What is Airflow

Apache Airflow is an open-source tool for creating, scheduling, and monitoring workflows.

Write workflows in Python, making it easy to connect with many technologies.

Web UI lets you view, manage, and debug workflows easily.

Flexible: Run it on your laptop or scale up for large, distributed systems.

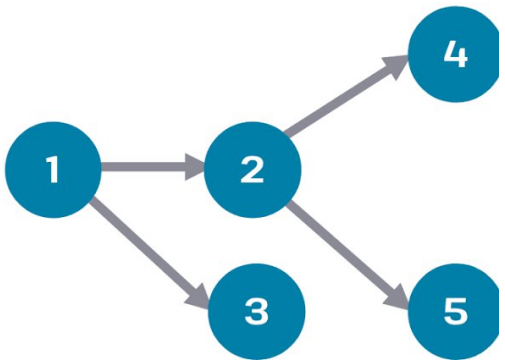
Workflows as code

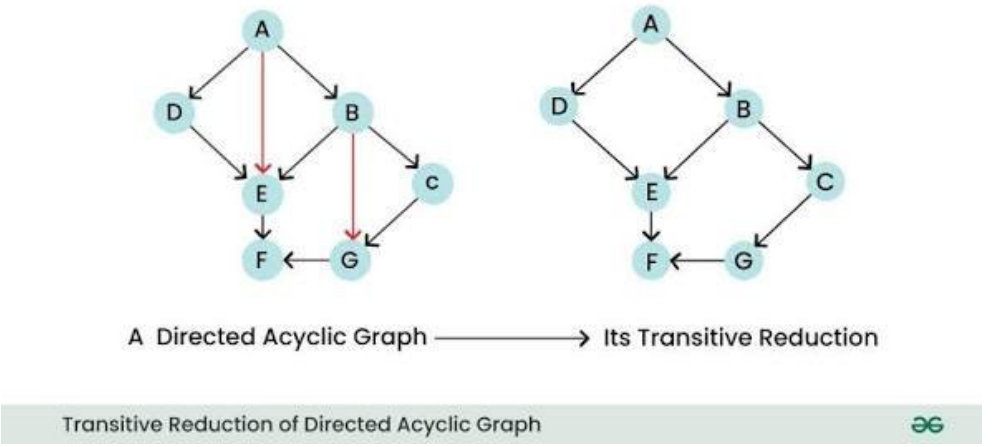
Airflow workflows are defined entirely in Python. This “workflows as code” approach brings several advantages:

- **Dynamic**: Pipelines are defined in code, enabling dynamic dag generation and parameterization.
- **Extensible**: The Airflow framework includes a wide range of built-in operators and can be extended to fit your needs.
- **Flexible**: Airflow leverages the Jinja templating engine, allowing rich customizations.

Dags – Directed Acyclic Graph

- A Directed Acyclic Graph (DAG) is a type of graph that consists of nodes (or vertices) and edges, where each edge has a direction, and there are no directed cycles.
- This means that if you follow the directed edges in the graph, you will never be able to return to a node you've already visited





Dags - Components

- **Components:**

- **Nodes (Vertices):** These represent entities, objects, tasks, or events within the system being modeled.
- **Directed Edges:** These are connections between nodes with a specified direction, indicating a one-way relationship or dependency. The arrow on the edge shows this direction.
- **Acyclic Nature:** The crucial feature of a DAG is the absence of cycles or loops. You cannot start at a node and follow the directed edges to get back to that same starting node.

Why DAGs are Useful ?

- DAGs are powerful tools for modeling systems where events or tasks happen in a specific order and dependencies exist. They are widely used in various fields:
  - **Task Scheduling:** DAGs are used to represent tasks and their dependencies, ensuring they are executed in the correct order.
  - **Data Processing Pipelines**: In data engineering, DAGs are used to define workflows for processing data, ensuring efficient handling of complex tasks.
  - **Version Control Systems:** Systems like Git use DAGs to track changes and manage branches effectively.
  - **Compiler Design:** DAGs are used in compilers to represent program structures and optimize code.
  - **Causal Structures:** DAGs are used to model causal relationships between events, particularly in fields like epidemiology and machine learning.
- **<u>Summing -</u>**
- *In essence, DAGs provide a clear and structured way to visualize and manage complex relationships, dependencies, and workflows without the possibility of getting caught in circular paths.*

DAG model

A DAG is a model that encapsulates everything needed to execute a workflow. Some DAG attributes include the following:

- **Schedule**: When the workflow should run.
- **Tasks**: tasks are discrete units of work that are run on workers.
- **Task Dependencies**: The order and conditions under which tasks execute.
- **Callbacks**: Actions to take when the entire workflow completes.
- **Additional Parameters**: And many other operational details.

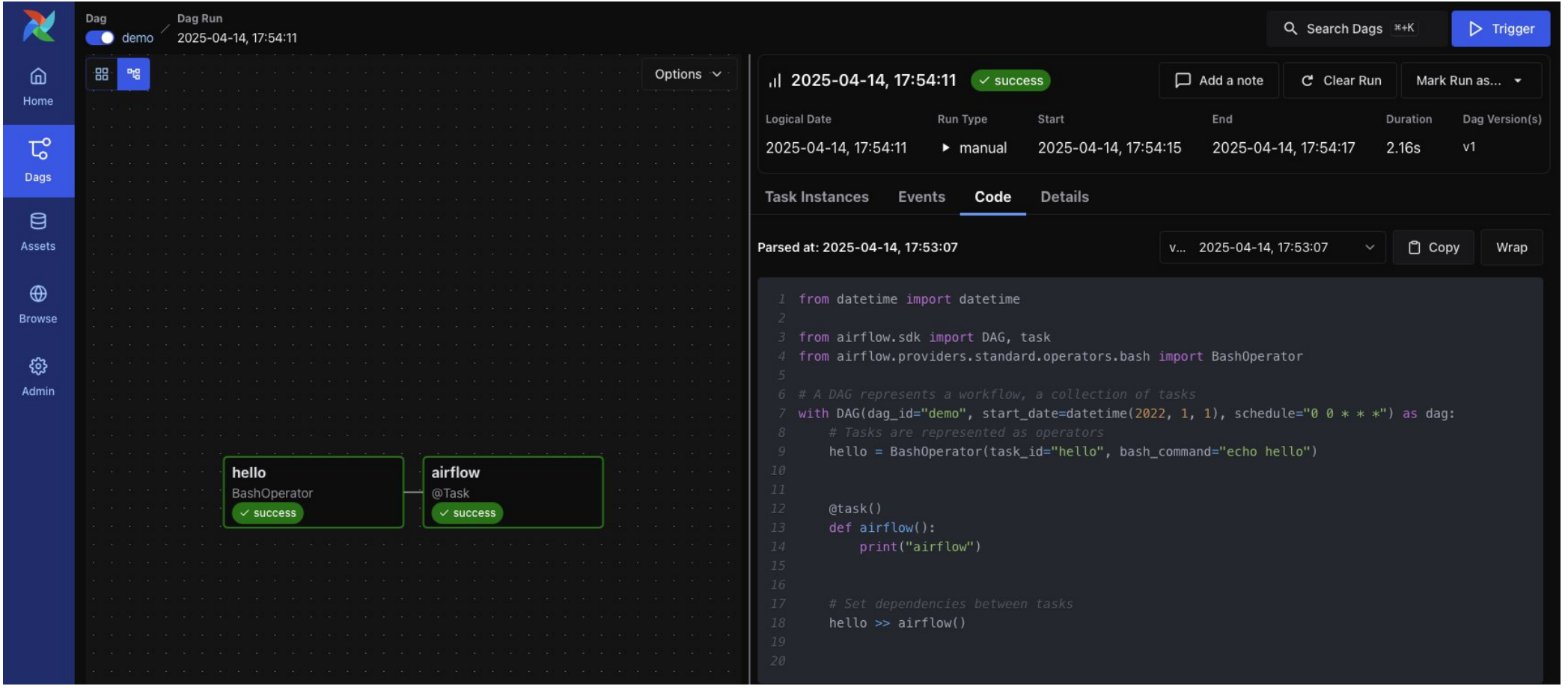
E.g DAG

DAG name

```python
from datetime import datetime

from airflow.sdk import DAG, task
from airflow.providers.standard.operators.bash import BashOperator

# A DAG represents a workflow, a collection of tasks
with DAG(dag_id="demo", start_date=datetime(2022, 1, 1), schedule="0 0 * * *") as dag:
    # Tasks are represented as operators
    hello = BashOperator(task_id="hello", bash_command="echo hello")

    @task()
    def airflow():
        print("airflow")

    # Set dependencies between tasks
    hello >> airflow()
```

Operator

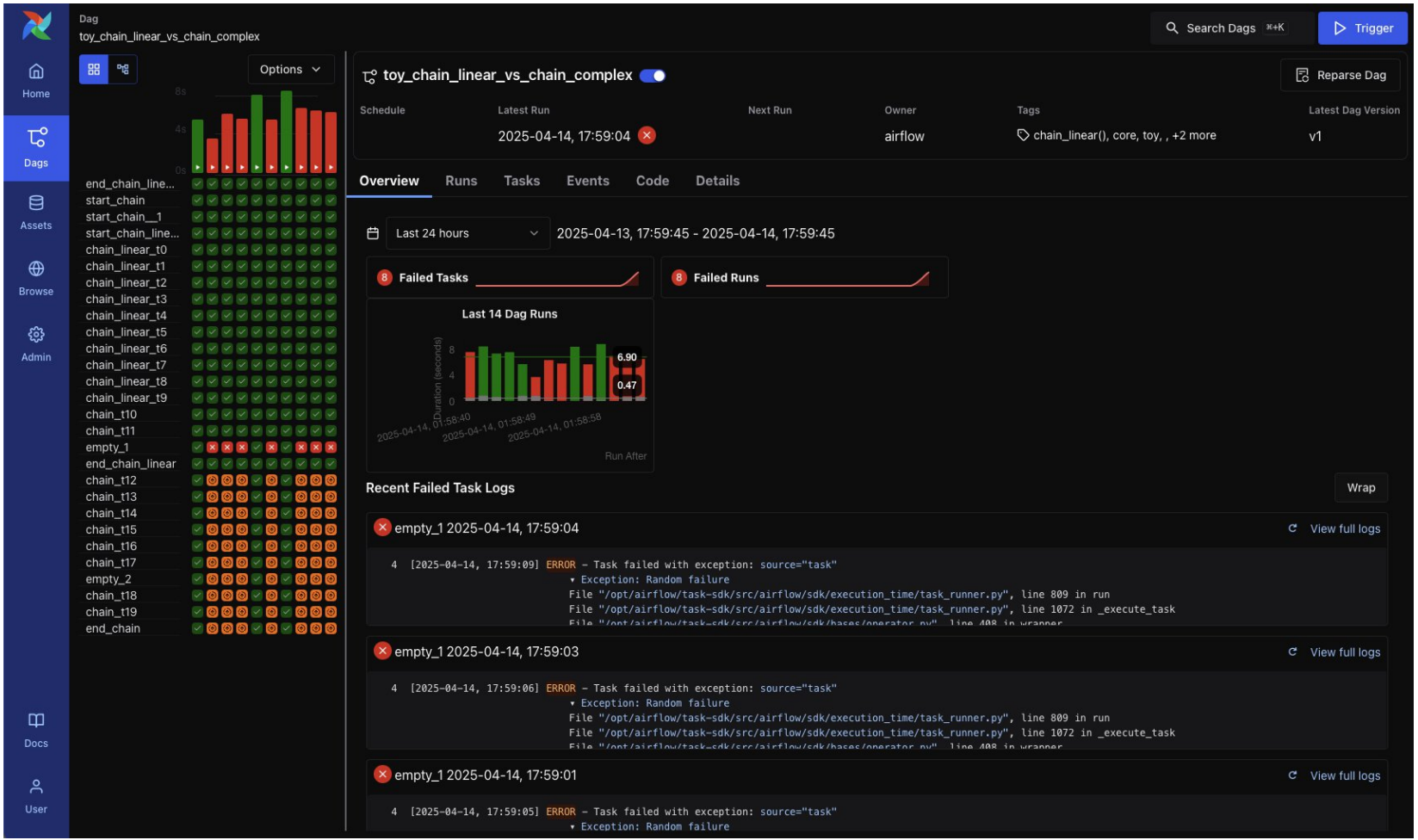
Dependency between 2 tasks


Dag
demo
Dag Run
2025-04-14, 17:54:11
Search Dags ⌘+K
Trigger
Home
Dags
Assets
Browse
Admin
Options
hello
BashOperator
success
airflow
@Task
success
2025-04-14, 17:54:11
success
Add a note
Clear Run
Mark Run as...
Logical Date
Run Type
Start
End
Duration
Dag Version(s)
2025-04-14, 17:54:11
manual
2025-04-14, 17:54:15
2025-04-14, 17:54:17
2.16s
v1
Task Instances
Events
Code
Details
Parsed at: 2025-04-14, 17:53:07
v... 2025-04-14, 17:53:07
Copy
Wrap
from datetime import datetime
from airflow.sdk import DAG, task
from airflow.providers.standard.operators.bash import BashOperator
# A DAG represents a workflow, a collection of tasks
with DAG(dag_id="demo", start_date=datetime(2022, 1, 1), schedule="0 0 * * *") as dag:
# Tasks are represented as operators
hello = BashOperator(task_id="hello", bash_command="echo hello")
@task()
def airflow():
print("airflow")
# Set dependencies between tasks
hello >> airflow()

Each column in the grid represents a single dag run.

While the graph and grid views are most commonly used,

Airflow provides several other views to help you monitor and troubleshoot workflows



Why Airflow ?

- **Airflow** helps you organize and run scheduled tasks (batch workflows).
- **Works well for tasks with a clear start and end** that need to run regularly.
- **Write workflows in Python code**—great for people who prefer coding over clicking.
- **Easy to track and share:**
  - Use version control to manage changes.
  - Many people can work on the same code.
  - Test workflows before running them.
  - Add new features or use existing ones.
- **Web interface:**
  - Start tasks manually, view logs, and check status.
  - Rerun failed or past tasks easily.
- **Highly customizable:**
  - Extend with your own plugins and components.
- **Open source and community-driven:**
  - Lots of free help and resources available.

Why not Airflow ?

- **Airflow** is made for tasks that run in batches and have a clear end, not for always-running or real-time jobs.
- **You can start workflows** using commands (CLI) or APIs, but Airflow is not meant for event-driven or streaming jobs.
- **Airflow works well with tools like Apache Kafka:**
  - Kafka handles real-time data.
  - Airflow picks up and processes that data later in batches.
- **Workflows are defined in code (Python):**
  - If you prefer clicking and not coding, Airflow may not be ideal.
  - The web UI helps manage workflows, but you still need to write code to create them.

Installation

Airflow® is tested with:

- Python: 3.9, 3.10, 3.11, 3.12
- Databases:
  - PostgreSQL: 12, 13, 14, 15, 16
  - MySQL: 8.0, Innovation
  - SQLite: 3.15.0+
- Kubernetes: 1.26, 1.27, 1.28, 1.29, 1.30

Version Life Cycle

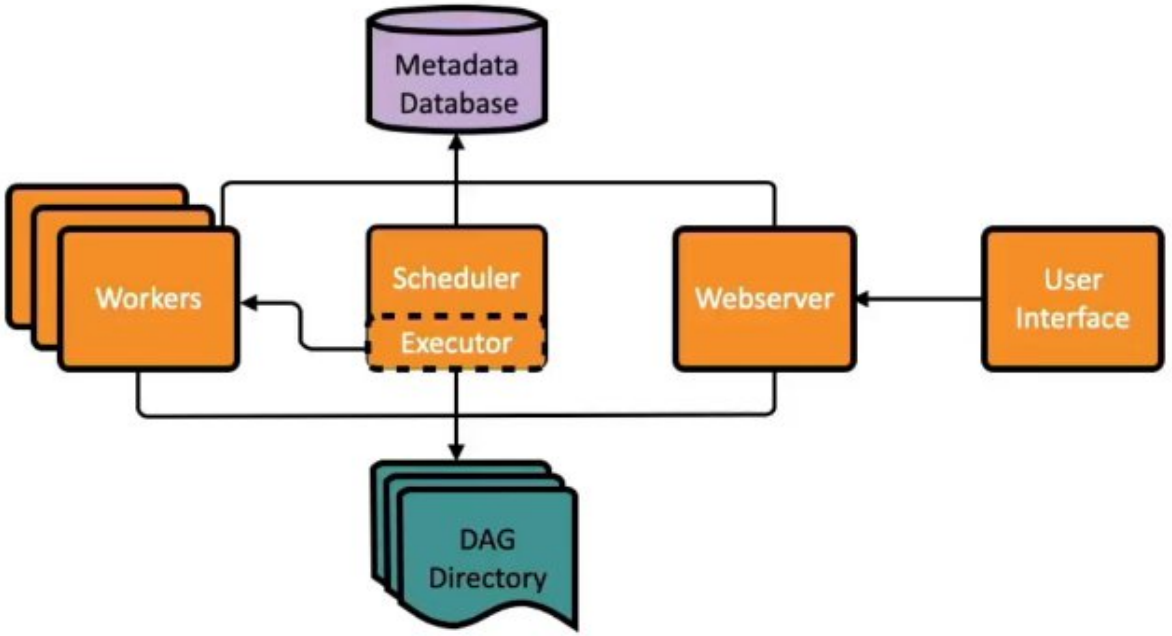
Apache Airflow® version life cycle:

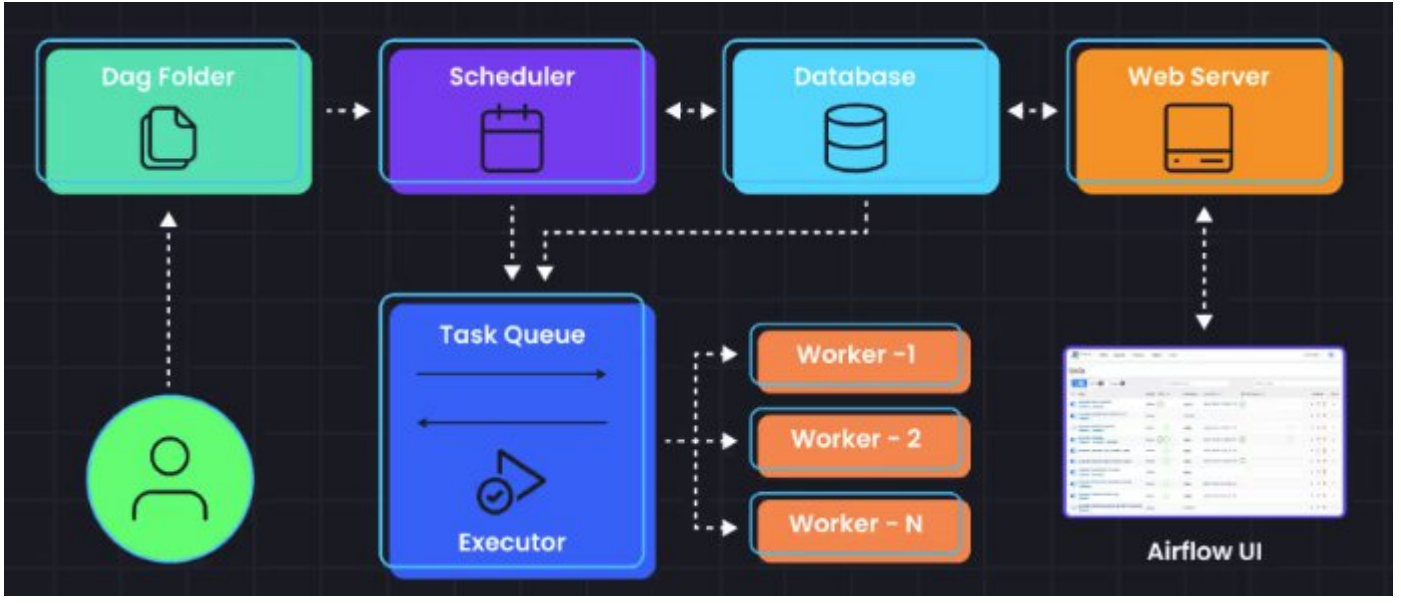
Version	Current Patch/Minor	State	First Release	Limited Maintenance	EOL/Terminated
3	3.0.2	Supported	Apr 22, 2025	TBD	TBD
2	2.11.0	Supported	Dec 17, 2020	Oct 22, 2025	Apr 22, 2026
1.10	1.10.15	EOL	Aug 27, 2018	Dec 17, 2020	June 17, 2021
1.9	1.9.0	EOL	Jan 03, 2018	Aug 27, 2018	Aug 27, 2018
1.8	1.8.2	EOL	Mar 19, 2017	Jan 03, 2018	Jan 03, 2018
1.7	1.7.1.2	EOL	Mar 28, 2016	Mar 19, 2017	Mar 19, 2017

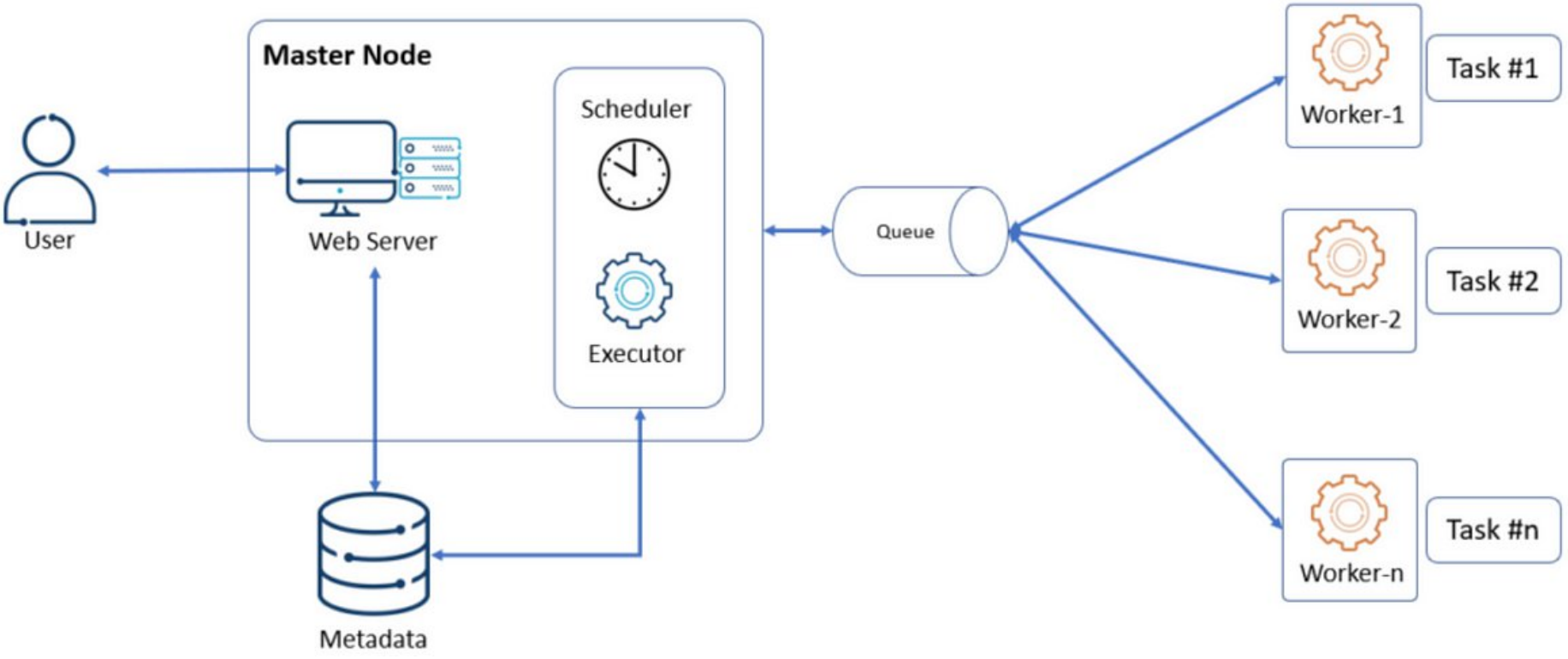
Upgrade

- Steps to be discussed

Airflow Architecture

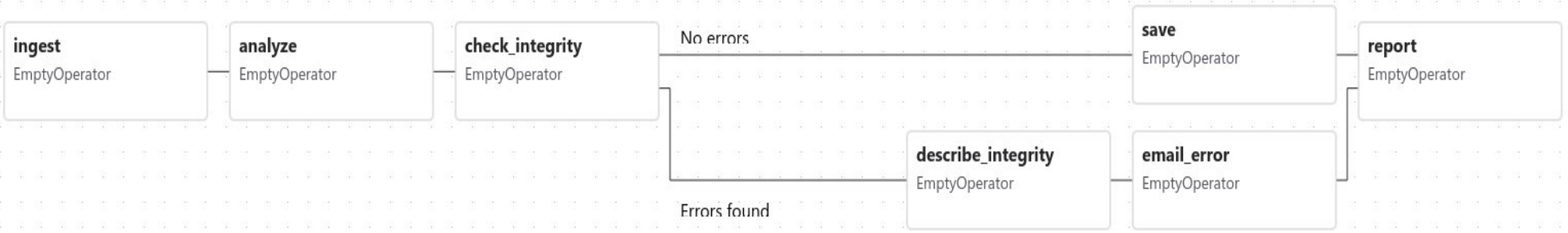



Dag Folder
Scheduler
Database
Web Server
Task Queue
Executor
Worker -1
Worker - 2
Worker - N
Airflow UI


Master Node
Scheduler
Executor
User
Web Server
Metadata
Queue
Worker-1
Worker-2
Worker-n
Task #1
Task #2
Task #n

Architecture

- Airflow is a platform that lets you build and run workflows.
- A workflow is represented as a DAG (a Directed Acyclic Graph), and contains individual pieces of work called Tasks, arranged with dependencies and data flows taken into account.



Terms

1) DAG

DAG specifies the dependencies between tasks, which defines the order in which to execute the tasks.

A DAG is like a recipe for a workflow. It defines the order in which tasks should run, without any loops.

2) Task

Tasks describe what to do, be it **fetching data,** running analysis, triggering other systems, or more.

A Task is a single step in a workflow, like running a script or moving a file.

- **Task A** is **upstream** of **Task B**.
- **Task B** is **downstream** of **Task A**.

Think of “upstream” as “must happen first”, and “downstream” as “comes after”.

3. Operator:

- An **Operator** is a template for what kind of work a task will do. For example, there are operators to run Python code, Bash commands, or move data.

4. Task Instance:

- A **Task Instance** is a single run of a task, on a specific date and time.

5. DAG Run:

- A **DAG Run** is one complete execution of the whole workflow (the DAG) for a specific schedule or trigger.

6. Scheduler:

- The **Scheduler** is the brain that decides when each task should run, based on the DAG's schedule and dependencies.

7. Executor:

- The **Executor** actually runs the tasks, possibly on different machines.

8. Web UI:

- The **Web UI** is a web dashboard to view, trigger, or manage workflows.

9. Trigger:

- To **Trigger** a DAG means to start it manually, outside its normal schedule.

10. Dependencies:

- **Dependencies** are the rules about which tasks must finish before others can start.

11. Upstream / Downstream:

- **Upstream** means tasks that must run before a given task.
- **Downstream** means tasks that must run after a given task.

12. XCom (Cross-Communication):

- **XCom** lets tasks pass small pieces of data to each other.

13. Schedule Interval:

- The **Schedule Interval** is how often the DAG should run (e.g., every hour, once a day, etc.).

14. Start Date / End Date:

- **Start Date** is when the DAG should start running.
- **End Date** is when it should stop.

15. Catchup:

- **Catchup** means whether Airflow should run missed DAG runs that didn't happen while the system was down.

16. Backfill:

- **Backfill** is running missed tasks from the past, based on the DAG's schedule.

17. Pool:

- A **Pool** limits how many tasks of a certain type can run at once.

18. SLA (Service Level Agreement):

- An **SLA** is a rule that says a task should finish within a certain time.

19. Sensor:

- A **Sensor** is a special operator that waits for something to happen, like a file appearing.

Airflow – Required Components

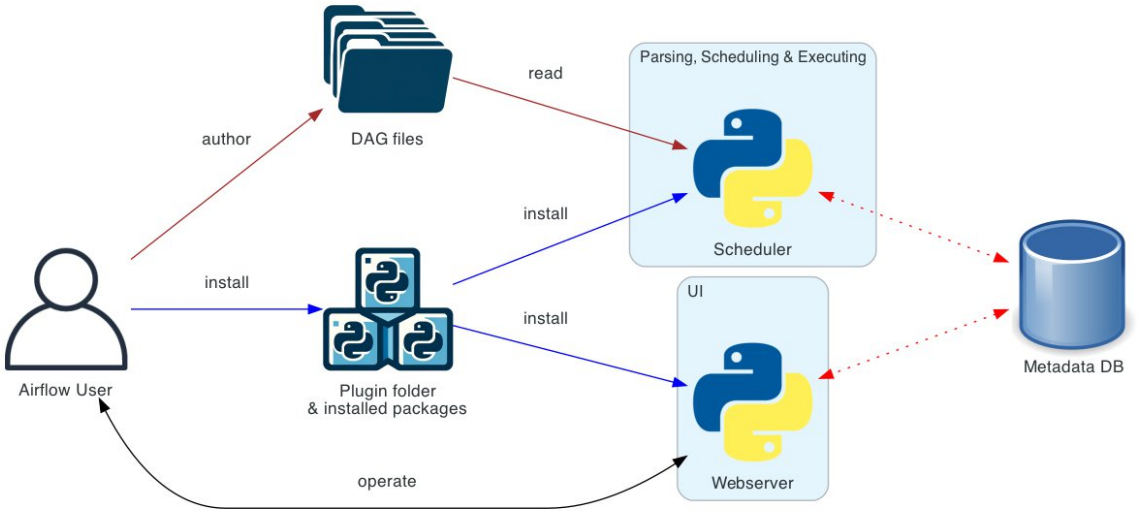
- **Scheduler**
  - Responsible for triggering scheduled workflows.
  - Submits tasks to the executor to run.
  - The executor is a configuration option of the scheduler (not a separate service) and runs within the scheduler process.
  - Multiple types of executors are available by default, or you can create your own.
- **DAG Processor**
  - Parses DAG files.
  - Serializes DAG information into the metadata database.
  - Essential for understanding and processing workflow definitions.
- **Webserver**
  - Provides a user-friendly web interface.
  - Allows users to inspect, trigger, and debug DAGs and tasks.
- **DAG Files Folder**
  - Directory containing your workflow (DAG) definitions.
  - The scheduler reads this folder to know which tasks to run and when.
- **Metadata Database**
  - Stores the state and history of all workflows and tasks.
  - All Airflow components interact with this database.
  - Setting up a backend database is mandatory for Airflow to function.

Airflow – Optional Components

- **Worker**
  - Executes tasks assigned by the scheduler.
  - In simple setups, the worker runs within the scheduler and is not separate.
  - For advanced setups (e.g., with CeleryExecutor or KubernetesExecutor), workers can run as separate long-running processes or Kubernetes pods, improving scalability.
- **Triggerer**
  - Handles deferred tasks using an asyncio event loop.
  - Not needed in basic setups that don’t use deferred tasks.
  - Only required if you use deferrable operators or triggers.
- **Plugins Folder**
  - Lets you extend and customize Airflow’s features (like installing additional packages).
  - Plugins are loaded by the scheduler, DAG processor, triggerer, and webserver.
  - Useful for adding new operators, hooks, sensors, or UI elements.

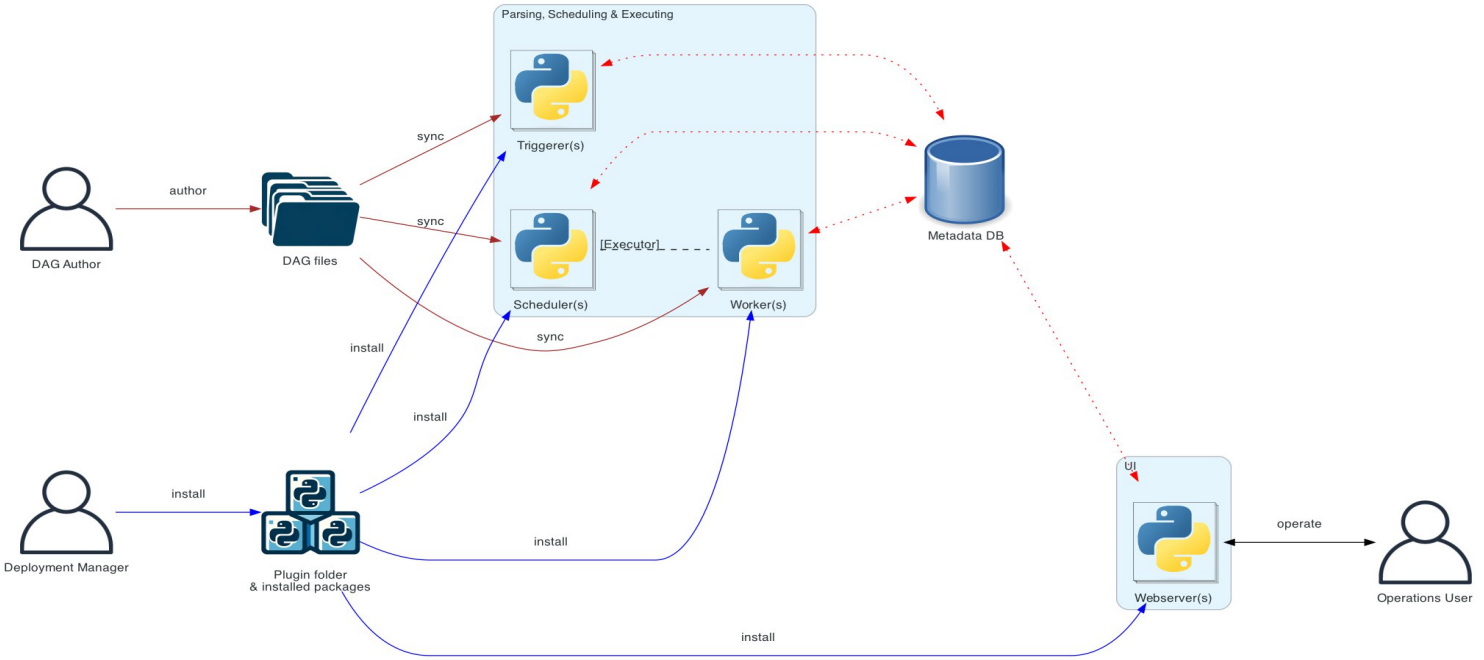
Deploying Airflow Components

Basic



Distributed Airflow architecture

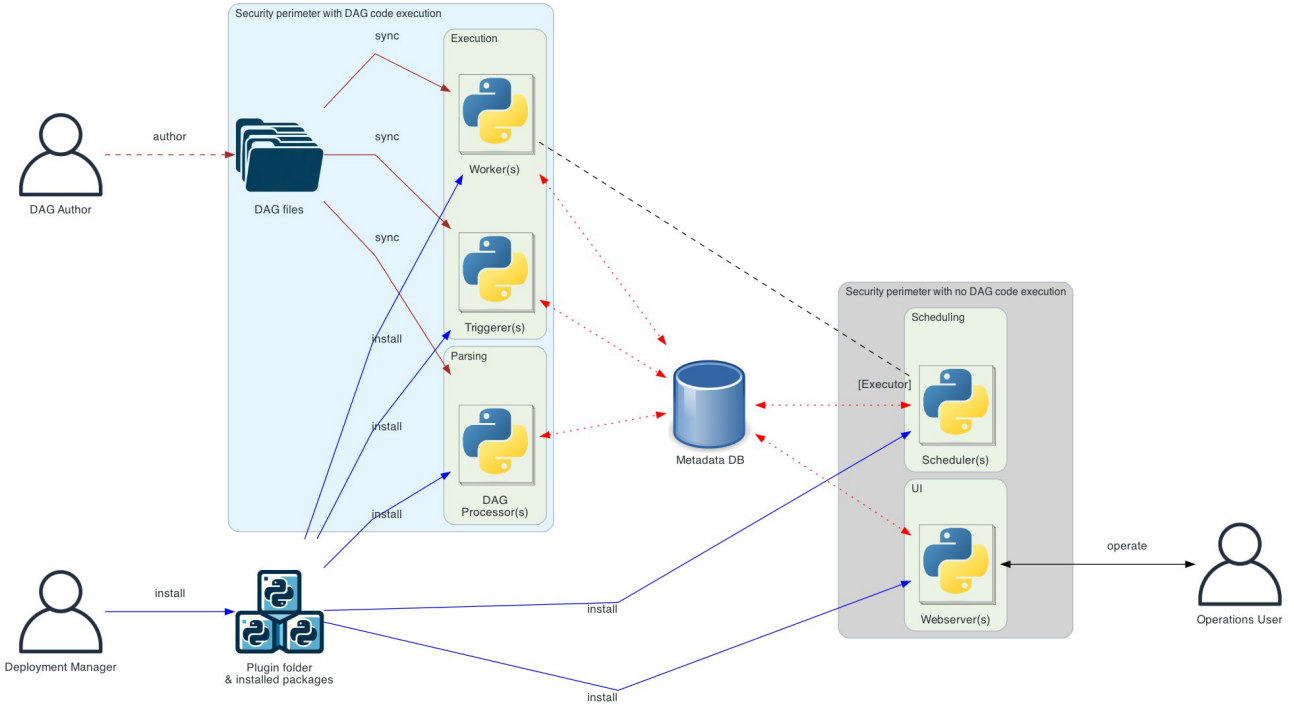
This is the architecture of Airflow where components of Airflow are distributed among multiple machines and where various roles of users are introduced - Deployment Manager, DAG author, Operations User.



Separate DAG processing architecture

In a more complex installation where security and isolation are important, you'll also see the standalone dag processor component that allows to separate scheduler from accessing DAG files.

This is suitable if the deployment focus is on isolation between parsed tasks.



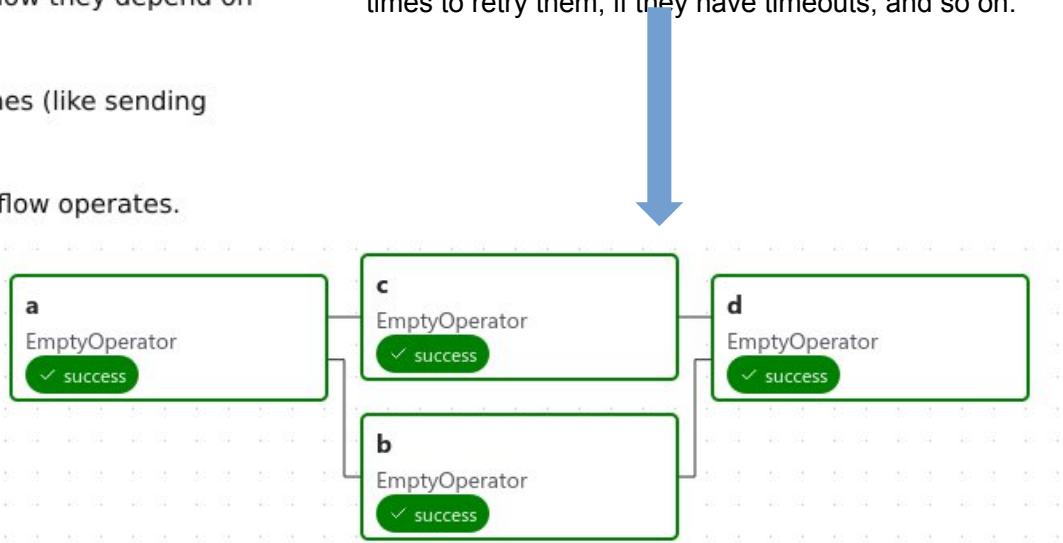
Dags

DAGs (Directed Acyclic Graphs) in Airflow

- **DAG** is a model that defines and manages the steps of a workflow in Airflow.
- **Key attributes of a DAG:**
  - **Schedule:** Specifies when the workflow should run (e.g., daily, hourly).
  - **Tasks:** The individual jobs or steps in the workflow, each doing a specific piece of work.
  - **Task Dependencies:** Defines the order in which tasks run and how they depend on each other.
  - **Callbacks:** Special actions that happen when the workflow finishes (like sending notifications).
  - **Additional Parameters:** Extra settings to control how the workflow operates.

It defines four Tasks - A, B, C, and D - and dictates the order in which they have to run, and which tasks depend on what others. It will also say how often to run the DAG - maybe “every 5 minutes starting tomorrow”, or “every day since January 1st, 2020”.

The DAG itself doesn’t care about what is happening inside the tasks; it is merely concerned with how to execute them - the order to run them in, how many times to retry them, if they have timeouts, and so on.



Declaring DAG

```python
import datetime

from airflow.sdk import DAG
from airflow.providers.standard.operators.empty import EmptyOperator

with DAG(
    dag_id="my_dag_name",
    start_date=datetime.datetime(2021, 1, 1),
    schedule="@daily",
):
    EmptyOperator(task_id="task")
```

```python
import datetime

from airflow.sdk import DAG
from airflow.providers.standard.operators.empty import EmptyOperator

my_dag = DAG(
    dag_id="my_dag_name",
    start_date=datetime.datetime(2021, 1, 1),
    schedule="@daily",
)
EmptyOperator(task_id="task", dag=my_dag)
```

```python
import datetime

from airflow.sdk import dag
from airflow.providers.standard.operators.empty import EmptyOperator


@dag(start_date=datetime.datetime(2021, 1, 1), schedule="@daily")
def generate_dag():
    EmptyOperator(task_id="task")


generate_dag()
```

Task Dependencies

- Task/Operator does not usually live alone;
- It has dependencies on other tasks (those upstream of it), and other tasks depend on it (those downstream of it).
- Declaring these dependencies between tasks is what makes up the DAG structure (the edges of the directed acyclic graph).

```
first_task >> [second_task, third_task]
third_task << fourth_task
```

OPerators

```
first_task.set_downstream([second_task, third_task])
third_task.set_upstream(fourth_task)
```

Cross Downstream , Complex dependencies

- If you want to make a *list of tasks depend on another list of tasks*.

```python
from airflow.sdk import cross_downstream

# Replaces
# [op1, op2] >> op3
# [op1, op2] >> op4
cross_downstream([op1, op2], [op3, op4])
```

Chaining together dependencies

```python
from airflow.sdk import chain

# Replaces op1 >> op2 >> op3 >> op4
chain(op1, op2, op3, op4)

# You can also do it dynamically
chain(*[EmptyOperator(task_id=f"op{i}") for i in range(1, 6)])
```

Loading DAG’s

Logic for Loading DAGs in Airflow

1. **DAG Folder Scanning**
   - Airflow regularly scans the folder(s) you specified for DAG files (Python scripts).
   - This is usually done by the **Scheduler** and **Webserver** processes.
2. **Parsing DAG Files**
   - Airflow imports (executes) each Python file in the DAGs folder.
   - Each file is expected to define one or more `DAG` objects.
   - Any code in the file (outside functions/classes) runs when the file is loaded.
3. **Extracting DAG Objects**
   - Airflow looks for variables in the file that are instances of the `DAG` class.
   - These `DAG` objects are then registered in Airflow’s internal registry.
4. **Serializing to Metadata Database**
   - The details of the DAGs (structure, tasks, schedule, etc.) are stored in the metadata database.
   - This allows other Airflow components (like the web UI and scheduler) to see and interact with the DAGs.
5. **Handling Updates**
   - When you add, update, or remove a DAG file, Airflow detects the change during the next scan.
   - It will reload the updated DAGs, add new ones, and remove deleted ones from the system.
6. **Syntax/Import Errors**
   - If a DAG file contains errors (like invalid Python code or import errors), Airflow will **skip** loading that DAG and show an error in the UI.

E.g Workflow

- You put a new Python file in the `dags/` folder, defining a new DAG.
- The scheduler and webserver **automatically scan** and import this file.
- If the file defines a valid `DAG` object, it appears in the Airflow UI and is ready to run.
- If you change the DAG (like adding new tasks), Airflow picks up the changes automatically after the next scan.

Key Points

- DAGs are just Python code.
- All top-level code in the file runs on import (so don’t put heavy code at the top level).
- DAG files are **reloaded frequently** to catch updates.
- DAGs are stored in the metadata DB after being parsed.

Multiple DAG’s per file ?

1. Multiple DAGs in a Single Python File

- **How it works:**
  - You can define more than one `DAG` object inside the same `.py` file in your `dags/` folder.
- **How Airflow loads it:**
  - When Airflow scans and imports that file, it will **find and register all `DAG` objects** created in that file.
  - All those DAGs will show up separately in the Airflow UI and will be scheduled independently.

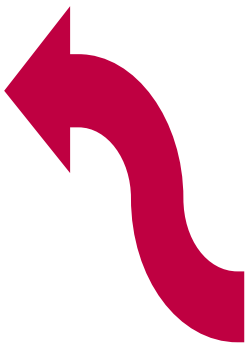
```
from airflow import DAG
from airflow.operators.empty import EmptyOperator
from datetime import datetime

with DAG('dag1', start_date=datetime(2024, 1, 1), schedule='@daily', catchup=False) as dag1:
    EmptyOperator(task_id='task1')

with DAG('dag2', start_date=datetime(2024, 1, 1), schedule='@hourly', catchup=False) as dag2:
    EmptyOperator(task_id='task2')
```

Spreading a Complex DAG Across Multiple Python Files

- **How it works:**
  - You can split a large or complex DAG into multiple files (for organization), but only **one file should define and instantiate the main `DAG` object**.
  - The main DAG file can import tasks or components from other Python files (modules).
- **How Airflow loads it:**
  - Airflow only loads files placed in the `dags/` folder.
  - If you split your code, make sure the file in the `dags/` folder imports everything it needs and creates the `DAG` object.
  - Supporting files (task definitions, helper functions) can be placed in a `dags/helpers/` or other subfolder and imported.

dags/my_complex_dag.py

```
from airflow import DAG

from datetime import datetime

from helpers.my_tasks import task_a, task_b      # Import tasks from another file

with DAG('complex_dag', start_date=datetime(2024, 1, 1), schedule='@daily',
catchup=False) as dag:

    task_a() >> task_b()
```



dags/helpers/my_tasks.py

```
from airflow.operators.empty import EmptyOperator


def task_a():

    return EmptyOperator(task_id='a')


def task_b():

    return EmptyOperator(task_id='b')
```

Airflow loads **my_complex_dag.py**, which defines the DAG and imports any required code from other files.

Key Takeaways

- **Multiple DAGs in One File:** All are loaded and registered.
- **One DAG Across Multiple Files:** Only the file that instantiates the `DAG` object (in `dags/`) is loaded as a DAG. The other files are just code helpers/modules.

Running DAG’s

- Dags will run in one of two ways:
- When they are triggered either manually or via the API
- On a defined schedule, which is defined as part of the DAG
- Dags do not require a schedule, but it’s very common to define one.
- E.g
- `with DAG("my_daily_dag", schedule="@daily"):`
- `with DAG("my_daily_dag", schedule="0 0 * * *"):`
- `with DAG("my_one_time_dag", schedule="@once"):`
- `with DAG("my_continuous_dag", schedule="@continuous"):`

Preset	Meaning
None	Don’t schedule (used for “manually/externally triggered” DAGs).
@once	Schedule once and only once.
@continuous	Run as soon as the previous run finishes.
@hourly	Run once an hour at the beginning of the hour.
@daily	Run once a day at midnight.
@weekly	Run once a week at midnight on Sunday morning.
@monthly	Run once a month at midnight on the first day of the month.
@quarterly	Run once a quarter at midnight on the first day.
@yearly	Run once a year at midnight on January 1.

When DAG runs

- **Each DAG execution creates a DAG Run**
  - Every time you run a DAG, a new instance called a “DAG Run” is created.
- **Parallel DAG Runs**
  - Multiple DAG Runs for the same DAG can run at the same time, each with its own context.
- **Data Interval**
  - Each DAG Run covers a specific period (data interval) for which the tasks operate (e.g. a single day).

When DAG runs

- **Backfilling**
  - Airflow can “backfill” a DAG, creating DAG Runs for previous periods (e.g., past 3 months), allowing historical data processing.
- **DAG Run Attributes**
  - Each DAG Run has:
    - **Start date:** When the run actually began
    - **End date:** When it finished
    - **Logical date:** (previously called execution date) — the intended schedule time or trigger time for that run
- **Task Instances**
  - Each DAG Run creates new Task Instances for every task defined in the DAG.
- **Manual vs. Scheduled Runs**
  - If manually triggered: Logical date = start date (when triggered)
  - If automatically scheduled: Logical date = start of the data interval; start date = logical date + scheduled interval

Default Arguments

```python
import pendulum

with DAG(
    dag_id="my_dag",
    start_date=pendulum.datetime(2016, 1, 1),
    schedule="@daily",
    default_args={"retries": 2},
):
    op = BashOperator(task_id="hello_world", bash_command="Hello World!")
    print(op.retries)  # 2
```

Default args

- `owner` **(str):** The owner of the task/DAG, usually a team or individual's name. Appears in the Airflow UI.
- `depends_on_past` **(bool):** If `True`, a task instance will only run if the *previous* task instance (for the immediately preceding `logical_date`) was successful. Useful for sequential, time-based dependencies.
- `email` **(str or list[str]):** Email address(es) to send alerts to. Requires Airflow to be configured for email.
- `email_on_failure` **(bool):** If `True`, sends an email when a task instance fails.
- `email_on_retry` **(bool):** If `True`, sends an email when a task instance is retried.
- `retries` **(int):** The number of times a task instance will retry upon failure.
- `retry_delay` **(timedelta):** The delay between retries. Often defined using `datetime.timedelta(minutes=5)` or similar.
- `retry_exponential_backoff` **(bool):** If `True`, the retry delay will increase exponentially (e.g., 1 min, then 2 min, then 4 min, etc.).

- `max_active_runs` **(int):** (This is a DAG parameter, not typically in `default_args`, but sometimes confused.) Controls the maximum number of active DAG runs at any given time for this specific DAG.
- `sla` **(timedelta):** Service Level Agreement. If a task doesn't complete within this time after its `logical_date`, an SLA miss alert is triggered.
- `queue` **(str):** The name of the queue to send the task to (relevant if you're using an executor that supports queues like Celery, Kubernetes, or Dask).
- `pool` **(str):** The name of a pool to which the task should be assigned. Pools control the maximum number of concurrently running tasks.
- `priority_weight` **(int):** A numerical weight that allows the scheduler to prioritize tasks. Higher weight means higher priority.
- `wait_for_downstream` **(bool):** If `True`, a task will wait for its *downstream* dependencies from the *previous* DAG run to complete before starting. Less common than `depends_on_past`.
- `trigger_rule` **(str):** Defines the conditions under which a task will run, based on the state of its upstream dependencies. Common values are `all_success` (default), `all_done`, `one_success`, `none_failed`, etc.

DAG decorator

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

# Define the DAG using the decorator
@dag(
    schedule="@daily",
    start_date=datetime(2024, 1, 1),
    catchup=False,
    tags=["example"]
)

def my_simple_decorator_dag():
    # Define tasks using the @task decorator
    @task
    def greet():
        print("Hello from a decorated DAG!")

    @task
    def add(a: int, b: int):
        return a + b
    # Task dependencies
    result = add(1, 2)
    greet() >> result

# Instantiate the DAG
dag = my_simple_decorator_dag()
```

How it works:

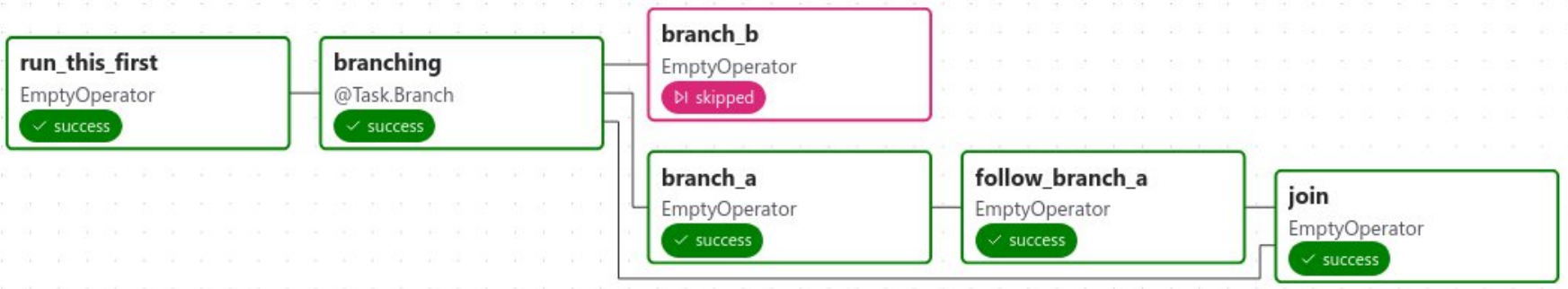
- The `@dag` decorator defines a DAG-generating function.
- Inside, you can define tasks using the `@task` decorator.
- Set dependencies as usual.
- At the end, you instantiate the DAG with `dag = my_simple_decorator_dag()` so Airflow can discover it.

Control Flow

- By default, a DAG will only run a Task when all the Tasks it depends on are successful. There are several ways of modifying this, however:
- Branching - select which Task to move onto based on a condition
- Trigger Rules - set the conditions under which a DAG will run a task
- Setup and Teardown - define setup and teardown relationships
- Latest Only - a special form of branching that only runs on dags running against the present
- Depends On Past - tasks can depend on themselves from a previous run

Branching

- We can make use of branching in order to tell the DAG not to run all dependent tasks, but instead to pick and choose one or more paths to go down.
- This is where the **@task.branch** decorator come in. e.g task_branch.py



```python
@task.branch(task_id="branch_task")
def branch_func(ti=None):
    xcom_value = int(ti.xcom_pull(task_ids="start_task"))
    if xcom_value >= 5:
        return "continue_task"
    elif xcom_value >= 3:
        return "stop_task"
    else:
        return None


start_op = BashOperator(
    task_id="start_task",
    bash_command="echo 5",
    do_xcom_push=True,
    dag=dag,
)

branch_op = branch_func()

continue_op = EmptyOperator(task_id="continue_task", dag=dag)
stop_op = EmptyOperator(task_id="stop_task", dag=dag)

start_op >> branch_op >> [continue_op, stop_op]
```

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

@dag(
    schedule="@daily",
    start_date=datetime.now() - timedelta(days=1),  # start_date is yesterday
    catchup=False,
    tags=["example", "branching"],
)

def branch_example_decorator():
    # Branching task
    @task.branch
    def choose_path(condition: str):
        if condition == "A":
            return "task_a"
        else:
            return "task_b"

    @task
    def task_a():
        print("Path A chosen!")

    @task
    def task_b():
        print("Path B chosen!")

    @task
    def after_branch():
        print("After branching.")

    # Task pipeline

    chosen = choose_path("A")
    a = task_a()
    b = task_b()
    after = after_branch()

    chosen >> [a, b] >> after

# Instantiate the DAG for Airflow to detect

dag = branch_example_decorator()
```

Latest Only

- There are situations, though, where you don’t want to let some (or all) parts of a DAG run for a previous date; in this case, you can use the LatestOnlyOperator.

```python
import datetime

import pendulum

from airflow.providers.standard.operators.empty import EmptyOperator
from airflow.providers.standard.operators.latest_only import LatestOnlyOperator
from airflow.sdk import DAG
from airflow.utils.trigger_rule import TriggerRule

with DAG(
    dag_id="latest_only_with_trigger",
    schedule=datetime.timedelta(hours=4),
    start_date=pendulum.datetime(2021, 1, 1, tz="UTC"),
    catchup=False,
    tags=["example3"],
) as dag:
    latest_only = LatestOnlyOperator(task_id="latest_only")
    task1 = EmptyOperator(task_id="task1")
    task2 = EmptyOperator(task_id="task2")
    task3 = EmptyOperator(task_id="task3")
    task4 = EmptyOperator(task_id="task4", trigger_rule=TriggerRule.ALL_DONE)

    latest_only >> task1 >> [task3, task4]
    task2 >> [task3, task4]
```

Depends on Past ?

What is `depends_on_past` in Airflow?

- `depends_on_past` is a task parameter in Airflow.
- When set to `True`, a task **will not run** for the current DAG run **unless its own previous run (from the prior schedule/interval) succeeded**.
- This is useful when each run of a task depends on the success of the same task in the previous period (e.g., data pipelines that must run in order without skipping).

How It Works

- **If `depends_on_past=False` (default):**
  - The task runs independently for each DAG run.
- **If `depends_on_past=True`:**
  - The task waits for its previous run to succeed before running again.

E.g

```python
from airflow import DAG
from airflow.operators.empty import EmptyOperator
from datetime import datetime, timedelta

with DAG(
    dag_id="depends_on_past_example",
    start_date=datetime(2024, 7, 1),
    schedule="@daily",
    catchup=True
) as dag:

    t1 = EmptyOperator(
        task_id="t1",
        depends_on_past=True  # This is the key!
    )
```

How this works:

- On July 2, Airflow will run `t1` for July 1.
- On July 3, `t1` will **only** run for July 2 if `t1` for July 1 succeeded.
- If `t1` failed or was skipped on July 1, it will **not** run on July 2, and so on.

In short:

`depends_on_past=True` enforces strict order and continuity for a task across scheduled runs.

Trigger Rules

What is `trigger_rule` in Airflow?

- `trigger_rule` is a property of Airflow tasks that controls **when a task should run** based on the state of its upstream (previous) tasks.
- By default, a task runs when **all its upstream tasks succeed** (`trigger_rule="all_success"`).
- You can change `trigger_rule` to customize how and when your task is triggered, especially for branching, error handling, or complex workflows.

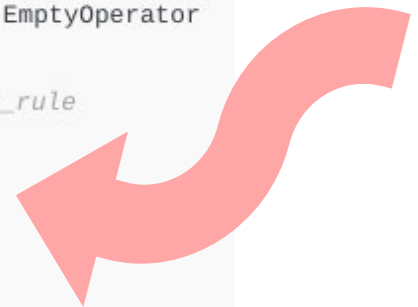
Common Trigger Rules

Trigger Rule	Description
`all_success`	(Default) Run when **all upstream tasks succeed**.
`all_failed`	Run when **all upstream tasks fail**.
`all_done`	Run when **all upstream tasks are finished** (success or fail).
`one_success`	Run when **any one upstream task succeeds**.
`one_failed`	Run when **any one upstream task fails**.
`none_failed`	Run when **no upstream tasks have failed** (some can be skipped).
`none_failed_or_skipped`	Run when **no upstream tasks failed or were skipped**.
`none_skipped`	Run when **no upstream tasks were skipped**.

✅ When to Use Specific Trigger Rules

Use Case	Trigger Rule
Send a notification if any task fails	`one_failed`
Clean up resources regardless of task results	`all_done` or `always`
Run task even if previous task is skipped	`none_skipped` or `all_done`
Conditionally execute a downstream task only if no failures	`none_failed`

```python
from airflow.operators.empty import EmptyOperator

# Example task with a custom trigger_rule
my_task = EmptyOperator(
    task_id="wait_for_any_failure",
    trigger_rule="one_failed"
)
```



my_task will run as soon as any upstream task fails.

In short:

`trigger_rule` lets you control exactly **when** your task will execute, depending on what happened to tasks before it.

Setup and teardown

What are Setup and Teardown Tasks in Airflow?

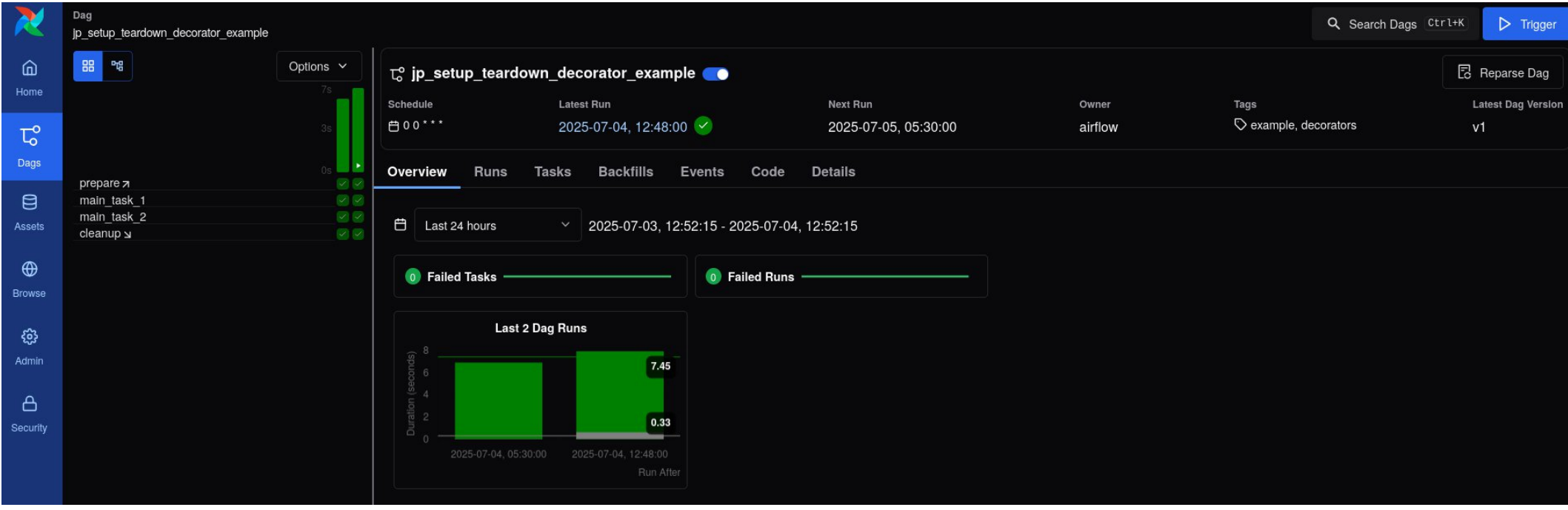
- **Setup task:** Runs before a set of tasks to prepare resources (e.g., start a cluster, create temp files).
- **Teardown task:** Runs after those tasks, regardless of their success/failure, to clean up (e.g., shut down cluster, delete temp files).
- You can define them for any part of your DAG—commonly used with Task Groups.

Setup and teardown

Key Points:

- **Setup/teardown tasks** ensure resources are always prepared/cleaned up, even if main tasks fail.
- You can use either **classic operators** or **decorators**—both are supported in Airflow 3.x.
- **Teardown** runs after all specified tasks, **even if** some fail/skipped.

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

@dag(
    start_date=datetime(2024, 7, 1),
    schedule="@daily",
    catchup=False,
    tags=["example", "decorators"],
)
def jp_setup_teardown_decorator_example():

    @setup
    def prepare():
        print("Setup: preparing resources!")

    @task
    def main_task_1():
        print("Doing main task 1...")

    @task
    def main_task_2():
        print("Doing main task 2...")

    @teardown
    def cleanup():
        print("Teardown: cleaning up resources!")

    # Workflow: setup -> (main tasks) -> teardown
    prepare() >> [main_task_1(), main_task_2()] >> cleanup()

dag = jp_setup_teardown_decorator_example()
```


Dag
jp_setup_teardown_decorator_example
Search Dags Ctrl+K
Trigger
Home
Dags
Assets
Browse
Admin
Security
Options
7s
3s
0s
prepare
main_task_1
main_task_2
cleanup
jp_setup_teardown_decorator_example
Reparse Dag
Schedule
0 0 * * *
Latest Run
2025-07-04, 12:48:00
Next Run
2025-07-05, 05:30:00
Owner
airflow
Tags
example, decorators
Latest Dag Version
v1
Overview
Runs
Tasks
Backfills
Events
Code
Details
Last 24 hours
2025-07-03, 12:52:15 - 2025-07-04, 12:52:15
0 Failed Tasks
0 Failed Runs
Last 2 Dag Runs
Duration (seconds)
8
6
4
2
0
7.45
0.33
2025-07-04, 05:30:00
2025-07-04, 12:48:00
Run After

Dynamic DAGs

- In Airflow 3.0.2, a "dynamic DAG" usually means creating tasks dynamically (such as in a loop) based on data, configuration, or other logic.
- This is fully supported and often used for ETL, batch, or ML workflows.

```
jilg@node-214:~/airflow_learning/dags$ cat jp_dynamic_dag.py
from airflow import DAG
from airflow.operators.python import PythonOperator
from datetime import datetime

# Example list of filenames (could come from anywhere)
file_list = ['file_a.csv', 'file_b.csv', 'file_c.csv']

def process_file(filename):
    print(f"Processing {filename}")

with DAG(
    dag_id="jp_dynamic_task_example",
    start_date=datetime(2024, 7, 1),
    schedule="@daily",
    catchup=False,
    tags=["dynamic", "example"],
) as dag:

    # Dynamically create a task for each file
    for fname in file_list:
        PythonOperator(
            task_id=f"process_{fname.replace('.', '_')}",
            python_callable=process_file,
            op_args=[fname],
        )
```

How this works:

- The loop creates one `PythonOperator` task per file.
- In the Airflow UI, you'll see `process_file_a_csv`, `process_file_b_csv`, etc.
- All tasks are independent and run in parallel unless you add dependencies.

TaskGroup

What is a TaskGroup in Airflow?

- **TaskGroup** is a feature in Airflow used to **logically group related tasks together** within a DAG.
- It’s a **visual and organizational tool**—not a new type of operator.
- In the Airflow UI, TaskGroups appear as collapsible sections, making large and complex DAGs much easier to read and manage.
- TaskGroups help you **avoid code duplication** and keep DAG definitions tidy, especially for repetitive patterns.

Typical Use Cases for TaskGroups

1. **Organizing Complex DAGs**
   - Break large workflows into sections (e.g., “Extract”, “Transform”, “Load”).
   - Keeps DAG diagrams uncluttered.
2. **Managing Repetitive Logic**
   - When you have the same logic applied to many items (e.g., process data for multiple regions or files), you can loop over a TaskGroup.
3. **Encapsulating Sub-Workflows**
   - Use TaskGroups to logically encapsulate steps like “Data Quality Checks” or “Notification Handling”.
4. **Easier Dependency Management**
   - You can set dependencies *to or from* entire groups, not just single tasks.

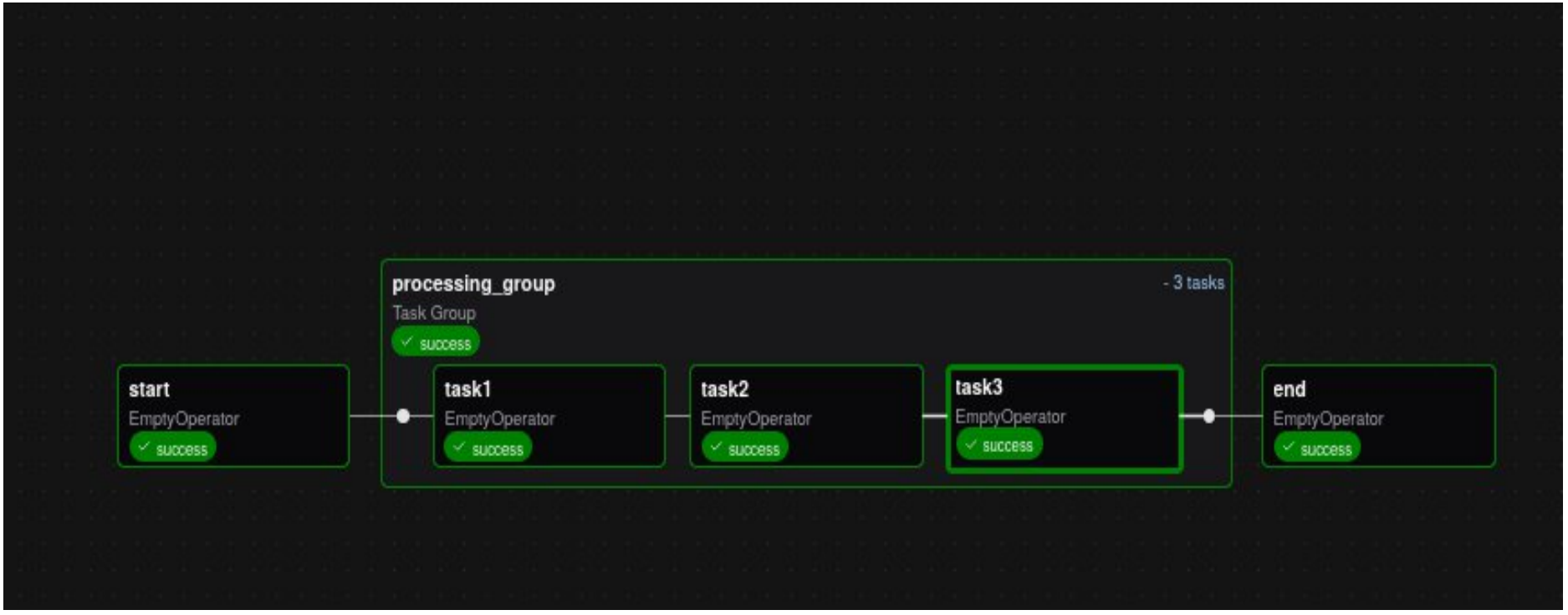
Real-World Examples

- **ETL Pipelines:**
  Use TaskGroups for each step: one for extraction, one for transformation, and one for loading data.
- **Parallel Processing:**
  Group all parallel tasks for different files/partitions into a single TaskGroup.
- **Multi-step Operations:**
  If each dataset requires validation, cleaning, and processing—wrap these steps in a TaskGroup and call it for each dataset.

```python
from airflow import DAG
from airflow.operators.empty import EmptyOperator
from airflow.utils.task_group import TaskGroup
from datetime import datetime

with DAG(
    dag_id="jp_taskgroup_example",
    start_date=datetime(2024, 7, 1),
    schedule="@daily",   # New style for Airflow 3.x
    catchup=False,
    tags=["example", "taskgroup"],
) as dag:

    start = EmptyOperator(task_id="start")

    # Define a TaskGroup
    with TaskGroup("processing_group") as processing_group:
        task1 = EmptyOperator(task_id="task1")
        task2 = EmptyOperator(task_id="task2")
        task3 = EmptyOperator(task_id="task3")
        task1 >> task2 >> task3

    end = EmptyOperator(task_id="end")

    # Set dependencies: start -> processing_group -> end
    start >> processing_group >> end
```

How This Works:

- `start` runs first.
- The three tasks inside `processing_group` run in order: `task1` → `task2` → `task3`.
- After the TaskGroup finishes, `end` runs.
- The group appears as a **collapsible box** in the Airflow UI for better organization and visualization.


processing_group
- 3 tasks
Task Group
success
start
EmptyOperator
success
task1
EmptyOperator
success
task2
EmptyOperator
success
task3
EmptyOperator
success
end
EmptyOperator
success

Edge Labels

- Use *edge labels to annotate dependencies between tasks or TaskGroups* in the Airflow UI, making DAG flows easier to understand at a glance.

```python
import pendulum
from airflow import DAG
from airflow.operators.empty import EmptyOperator
from airflow.utils.edgemodifier import Label

with DAG(
    "jp_example_branch_labels",
    schedule="@daily",
    start_date=pendulum.datetime(2021, 1, 1, tz="UTC"),
    catchup=False,
    tags=["example", "edgelabel"],
) as dag:
    ingest = EmptyOperator(task_id="ingest")
    analyze = EmptyOperator(task_id="analyze")
    check = EmptyOperator(task_id="check_integrity")
    describe = EmptyOperator(task_id="describe_integrity")
    error = EmptyOperator(task_id="email_error")
    save = EmptyOperator(task_id="save")
    report = EmptyOperator(task_id="report")

    ingest >> analyze >> check
    check >> Label("No errors") >> save >> report
    check >> Label("Errors found") >> describe >> error >> report
```

```
jilg@node-214:~/airflow_learning/dags$ cat jp_branch_labels_2.py
from airflow import DAG
from airflow.operators.empty import EmptyOperator
from airflow.utils.edgemodifier import Label
from datetime import datetime

with DAG(
    dag_id="jp_edge_label_simple_example",
    schedule="@daily",
    start_date=datetime(2024, 7, 1),
    catchup=False,
    tags=["jpsimple","example", "edgelabel"],
) as dag:
    start = EmptyOperator(task_id="start")
    quality_check = EmptyOperator(task_id="quality_check")
    pass_step = EmptyOperator(task_id="pass")
    fail_step = EmptyOperator(task_id="fail")
    end = EmptyOperator(task_id="end")

    start >> quality_check
    quality_check >> Label("passed") >> pass_step >> end
    quality_check >> Label("failed") >> fail_step >> end
```

- The `quality_check` task splits into two branches:
  - If it **passes**, the edge is labeled "passed" to `pass`.
  - If it **fails**, the edge is labeled "failed" to `fail`.
- Both paths then join at `end`.

DAG Doc

- We can easily add DAG documentation and task documentation using the doc_md attribute (for Markdown) or doc (for plain text).
- These docs are visible in the Airflow UI and are very useful for team communication and future you!

attribute	rendered to
doc	monospace
doc_json	json
doc_yaml	yaml
doc_md	markdown
doc_rst	reStructuredText

There are a set of special task attributes that get rendered as rich content if defined:

```
jilg@node-214:~/airflow_learning/dags$ cat jp_dag_documentation.py
from airflow import DAG
from airflow.operators.empty import EmptyOperator
from datetime import datetime

with DAG(
    dag_id="jp_dag_doc_example",
    start_date=datetime(2024, 7, 1),
    schedule="@daily",
    catchup=False,
    tags=["doc", "example"],
    doc_md="""
# Example DAG Documentation

This DAG is an example of how to add documentation in **Markdown**.

- You can use bullet points
- Add code: `print('Hello Airflow!')`
- Use headers, links, and more.

For details, see the [Airflow documentation](https://airflow.apache.org/docs/).
"""
) as dag:
    task1 = EmptyOperator(task_id="task1")
    task2 = EmptyOperator(task_id="task2")
    task1 >> task2
```

Task doc

```
jilg@node-214:~/airflow_learning/dags$ cat jp_task_documentation.py
from airflow import DAG
from airflow.operators.empty import EmptyOperator
from datetime import datetime

with DAG(
    dag_id="jp_task_doc_example",
    start_date=datetime(2024, 7, 1),
    schedule="@daily",
    catchup=False,
    tags=["doc", "example"],
) as dag:
    start = EmptyOperator(task_id="start")
    process = EmptyOperator(task_id="process")
    end = EmptyOperator(task_id="end")

    # Add documentation to tasks
    start.doc_md = """
**Start Task**

This task marks the beginning of the workflow.
"""

    process.doc_md = """
**Process Task**

This task processes the main data.
You can include:
- *Markdown*
- Code examples
- Tables

| Step | Description      |
|------|------------------|
| 1    | Initial process  |
| 2    | Further process  |
"""

    end.doc_md = """
**End Task**

The workflow finishes here.
"""

    start >> process >> end
```

✅ What are Airflow Variables?

- Stored in Airflow's **metadata database**
- Used to **store and retrieve config values** across your DAGs
- Can be managed via:
  - **UI** → Admin → Variables
  - **CLI** → `airflow variables`
  - **Code** → `Variable.get()`

📌 Step 1: Add a Variable

Go to **Airflow UI → Admin → Variables**, and add:

Key	Value
`my_name`	`John`

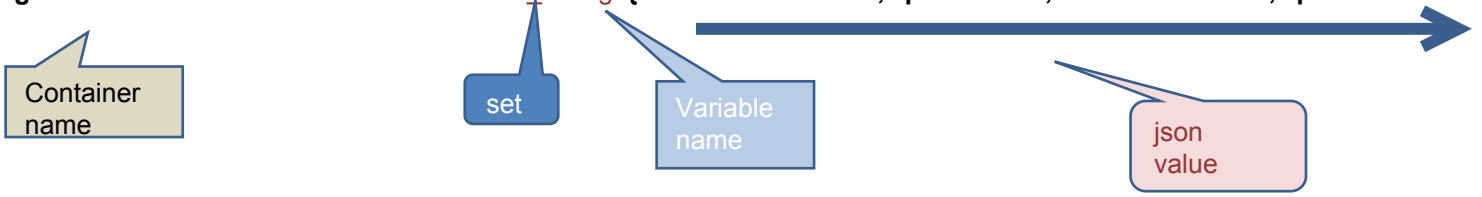
Set via CLI

airflow variables set my_name John

```
jilg@node-214:~/airflow_learning/dags$ cat jp_variable_def.py
from airflow import DAG
from airflow.decorators import task
from airflow.sdk import Variable
from datetime import datetime, timedelta

with DAG(
    dag_id="jp_variable_usage",
    start_date=datetime.now() - timedelta(days=1),
    schedule=None,
    catchup=False,
    tags=["example"],
) as dag:

    @task
    def greet():
        # Fetch variable from Airflow
        name = Variable.get("my_name", default="Guest")
        print(f"Hello, {name}!")

    greet()
```

JSON Variable Usage

```
docker  exec -it airflow_learning-airflow-worker-1 airflow variables set db_config '{"host": "localhost", "port": 5432, "user": "airflow", "password": "secret"}'
```



OR set via

Web UI: **Admin → Variables → Add Variable**

In **Airflow 3.0.2**, **Connections** are used to securely store credentials and configuration details needed to connect to external systems like:

- Databases (PostgreSQL, MySQL, etc.)
- Cloud services (AWS, GCP, Azure)
- Message brokers (RabbitMQ, Kafka)
- APIs (HTTP, REST, custom)

🧩 What Is a Connection in Airflow?

A **Connection** in Airflow is an object that contains:

Field	Purpose
`Conn Id`	Unique name to reference the connection
`Conn Type`	Type of service (e.g., `postgres`, `aws`, `http`)
`Host`	Hostname or IP of the service
`Schema`	DB name or schema (optional)
`Login`	Username
`Password`	Password
`Port`	Port number
`Extra`	JSON config for other options (e.g., region, keyfile)

✅ Ways to Manage Connections

Method	Usage
Web UI	Admin → Connections
CLI	`airflow connections` command
Environment Variables	Use `AIRFLOW_CONN_<CONN_ID>`
Code	Using `airflow.sdk.Connection` (in Airflow 3.0)

airflow connections add 'my_postgres' --conn-uri 'postgresql://airflow:airflowpass@localhost:5432/mydb'

We can now use `conn_id='my_postgres'` in operators or hooks.

.airflowignore

What is `.airflowignore`?

- `.airflowignore` is a special file you can place inside your Airflow DAGs directory (or any of its subdirectories).
- It tells Airflow to **ignore specific files or folders** when searching for and loading DAGs.
- It works similarly to `.gitignore` in Git, but is used only for controlling which Python scripts and subdirectories Airflow parses for DAGs.

.airflowignore

Why is it Useful?

- **Performance:** Exclude large, unnecessary, or temporary files that would slow down DAG parsing.
- **Error Prevention:** Avoid Airflow import errors by skipping files that are not meant to be parsed as DAGs (like helper scripts, test files, or data files).
- **Cleanliness:** Keep your Airflow UI uncluttered by preventing accidental DAG registration for scripts not intended to be DAGs.

.airflowignore

How Does It Work?

- When Airflow scans your `dags/` folder, it reads the `.airflowignore` file (if present).
- Each line in `.airflowignore` is a **pattern** (glob or regular expression) describing which files or folders to ignore.
- If a file or folder matches any pattern in `.airflowignore`, Airflow will **skip** it and not attempt to import/parse it.

.airflowignore

Where to Place It?

- Place a `.airflowignore` file in **any directory** inside your DAGs folder (including subfolders).
- The rules in `.airflowignore` apply **only to that directory and its subdirectories**.

What Patterns Can You Use?

- Simple filename matches: `test_*.py`
- Directory matches: `archive/`
- Wildcards and globs: `*.bak`, `tmp*`
- Regex: `.*_helper\.py`

E.g

```
dags/
├── my_etl_dag.py
├── data/
│   ├── input.csv
│   └── tmp_script.py
├── test_dag.py
└── helpers/
    └── dag_helper.py
```

You want Airflow to **only** load `my_etl_dag.py` as a DAG.
You want to ignore:

- All files in the `data/` directory
- Any files in `helpers/`
- All files starting with `test_`

Place this file as dags/.airflowignore

```
# Ignore test files
test_*

# Ignore data and helper directories
data/
helpers/
```

Now, only `my_etl_dag.py` is discovered as a DAG.

DAG dependency

- In Apache Airflow, a DAG dependency refers to the relationships that define the order in which tasks or even entire DAGs (Directed Acyclic Graphs) should run.

1. Within a Single DAG (Task Dependency)

- A **DAG** is a collection of tasks organized in a way that clearly shows their relationships and execution order.
- **Task dependencies** mean that one task will only run after another has successfully finished (or failed, depending on configuration).

```
task_a >> task_b  # task_b runs after task_a
task_b >> task_c  # task_c runs after task_b
```

Here, the order is: `task_a` → `task_b` → `task_c`.

This is called a **task-level dependency**.

Between Multiple DAGs (DAG Dependency)

- Sometimes you want one **entire DAG** to run after another DAG finishes.
- Airflow doesn't provide native cross-DAG dependencies, but you can use **sensors** like `ExternalTaskSensor` or trigger one DAG from another.

```
from airflow.sensors.external_task import ExternalTaskSensor

wait_for_dag1 = ExternalTaskSensor(
    task_id='wait_for_dag1',
    external_dag_id='dag1',
    external_task_id='final_task_of_dag1',
    dag=dag2,
)
```

This makes a task in `dag2` wait for a task in `dag1` to complete.

- This is called a **DAG-level dependency**.

Summary Table

Type	Example	What it Means
Task dependency	`task1 >> task2`	`task2` runs after `task1`
DAG dependency	ExternalTaskSensor	One DAG waits for another DAG/task

In Short

- **Task dependency**: Order of tasks within the same DAG.
- **DAG dependency**: Order between different DAGs.

DAG pausing, deactivation and deletion

1. Pausing a DAG

- You can **pause** a DAG from the UI or API.
- When paused, the DAG is still in Airflow but won’t run on schedule.
- You can still trigger it manually.
- Paused DAGs appear in the "Paused" tab; unpaused ones are in "Active."
- When paused, running tasks finish, and scheduled tasks wait until you unpause.

DAG pausing, deactivation and deletion

2. Deactivating a DAG

- **Deactivate** a DAG by **removing its file** from the `DAGS_FOLDER`.
- Airflow notices the file is gone and marks the DAG as deactivated.
- Historical data and metadata remain in the database.
- When you re-add the file, the DAG becomes active again with its history.
- Deactivated DAGs don’t show in the UI, except for their past run records.

DAG pausing, deactivation and deletion

3. Deleting a DAG

- **Delete metadata** using UI or API, but if the file stays in `DAGS_FOLDER`, it reappears after refresh.
- To **fully delete** a DAG and its history:
  1. Pause the DAG.
  2. Delete its metadata from the database (UI/API).
  3. Remove the DAG file from the `DAGS_FOLDER`.

Summary Table

Action	How to do it	Effect
Pause	UI/API	Stops schedule, manual runs possible
Deactivate	Remove file from folder	Disappears, keeps history, not in UI
Delete	Remove metadata & file	DAG and history are fully removed

pause/unpause activate/deactivate DAG

1. Pausing and Unpausing (Activating) DAGs via Airflow Web UI

To Pause a DAG:

1. **Open Airflow Web UI** (`http://<your-airflow-host>:8080` by default).
2. Go to the **DAGs** page (main dashboard).
3. Find your DAG in the list.
4. **Click the toggle button** in the “Paused” column (the switch icon).
    - If the toggle is **blue/on**, the DAG is active (unpaused).
    - If the toggle is **grey/off**, the DAG is paused.
5. **Click to turn it off** (grey) to pause the DAG.
    - Paused DAGs won’t run automatically but can be triggered manually.

pause/unpause activate/deactivate DAG

To Unpause (Activate) a DAG:

1. On the same DAGs page, **find your paused DAG**.
2. **Click the toggle** in the “Paused” column to turn it **blue/on**.
    - The DAG will now be scheduled and will run as per its schedule.

pause/unpause activate/deactivate DAG

2. Activating or Deactivating DAGs

- **Activate** a DAG:
  - Just make sure the DAG file exists in your `DAGS_FOLDER` and is not paused in the UI.
- **Deactivate** a DAG:
  - **You CANNOT deactivate from the UI.**
  - To deactivate, **remove the DAG file** from the `DAGS_FOLDER` on the server (Airflow will auto-detect and hide it).

Summary Table

Action	How to do it in UI
Pause DAG	Toggle "Paused" switch to off (grey)
Unpause DAG	Toggle "Paused" switch to on (blue)
Activate DAG	Ensure file is in `DAGS_FOLDER` and unpaused
Deactivate DAG	Remove file from `DAGS_FOLDER` (not in UI)

Tip:

- Pausing/unpausing is always done via the Web UI.
- Activating/deactivating (in the Airflow technical sense) is file-based and not controlled by the UI.

DAG Auto-pause

What is DAG Auto-pausing (Experimental)?

- This feature **automatically pauses a DAG** if it fails too many times in a row.
- It helps prevent broken or failing DAGs from running endlessly and consuming resources.

How does it work?

- There is a configuration option:
  `max_consecutive_failed_dag_runs_per_dag`
  - Set this in your `airflow.cfg` file.
  - Example:
    If you set `max_consecutive_failed_dag_runs_per_dag = 3`,
    Airflow will **auto-pause** any DAG that fails 3 times in a row.
- You can also override this per DAG by passing a DAG argument:
  `max_consecutive_failed_dag_runs`
  - This value (if set) will be used **instead of** the global setting for that specific DAG.

Set globally (airflow.cfg):

```
[scheduler]
max_consecutive_failed_dag_runs_per_dag = 3
```

Override in DAG definition:

```python
from airflow import DAG

with DAG(
    'example_dag',
    max_consecutive_failed_dag_runs=5,  # This DAG will auto-pause after 5 failed runs
    ...
) as dag:
    ...
```

Summary Table

Setting	Where to set	Effect
`max_consecutive_failed_dag_runs_per_dag`	`airflow.cfg` (global)	All DAGs auto-pause after N failures
`max_consecutive_failed_dag_runs`	In DAG code (per DAG)	Only this DAG auto-pauses after N failures

Note:

- This feature is **experimental**—test it before using in production.
- It is useful for auto-disabling problematic DAGs without manual intervention.

DAG Runs

DAG Run

A DAG Run is an instance of a DAG at a **specific point in time.**

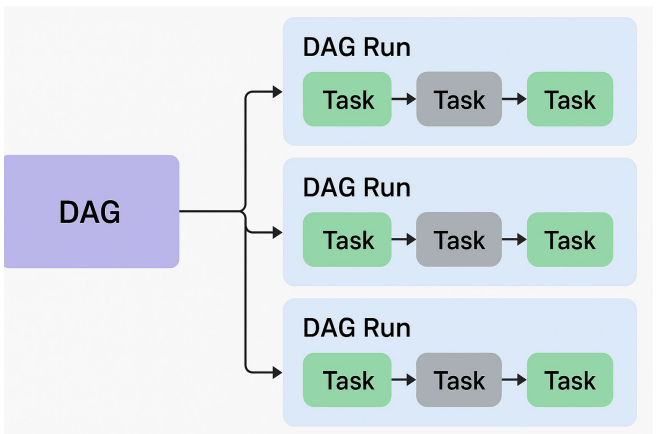
Every time a DAG is triggered (manually or scheduled), a new DAG Run is created.

The DAG Run executes all tasks defined inside the DAG.

The status of the DAG Run (e.g., success, failed) is determined by the states of its tasks.

Each DAG Run is independent of others — they don't affect each other.

You can have multiple DAG Runs of the same DAG running in parallel (e.g., for different dates or manual triggers).



DAG Run Status

A DAG Run status is set only after all tasks finish executing.

The final status depends on the states of the tasks and their dependencies.

A DAG Run reaches a terminal state when all tasks are in terminal states: success, failed, or skipped.

Status evaluation is based on leaf nodes (tasks with no downstream dependencies).

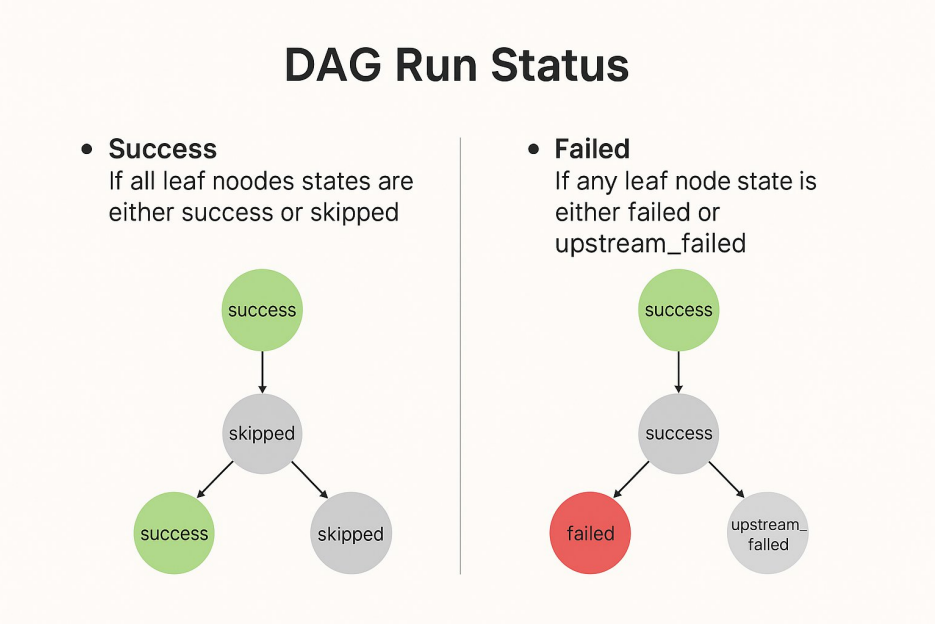
Terminal States for DAG Run

success

All leaf nodes are in state success or skipped.

failed

Any leaf node is in state failed or upstream_failed.



📌 Data Interval in Airflow

- ✅ Each **DAG run** is tied to a **data interval** — a specific **time range** it processes data for.
- ⏰ For example, with `@daily` schedule:
  - A DAG run on **2020-01-01** covers data from **2020-01-01 00:00** to **2020-01-02 00:00**.
- 🕓 A DAG run is **scheduled to start only after** its data interval **ends**.
  - So the run for **2020-01-01** actually **starts on 2020-01-02**.
- 🧠 This ensures the DAG run can **process all data** generated during the full interval.

📅 Important Terms:

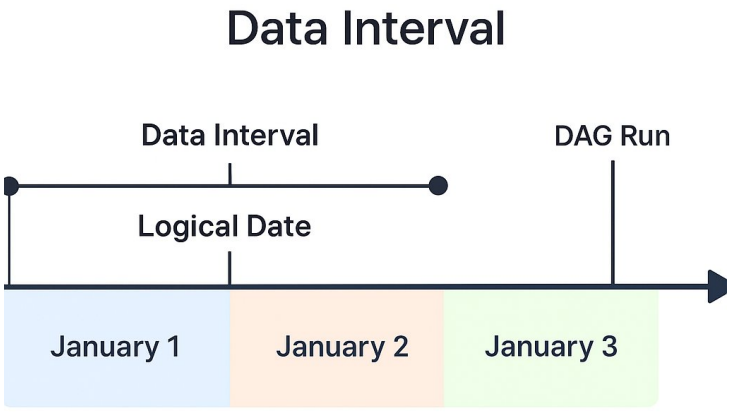
- **Data Interval** → The time window (start to end) the DAG run is responsible for processing.
- **Logical Date** (`execution_date` in older versions) →
  - Refers to the **start of the data interval**, **not the actual execution time**.

🚦 start_date Behavior:

- The `start_date` in the DAG defines **when the first data interval starts**,
  but **not** when the DAG actually runs.
- The first DAG run is scheduled **after the first data interval ends**,
  i.e., **one interval after** `start_date`.

✅ Example for Clarity:

DAG Config	Value
`schedule_interval`	`@daily`
`start_date`	`2024-01-01 00:00:00`
➡️ First DAG Run	for **2024-01-01**
⏱️ Actually Runs On	**2024-01-02 00:00:00**



📌 Catchup in Airflow (

- 🗓️ A DAG with a `start_date`, optional `end_date`, and a schedule (e.g. `@daily`) defines **a series of time intervals**.
- 🔁 The **scheduler turns each interval into a separate DAG Run** when the DAG is activated.

⚙️ Default Behavior (`catchup_by_default=False`)

- By default, the **scheduler creates only the latest DAG run** for the most recent interval.
- It **does not backfill** missed intervals unless explicitly configured.

✅ What is `catchup=True`?

- If you set `catchup=True` in the DAG definition:
  - Airflow **creates DAG runs for every missed interval** since `start_date`.
  - This is known as **"Catchup"** or **"backfilling"**.

❌ When to Avoid Catchup

- If your DAG logic **uses the current time (`now()`) instead of the logical execution time**, it may process incorrect data during catchup.
- In such cases, it's **recommended to disable catchup** using:

catchup=False

✅ When to Use Catchup

- If your DAG:
  - Is **time-partitioned** (e.g., processes data for each day/hour)
  - Uses `data_interval_start` or `logical_date`
  - Can safely process historical intervals
- Then you should enable:

catchup=True

🧠 Quick Comparison

Scenario	`catchup=True`	`catchup=False`
Backfill missed intervals	✅ Yes	❌ No
Create only latest DAG run	❌ No	✅ Yes
Default behavior (Airflow)	❌ No (by config default)	✅ Yes (if not changed)

📌 Backfill Explained Simply

- 🔄 **Backfill** means running a DAG for **past (historical) dates** that were not previously run.
- 📅 Example:
  - DAG has `start_date = 2024-11-21`
  - You need output from **2024-10-21 to 2024-11-20**
  - You can **backfill** this range manually
- 🛠️ Backfill is useful when:
  - Data for earlier dates is now available
  - A new DAG is added but historical runs are needed
  - A task or DAG was missed or failed in the past

```
airflow backfill create --dag-id <DAG_ID> \
                        --start-date <YYYY-MM-DD> \
                        --end-date <YYYY-MM-DD>
```

- This triggers **DAG runs for all intervals** between the start and end date.

⚠️ Notes

- Make sure your DAG logic handles historical intervals properly (i.e., uses `data_interval_start` not `datetime.now()`).
- If `catchup=True` is set, backfilling may also happen automatically on first DAG activation.

🔁 Catchup vs Backfill – Key Differences

Feature	Catchup	Backfill
📌 **Definition**	Auto-creation of DAG runs for **missed past intervals** when a DAG is turned on or scheduled	Manual triggering of DAG runs for **past dates** via CLI or API
⚙️ **How it happens**	Controlled by `catchup=True` in DAG definition	Executed manually using `airflow backfill create`
🧠 **When used**	When a DAG is first scheduled, or re-enabled after being paused	When historical data needs to be processed retroactively
📆 **Execution scope**	From `start_date` up to **current time**, unless restricted	From any specified `--start-date` to `--end-date`
🔌 **Manual/Automatic**	**Automatic** (handled by the scheduler)	**Manual** (user-triggered)
🛠️ **Control granularity**	Less granular – covers all missed runs	More granular – you choose exact date range
⚠️ **DAG must support**	Time-partitioned, interval-based logic (e.g., using `data_interval_start`)	Same requirement – must not rely on `datetime.now()`

🧠 Think of it Like This:

- **Catchup** = "Let Airflow automatically handle past missed runs"
- **Backfill** = "I want to manually re-run for specific past intervals"

🔁 Re-running Tasks in Airflow

- ✅ If a task **fails** during a DAG run, you can **re-run it** after fixing the issue.
- 🔍 First, check the **logs** to identify and fix the root cause of the failure.
- 🧹 Then, go to **Tree View** or **Graph View** in the Airflow UI → click the **failed task** → click **Clear**.
- 🔄 **Clearing a task**:
  - **Increments** the task's `try_number`
  - Sets `max_tries = 0`
  - Sets its **state to** `None`
  - Triggers the **executor to re-run the task**

🛠️ Re-run Options in UI

When clearing, you can choose additional scope:

Option	What it Does
✅ **Past**	Re-runs all past instances of the task (before the latest DAG run)
✅ **Future**	Re-runs all future instances of the task (after the latest DAG run)
✅ **Upstream**	Re-runs this task and all **upstream (previous)** tasks
✅ **Downstream**	Re-runs this task and all **downstream (next)** tasks
✅ **Recursive**	Re-runs **parent and child DAG tasks** (in SubDAGs or external DAGs)
✅ **Failed**	Re-runs **only the failed tasks** in the most recent DAG run

💡 Use Cases

- A task failed due to a temporary DB/network issue → fix it → **Clear and re-run**
- Task logic changed → Clear and re-run the task and its **downstream**
- A DAG was partially successful → use **"Failed"** to retry only failed tasks

Re-run Tasks

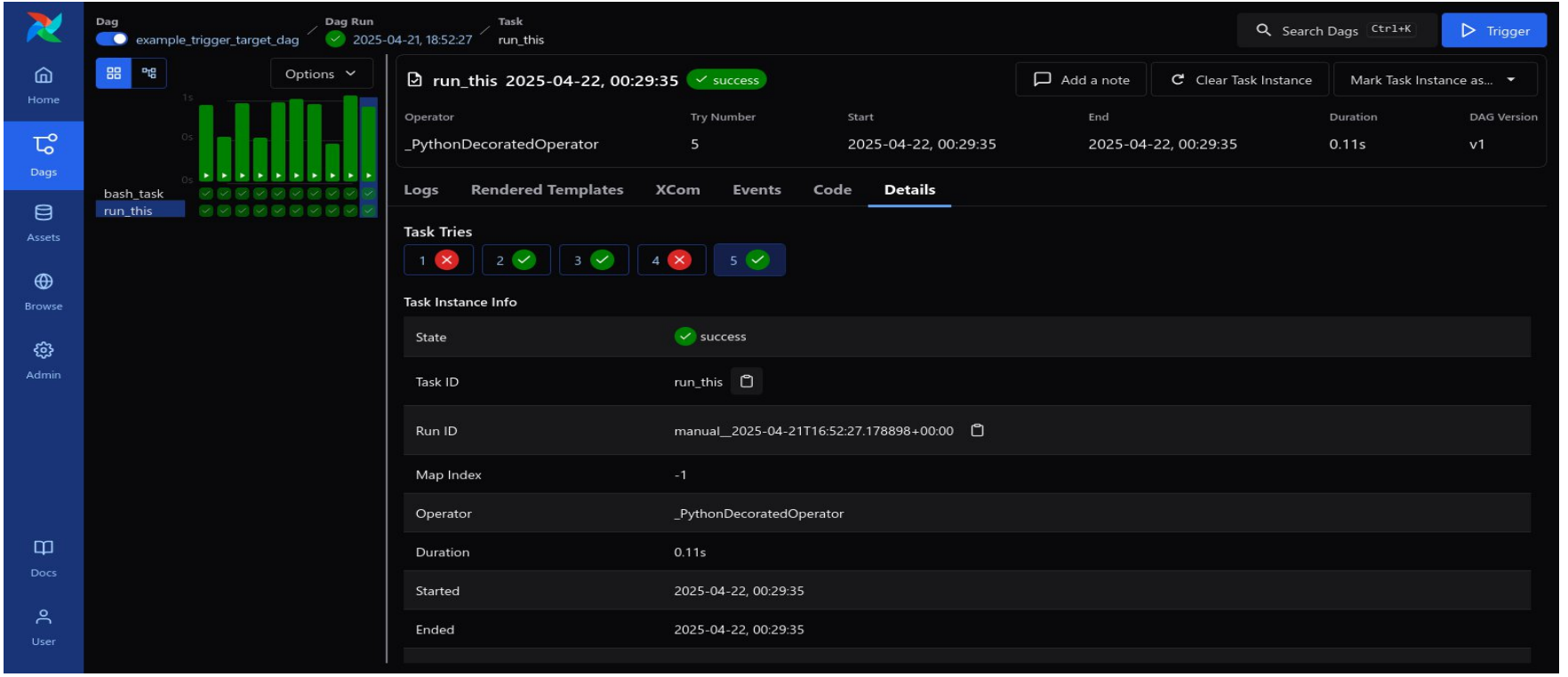
For the specified dag_id and time interval, the command clears all instances of the tasks matching the regex.

```
airflow tasks clear dag_id \
    --task-regex task_regex \
    --start-date START_DATE \
    --end-date END_DATE
```

```
airflow tasks clear --help
```

Task Instance History

When a task instance retries or is cleared, the task instance history is preserved. You can see this history by clicking on the task instance in the Grid view.



External Triggers

DAG Runs can also be created manually through the CLI.

airflow dags trigger --logical-date logical_date run_id

DAG Runs created externally to the scheduler get associated with the trigger's timestamp and are displayed in the UI alongside scheduled DAG runs.

The logical date passed inside the DAG can be specified using the -e argument.

The default is the current date in the UTC timezone.

We can also manually trigger a DAG Run using the web UI (tab **Dags** -> column **Links** -> button **Trigger Dag**)

Passing Parameters when triggering dags

- When triggering a DAG from the CLI, the REST API or the UI, it is possible to pass configuration for a DAG Run as a JSON blob.

```
import pendulum

from airflow.sdk import DAG
from airflow.providers.standard.operators.bash import
BashOperator

dag = DAG(
    "jp_parameterized_dag",
    schedule=None,
    start_date=pendulum.datetime(2021, 1, 1, tz="UTC"),
    catchup=False,
)

parameterized_task = BashOperator(
    task_id="parameterized_task",
    bash_command="echo value: {{ dag_run.conf['conf1'] }}",
    dag=dag,
)
```

Template:
`{{ dag_run.conf['conf1'] }}`
is a **Jinja templated** value

📦 How It Works

This DAG is meant to be **triggered manually** like this:

```
airflow dags trigger example_parameterized_dag \
  --conf '{"conf1": "Hello World"}'
```

```
value: Hello World
```

✅ Use Case

This is a **parameterized DAG**, useful when:

- You want to run the same logic but **with different inputs**
- E.g., load data for a specific customer, file name, or date

Tasks

- A **Task** is the **smallest unit of execution** in Airflow.
- Tasks are placed inside a **DAG**, and are connected using **upstream/downstream dependencies** to define their **execution order**.

🧱 Types of Tasks

1. **Operators**
   - Predefined templates for common actions
   - Example: `BashOperator`, `PythonOperator`
2. **Sensors**
   - Special kind of operator that **waits for an external event**
   - Example: `FileSensor`, `S3KeySensor`
3. **TaskFlow @task-decorated functions**
   - Your own **custom Python functions** converted into tasks
   - Part of the TaskFlow API

🧬 Behind the Scenes

- All task types are subclasses of `BaseOperator`
- **Operators** and **Sensors** are like **templates**
- When you use them in a DAG, you are actually **instantiating a Task**

Relationships

- The key part of using Tasks is defining how they relate to each other - their *dependencies*, or as we say in Airflow, their *upstream* and *downstream* tasks.
- We declare your Tasks first, and then you declare their dependencies second.

There are two ways of declaring dependencies - using the `>>` and `<<` (bitshift) operators:

```
first_task >> second_task >> [third_task, fourth_task]
```

```
first_task.set_downstream(second_task)
third_task.set_upstream(second_task)
```

By default, a Task will run when all of its upstream (parent) tasks have succeeded, but there are many ways of modifying this behaviour to add branching, to only wait for some upstream tasks, or to change behaviour based on where the current run is in history.

Task Instances

- As DAG is instantiated into a each time it runs, the *tasks under a DAG are instantiated into Task Instances*.
- An instance of a Task is a specific run of that task for a given DAG (and thus for a given data interval).
- Possible states for a Task Instance are

  - `none` : The Task has not yet been queued for execution (its dependencies are not yet met)
  - `scheduled` : The scheduler has determined the Task's dependencies are met and it should run
  - `queued` : The task has been assigned to an Executor and is awaiting a worker
  - `running` : The task is running on a worker (or on a local/synchronous executor)
  - `success` : The task finished running without errors
  - `restarting` : The task was externally requested to restart when it was running
  - `failed` : The task had an error during execution and failed to run
  - `skipped` : The task was skipped due to branching, LatestOnly, or similar.
  - `upstream_failed` : An upstream task failed and the Trigger Rule says we needed it
  - `up_for_retry` : The task failed, but has retry attempts left and will be rescheduled.
  - `up_for_reschedule` : The task is a Sensor that is in `reschedule` mode
  - `deferred` : The task has been deferred to a trigger
  - `removed` : The task has vanished from the DAG since the run started

'task' should flow from none, to scheduled, to queued, to running, and finally to success

Operators

- An Operator is conceptually a template for a predefined , that you can just define declaratively inside your DAG

```
with DAG("my-dag") as dag:
    ping = HttpOperator(endpoint="http://example.com/update/")
    email = EmailOperator(to="admin@example.com", subject="Update complete")

    ping >> email
```

- **BashOperator** - executes a bash command
- **PythonOperator** - calls an arbitrary Python function

- **EmailOperator**
- **HttpOperator**
- **SQLExecuteQueryOperator**
- **DockerOperator**
- **HiveOperator**
- **S3FileTransformOperator**
- **PrestoToMySqlOperator**
- **SlackAPIOperator**

Jinja Templating

Suppose we want to pass the start of the data interval as an environment variable to a Bash script using the BashOperator:

```
# The start of the data interval as YYYY-MM-DD
date = "{{ ds }}"
t = BashOperator(
    task_id="test_env",
    bash_command="/tmp/test.sh ",
    dag=dag,
    env={"DATA_INTERVAL_START": date},
)
```

Here, {{ ds }} is a templated variable, and because the env parameter of the BashOperator is templated with Jinja, the data interval's start date will be available as an environment variable named DATA_INTERVAL_START in your Bash script.

E.g jp_jinja_template-1.py

airflow dags trigger jp_jinja_template-1.py --conf '{"name": "Jagjit"}'

Sensors

📌 What Are Sensors?

- Sensors are a **special type of Operator** used to **wait for something to happen**.
- Common use cases:
  - Wait for a **file to appear**
  - Wait for a **database update**
  - Wait for a **time window** or **external event**

⏳ How Sensors Work

- A sensor **waits** and only **succeeds when the condition is met**.
- Once successful, the sensor **unblocks downstream tasks**.

⚙️ Sensor Modes

Mode	Behavior
`poke`	🟡 Default mode – sensor **holds a worker slot** the entire time
`reschedule`	🟢 Sensor **releases the worker slot** between checks

- Use `poke` for **frequent checking** (e.g., every 1–5 seconds)
- Use `reschedule` for **less frequent polling** (e.g., every 60+ seconds)

🧰 Types of Built-in Sensors

Airflow includes many pre-built sensors, like:

- `FileSensor`
- `S3KeySensor`
- `ExternalTaskSensor`
- `HttpSensor`
- `TimeDeltaSensor`

You can also find more via **Airflow providers**.

E.g

jp_sensor_example-1.py (on ubuntu 214)

jp_aws_s3_eg.py (on deb 33)

jp_aws_sensor_s3_eg.py (on deb 33)

XCom

🔄 What is XCom in Apache Airflow?

XCom stands for **Cross-Communication** in Airflow. It's a mechanism that allows **tasks to share small pieces of data** with each other.

Think of it like a **shared memory** that one task can `push` data into and another task can `pull` data from.

📌 Why Use XCom?

- To **pass data** between tasks (e.g., results of a Python function)
- To **drive logic** in downstream tasks based on upstream results
- To **avoid storing temporary files** or using external storage for small data

🧠 How XCom Works

- XComs are **key-value pairs** stored in Airflow's metadata database.
- Every XCom has:
  - `key` – a string (e.g. `"result"`)
  - `value` – the actual data (must be serializable)
  - `task_id` – who pushed it
  - `dag_id` and `execution_date` – when and where it was created

Ubuntu 214-

jp_xcom_eg-1.py

🔧 Basic XCom Methods

Action	Method
Push value	`ti.xcom_push(key, value)`
Pull value	`ti.xcom_pull(task_ids, key)`

What is Taskflow in Airflow?

Taskflow API is a way to write your DAGs (workflows) using plain Python functions.
It lets you create tasks as Python functions (decorated with `@task`), and pass data between them directly as arguments—making your code more readable, modular, and easier to maintain.

Traditional Airflow: You create tasks using Operators (e.g., `PythonOperator`, `BashOperator`), and passing data between tasks is more manual (using XComs).

Taskflow: You just write Python functions, Airflow handles data passing for you, and your DAGs look like regular Python code.

Why use Taskflow?

- **Cleaner, more Pythonic code**
- **Easier data passing** between tasks (function return values become inputs to next tasks)
- **Better debugging and testing** (since tasks are just functions)

How does it work?

You decorate your functions with `@task` (from `airflow.decorators`).

You can then chain them together by passing outputs directly as inputs.

Taskflow eg

- **Use case**:
Let's say you want to create a DAG that:
- Gets a list of numbers
- Doubles each number

Prints the final list

Ubuntu 214

jp_taskflow_eg-1.py

How it works:

- `get_numbers()` returns a list.
- `double_numbers()` takes the output of `get_numbers` as input, doubles each value.
- `print_result()` prints the output.

The magic: **Airflow handles all the data passing between these tasks.**

You get proper task dependencies in the UI, and can view/log outputs.

Summary Table

Feature	Classic Airflow	Taskflow API
Define a Task	PythonOperator	`@task`-decorated function
Pass Data	XComs	Return values, arguments
Readability	More boilerplate	More Pythonic, concise
Testing Functions	Less direct	Can test functions directly

When should you use Taskflow?

- Whenever your workflow logic fits naturally as chained Python functions
- When you want clear, modular code with built-in data passing

What is Logging in Airflow?

Logging in Airflow means capturing all the information about what's happening when your DAGs and tasks run—success, errors, warnings, outputs, etc.
This helps in:

- Debugging failures
- Auditing workflow execution
- Monitoring task progress

Airflow logging writes log files for each DAG run and task instance, by default to local disk. Logs can also be sent to remote storage like S3, GCS, or Elasticsearch.

How Logging Works in Airflow

- Every time a task runs, Airflow writes log messages.
- You can view logs in the Airflow **web UI** by clicking on a DAG > Task Instance > “Log”.
- You can also add your own log messages inside your Python functions using Python’s standard `logging` module.

Default log location:

- Local executor: `~/airflow/logs/`
- Inside the logs directory: `/logs/{dag_id}/{task_id}/{execution_date}/...`

You can configure logging settings in `airflow.cfg` under `[logging]`.

Logging Use Case

- Let’s say you have a DAG that processes some numbers, and you want to log:
- When the process starts and ends
- The numbers being processed

E.g ubuntu 214

- jp_logging_eg_1.py

How to View Logs

1. Go to Airflow Web UI → DAGs → Find your DAG (`logging_taskflow_dag`)
2. Trigger the DAG (if `schedule=None` , you can trigger manually)
3. Click the task (`process_numbers`)
4. Click **Log** to see the log output

You’ll see your `logger.info` messages, and also Airflow’s own logs (when task started, ended, etc.).

What is an Executor in Airflow?

An **executor** in Airflow is the component that actually runs your tasks.

- It decides **how** and **where** your task code will be executed.
- It connects the **scheduler** (which decides *what* to run and *when*) to the actual task execution engine (*how* to run).

Think of it as the "worker manager":

- The executor tells Airflow:
  “Run this task now on this worker/process/thread/machine.”

Types of Executors (Airflow 3.0.2)

- **SequentialExecutor**
  - Runs one task at a time (no parallelism).
  - Default for new installs and for debugging/testing only.
- **LocalExecutor**
  - Runs tasks in parallel, on the same machine where scheduler runs.
  - Good for small/medium setups.
- **CeleryExecutor**
  - Runs tasks on a distributed set of workers (can be multiple machines).
  - Best for production/scaling out.
- **KubernetesExecutor**
  - Each task runs as a pod in a Kubernetes cluster.

ubuntu214
jp_executor_eg-1.py

Let’s say you have a DAG with 3 independent tasks:

```python
from airflow import DAG
from airflow.operators.bash import BashOperator
from datetime import datetime

with DAG(
    'example_executor_dag',
    start_date=datetime(2024, 1, 1),
    schedule_interval=None,
    catchup=False,
) as dag:
    t1 = BashOperator(
        task_id='task1',
        bash_command='echo "Task 1 running"'
    )
    t2 = BashOperator(
        task_id='task2',
        bash_command='echo "Task 2 running"'
    )
    t3 = BashOperator(
        task_id='task3',
        bash_command='echo "Task 3 running"'
    )
```

If you use SequentialExecutor (default in some setups):

- Only one task (t1, t2, or t3) runs at a time, even if your machine has many CPU cores.
- Next task starts **only after previous finishes**.

If you use LocalExecutor (set in `airflow.cfg`):

- All three tasks can run **in parallel** (at the same time), up to your parallelism limits and available cores.
- This means the DAG finishes much faster!

How to change the executor?

1. **Open** `airflow.cfg`
2. Find the `[core]` section, then set:

```ini
executor = LocalExecutor
```

3. Restart the Airflow webserver and scheduler.

Executor	How it works	When to use
SequentialExecutor	One task at a time	Debug, development
LocalExecutor	Many tasks in parallel	Small/medium, single server
CeleryExecutor	Distributed, many workers	Production, scale-out
KubernetesExecutor	Each task in a pod	Cloud-native, Kubernetes

In short:

- **Executor** = "How do we run tasks?"
- Choose **Sequential** for testing, **Local** for small real use, **Celery/Kubernetes** for big setups.
- For beginners, **LocalExecutor** is a great starting point to see parallelism in action.