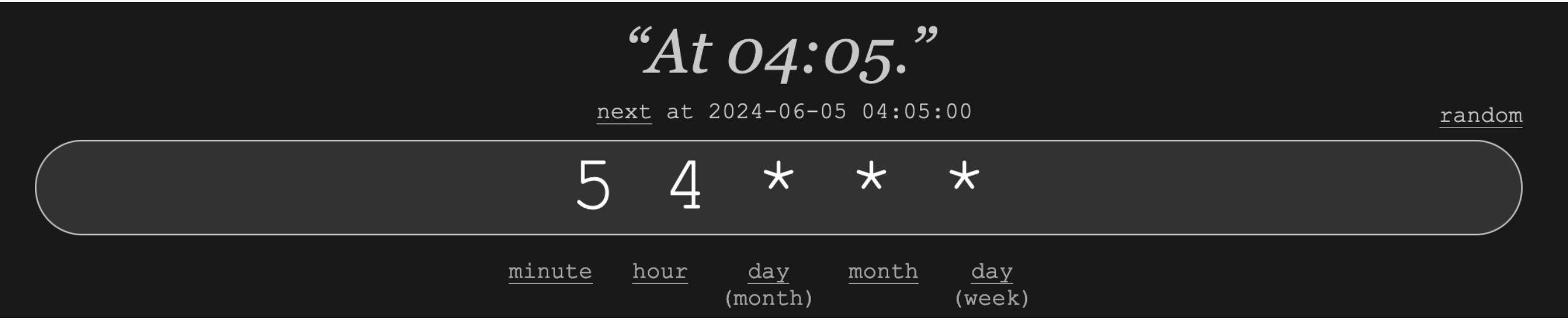
TaskFlow takes care of moving inputs and outputs between your Tasks using XComs for you, as well as automatically calculating dependencies - when you call a TaskFlow function in your DAG file, rather than executing it, you will get an object representing the XCom for the result (an `XComArg`), that you can then use as inputs to downstream tasks or operators.

Airflow Scheduler and CRON

DAGS can be scheduled through datetime or **Cron**.

Cron expression is a string comprising of five fields separated by white space that represents time.

https://crontab.guru/



Catchup and Backfill

Catchup refers to the process where Airflow tries to execute all the missing task instances from the start date to the current date whenever a DAG (Directed Acyclic Graph) is triggered or scheduled.

By default, Airflow is set to catch up on all the runs that were supposed to have happened according to the schedule interval but didn't because the scheduler was down or the DAG was paused.

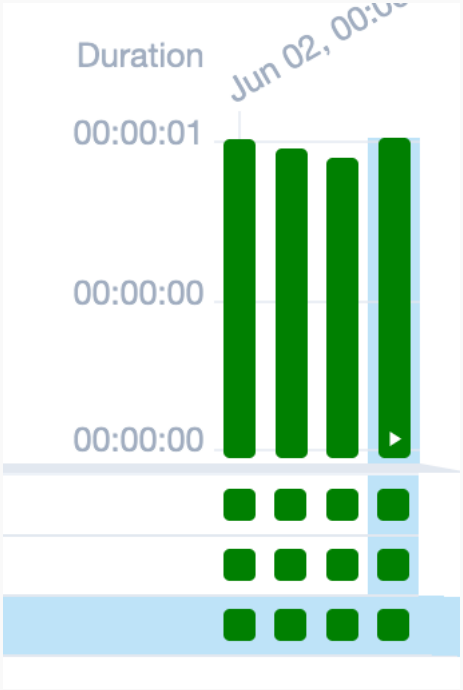
Backfill is a specific term in Airflow used for manually running DAG runs for a specified date range. This is often used to retroactively fill in gaps in your DAG's execution history.

In other words:

Catchup: Automatically tries to run all missed DAG runs based on the DAG's schedule interval from the start date to the present.

Backfill: Manually triggered process to run DAG instances for a specified date range.

Backfill - docker exec -it <container-id> bash
airflow dags backfill -s <start-date> -e <end-date> <dag_id>



Hooks

Airflow hooks are a core part of Apache Airflow's extensibility and allow you to interface with external systems. They encapsulate the logic needed to connect to and interact with these systems, such as databases, file storage, and external APIs. By using hooks, you can simplify the process of writing tasks in Airflow that need to interact with these systems, making your DAGs (Directed Acyclic Graphs) cleaner and more modular.

Hooks

First we must create a directory “plugin/hooks” where we can create out “ingestion_api_hook.py”

```python
from airflow.hooks.http_hook import HttpHook
import json

class IngestionApiHook(HttpHook):
    def __init__(self, http_conn_id, method='GET'):
        super().__init__(method=method, http_conn_id=http_conn_id)

    def run(self, endpoint, data=None, headers=None):
        self.method = self.method.upper()
        if self.method == 'POST' and data:
            response = self.run(endpoint, data=json.dumps(data), headers=headers)
        else:
            response = self.run(endpoint, headers=headers)
        return response.json()

```

Hooks

Second 3. Define an HTTP Connection in Airflow
This is done via the Airflow UI

1. Go to the Airflow web UI.
2. Navigate to **Admin** -> **Connections**.
3. Click the + button to add a new connection.
4. Set the **Conn Id** to a unique identifier, e.g., `my_api_conn_id`.
5. Set the Conn Type to `HTTP`.
6. Fill in the required fields, such as **Host**, and optionally **Login** and **Password** if needed. For internal have host.docker.internal

Docker Operator

The Airflow **Docker Operator** is a component of Apache Airflow that allows you to run tasks inside Docker containers. This operator is useful when you need to ensure that the execution environment for your tasks is consistent, isolated, and reproducible.

REQUIRES ABIT OF WORKAROUND IN YOUR YAML:
In your airflow docker-compose.yaml add the line
/var/run/docker.sock:/var/run/docker.sock
This is to open the docker socket

In your DAG use the DockerOperator

```
volumes:
  - ${AIRFLOW_PROJ_DIR:-.}/dags:/opt/airflow/dags
  - ${AIRFLOW_PROJ_DIR:-.}/logs:/opt/airflow/logs
  - ${AIRFLOW_PROJ_DIR:-.}/config:/opt/airflow/config
  - ${AIRFLOW_PROJ_DIR:-.}/plugins:/opt/airflow/plugins
  - /var/run/docker.sock:/var/run/docker.sock
```

```
run_custom_docker = DockerOperator(
    task_id='docker_pipeline',
    image='ingestion:latest',
    api_version='auto',
    auto_remove=True,
    command='python ingestion.py',
    docker_url='unix://var/run/docker.sock',
    network_mode='ai23network',
    environment={
        'EXECUTION_DATE': '{{ ds }}',
        'LOCATION': 'Stockholm'
    },
    dag=dag
)
```