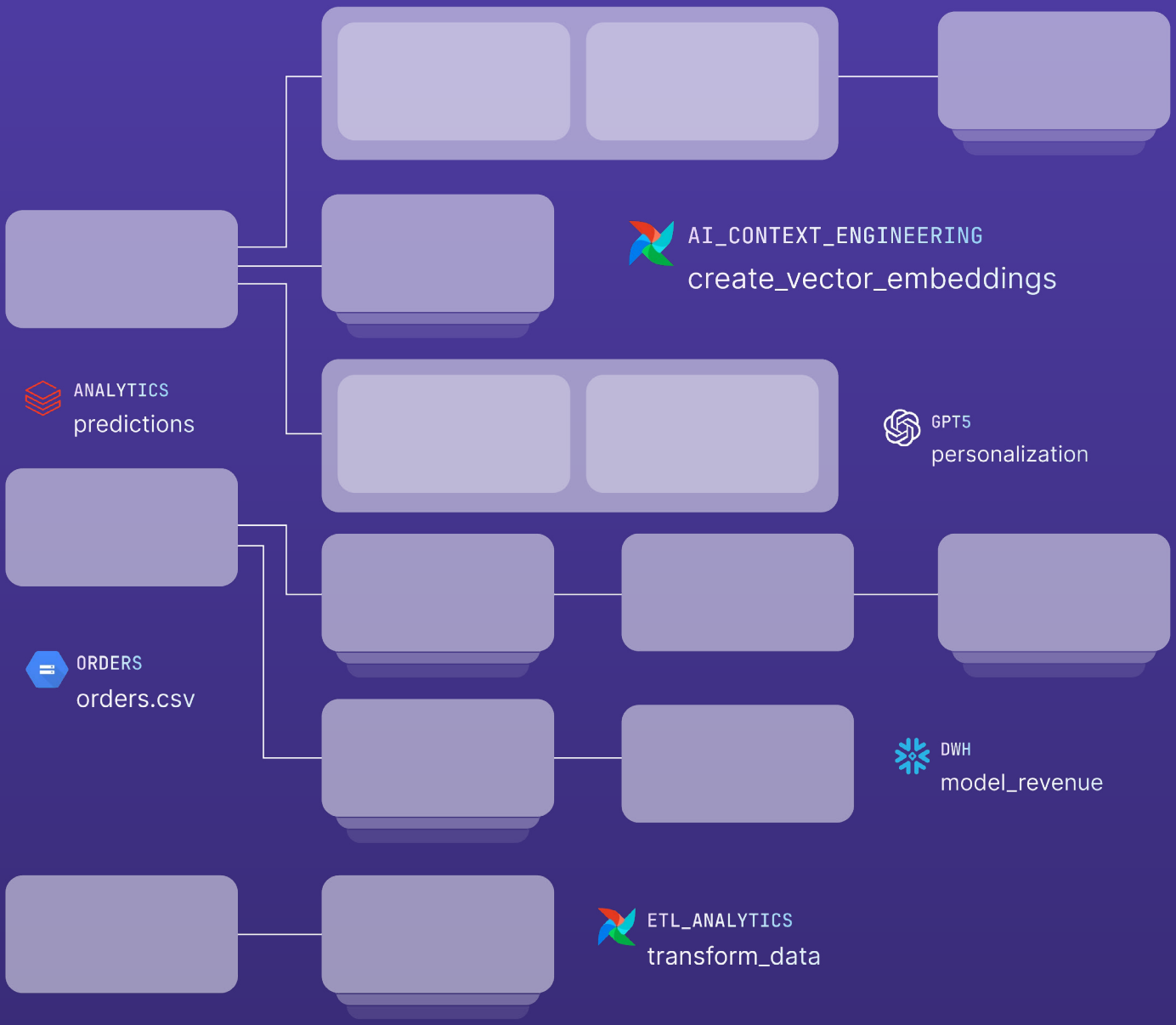

The Definitive Guide to Apache Airflow® 3 Dags

ASTRONOMER

Editor's Note

Welcome to the Ultimate guide to Apache Airflow® Dags, brought to you by the Astronomer team. This eBook covers everything you need to know to work with Dags, from the building blocks they consist of to best practices for writing them, making them dynamic and adaptable, testing and debugging them, and more. It's a guide written by practitioners, for practitioners.

For more information check out:

- Astronomer webinars: 1h deep-dives into Airflow and Astro related topics.
- Astronomer Academy: Structured Airflow courses and certifications.
- Astro CLI: A free and open-source tool to quickly spin up a local Airflow development environment running in Docker.
- Free trial of Astro: The best place to run Airflow in production.

Introduction to Dags

1

KEY CONCEPTS

- **Dag**: In Airflow, a Dag is an individual pipeline. Dags are written in Python, and instantiated with a schedule which defines when the Dag runs.
- **Dag run**: an instance of a Dag running at a specific point in time.
- **Task**: Dags consist of tasks, each performing one step in the pipeline.
- **Task instance**: an instance of a task running at a specific point in time.

What is a Dag?

In Airflow, a *Dag* is a data pipeline or workflow. Dags are the main organizational unit in Airflow; they contain a collection of tasks and dependencies that you want to execute on a schedule.

A Dag is defined in Python code and visualized in the Airflow UI. Dags can be as simple as a single task or as complex as hundreds or thousands of tasks with complicated dependencies.

The following screenshot shows a complex Dag graph in the Airflow UI. After reading this chapter, you'll be able to understand the elements in this graph, as well as know how to define Dags and use Dag parameters.

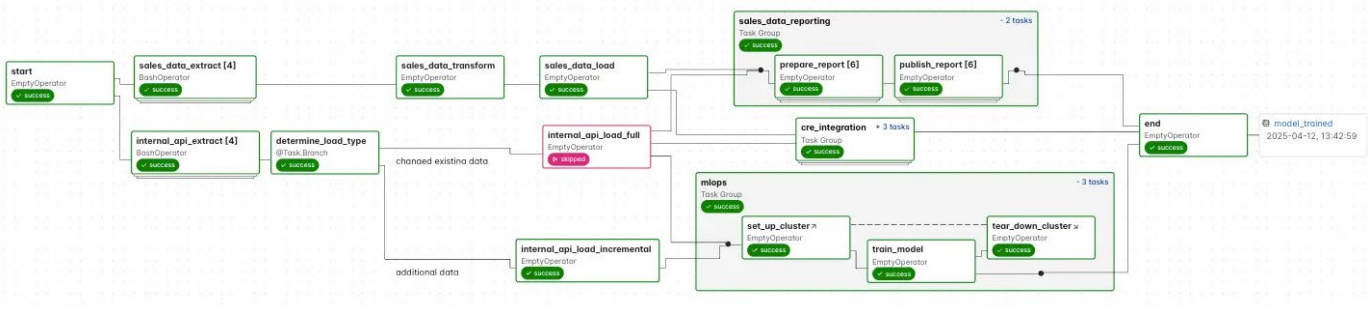

Figure 1: *Screenshot of the Airflow UI showing a complex Dag run graph.*

There are two requirements for the structure of a Dag in Airflow: there needs to be a clear direction between tasks and there can be no circular dependencies. The Dag in the screenshot below fulfills these two requirements:

- **Clear direction:** The direction of Dags is from left to right by default. In this Dag, first `task_1` and `task_2` run in parallel then, after `task_2` finishes successfully `task_3` runs. `task_4` and `task_5` depend on `task_3` and start running as soon as `task_3` has completed successfully. Lastly, `task_6` runs after successful completion of `task_5` and `task_4`.
- **No circular dependencies**: No task in this Dag is upstream of a task it ultimately depends on, a dependency like that would cause an error in Airflow, not allowing the Dag to run. An example of a circular dependency would be if `task_3` would be made dependent on successful completion of `task_6`.

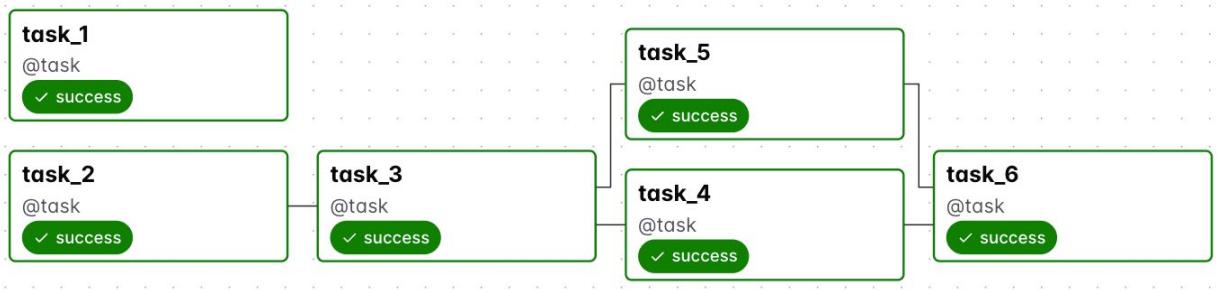

Figure 2: *Screenshot of the Airflow UI showing a Dag run graph with 6 tasks.*

The term **Dag** (**D**irected **A**cyclic **G**raph) originated in mathematics, where it describes a structure consisting of nodes and edges that is both **directed** (there is a clear flow of direction between nodes) and **acyclic** (there are no circles in the flow between nodes).

Aside from these two requirements, a Dag can be as simple or as complicated as you need! You can define tasks that run in parallel or sequentially, implement conditional branches, and visually group tasks together in task groups. A Dag can also consist of only one task, which is the case when using the asset-oriented approach to writing Dags.

Each task in a Dag performs one unit of work. Tasks can be anything from a simple Python function to a complex data transformation or a call to an external service.

In Airflow, Dags:

- can be written using two different approaches: task-oriented or asset-oriented, the latter of which is new to Airflow 3
- contain tasks defined using building blocks: Python classes (incl. operators, sensors, decorators and deferrable operators)
- show dependencies between the building blocks
- are scheduled based on time or updates to assets

The following screenshot shows a simple Dag graph.

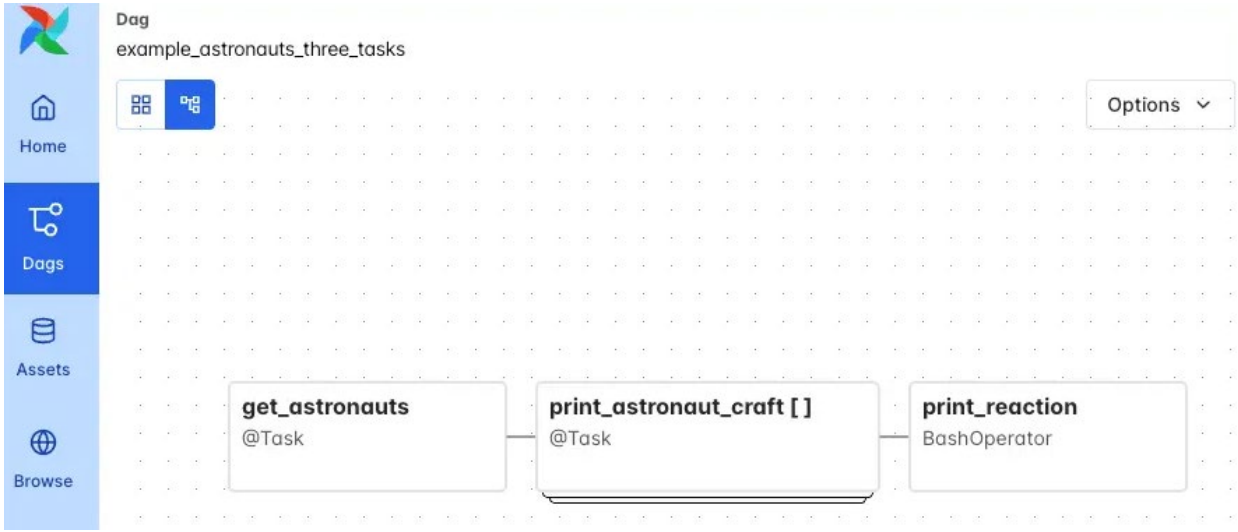

Figure 3: *Screenshot of the Airflow UI showing a Dag graph with 3 tasks.*

By default, Dags are read from left to right and all upstream tasks need to be successful for a downstream task to run. In this Dag there are 3 sequential tasks:

- `get_astronauts`: Has no upstream tasks and one downstream task, `print_astronaut_craft`
- `print_astronaut_craft`: Has `get_astronauts` as its upstream task and `print_reaction` as its downstream task
- `print_reaction`: Has `print_astronaut_craft` as its upstream task and no downstream tasks

Why use Airflow to run your pipelines?

Apache Airflow® is a platform for programmatically authoring, scheduling, and monitoring workflows. It is especially useful for creating and orchestrating complex data pipelines.

Data orchestration sits at the heart of any modern data stack and provides elaborate automation of data pipelines. With orchestration, actions in your data pipeline become aware of each other and your data team has a central location to monitor, edit, and troubleshoot their workflows.

Airflow provides many benefits, including:

- **Data pipelines as code:** code-based pipelines are dynamic, extensible and you can manage them using software-engineering best practices like version control and CICD.
- **Tool agnosticism**: Airflow can connect to any application in your data ecosystem that allows connections through an API. Prebuilt operators exist to connect to many common data tools.
- **High extensibility**: Since Airflow pipelines are written in Python, you can build on top of the existing codebase and extend the functionality of Airflow to meet your needs. Anything you can do in Python, you can do in Airflow.
- **Infinite scalability**: Given enough computing power, you can orchestrate as many processes as you need, no matter the complexity of your pipelines.
- **Dynamic data pipelines**: Airflow offers the ability to create dynamic tasks to adjust your workflows based on the data you are processing at runtime.
- **Active OSS community**: With millions of users and thousands of contributors, Airflow is here to stay and grow. Join the Airflow Slack to become part of the community.
- **Observability**: The Airflow UI provides an immediate overview of all your data pipelines and can provide the source of truth for workflows in your whole data ecosystem.

What is a Dag run?

A *Dag run* is an instance of a Dag running at a specific point in time. A *task instance* is an instance of a task running at a specific point in time.

Each Dag run has a unique `dag_run_id` and contains one or more task instances. The history of previous Dag runs is stored in the Airflow metadata database.

In the Airflow UI, you can view previous runs of a Dag in the grid view and select individual Dag runs by clicking on their respective duration bar. You can view the graph of a Dag in the graph view and select different previous runs using the `Options` menu.

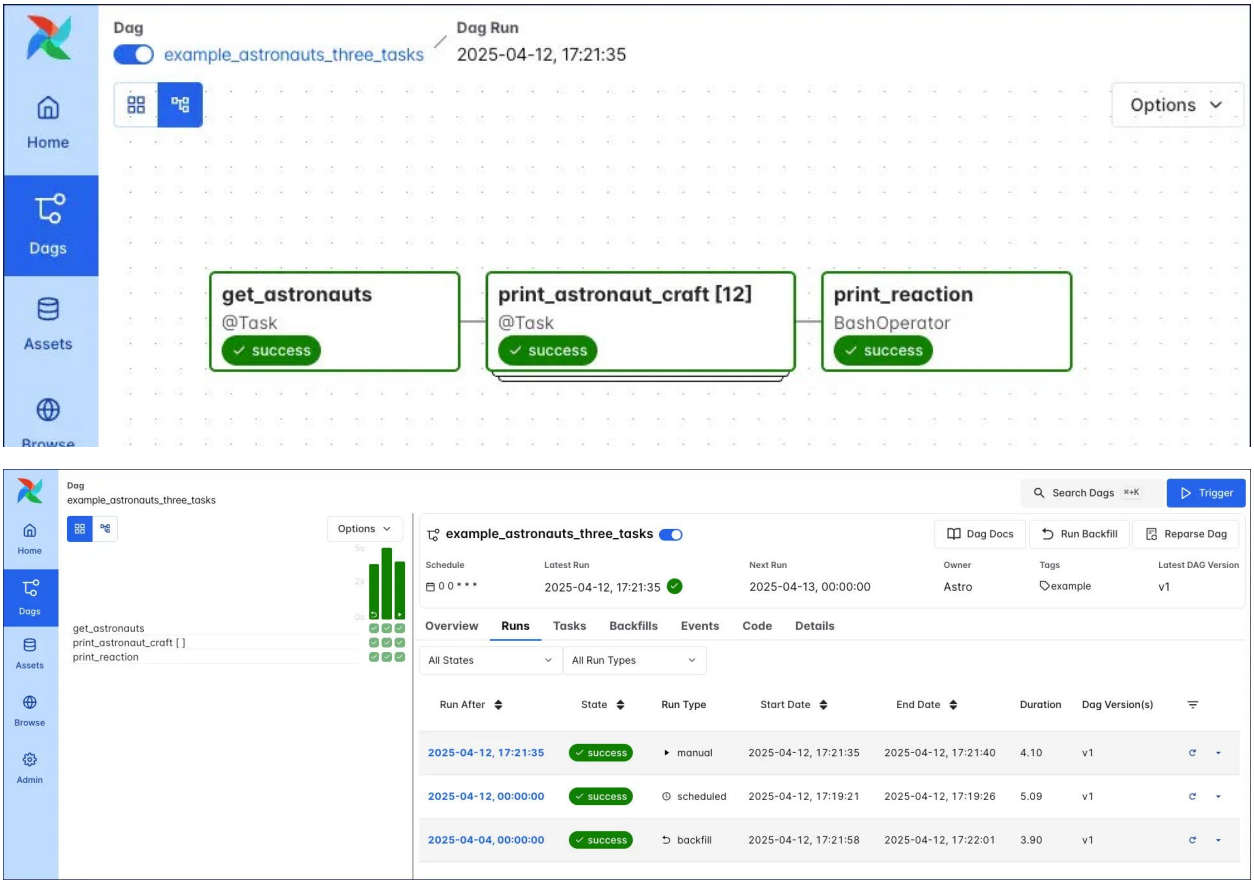

Figure 4: *2 Screenshots of the Airflow UI showing a Dag run with 3 tasks. The first screenshot shows the graph view, the second screenshot the grid view of the same Dag.*

Dag run properties

A Dag run graph in the Airflow UI contains information about the Dag run, as well as the status of each task instance in the Dag run. The following screenshot shows the same Dag as in the previous section, but with annotations explaining the information shown in the Airflow UI.

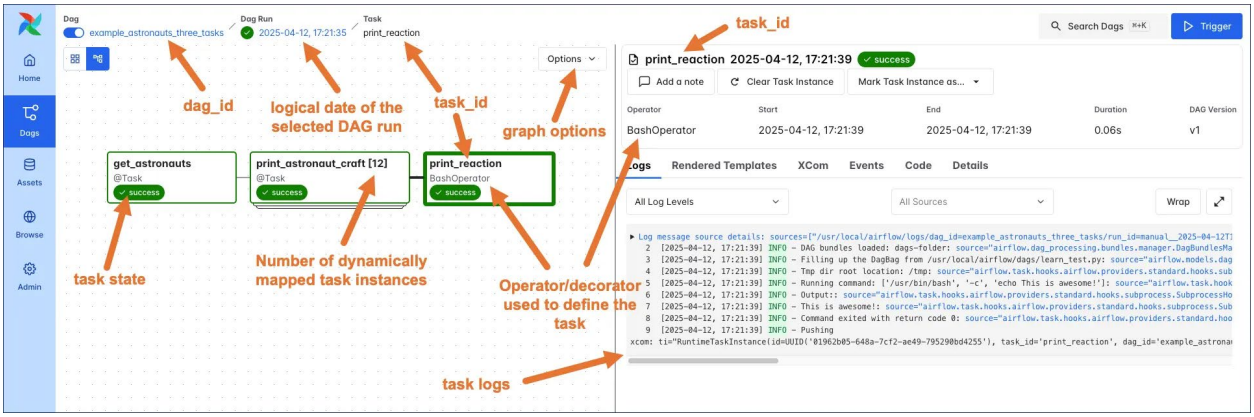

Figure 5: *Screenshot of the Airflow UI showing an annotated Dag run graph.*

- `dag_id`: The unique identifier of the Dag.
- `logical_date`: The point in time for which the Dag run is scheduled. This date and time is not necessarily the same as the actual moment the Dag run is executed. See Scheduling Dags for more information.
- `task_id`: The unique identifier of the task.
- **task state**: The status of the task instance in the Dag run. Possible states are running, success, failed, skipped, restarting, up_for_retry, upstream_failed, queued, scheduled, none, removed, deferred, and up_for_reschedule, they each cause the border of the node to be colored differently. See the OSS documentation on task instances for an explanation of each state.

A Dag can be triggered in 4 different ways:

- **Scheduled**: A Dag can be triggered based on the Dag's defined schedule. This is the default method for running Dags.
- **Manual**: A common way to trigger Dags in development is manually by a user using the Airflow UI or the Airflow CLI. In many production implementations Dags are triggered on-demand using the Airflow REST API. Manually triggered Dag runs include a play icon on the Dag run duration bar.
- **Asset triggered**: Dags can also run on data-aware/data-driven schedules using assets. Dag runs triggered by an asset include an asset icon on their Dag run duration bar.

- **Backfill**: The first Dag run was created using a backfill. Backfilled Dag runs include a curved arrow on their Dag run duration bar. In Airflow 3, you can create backfills using the Airflow REST API, Airflow UI or the Airflow CLI.

A Dag run can have the following statuses:

- **Queued**: The time after which the Dag run can be created has passed but the scheduler has not created task instances for it yet.
- **Running**: The Dag run is eligible to have task instances scheduled.
- **Success**: All task instances are in a terminal state (`success`, `skipped`, `failed` or `upstream_failed`) and all leaf tasks (tasks with no downstream tasks) are either in the state `success` or `skipped`. In the previous screenshot, all four Dag runs were successful.
- **Failed**: All task instances are in a terminal state and at least one leaf task is in the state `failed` or `upstream_failed`.

Complex Dag runs

When you start writing more complex Dags, you will see additional Airflow features that are visualized in the Dag run graph. The following screenshot shows the same complex Dag as in the overview but with annotations explaining the different elements of the graph. Don't worry if you don't know about all these features yet - you will learn about them as you become more familiar with Airflow.

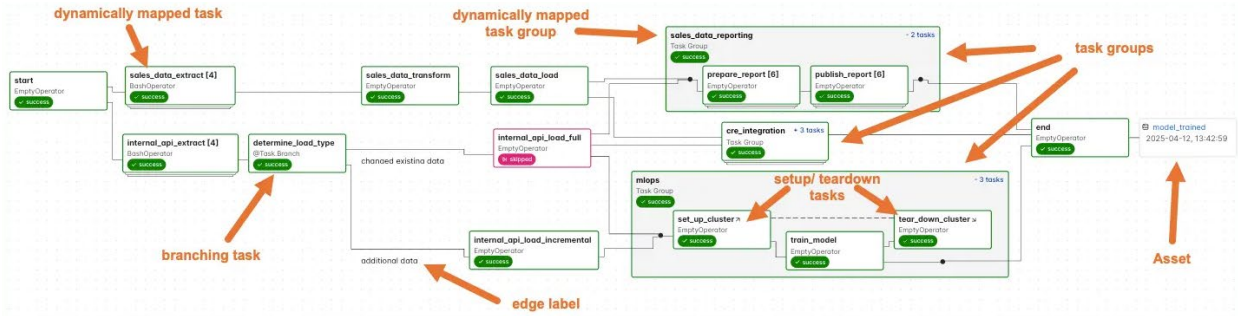

Figure 6: *Screenshot of the Airflow UI showing an annotated complex Dag run graph.*

Some more complex features visible in this Dag graph are:

- **Dynamically mapped tasks**: A dynamically mapped task is created dynamically at runtime based on user-defined input. The number of dynamically mapped task instances is shown in brackets (`[ ]`) behind the task ID.
- **Branching tasks**: A branching task creates a conditional branch in the Dag.

- **Edge labels**: Edge labels appear on the edge between two tasks. These labels are often helpful to annotate branch decisions in a Dag graph.
- **Task groups**: A task group is a tool to logically and visually group tasks in an Airflow Dag.
- **Setup/teardown tasks**: When using Airflow to manage infrastructure, it can be helpful to define tasks as setup and teardown tasks to take advantage of additional intelligent dependency behavior. Setup and teardown tasks appear with diagonal arrows next to their task IDs and are connected with a dotted line. See the Use setup and teardown tasks in Airflow guide in the Astronomer documentation for more information.
- **Assets**: Assets are shown in the Dag graph. If a Dag is scheduled on an asset, it is shown upstream in the Dag graph. If a task in the Dag updates an asset, it is shown after the respective task as in the previous screenshot.

You can learn more about how to set complex dependencies between tasks and task groups in the Task dependencies chapter.

Writing a simple Dag

Apache Airflow 3 offers two different approaches to writing Dags:

1. Task-oriented approach: Define a Dag object (using the `@dag` decorator) and fill it with tasks. This way of defining a Dag already existed in Airflow 2.
2. Asset-oriented approach: Use the shorthand `@asset` to define one Dag containing one task. When the task completes successfully it will also produce an event to an Airflow Asset. This way of defining a Dag is new in Airflow 3.

The table on the next page shows a side-by-side comparison of the two approaches. Generally most pipelines can be written using either approach and which to pick comes down to personal preference. Note that assets are under development and additional features such as asset partitioning are planned for coming minor releases.

	Task-oriented approach	Asset-oriented approach
Mindset	"Action" first - you are thinking about what each step in your pipeline *does*.	"Data first" - you are thinking about the *data* each step in your pipeline *produces*.
Boilerplate Code	A little bit, but tasks can be reused across Dags in a modular approach.	Minimal
Adapt pipelines to your data at runtime	Possible with dynamic task mapping	Not possible in Airflow 3.0.
Build data-driven pipelines	Within a Dag, tasks can have complex dependencies between them and Dags can be connected using data-driven scheduling.	Assets can be scheduled depending on each other using the same data-driven scheduling practices.
Dependency visualization	Task dependencies are shown in graph view for each Airflow Dag, cross-Dag dependencies created through assets are shown in asset graphs.	Dependencies between assets are shown in asset graphs.

Task-oriented approach

A Dag can be defined with a Python file placed in an Airflow project's Dag bundle, when using the Astro CLI, the default Dag bundle is the `dags` folder. Airflow automatically parses all files in this folder every 5 minutes to check for new Dags, and it parses existing Dags for code changes every 30 seconds. You can force a new Dag parse using `airflow dags reserialize`, or `astro dev run dags reserialize` when using the Astro CLI.

There are two types of syntax you can use to structure your Dag when using the task-oriented approach. Which one to use is a matter of personal preference:

- **TaskFlow API**: The TaskFlow API contains the `@dag` and `@task` decorators.
    - A function decorated with `@dag` defines a Dag.
    - A function decorated with `@task` defines an Airflow task.
    - All tasks that are instantiated/called within the context of the Dag function are part of the Dag.
    - Note that you need to call the decorated function for Airflow to register the Dag or task.

- **Traditional syntax**:
    - You can create a Dag by instantiating a Dag context using the `DAG` class and defining tasks within that context.

- You can create Airflow tasks using traditional Airflow operators (including sensors and deferrable operators), which are Python classes simplifying common actions in Airflow.
- All tasks that are instantiated/called within the Dag context are part of the Dag.

Any custom Python function can be turned into an Airflow task by decorating it with `@task` or providing it to the `python_callable` parameter of the PythonOperator. Note Dags can contain different types of building blocks that can depend on each other: you can freely mix the TaskFlow API and traditional syntax!

The following is an example of the same simple Dag written using the TaskFlow API (`@dag` and `@task`) and traditional syntax (`DAG` and the `PythonOperator`)

This simple Dag is written using the TaskFlow API and consists of two tasks running sequentially.

```
from airflow.sdk import dag, task, chain
from pendulum import datetime



# Define the Dag function with a set of parameters
@dag(
    start_date=datetime(2025, 8, 1),
    schedule="@daily"
)
def taskflow_dag():
    # Define tasks within the Dag context
    @task
    def my_task_1():
        import time  # import packages only needed in the task function
        time.sleep(5)
        print(1)


    @task
    def my_task_2():
```

```
        print(2)


    # Define dependencies and call task functions
    chain(my_task_1(), my_task_2())


# Call the Dag function
taskflow_dag()
```

This simple Dag, defined using the traditional syntax, accomplishes the same as the Dag above.

```
# Import all packages needed at the top level of the Dag
from airflow.sdk import DAG
from airflow.providers.standard.operators.python import PythonOperator
from pendulum import datetime



def my_task_1_func():
    import time  # import packages only needed in the task function

    time.sleep(5)
    print(1)



# Instantiate the Dag
with DAG(
    dag_id="traditional_syntax_dag",
    start_date=datetime(2025, 8, 1),
    schedule="@daily"
):
```

```
    # Instantiate tasks within the Dag context
    _my_task_1 = PythonOperator(
        task_id="my_task_1",
        python_callable=my_task_1_func,
    )

    _my_task_2 = PythonOperator(
        task_id="my_task_2",
        python_callable=lambda: print(2),
    )

    # Define dependencies
    _my_task_1 >> _my_task_2
```

TIP

Astronomer recommends creating one Python file for each Dag and naming it after the `dag_id` as a best practice for organizing your Airflow project. For certain advanced use cases it may be appropriate to dynamically generate Dags using Python code, see Programmatically generate Dags for more information.

Asset-oriented approach

The asset-oriented approach, which is new in Airflow 3 allows you to quickly define a Dag containing one task that updates one Asset with minimal boilerplate code.

The code above creates one Dag with the Dag ID `my_asset_1` containing one task with the task ID `my_asset_1` that produces updates to the Airflow Asset `my_asset_1`.

```
from airflow.sdk import asset


@asset(schedule=None, tags=["basic_asset_example"])
def my_asset_1():
    print("Add any Python Code!")
```

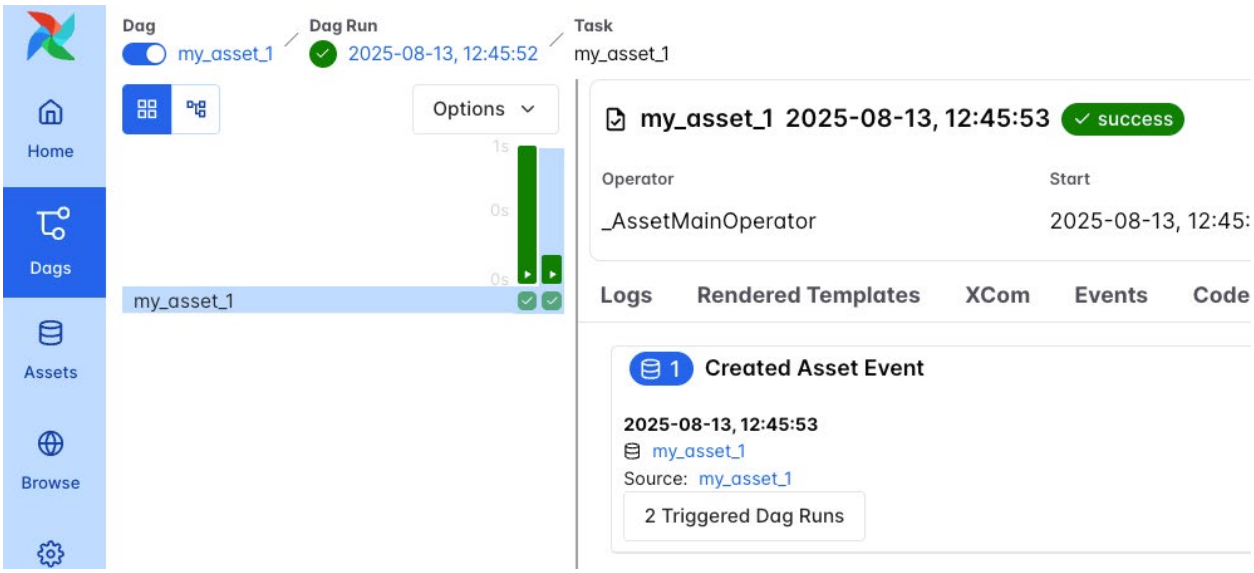

Figure 7: *Screenshot of the Airflow UI showing a Dag created with an @asset decorator on a function named my_asset_1. Note how the ID of the Dag, of the singular task and the Asset object are all the same.*

Assets can run on a schedule like regular Dags, including data-driven schedules, i.e. an `@asset` can be scheduled based on updates to `Assets`.

It is also possible to push data to XCom from assets and retrieve them using a cross-Dag XCom pull, as well as attach small amounts of `Metadata` to the extra dictionary of the Asset event.

The following code snippet shows several assets that depend on each other, push to and pull from XCom and use the extra dictionary of the Asset event. You can also find this example here on GitHub.

```
from airflow.sdk import asset, Metadata, Asset


@asset(schedule=None, tags=["basic_asset_example"])
def my_asset_1():
    print("Add any Python Code!")


@asset(schedule=[my_asset_1], tags=["basic_asset_example"])
def my_asset_2():
    # you can return data from an @asset decorated function
    # to push it to XCom
    return {"a": 1, "b": 2}


@asset(schedule=[my_asset_1, my_asset_2], tags=["basic_asset_example"])
def my_asset_3(context):
    # you can pull the data from XCom via the context
    # note that this is a cross-dag xcom pull
    data = context["ti"].xcom_pull(
        dag_id="my_asset_2",
        task_ids=["my_asset_2"],
        key="return_value",
        include_prior_dates=True,
    )[0]

    print(data)


@asset(schedule=[my_asset_1], tags=["basic_asset_example"])
```

```python
def my_asset_4(context):
    # you can also attache Metadata to the Asset Event
    yield Metadata(Asset("my_asset_4"), {"a": 1, "b": 2})


# note that using Asset("my_asset_4") is equivalent to passing the my_asset_4
# function directly. This is handy if your assets are defined in seperate
# Python files.
@asset(schedule=[Asset("my_asset_4")], tags=["basic_asset_example"])
def my_asset_5(context):
    # and retrieve the Metadata from the Asset Event
    metadata = context["triggering_asset_events"][Asset("my_asset_4")][0].extra
    print(metadata)
```

In the Airflow 3 UI you can view asset dependencies by selecting a graph from the list shown under the `Assets` button. This is the same view that shows you asset based dependencies of Dags written with the task-oriented approach. You can even create dependencies between Dags written with the task-oriented and with the asset-oriented approach.

The following asset graph is created by the example code above.

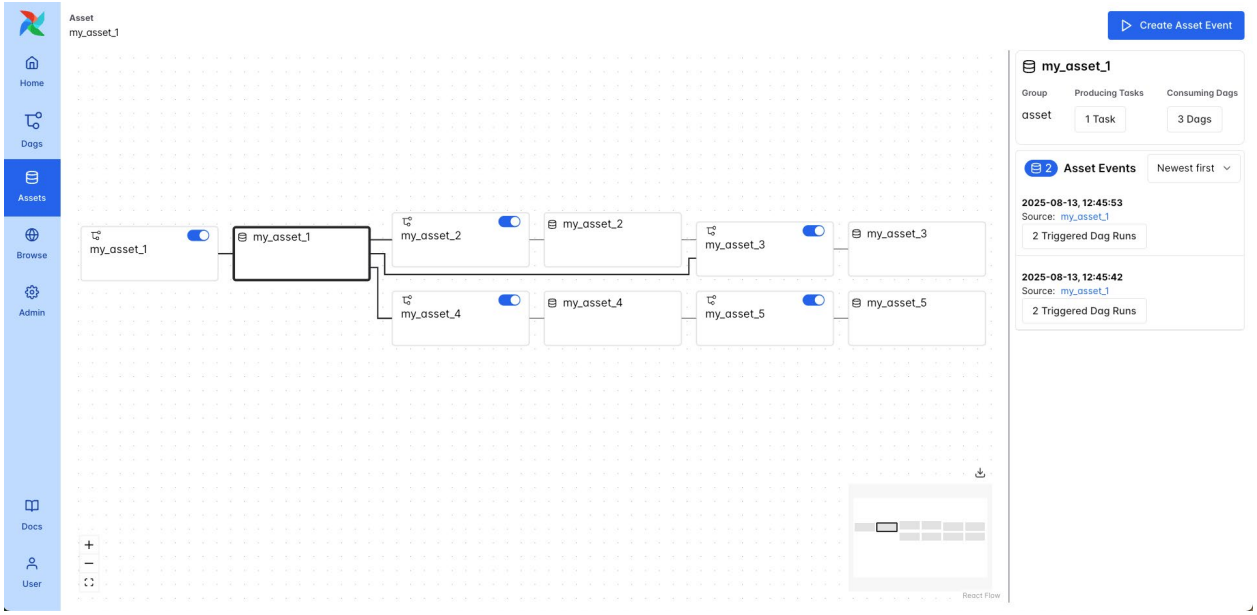

Figure 8: *Asset graph created from the 5 `@asset`s defined in the code snippet above.*

Dag-level parameters

In Airflow, you can configure when and how your Dag runs by setting parameters in the Dag object. Dag-level parameters affect how the entire Dag behaves, as opposed to task-level parameters which only affect a single task.

The Dags in the previous section have the following basic parameters defined. It's best practice to always define these parameters for any Dags you create:

- `dag_id`: The name of the Dag. This must be unique for each Dag in the Airflow environment. When using the `@dag` decorator and not providing the `dag_id` parameter name, the function name is used as the `dag_id`.
- `start_date`: The date and time after which the Dag starts being scheduled.
- `schedule`: The schedule for the Dag. There are many different ways to define a schedule, see Scheduling in Airflow for more information.

Other Dag parameters

Besides the three Dag parameters above, additional parameters are available that help you modify how your Dag behaves in the UI, interacts with Jinja templates, scales, notifies you about its state and more.

UI PARAMETERS

Parameter	Description
`description`	A short string that is displayed in the Airflow UI next to the Dag name.
`doc_md`	A string that is rendered as Dag documentation in the Airflow UI. Tip: use `__doc__` to use the docstring of the Python file. It is a best practice to give all your Dags descriptive Dag documentation.
`tags`	A list of tags shown in the Airflow UI to help with filtering Dags.

PARAMETERS RELATING TO JINJA TEMPLATING

Parameter	Description
`template_searchpath`	A list of folders where Jinja looks for templates. The path of the Dag file is included by default.
`template_undefined`	The behavior of Jinja when a variable is undefined. Defaults to StrictUndefined.
`render_template_as_native_obj`	Whether to render Jinja templates as native Python objects instead of strings. Defaults to `False`.
`user_defined_macros`	A dictionary of macros that are available in the Dag's Jinja templates. Use `user_defined_filters` to add filters and `jinja_environment_kwargs` for additional Jinja configuration. See Macros: using custom functions and variables in templates.

For more information on Jinja templating in Airflow, see the Use Airflow templates guide.

DAG SCALING PARAMETERS

Parameter	Description
`max_active_tasks`	The number of task instances allowed to run concurrently in one run of this Dag.
`max_active_runs`	The number of active Dag runs allowed to run concurrently for this Dag.
`max_consecutive_failed_dag_runs`	The maximum number of consecutive failed Dag runs, after which the scheduler will disable this Dag.

CALLBACK PARAMETERS

Parameter	Description
`on_success_callback`	A function to be executed after completion of a successful Dag run.
`on_failure_callback`	A function to be executed after a failed Dag run.

On Astro, you can use Astro alerts instead of or in addition to Airflow callbacks. See When to use Airflow or Astro alerts for your pipelines on Astro for more information.

MISCELLANEOUS

Parameter	Description
`end_date`	A function to be executed after completion of a successful Dag run.
`catchup`	Whether the scheduler should backfill all missed Dag runs between the current date and the start date when the Dag is unpaused. This parameter defaults to `False` in Airflow 3. See Catchup for more information.
`default_args`	A function to be executed after a failed Dag run.
`params`	A dictionary of Dag-level Airflow params. See Airflow params for more information.
`dagrun_timeout`	The time it takes for a Dag run of this Dag to time out and be marked as `failed`.
`access_control`	Specify optional permissions for roles specific to an individual Dag. See Dag-level permissions. This cannot be implemented on Astro. Astronomer recommends customers to use Astro's RBAC features instead.
`is_paused_upon_creation`	Whether the Dag is paused when it is created. When not set, the Airflow config `core.dags_are_paused_at_creation` is used, which defaults to `True`.

Parameter	Description
`auto_register`	Defaults to `True` and can be set to `False` to prevent Dags using a with context from being automatically registered which can be relevant in some advanced dynamic Dag generation use cases. See Registering dynamic Dags.
`fail_fast`	You can set this parameter to `True` to stop Dag execution as soon as one task in this Dag fails. Any tasks that are still running are marked as `failed`, and any tasks that have not run yet are marked as `skipped`. Note that you cannot have any trigger rule other than `all_success` in a Dag with `fail_fast` set to `True`.
`dag_display_name`	Airflow 2.9+ allows you to override the dag_id to display a different Dag name in the Airflow UI. This parameter allows special characters.

Want to know more?

Checkout our additional introductory resources about Airflow Dags

- Practical Guide to Apache Airflow® 3
- Airflow: Dags 101 Astronomer Academy Module
- An introduction to Apache Airflow guide
- Introduction to Airflow Dags guide
- Dag-level parameters in Airflow guide
- An introduction to the Airflow UI guide

Dag Building Blocks

2

KEY CONCEPTS

- **Operators**: Python classes that contain code to create an Airflow task by instantiating the class and providing arguments to its parameters.
- **Decorators**: Python decorators that turn Python functions into Airflow tasks. The base decorator is `@task`.
- **Sensors**: operators that wait for a specific state to be reached before running.
- **Deferrable Operators**: operators that leverage the Python asyncio library to run tasks waiting for an external resource to finish. This frees up your workers and allows you to use resources more effectively.
- **Hooks**: Python classes that connect to external tools using an Airflow connection connection ID. The class methods are used to interact with the external tool. Hooks cannot create Airflow tasks by themselves, they are used in operators, in `@task` decorated tasks or in `@asset`s.
- **Airflow provider package**: A Python package containing a set of operators, decorators, hooks and other modules relating to an external tool. Provider packages can be pip installed on top of core Airflow to extend its functionality. If you are using the Astro CLI to run Airflow, install provider packages by adding them to the `requirements.txt` file.
- Astronomer Registry: A comprehensive collection of available provider packages and their modules.

Airflow building blocks connect to external data tools using Airflow connections.

Some use cases require isolated environments to run different tasks, for example due to Python package conflicts. See the Run tasks in an isolated environment in Apache Airflow guide in our documentation for more information.

Airflow Operators

Operators are one of the fundamental building blocks of Airflow Dags. They contain the logic of how data is processed in a pipeline.

There are many different types of operators available in Airflow. Some operators such as the PythonOperator execute general code provided by the user, while other operators perform very specific actions such as transferring data from one system to another. Two important subgroups of operators are Sensors and Deferrable Operators.

You can find a comprehensive list of available operators in the Astronomer Registry.

Operators are Python classes that encapsulate logic to do a unit of work. They can be viewed as a wrapper around each unit of work that defines the actions that will be completed and abstract the majority of code you would typically need to write. When you create an instance of an operator in a Dag and provide it with its required parameters, it becomes a task.

All operators inherit from the abstract BaseOperator class, which contains the logic to execute the work of the operator within the context of a Dag.

The following are some of the most frequently used Airflow operators:

- PythonOperator: Executes a Python function.

```
# from airflow.providers.standard.operators.python import PythonOperator


def _say_hello():
    return “hello”



say_hello = PythonOperator(
    task_id=”say_hello”,
    python_callable=_say_hello
)
```

- BashOperator: Executes a bash command or script. See also Using the BashOperator in our documentation.

```
# from airflow.providers.standard.operators.bash import BashOperator


bash_task = BashOperator(
    task_id="bash_task",
    bash_command="echo $MY_VAR",
    env={"MY_VAR": "Hello World"}
)
```

- KubernetesPodOperator: Executes a task defined as a Docker image in a Kubernetes Pod. See also Use the KubernetesPodOperator in our documentation.

```
# from airflow.providers.cncf.kubernetes.operators.pod import KubernetesPodOperator


example_kpo = KubernetesPodOperator(
    kubernetes_conn_id="k8s_conn",
    image="hello-world",
    name="airflow-test-pod",
    task_id="task-one",
    is_delete_operator_pod=True,
    get_logs=True,
)
```

Operators typically only require a few parameters. Keep the following considerations in mind when using Airflow operators:

- The Airflow standard provider package includes basic operators such as the PythonOperator and BashOperator. The standard provider package is installed by default in most Airflow installations, including the environment you get when using the Astro CLI. Other operators are part of additional provider packages, which you must install separately. For example, the `CopyFromExternalStageToSnowflakeOperator` is part of the Snowflake provider package.
- If an operator exists for your specific use case, you should use it instead of your own Python functions. This makes your Dags easier to read and maintain and you profit from the open-source community maintaining and testing the operator.
- Deferrable Operators are a type of operator that releases their worker slot while waiting for their work to be completed. Many operators can be turned into deferrable operators by setting the `deferrable` parameter to `True`. This can result in cost savings and greater scalability. Astronomer recommends using deferrable operators whenever one exists for your use case and your task takes longer than a minute. You must have a triggerer running to use deferrable operators.
- Any operator that interacts with a service external to Airflow typically requires a connection so that Airflow can authenticate to that external system. See Airflow connections.

If an operator doesn't exist for your use case, you can create a custom operator. See Custom hooks and operators in our documentation.

Airflow Decorators

Airflow Decorators are part of the TaskFlow API, which is a functional API for using decorators to define Dags and tasks, simplifying the process for passing data between tasks and defining dependencies.

You can use TaskFlow decorator functions (for example, `@task`) to pass data between tasks by providing the output of one task as an argument to another task. Decorators are a simpler, cleaner way to define your tasks and Dags and can be used in combination with traditional operators.

In general, whether you use the TaskFlow API is a matter of your own preference and style. In most cases, a TaskFlow decorator and the corresponding traditional operator will have the same functionality.

What is a decorator in Python?

In Python, decorators are functions that take another function as an argument and extend the behavior of that function. For example, the `@multiply_by_100_decorator` in the example code below takes any function as the `decorated_function` argument and returns the result of that function multiplied by 100.

```
# definition of the decorator function
def multiply_by_100_decorator(decorated_function):
    def wrapper(num1, num2):
        result = decorated_function(num1, num2) * 100
        return result

    return wrapper



# definition of the `add` function decorated with the `multiply_by_100_decorator`
@multiply_by_100_decorator
def add(num1, num2):
    return num1 + num2



# definition of the `subtract` function decorated with the `multiply_by_100_decorator`
```

```
@multiply_by_100_decorator
def subtract(num1, num2):
    return num1 - num2



# calling the decorated functions
print(add(1, 9))  # prints 1000
print(subtract(4, 2))  # prints 200
```

In the context of Airflow, decorators contain more functionality than this simple example, but the basic idea is the same: the Airflow decorator function extends the behavior of a normal Python function to turn it into an Airflow task, task group or Dag.

How to use Airflow Decorators

The TaskFlow API allows you to write your Python tasks with decorators. It handles passing data between tasks using XCom and infers task dependencies automatically.

Using decorators to define your Python functions as tasks is easy, just put `@task` on top of your Python function:

```
# from airflow.sdk import task


@task
def say_hello():
    return “hello”
```

The task that the code above creates is equivalent to using the PythonOperator with the `say_hello` function as `python_callable`:

```
# from airflow.providers.standard.operators.python import PythonOperator


def _say_hello():
    return “hello”



say_hello = PythonOperator(
    task_id=”say_hello”,
    python_callable=_say_hello
)
```

Using decorators eliminates the need to explicitly instantiate the equivalent traditional operator, pull data from XCom and set dependencies explicitly, and is easier to read.

The following two Dags, while functionally equivalent, show the advantages of using decorators over operators.

TaskFlowAPI version, using the `@task` decorator:

```
from airflow.sdk import dag, task



@dag
def task_decorator_dag():

    @task
    def t1() -> str:
        return “Hi”
```

```
    @task
    def t2(input_string: str):
        print(input_string)

    # the dependency is automatically inferred by Airflow
    # and the output from t1 is passed to t2
    t2(input_string=t1())


task_decorator_dag()
```

Traditional syntax version, using the PythonOperator:

```
from airflow.sdk import dag, chain
from airflow.providers.standard.operators.python import PythonOperator


def t1_func():
    return “Hi”


def t2_func(ti):
    # explicitely pulling information from the XCom table in the Airflow metadata database
    return_value_t1 = ti.xcom_pull(task_ids=”t1”)
    print(return_value_t1)


@dag
```

```
def python_operator_dag():
    _t1 = PythonOperator(task_id="t1", python_callable=t1_func)

    _t2 = PythonOperator(task_id="t2", python_callable=t2_func)

    # explicit definition of task dependencies
    chain(_t1, _t2)


python_operator_dag()
```

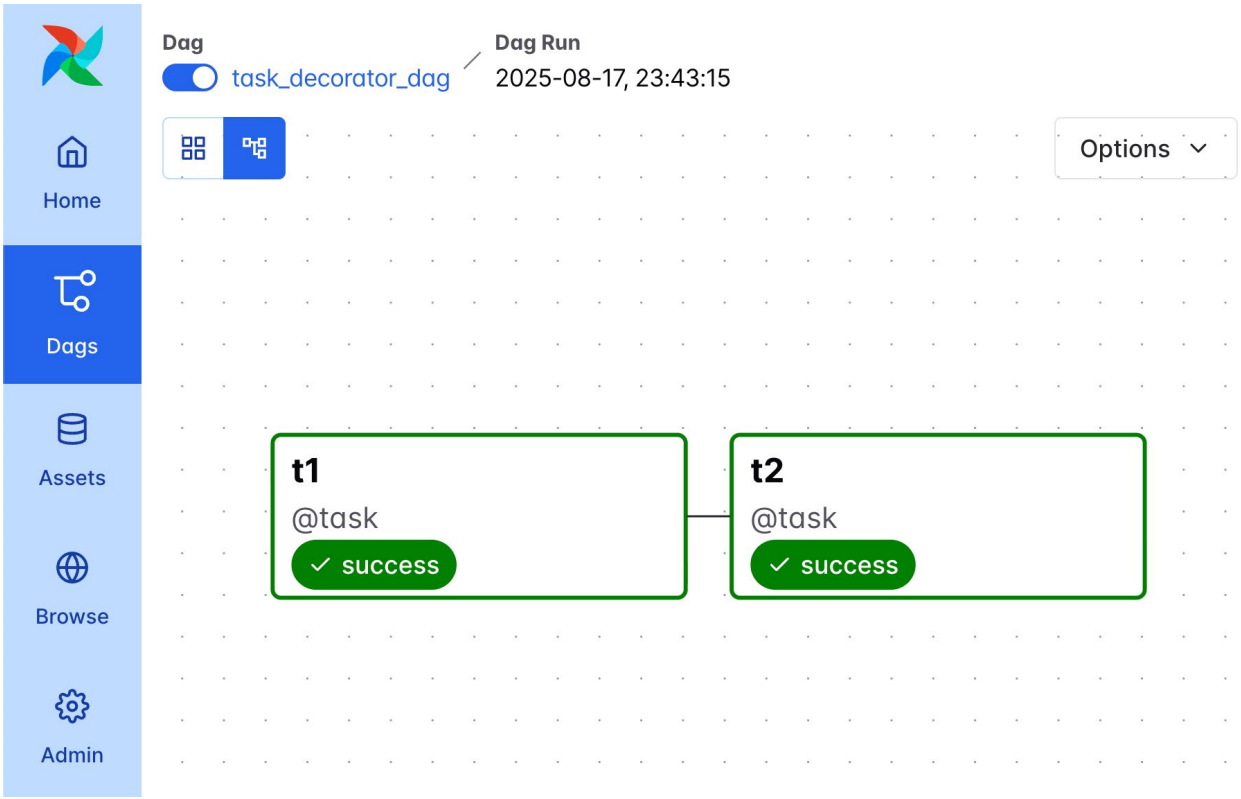

Figure 1: *Screenshot of the resulting Dag graph from the code TaskFlowAPI code example above.*

Here are some other things to keep in mind when using decorators:

- You must call all decorated functions in your Dag file so that Airflow can register the task or Dag. For example, `task_decorator_dag()` is called at the end of the previous example to call the Dag function and `t1()` and `t2()` are called inside the Dag function to register the respective Airflow tasks.
- When you define a task, the `task_id` defaults to the name of the function you decorated. If you want to change this behavior, you can pass a `task_id` to the decorator Similarly, other BaseOperator task-level parameters, such as retries or pool, can be defined within the decorator:

```
@task(
    task_id="say_hello_world"
    retries=3,
    pool="my_pool",
)
def taskflow_func():
    retun "Hello World"

taskflow_func()  # this creates a task with the task_id `say_hello_world`
```

You can override task-level parameters when you call the task by using the `.override()` method. The `()` at the end of the line calls the task with the overridden parameters applied.

```
# this creates a task with the task_id `greeting`
taskflow_func.override(retries=5, pool="my_other_pool", task_id="greeting")()
```

If you call the same task multiple times and do not override the `task_id`, Airflow creates multiple unique task IDs by appending a number to the end of the original task ID (for example, `say_hello`, `say_hello__1`, `say_hello__2`, etc). You can see the result of this in the following example:

```
# task definition
@task
def say_hello(dog):
    return f"Hello {dog}!"


### calling the task 4 times, creating 4 tasks in the Dag
# this task will have the id `say_hello` and print "Hello Avery!"
say_hello("Avery")
# this task will have the id `greet_dog` and print "Hello Piglet!"
say_hello.override(task_id="greet_dog")("Piglet")
# this task will have the id `say_hello__1` and print "Hello Peanut!"
say_hello("Peanut")
# this task will have the id `say_hello__2` and print "Hello Butter!"
say_hello("Butter")
```

You can decorate a function that is imported from another file as shown in the following code snippet:

```
from include.my_file import my_function


@task
def taskflow_func():
    my_function()
```

This is recommended in cases where you have lengthy Python functions since it will make your Dag file easier to read.

You can assign the output of a called decorated task to a Python object to be passed as an argument into another decorated task. This is helpful when the output of one decorated task is needed in several downstream functions.

```
@task
def get_fruit_options():
    return [“peach”, “raspberry”, “pineapple”]


@task
def eat_a_fruit(fruit_list):
    index = random.randint(0, len(list) - 1)
    print(f”I’m eating a {list[index]}!”)


@task
def gift_a_fruit(fruit_list):
    index = random.randint(0, len(list) - 1)
    print(f”I’m giving you a {list[index]}!”)


# you can assign the output of a decorated task to a Python object to pass it to one or more downstream tasks as inpu
_get_fruit_options = get_fruit_options()
eat_a_fruit(fruit_list=_get_fruit_options)
gift_a_fruit(fruit_list=_get_fruit_options)
```

Available Airflow decorators

There are several decorators available to use with Airflow. Some of the most commonly used decorators are:

- Dag decorator (`@dag()`), which creates a Dag.
- TaskGroup decorator (`@task_group()`), which creates a task group.
- Task decorator (`@task()`), which creates a Python task.
- Bash decorator (`@task.bash()`) which creates a BashOperator task.
- Python Virtual Env decorator (`@task.virtualenv()`), which runs your Python task in a virtual environment.
- Branch decorator (`@task.branch()`), which creates a branch in your Dag based on an evaluated condition.
- Kubernetes pod decorator (`@task.kubernetes()`), which runs a KubernetesPodOperator task.
- Sensor decorator (`@task.sensor()`), which turns a Python function into a sensor.
- The `@asset` decorator is a special case of decorator used to write Dags with the asset-oriented approach; it creates one Dag containing one task that updates one asset.

See create your own custom task decorator in the Airflow documentation to learn how to create custom decorators.

Airflow Sensors

Sensors are a special kind of operator that are designed to wait for something to happen. When sensors run, they check to see if a certain condition is met before they are marked successful and let their downstream tasks execute.

All sensors inherit from the BaseSensorOperator and have the following parameters:

- `mode`: How the sensor operates. There are two types of modes:
    - poke: This is the default mode. When using poke, the sensor occupies a worker slot for the entire execution time and sleeps between pokes. This mode is best if you expect a short runtime for the sensor.
    - reschedule: When using this mode, if the criteria is not met then the sensor releases its worker slot and reschedules the next check for a later time. This mode is best if you expect a long runtime for the sensor, because it is less resource intensive and frees up workers for other tasks.

- `poke_interval`: When using poke mode, this is the time in seconds that the sensor waits before checking the condition again. The default is 60 seconds.
- `exponential_backoff`: When set to True, this setting creates exponentially longer wait times between pokes in poke mode.
- `max_wait`: The maximum wait time between pokes in seconds or as a timedelta object.
- `timeout`: The maximum amount of time in seconds that the sensor checks the condition. If the condition is not met within the specified period, the task fails.
- Parameters relating sensor failure modes.

Different types of sensors have different implementation details.

Many sensors have a `deferrable` parameter, which when set to `True` will turn the sensor into a deferrable operator. This functionality is implemented on a per-sensor basis and may not be available for all sensors.

COMMONLY USED SENSORS

Many Airflow provider packages contain sensors that wait for various criteria in different source systems. The following are some examples of commonly used sensors:

- @task.sensor decorator: Allows you to turn any Python function that returns a `PokeReturnValue` into an instance of the BaseSensorOperator class. This way of creating a sensor is useful when checking for complex logic or if you are connecting to a tool via an API that has no specific sensor available.
- S3KeySensor: Waits for a key (file) to appear in an Amazon S3 bucket. This sensor is useful if you want your Dag to process files from Amazon S3 as they arrive.
- HttpSensor: Waits for an API to be available. This sensor is useful if you want to ensure your API requests are successful.
- SqlSensor: Waits for data to be present in a SQL table. This sensor is useful if you want your Dag to process data as it arrives in your database.

To review the available Airflow sensors, see the Astronomer Registry.

Sensor decorator

If no sensor exists for your use case, you can create your own using either the @task.sensor decorator or the PythonSensor. The @task.sensor decorator returns a `PokeReturnValue` directly and the PythonSensor takes a `python_callable` that returns `True` or `False`.

The following Dag shows how to use the `@task.sensor` decorator:

```
"""
### Create a custom sensor using the @task.sensor decorator

This Dag showcases how to create a custom sensor using the @task.sensor decorator
to check the availability of an API.
"""

from airflow.sdk import dag, task, PokeReturnValue
from pendulum import datetime
import requests


@dag(start_date=datetime(2025, 8, 1), schedule="@daily")
def sensor_decorator():
    # supply inputs to the BaseSensorOperator parameters in the decorator
    @task.sensor(poke_interval=30, timeout=3600, mode="poke")
    def check_dog_availability() -> PokeReturnValue:
        r = requests.get("https://random.dog/woof.json")
        print(r.status_code)

        # set the condition to True if the API response was 200
        if r.status_code == 200:
```

```
            condition_met = True
            operator_return_value = r.json()
        else:
            condition_met = False
            operator_return_value = None
            print(f"Woof URL returned the status code {r.status_code}")

        # the function has to return a PokeReturnValue
        # if is_done = True the sensor will exit successfully, if
        # is_done=False, the sensor will either poke or be rescheduled
        return PokeReturnValue(is_done=condition_met, xcom_value=operator_return_value)

    # print the URL to the picture
    @task
    def print_dog_picture_url(url):
        print(url)

    print_dog_picture_url(check_dog_availability())

sensor_decorator()
```

Here, the `@task.sensor` decorates the `check_dog_availability()` function, which checks if a given API returns a `200` status code. If the API returns a `200` status code, the sensor task is marked as successful. If any other status code is returned, the sensor pokes again after the `poke_interval` has passed.

The optional `xcom_value` parameter in PokeReturnValue defines what data will be pushed to XCom once `is_done=true`. You can use the data that was pushed to XCom in any downstream tasks.

Sensor best practices

When using sensors, keep the following in mind to avoid potential performance issues:

- Always define a meaningful `timeout` parameter for your sensor. The default for this parameter is seven days, which is a long time for your sensor to be running. When you implement a sensor, consider your use case and how long you expect the sensor to wait and then define the sensor's timeout accurately.
- Whenever possible and especially for long-running sensors, use deferrable operators instead. If no deferrable operator is available for your use case and you do not want to create your own, use the Sensor in reschedule mode so it is not constantly occupying a worker slot. This helps avoid deadlocks in Airflow where sensors take all of the available worker slots.
- If your `poke_interval` is very short (less than about 5 minutes), use the poke mode. Using reschedule mode in this case can overload your scheduler.
- Define a meaningful `poke_interval` based on your use case. There is no need for a task to check a condition every 60 seconds (the default) if you know the total amount of wait time will be 30 minutes.

Sensor failure modes

When using sensors, there are different options to define its behavior in case of an exception raised within the sensor.

- `soft_fail=True`: If an exception is raised within the task, it is marked as skipped, affecting downstream tasks according to their defined trigger rules.
- `silent_fail=True`: If an exception is raised in the poke method that is not one of: AirflowSensorTimeout, AirflowTaskTimeout, AirflowSkipException or AirflowFailException, the sensor will log the error but continue its execution.
- `never_fail=True`: If the poke method raises any exception, the sensor task is skipped. This parameter is mutually exclusive with `soft_fail`.

Deferrable Operators

Deferrable operators leverage the Python asyncio library to efficiently run tasks waiting for an external resource to finish. This frees up your workers and allows you to use resources more effectively. In this guide, you'll review deferrable operator concepts and learn how to use deferrable operators in your Dags. Tasks that are in a deferred state are shown as purple in the Airflow UI.

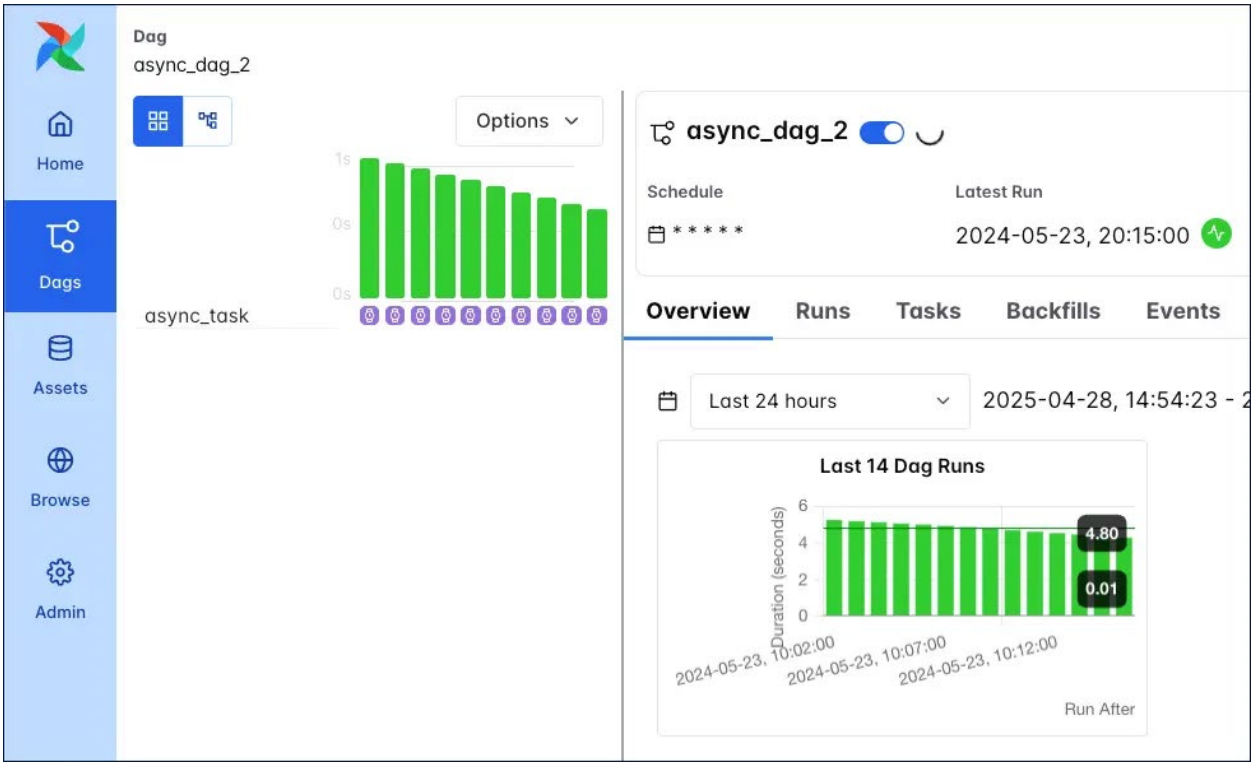

Figure 2: *Screenshot of the Airflow UI showing several Dag runs containing a deferred (purple) task.*

Terms and concepts

Review the following terms and concepts to gain a better understanding of deferrable operator functionality:

- asyncio: A Python library used as the foundation for multiple asynchronous frameworks. This library is core to deferrable operator functionality, and is used when writing triggers.
- **Triggers**: Small, asynchronous sections of Python code. Due to their asynchronous nature, they coexist efficiently in a single process known as the triggerer.
- **Triggerer**: An Airflow service similar to a scheduler or a worker that runs an asyncio event loop in your Airflow environment. Running a triggerer is essential for using deferrable operators.
- **Deferred**: An Airflow task state indicating that a task has paused its execution, released the worker slot, and submitted a trigger to be picked up by the triggerer process.

The terms deferrable, async, and asynchronous are used interchangeably and have the same meaning in the context of Airflow.

With traditional operators, a task submits a job to an external system such as a Spark cluster and then polls the job status until it is completed. Although the task isn't doing significant work, it still occupies a worker slot during the polling process. As worker slots are occupied, tasks are queued and start times are delayed. The following image illustrates this process:

Figure 3: *Graph showing a regular operator using up a worker slot while polling a spark cluster for a job status.*

With deferrable operators, worker slots are released when a task is polling for the job status. When the task is deferred, the polling process is offloaded as a trigger to the triggerer, and the worker slot becomes available. The triggerer can run many asynchronous polling tasks concurrently, and this prevents polling tasks from occupying your worker resources. When the terminal status for the job is received, the operator resumes the task, taking up a worker slot while it finishes. The following image illustrates the process:

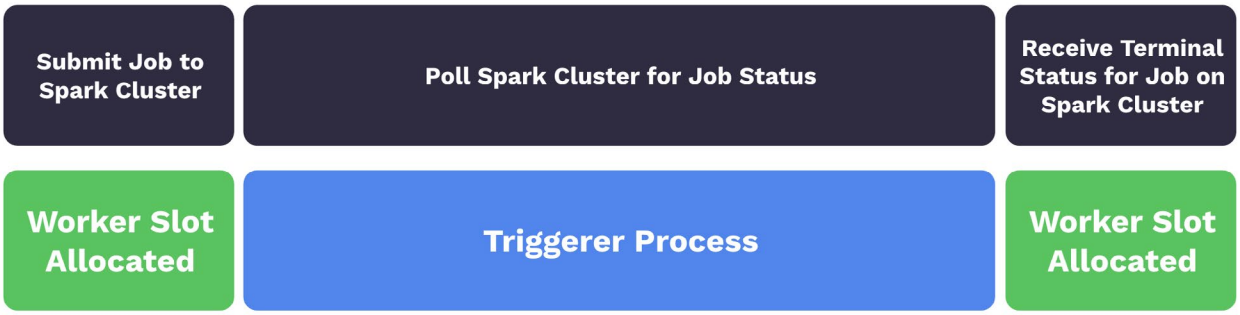

Figure 4: *Graph showing a deferrable operator releasing its worker slot while polling a spark cluster for a job status. The polling occurs in the triggerer process.*

Use deferrable Operators

Deferrable operators should be used whenever you have tasks that occupy a worker slot while polling for a condition in an external system. For example, using deferrable operators for sensor tasks can provide efficiency gains and reduce operational costs.

START A TRIGGERER

To use deferrable operators, you must have a triggerer running in your Airflow environment. If you are running Airflow on Astro or using the Astro CLI, the triggerer runs automatically if you are on Astro Runtime 4.0 and later.

If you are not using Astro, run `airflow triggerer` to start a triggerer process in your Airflow environment.

As tasks are raised into a deferred state, triggers are registered in the triggerer. You can set the number of concurrent triggers that can run in a single triggerer process with the default_capacity configuration setting in Airflow. This config can also be set with the `AIRFLOW__TRIGGERER__DEFAULT_CAPACITY` environment variable. The default value is 1000.

Triggers are designed to be highly available. You can implement this by starting multiple triggerer processes. Similar to the HA scheduler, Airflow ensures that they co-exist with correct locking and high availability. See High Availability in the Airflow documentation for more information on this topic.

USE DEFERRABLE VERSIONS OF OPERATORS

Many Airflow operators, for example the TriggerDagRunOperator and the WasbBlobSensor, can be set to run in deferrable mode using the `deferrable` parameter. You can check if the operator you want to use has a deferrable parameter in the Astronomer Registry.

To always use the deferrable version of an operator if it's available in Airflow 2.7+, set the Airflow config `operators.default_deferrable` to `True`. You can do so by defining the following environment variable in your Airflow environment:

```
AIRFLOW__OPERATORS__DEFAULT_DEFERRABLE=True
```

After you set the variable, all operators with a deferrable parameter will run as their deferrable version by default. You can override the config setting at the operator level using the deferrable parameter directly:

```
trigger_dag_run = TriggerDagRunOperator(
   task_id="task_in_downstream_dag",
   trigger_dag_id="downstream_dag",
   wait_for_completion=True,
   poke_interval=20,
   deferrable=False,  # turns off deferrable mode just for this operator instance
)
```

You can find a list of operators that support deferrable mode in the Airflow documentation.

Previously, before the deferrable parameter was available in regular operators, deferrable operators were implemented as standalone operators, usually with an -Async suffix. Some of these operators are still available. For example, the DateTimeSensor does not have a deferrable parameter, but has a deferrable version called DateTimeSensorAsync.

You can also create your own deferrable operator. See Create a deferrable operator in our documentation for template code.

Hooks

A hook is an abstraction over a specific API that allows Airflow to interact with an external system. Hooks are built into many operators, but they can also be used directly in Dag code either inside functions decorated with Airflow decorators or in the `python_callable` provided to a PythonOperator. Hooks standardize how Airflow interacts with external systems and using them makes your Dag code cleaner, easier to read, and less prone to errors than when using an external API directly.

The code snippet below shows the S3Hook used in an `@task` decorated function to read the information from two files located in two different S3 buckets and return the response:

```
@task
def read_keys_from_s3():
    from airflow.providers.amazon.aws.hooks.s3 import S3Hook


    s3_hook = S3Hook(aws_conn_id="aws_conn")
    response_file_1 = s3_hook.read_key(
        key="folder_A/file_A", bucket_name="bucket_A"
    )
    response_file_2 = s3_hook.read_key(
        key="folder_B/file_B", bucket_name="bucket_B"
    )
```

```
    response = {
        "content_A": response_file_1,
        "content_B": response_file_2,
    }
    return response
```

Over 300 hooks are available in the Astronomer Registry. If a hook isn't available for your use case, you can write your own and share it with the community.

To instantiate a hook, you typically only need a connection ID to connect with an external system. In the example above `aws_conn` is the connection ID used.

All hooks inherit from the BaseHook class, which contains the logic to set up an external connection with a connection ID. On top of making the connection to an external system, individual hooks can contain additional methods to perform various actions within the external system. These methods might rely on different Python libraries for these interactions. For example, the S3Hook relies on the boto3 library to interact with Amazon S3.

Want to know more?

- Airflow: Deferrable Operators Astronomer Academy Module
- Airflow: Taskflow API Astronomer Academy Module
- Airflow: Sensors Astronomer Academy Module
- Airflow operators guide
- Introduction to the TaskFlow API and Airflow decorators guide
- Airflow sensors guide
- Deferrable operators guide
- Airflow hooks guide
- Custom hooks and operators guide

Airflow Connections

3

KEY CONCEPTS

- **Airflow Connection**: Sets of configurations used to connect with other tools in the data ecosystem.
- **Connection ID**: A unique string identifying an Airflow Connection. The connection ID is provided to Airflow modules like operators and hooks to use a connection.

Connection options

Connections in Airflow are sets of configurations used to connect with other tools in the data ecosystem. Most hooks and operators rely on connections to send and retrieve data from external systems.

An Airflow connection is a set of configurations that send requests to the API of an external tool. In most cases, a connection requires login credentials or a private key to authenticate Airflow to the external tool.

Airflow connections can be created by using one of the following methods:

- Astro Environment Manager (recommended for Astro customers to manage connections)
- Airflow UI
- Environment variables
- Airflow REST API
- A secrets backend (a system for managing secrets external to Airflow)
- Airflow CLI
- airflow_settings.yaml file (for Astro CLI users)

Each connection has a unique `conn_id` which can be provided to operators and hooks that require a connection.

There are a couple of ways to find the information you need to provide for a particular connection type:

- Open the relevant provider page in the Astronomer Registry and go to the first link under **Helpful Links** to access the Apache Airflow documentation for the provider. Most commonly used providers will have documentation on each of their associated connection types. For example, you can find information on how to set up different connections to Azure in the Azure provider docs.
- Check the documentation of the external tool you are connecting to and see if it offers guidance on how to authenticate.
- Refer to the source code of the hook that is being used by your operator.

Defining connections in the Airflow UI

The most common way of defining a connection is using the Airflow UI. Go to **Admin > Connections**.

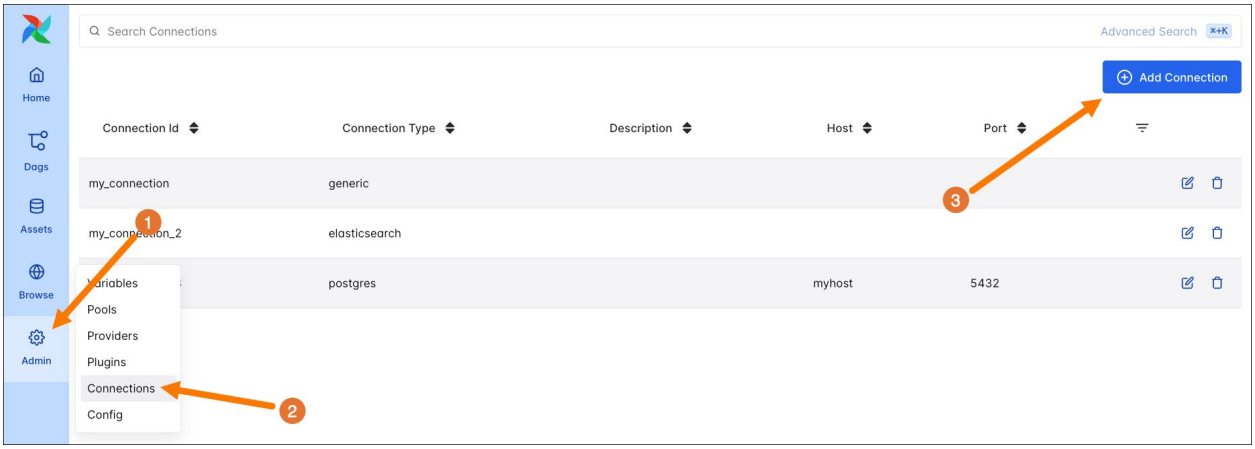

Figure 1: *Screenshot of the Airflow UI showing how to navigate to the Connections view.*

Airflow doesn't provide any preconfigured connections. To create a new connection, click the blue **+** Add Connection button.

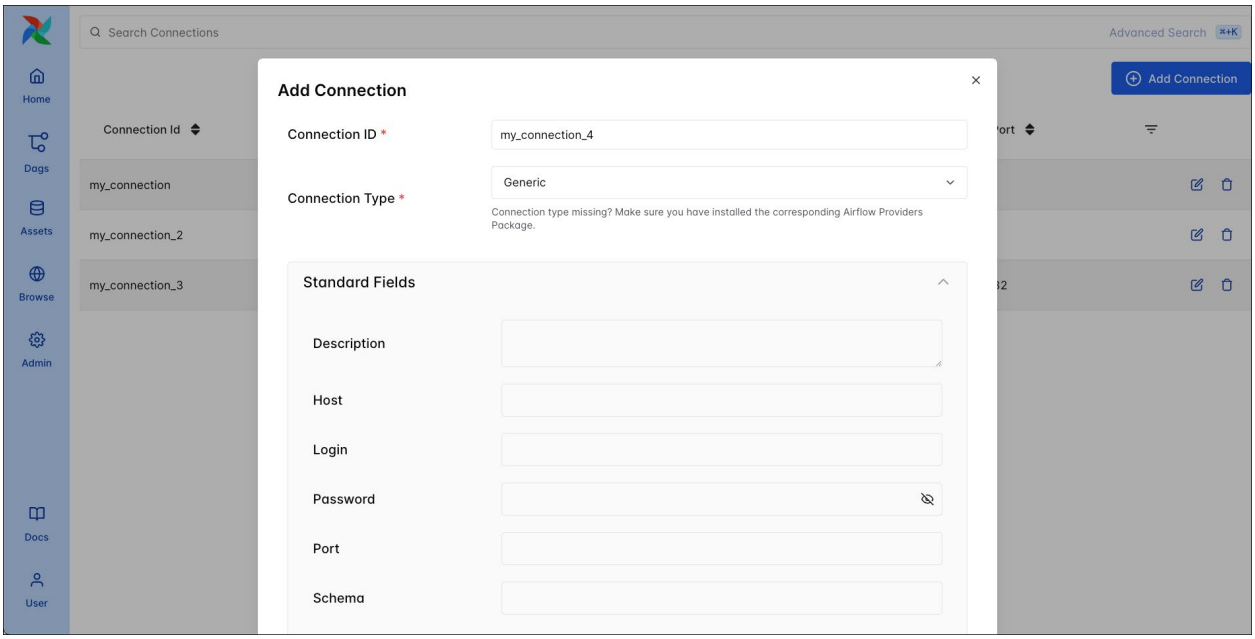

Figure 2: *Screenshot of the Airflow UI showing an example connection form.*

As you update the **Connection Type** field, notice how the other available fields change. Each connection type requires different kinds of information. Specific connection types are only available in the dropdown list when the relevant Airflow provider package is installed in your environment.

You don't have to specify every field for most connections. For example, to set up a connection to a PostgreSQL database, you can reference the PostgreSQL Airflow provider documentation to learn that the connection requires a Host, a user name as login, and a password in the password field.

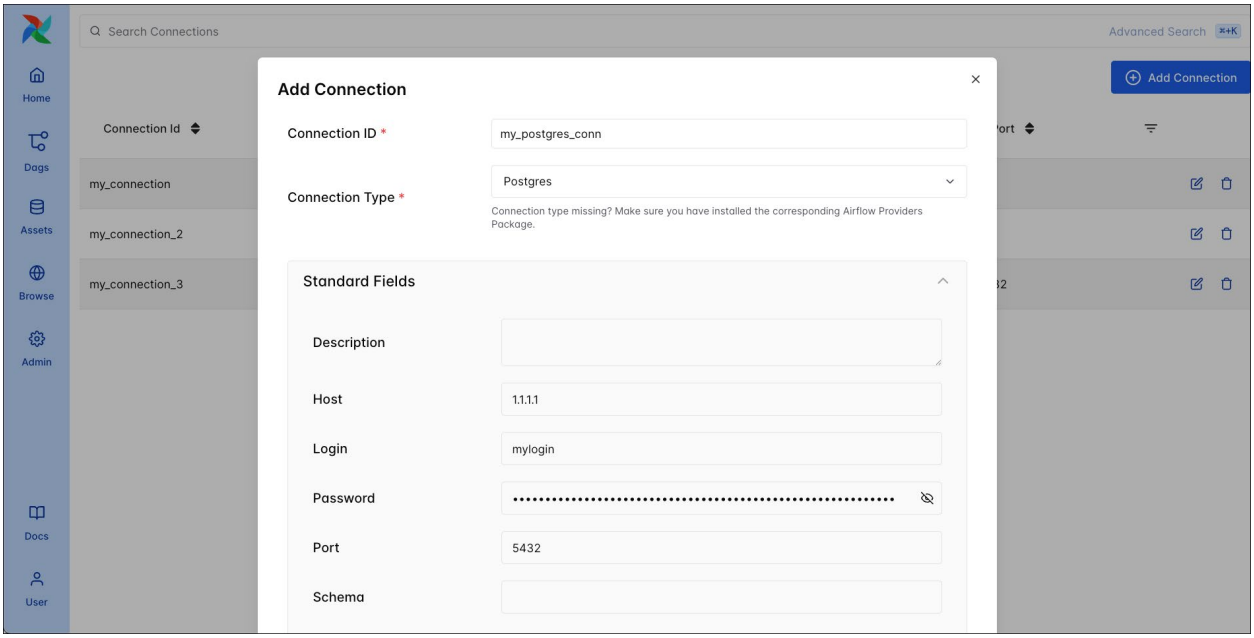

Figure 3: *Screenshot of the Airflow UI showing an example Postgres connection.*

Any parameters that don't have specific fields in the connection form can be defined in the **Extra** field as a **JSON dictionary**. For example, you can add the `sslmode` or a client `sslkey` in the Extra field of your PostgreSQL connection.

Define connections with environment variables

Connections can also be defined using environment variables. If you use the Astro CLI, you can use the `.env` file for local development or specify environment variables in your project's Dockerfile.

> If you are synchronizing your project to a remote repository, don't save sensitive information in your Dockerfile. In this case, use either a secrets backend, Airflow connections defined in the UI, or the Astro Environment Manager.

The environment variable used for the connection must be formatted as `AIRFLOW_CONN_MYCONNID` and can be provided as a Uniform Resource Identifier (URI) or in JSON.

URI is a format designed to contain all necessary connection information in one string, starting with the connection type, followed by login, password, and host. In many cases a specific port, schema, and additional parameters must be added.

```
# the general format of a URI connection that is defined in your Dockerfile

.env AIRFLOW_CONN_MYCONNID='my-conn-type://login:password@host:port/
schema?param1=val1&param2=val2'


# an example of a connection to snowflake defined as a URI AIRFLOW_CONN_SNOWFLAKE_
CONN='snowflake://LOGIN:PASSWORD@/?account=xy12345&region=eu-central-1'
```

Connections can also be provided to an environment variable as a JSON dictionary:

```
# example of a connection defined as a JSON file in your `.env` file

AIRFLOW_CONN_MYCONNID='{

    "conn_type": "my-conn-type",

    "login": "my-login",

    "password": "my-password",

    "host": "my-host",

    "port": 1234,

    "schema": "my-schema",

    "extra": {

        "param1": "val1",

        "param2": "val2"

    }

}'
```

Masking sensitive information

Connections often contain sensitive credentials. By default, Airflow hides the password field in the UI and in the Airflow logs. If `AIRFLOW__CORE__HIDE_SENSITIVE_VAR_CONN_FIELDS` is set to `True`, values from the connection's Extra field are also hidden if their keys contain any of the words listed in `AIRFLOW__CORE__SENSITIVE_VAR_CONN_NAMES`. You can find more information on masking, including a list of the default values in this environment variable, in the Airflow documentation on Masking sensitive data.

Want to know more?

- Airflow: Connections 101 Astronomer Academy Module
- How to manage connections in Airflow webinar
- Manage connections in Apache Airflow guide

Writing Dags

4

KEY CONCEPTS

- Top-level code: Code that is not contained in a task and is executed every time the Dag file is parsed. You should avoid having long running top-level code, especially code that connects to external tools
- Three core best practices: keep your tasks atomic, idempotent and modular.
- Automatic retries: a set of configurations that make individual Airflow tasks run again automatically after a failure.
- Software development best practices: Dag code should be treated like any software code: keep it in version control and use CICD best practices.
- Task dependencies: In Airflow you can set dependencies between tasks using `chain` functions or bitshift operators
- Trigger rules: By default an Airflow task only runs when all tasks it depends on have completed successfully. Trigger rules allow you to modify this behavior.
- Branching: Airflow branching allows you to execute different tasks based on conditional logic evaluated at runtime.
- Task groups: Airflow tasks can be assigned to groups to make Dags easier to read.
- XCom: XCom stands for cross-communication and is Airflow's way of passing information between tasks.

Dag writing Best Practices in Apache Airflow

Three core Dag writing best practices

There are three best practices to follow, no matter the purpose of your Airflow pipeline:

- **Atomicity**: Atomicity refers to designing each task to complete one specific function in the pipeline. In an ETL or ELT pipeline, each task in your Dag, ideally accomplishes one step: extracting, transforming, loading, or an ETL adjacent task such as sending a report. This atomic structure gives you full observability over your pipeline and enables you to rerun only failed tasks without having to repeat earlier steps, saving time and compute.
- **Idempotency**: This is just a fancy way of saying "Same in - same out". When creating an Airflow pipeline, you ideally have idempotent tasks and Dags, meaning with the same input your task always has the same output. In ETL pipelines it is common to use timestamps to partition your data. When using Airflow you can pull different timestamps relating to the Dag run time from the Airflow context, for example the logical_date to partition your data, ensuring that the partition stays the same even if the Dag run is rerun months later.
- **Modularity**: Don't repeat yourself - this software engineering best practice holds true for Airflow Dags. Since Dags are defined in Python code, you can modularize your functions to be imported into several Dags. And if you'd like to standardize how your team interacts with tools you can write your own custom operator classes.

A best practice closely related to modularity is to treat your Dags like config files. Try to avoid having supporting code like long SQL statements or Python scripts inside your Dag code. Instead, modularize them in a separate location like an include folder to make the Dag easier to read and enable reuse of your scripts across Dags.

Avoid top-level code

By default Airflow parses all Dag files every 30 seconds. During this parse all top-level code, meaning all code outside of Airflow tasks, is executed. If you have long-running code at the top-level, it can cause Airflow to slow down, and if you make connections and calls to external systems you can even start incurring costs!

```
"""WARNING: BAD PRACTICE"""
...
# Bad practice: top-level code in a Dag file
hook = PostgresHook("postgres_conn")
results = hook.get_records("SELECT * FROM my_table;")
@dag(
    ...
)
def bad_dag():
    ...
```

A common reason users resort to top-level code when connecting to databases is the need to dynamically generate tasks. For instance, they may want to create one loading task per table in a schema but do not know the number or names of the tables in advance. A best practice for creating pipelines that can adapt to such dynamic data at runtime is to use Dynamic Task Mapping, which avoids top-level code and provides full visibility into the task copies created during each Dag run.

Automatic retries

Airflow allows you to define how often a task should be retried automatically before it fails, as well as how long to wait between retries by using the retries and retry_delay task parameters. Retries are especially important to make ETL and ELT pipelines that might run into concurrency limitations of databases more resilient.

You can set retries for your whole Airflow environment by using a config variable:

```
AIRFLOW__CORE__DEFAULT_TASK_RETRIES=5
```

If you want all tasks in one of your Dags to retry a different number of times you can override the config at the Dag level using the default_args dictionary and at the task level by using task parameters.

```
# from airflow.sdk import dag
# from datetime import timedelta

@dag(
    ...
    default_args={
        "retries": 3,
        "retry_delay": timedelta(seconds=30)
    },
)



# from airflow.sdk import task
# from datetime import timedelta

@task(retries=2, retry_delay=timedelta(minutes=10))
def extract():
    # your code
    return "hi"
```

For more information on Airflow retries, see our documentation.

Software Development Lifecycle best practices

Airflow offers the great advantage of being able to define your pipelines as Python code and you should treat your Dag code, including supporting code like SQL statements, like any application code.

This means you should:

- Store your code in a **version control system** like git.
- Use at least two **environments**, a development and production environment.
- Use tests on your Dag code.
- Deploy your code with automated testing in a **CICD** pipeline.

If you are not familiar with version control systems, see git's entry level explanation videos. In short, version control allows you to track changes to your code, making it possible to reverse specific edits or merge updates from different developers working on separate branches.

When creating ETL and ELT pipelines with Airflow you should have at least two branches for your code, the development (dev) branch where you make changes to test in a local development environment and the production (prod) branch where your code gets promoted to after thorough testing.

Deploying from the production branch to the production environment should be automatic. Astro customers can use the GitHub integration or our CICD template scripts.

As soon as you are working with a larger number of Dags, in a team and/or deploy business-critical Dags, we recommend having at least three environments:

- **Dev**: mapped to your local development environment.
- **Staging**: mapped to a cloud environment that serves as end-to-end testing.
- **Production**: mapped to a cloud environment that powers your data products in production.

When your code changes are promoted from a lower to a higher environment they go through a CICD process including automated testing.

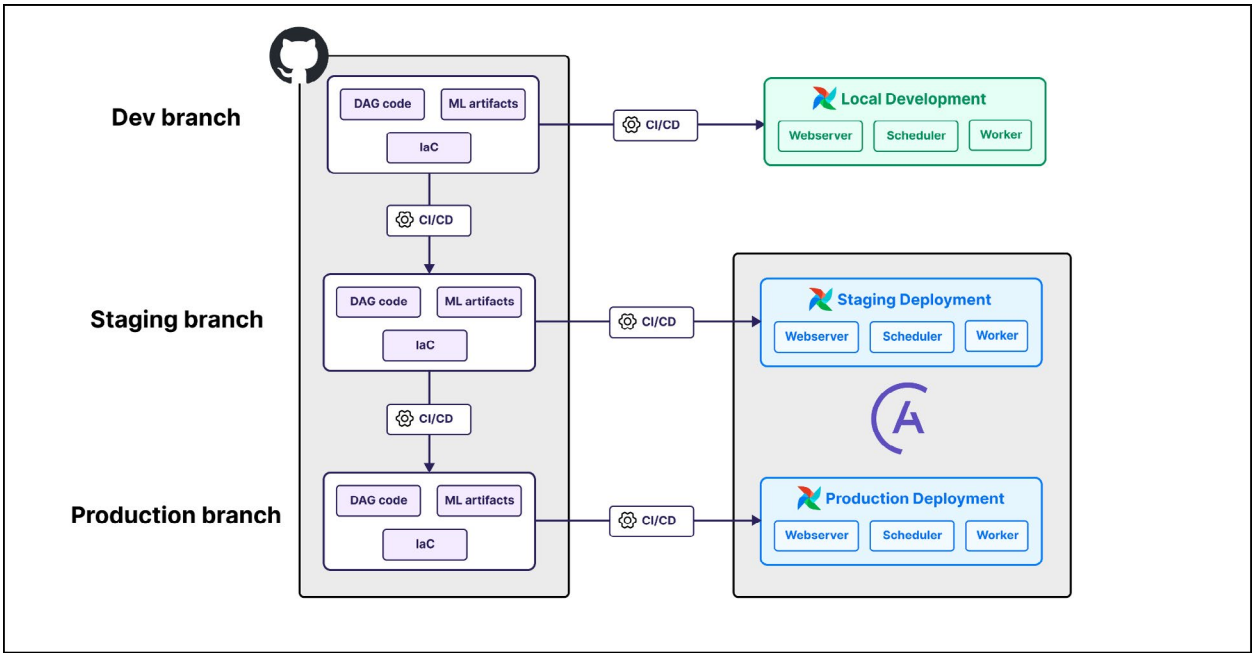

Figure 1. *Graph showing a best practice Airflow setup with three environments, version control and CICD.*

Task dependencies

In Airflow Dags, each task is a node in the graph and dependencies are the directed edges that determine how to move through the graph.

The following terms are commonly used to describe task dependencies:

- **Upstream task**: A task that must reach a specified state before a dependent task can run.
- **Downstream task**: A dependent task that cannot run until an upstream task reaches a specified state.

The focus of this section is dependencies between tasks in the same Dag. If you need to implement dependencies between Dags, using an Asset schedule is recommended. For other options see the Cross-Dag dependencies guide in our documentation.

Basic task dependencies

Basic dependencies between Airflow tasks can be set in the following ways:

- Using the `chain` function
- Using bit-shift operators (`<<` and `>>`)
- Using the `set_upstream` and `set_downstream` methods

For example, if you have a Dag with four sequential tasks (`t0` ,`t1`, `t2`, `t3`) the dependencies can be set in the following ways:

- Using the `chain` function:

```
# from airflow.sdk import chain


chain(t0, t1, t2, t3)
```

- Using `set_downstream()`:

```
t0.set_downstream(t1)

t1.set_downstream(t2)

t2.set_downstream(t3)
```

- Using `set_upstream()`:

```
t3.set_upstream(t2)

t2.set_upstream(t1)

t1.set_upstream(t0)
```

- Using `>>`:

```
t0 >> t1 >> t2 >> t3
```

- Using `<<`:

```
t3 << t2 << t1 << t0
```

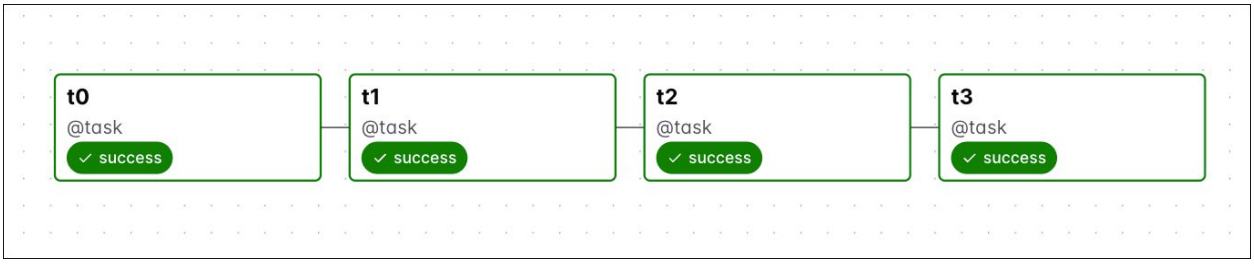

Figure 2: *Screenshot of the Airflow Dag generated using the above dependency setting methods.*

Astronomer recommends using a single method consistently.

To set a dependency where two downstream tasks are dependent on the same upstream task, use lists. For example:

```
chain(t0, t1,  [t2, t3])
```

This results in the Dag shown in the following image:

Figure 3: *Screenshot of the Airflow Dag generated using the above dependency setting methods.*

When you use bit-shift operators and the `.set_upstream` and `.set_downstream` methods, you can't set dependencies between two lists. For example, `[t0, t1] >> [t2, t3]` returns an error. To set dependencies between lists, use the chain functions described in the following section.

Chain functions

Chain functions (`chain` and `chain_linear`) are utilities that let you set dependencies between several tasks or lists of tasks. A common reason to use chain functions over bit-shift operators is to create dependencies for tasks that were created in a loop and are stored in a list.

```
from airflow.sdk import chain


list_of_tasks = []

for i in range(5):

    if i % 3 == 0:

        ta = EmptyOperator(task_id=f"ta_{i}")
```

```
        list_of_tasks.append(ta)
    else:
        ta = EmptyOperator(task_id=f"ta_{i}")
        tb = EmptyOperator(task_id=f"tb_{i}")
        tc = EmptyOperator(task_id=f"tc_{i}")
        list_of_tasks.append([ta, tb, tc])

chain(*list_of_tasks)
```

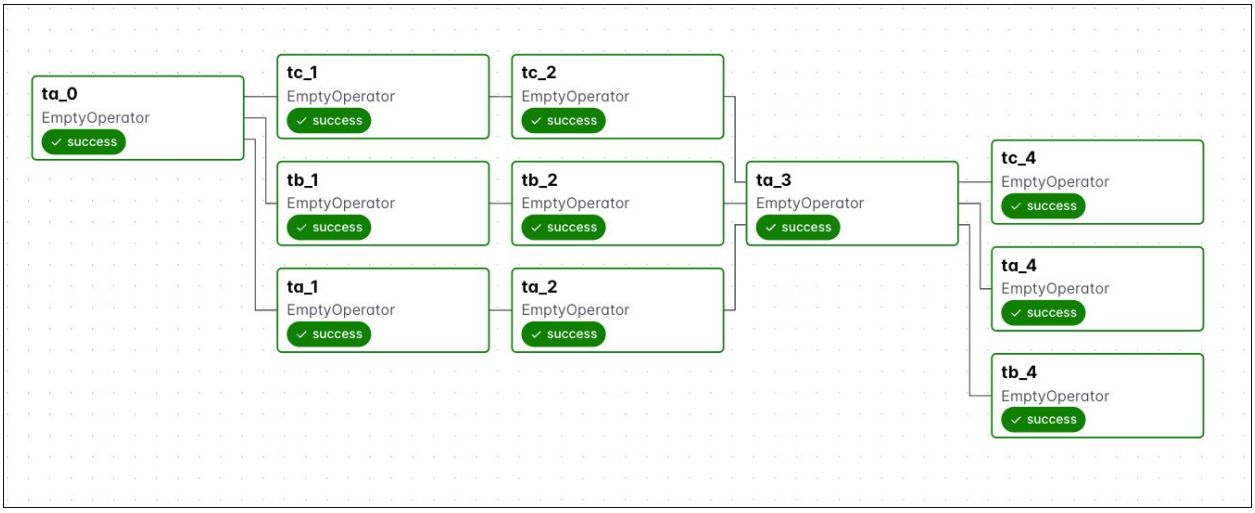

Figure 4: *Screenshot of the Airflow Dag generated from the code above.*

CHAIN VS CHAIN_LINEAR

To set parallel dependencies between tasks and lists of tasks of the same length, use the `chain` function:

```
# from airflow.sdk import chain

chain(t0, t1, [t2, t3, t4], [t5, t6, t7], t8)
```

This code creates the following Dag structure:

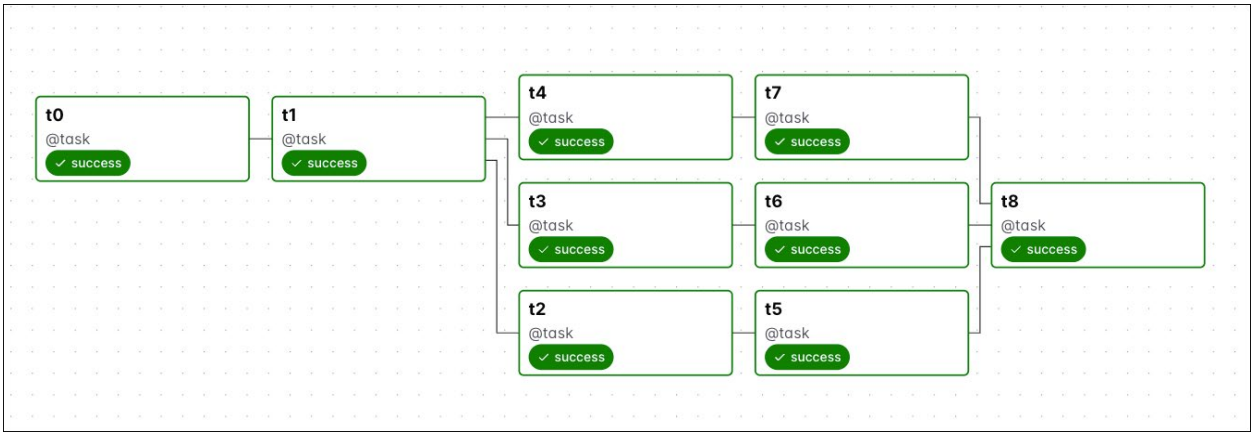

Figure 5: *Screenshot of the Airflow Dag generated from the dependency definition shown above.*

When you use the chain function, any lists or tuples that are set to depend directly on each other **need to be the same length**.

```
chain([t0, t1], [t2, t3])  # this code will work

chain([t0, t1], [t2, t3, t4])  # this code will cause an error

chain([t0, t1], t2, [t3, t4, t5])  # this code will work
```

To set interconnected dependencies between tasks and lists of tasks, use the `chain_linear()` function. This function is available in Airflow 2.7+, in older versions of Airflow you can set similar dependencies between two lists at a time using the cross_downstream() function.

Replacing `chain` in the previous example with `chain_linear` creates dependencies where each element in the downstream list will depend on each element in the upstream list.

```
# from airflow.sdk import chain_linear

chain_linear(t0, t1, [t2, t3, t4], [t5, t6, t7], t8)
```

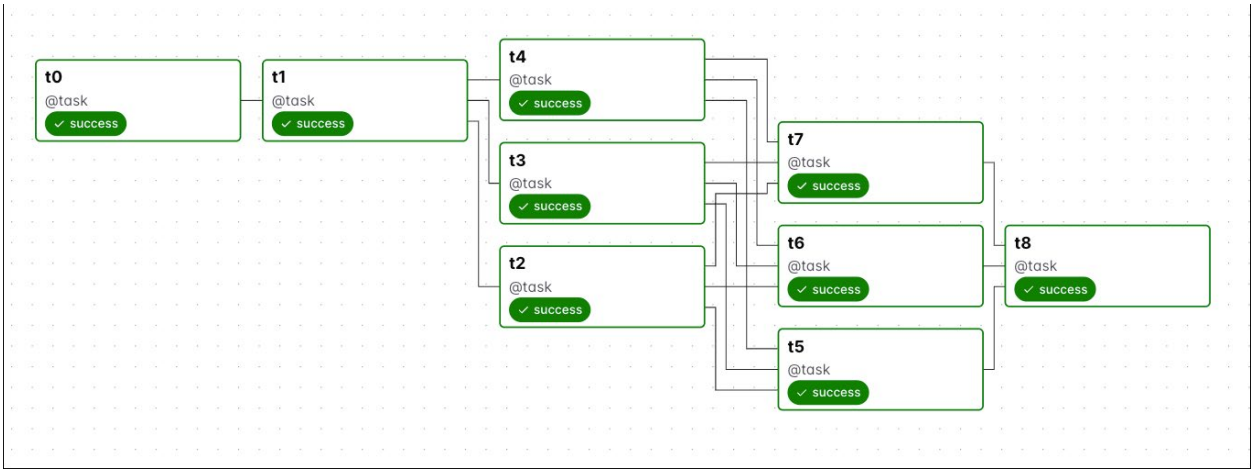

Figure 6: *Screenshot of the Airflow Dag generated from the dependency definition shown above.*

The `chain_linear` function can accept lists of any length in any order. For example, the following arguments are valid:

```
chain_linear([t0, t1], [t2, t3, t4])
```

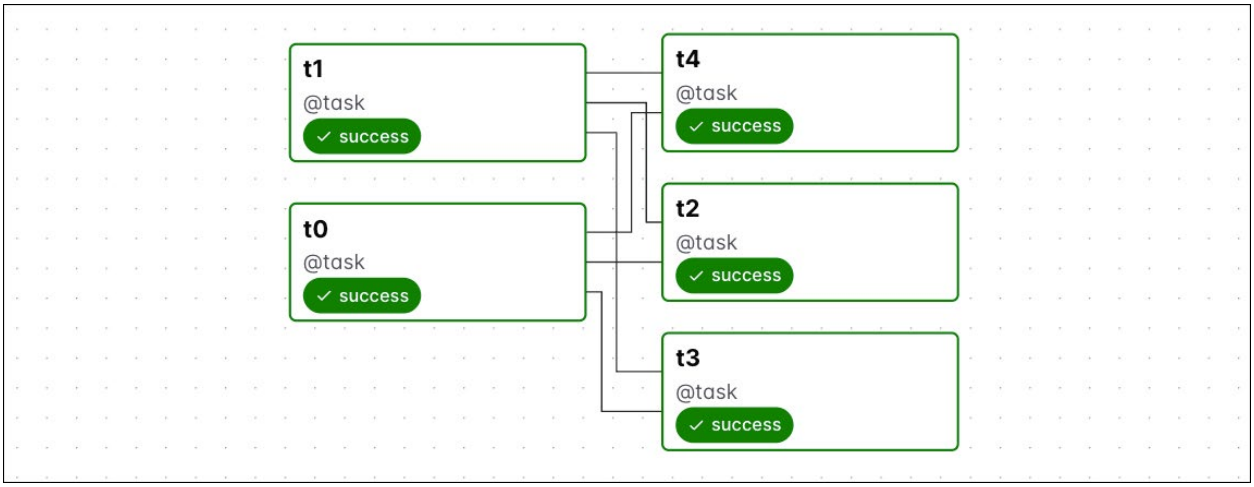

Figure 7: *Screenshot of the Airflow Dag generated from the dependency definition shown above.*

Inferred dependencies by @task

The `@task` decorator allows you to easily turn Python functions into Airflow tasks without explicitly setting the dependencies.

```
# from airflow.sdk import task

@task
def get_num():
    return 42

@task
def add_one(num):
    return num + 1

@task
def add_two(num):
    return num + 2

num = get_num()
add_one(num)
add_two(num)
```

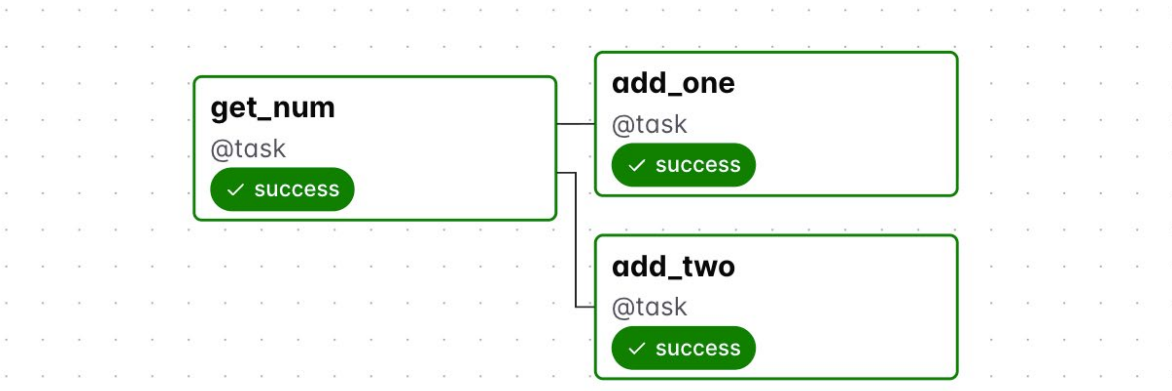

Figure 8: *Screenshot of the Airflow Dag generated from the code above.*

If your Dag contains a mix of Python function tasks defined with decorators and tasks defined with traditional operators, you can set the dependencies by assigning the decorated task invocation to a variable and then defining the dependencies using any dependency definition method.

```
# from airflow.sdk import task
# from airflow.providers.standard.operators.bash import BashOperator

@task
def t0():
    return “hi”

t1 = BashOperator(
    task_id=”t1”,
    bash_command=”echo ‘hello’”
)

_t0 = t0()
chain(_t0, t1)
```

Trigger Rules

In Airflow, trigger rules are used to determine when a task should run in relation to the previous task. By default, Airflow runs a task when all directly upstream tasks are successful. However, you can change this behavior using the trigger_rule parameter in the task definition.

DEFINE A TRIGGER RULE

You can override the default trigger rule by setting the `trigger_rule` parameter in the task definition.

```
# from airflow.sdk import task, chain

@task
def upstream_task():
    return “Hello...”

@task(trigger_rule=”all_success”)
```

```
def downstream_task():
    return " World!"

chain(upstream_task(), downstream_task())


# from airflow.providers.standard.operators.empty import EmptyOperator

upstream_task = EmptyOperator(task_id="upstream_task")
downstream_task = EmptyOperator(
    task_id="downstream_task",
    trigger_rule="all_success"
)
chain(upstream_task, downstream_task)
```

AVAILABLE TRIGGER RULES IN AIRFLOW

The following trigger rules are available:

- `all_success`: (default) The task runs only when all upstream tasks have succeeded.
- `all_failed`: The task runs only when all upstream tasks are in a failed or upstream_failed state.
- `all_done`: The task runs once all upstream tasks are done with their execution.
- `all_skipped`: The task runs only when all upstream tasks have been skipped.
- `one_failed`: The task runs when at least one upstream task has failed.
- `one_success`: The task runs when at least one upstream task has succeeded.
- `one_done`: The task runs when at least one upstream task has either succeeded or failed.
- `none_failed`: The task runs only when all upstream tasks have succeeded or been skipped.
- `none_failed_min_one_success`: The task runs only when all upstream tasks have not failed or upstream_failed, and at least one upstream task has succeeded.
- `none_skipped`: The task runs only when no upstream task is in a skipped state.
- `always`: The task runs at any time.

Trigger rules are especially important when using Airflow branching to ensure that tasks downstream of the branching task run correctly.

Branching in Airflow

When designing your data pipelines, you may encounter use cases that require more complex task flows than “Task A > Task B > Task C”. For example, you may have a use case where you need to decide between multiple tasks to execute based on the results of an upstream task. Or you may have a case where part of your pipeline should only run under certain external conditions. This can be achieved using branching tasks.

The most common way to implement branching in Airflow is to use the `@task.branch` decorator, which is the decorator version of the BranchPythonOperator. `@task.branch` accepts any Python function as an input as long as the function returns a **list of valid IDs for Airflow tasks** that the Dag should run after the function completes.

The following example uses the `choose_branch` task that returns one set of task IDs if the input provided by the `upstream` task is greater than 0.5 and a different set if the result is less than or equal to 0.5. Note that all task IDs returned by the branching task need to be valid IDs of tasks in the same Dag.

```
from airflow.sdk import dag, task, chain


@dag
def branching_example():


    @task
    def upstream():
        return 1


    @task.branch
    def choose_branch(decision_input: float):
        if decision_input > 0.5:
            return [“task_a”, “task_b”]
        return [“task_c”]


    @task
    def task_a():
        print(“Running task A”)


    @task
    def task_b():
        print(“Running task B”)
```

```
    @task
    def task_c():
        print("Running task C")

    @task(trigger_rule="none_failed")
    def downstream():
        print("Running downstream")

    _upstream = upstream()
    _choose_branch = choose_branch(decision_input=_upstream)

    chain(_choose_branch, [task_a(), task_b(), task_c()], downstream())


branching_example()
```

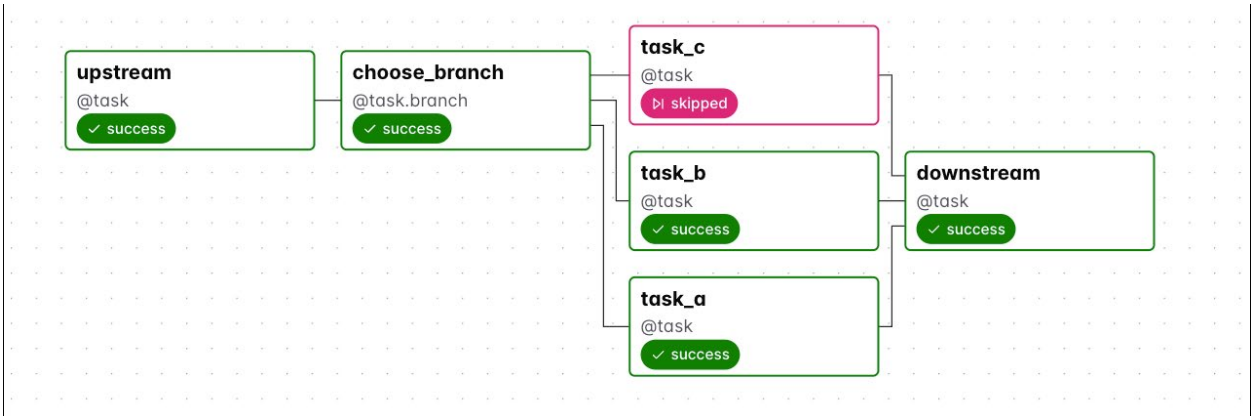

Figure 9: *Screenshot of the Airflow Dag generated from the code above.*

Whether you want to use the decorated version or the traditional operator is a question of personal preference.

Branching often leads to one or more tasks being skipped, which can cause downstream tasks to be skipped as well. To prevent accidental skipping, adjust the trigger rule of downstream tasks to run even if some of their upstream tasks were skipped. In the Dag example above this is accomplished for the `downstream` task by setting its trigger rule to `none_failed`.

@task.run_if / @task.skip_if

Airflow 2.10 added the possibility to run or skip individual `@task` decorated Airflow tasks at runtime based on whether a function provided to an additional decorator returns true or false.

The code snippet below shows how to use the `@task.skip_if` decorator with the `skip_decision` function to skip a task based on its task id.

```
def skip_decision(context):
    task_id_ending_to_skip = "_skip_me"
    return context["task_instance"].task_id.endswith(task_id_ending_to_skip)


@task.skip_if(skip_decision)
@task
def say_bye():
    return "hello!"


say_bye.override(task_id="say_bye_skip_me")()  # this task will be skipped
say_bye.override(task_id="say_bye_1234")()  # this task will run
```

The `@task.run_if` decorator functions analogously, in the below example only running tasks whose `task_id` ends with `_do_run`.

```
@task.run_if(lambda context: context["task_instance"].task_id.endswith("_do_run"))
@task
def say_hello():
    return "hello!"


say_hello.override(task_id="say_hi_do_run")()  # this task will run
say_hello.override(task_id="say_hi_1234")()   # this task will be skipped
```

Other branch operators

Airflow offers a few other branching operators that work similarly to the BranchPythonOperator but for more specific contexts:

- BranchSQLOperator: Branches based on whether a given SQL query returns true or false.
- BranchDayOfWeekOperator: Branches based on whether the current day of week is equal to a given `week_day` parameter.
- BranchDateTimeOperator: Branches based on whether the current time is between `target_lower` and `target_upper` times.
- BranchExternalPythonOperator: Branches based on a Python function like the BranchPythonOperator, but runs in a preexisting virtual environment like the ExternalPythonOperator.
- BranchPythonVirtualenvOperator: Branches based on a Python function like the BranchPythonOperator, but runs in a newly created virtual environment like the PythonVirtualenvOperator. The environment can be cached by providing a `venv_cache_path`.

Task Groups

Airflow task groups are a tool to organize tasks into groups within your Dags. Using task groups allows you to:

- Organize complicated Dags, visually grouping tasks that belong together in the Airflow UI.
- Apply `default_args` to sets of tasks, instead of at the Dag level.
- Dynamically map over groups of tasks, enabling complex dynamic patterns. See Dynamically map task groups.
- Turn task patterns into modules that can be reused across Dags or Airflow instances.

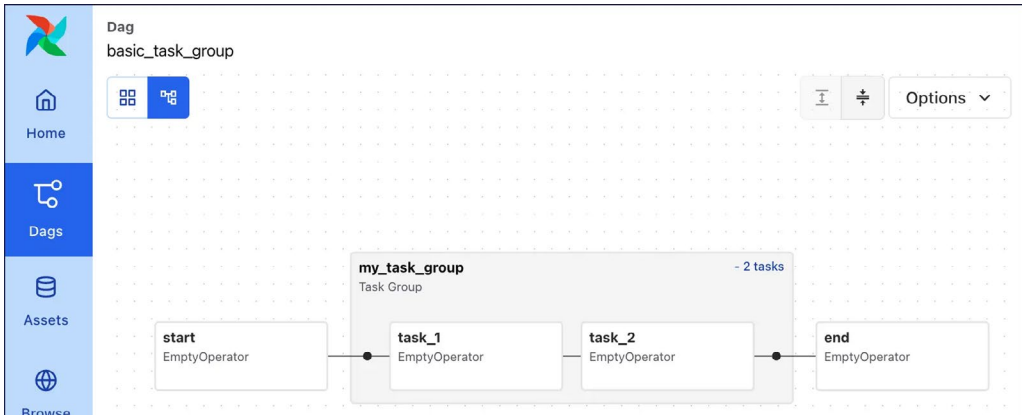

Figure 10: *Screenshot of a Dag with 4 tasks, two of which (`task_1`, `task_2`) are part of a task group (`my_task_group`).*

When to use task groups

Task groups are most often used to visually organize complicated Dags. For example, you might use task groups:

- In big ELT/ETL Dags, where you have a task group per table or schema.
- In MLOps Dags, where you have a task group per model being trained.
- In Dags owned by several teams, where you have task groups to visually separate the tasks that belong to each team. Although in this case, it might be better to separate the Dag into multiple Dags and use Assets to connect them.
- When you are using the same patterns of tasks in multiple Dags and want to create a reusable module.
- When you have an input of unknown length, for example an unknown number of files in a directory. You can use task groups to dynamically map over the input and create a task group performing sets of actions for each file. This is the only way to dynamically map sequential tasks in Airflow.

Define task groups

There are two ways to define task groups in your Dags:

- Use the `TaskGroup` class to create a task group context.
- Use the `@task_group` decorator on a Python function.

In most cases, it is a matter of personal preference which method you use. The only exception is when you want to dynamically map over a task group; this is possible only when using `@task_group`.

The following code shows how to instantiate a simple task group containing two sequential tasks. You can set dependencies both within and between task groups in the same way that you can with individual tasks.

In the Airflow UI you can collapse and expand task groups as well as clear them to rerun all tasks they contain.

```
# from airflow.sdk import task_group, chain


t0 = EmptyOperator(task_id='start')


# Start task group definition
@task_group(group_id='my_task_group')
def tg1():
    t1 = EmptyOperator(task_id='task_1')
    t2 = EmptyOperator(task_id='task_2')


    chain(t1, t2)
```

```
# End task group definition

t3 = EmptyOperator(task_id='end')

# Set task group's (tg1) dependencies
chain(t0, tg1(), t3)



# from airflow.sdk import TaskGroup

t0 = EmptyOperator(task_id='start')

# Start task group definition
with TaskGroup(group_id='my_task_group') as tg1:
    t1 = EmptyOperator(task_id='task_1')
    t2 = EmptyOperator(task_id='task_2')

    t1 >> t2
# End task group definition

t3 = EmptyOperator(task_id='end')

# Set task group's (tg1) dependencies
t0 >> tg1 >> t3
```

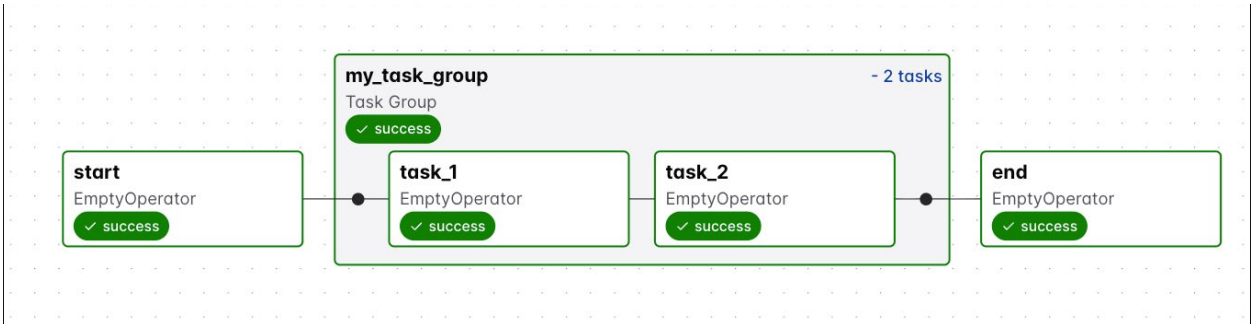

Figure 11: *Screenshot of the Airflow Dag generated from the code above.*

Task group parameters

You can use parameters to customize individual task groups. The two most important parameters are the `group_id` which determines the name of your task group, as well as the `default_args` which will be passed to all tasks in the task group. The following examples show task groups with some commonly configured parameters:

```
@task_group(
    group_id="task_group_1",
    default_args={"conn_id": "postgres_default"},
    prefix_group_id=True,
    # parent_group=None,
    # dag=None,
)
def tg1():
    t1 = EmptyOperator(task_id="t1")

tg1()
```

task_id in task groups

When your task is within a task group, your callable `task_id` will be `group_id.task_id`. This ensures the `task_id` is unique across the Dag. It is important that you use this format when referring to specific tasks when working with XComs or branching. You can disable this behavior by setting the task group parameter `prefix_group_id=False`.

For example, the `task_1` task in the following Dag has a `task_id` of `my_outer_task_group.my_inner_task_group.task_1`.

```
@task_group(group_id="my_outer_task_group")
def my_outer_task_group():
    @task_group(group_id="my_inner_task_group")
    def my_inner_task_group():
        EmptyOperator(task_id="task_1")

    my_inner_task_group()

my_outer_task_group()
```

Passing data through task groups

When you use the `@task_group` decorator, you can pass data through the task group just like with regular `@task` decorators:

```
from airflow.sdk import dag, task, task_group
from pendulum import datetime
import json



@dag
def task_group_example():
    @task
    def extract_data():
        data_string = '{"1001": 301.27, "1002": 433.21, "1003": 502.22}'
        order_data_dict = json.loads(data_string)
        return order_data_dict

    @task
    def transform_sum(order_data_dict: dict):
        total_order_value = 0
        for value in order_data_dict.values():
            total_order_value += value

        return {"total_order_value": total_order_value}

    @task
    def transform_avg(order_data_dict: dict):
        total_order_value = 0
        for value in order_data_dict.values():
            total_order_value += value
            avg_order_value = total_order_value / len(order_data_dict)

        return {"avg_order_value": avg_order_value}

    @task_group
    def transform_values(order_data_dict):
```

```
        return {
            "avg": transform_avg(order_data_dict),
            "total": transform_sum(order_data_dict),
        }

    @task
    def load(order_values: dict):
        print(
            f"""Total order value is: {order_values['total']['total_order_value']:.2f}
            and average order value is: {order_values['avg']['avg_order_value']:.2f}"""
        )

    load(transform_values(extract_data()))


task_group_example()
```

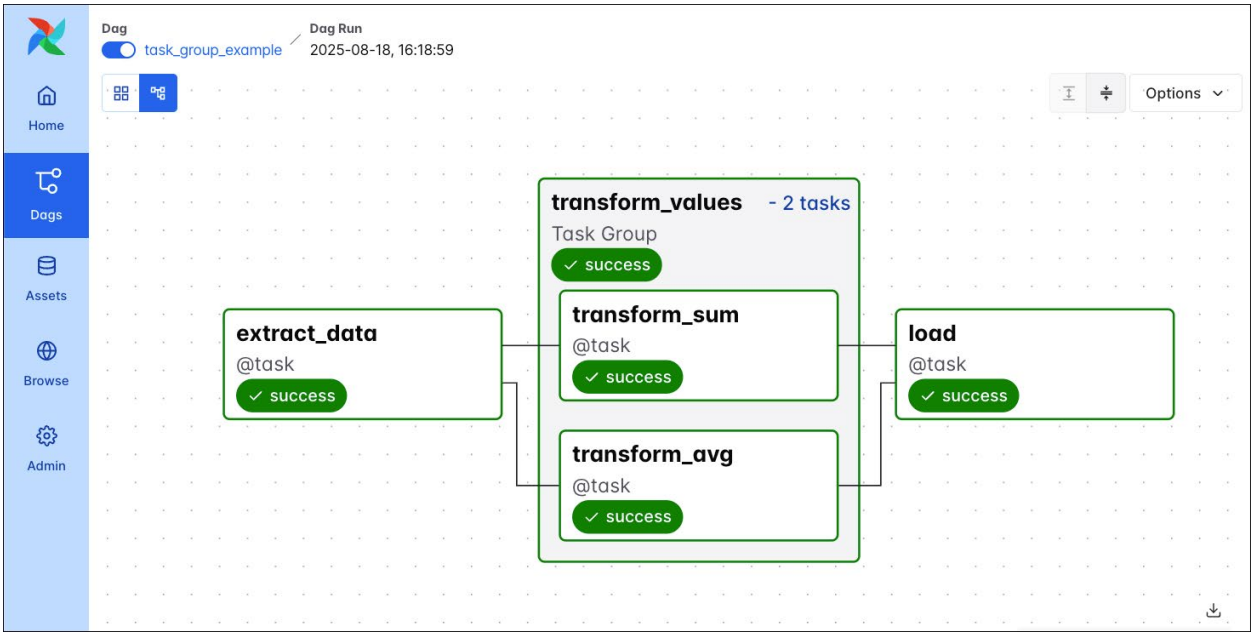

Figure 12: *Screenshot of the Airflow Dag generated from the code above.*

There are a few things to consider when passing information into and out of task groups:

- If downstream tasks require the output of tasks that are in the task group decorator, then the task group function must return a result. In the previous example, a dictionary with two values was returned, one from each of the tasks in the task group, that were then passed to the downstream `load()` task.
- If your task group function returns an output that another task takes as an input, Airflow can infer the task group and task dependency with the TaskFlow API. If your task group function's output isn't used as a task input, you must explicitly define your dependencies.

Nest task groups

You can nest task groups by defining a task group indented within another task group. There is no limit to how many levels of nesting you can have.

```
groups = []
for g_id in range(1,3):
    @task_group(group_id=f"group{g_id}")
    def tg1():
        t1 = EmptyOperator(task_id="task1")
        t2 = EmptyOperator(task_id="task2")
        sub_groups = []
        for s_id in range(1,3):
            @task_group(group_id=f"sub_group{s_id}")
            def tg2():
                st1 = EmptyOperator(task_id="task1")
                st2 = EmptyOperator(task_id="task2")
                st1 >> st2
            sub_groups.append(tg2())
        t1 >> sub_groups >> t2
    groups.append(tg1())
groups[0] >> groups[1]
```

The following image shows the nested task groups in the Airflow UI:

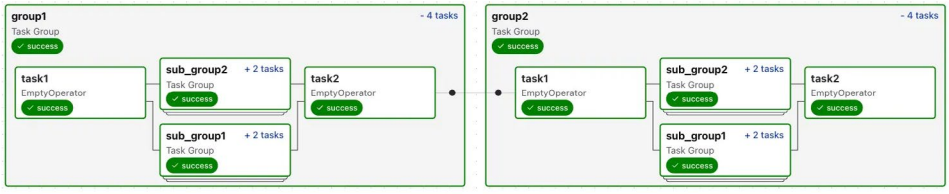

Figure 13: *Screenshot of the Airflow Dag generated from the code above.*

Task group dependencies

This section will explain how to set dependencies between task groups.

Dependencies can be set both inside and outside of a task group. For example, in the following Dag code there is a start task, a task group with two dependent tasks, and an end task. All of these tasks need to happen sequentially. The dependencies between the two tasks in the task group are set within the task group's context (`t1 >> t2`). The dependencies between the task group and the start and end tasks are set within the Dag's context (`t0 >> tg1() >> t3`).

```
t0 = EmptyOperator(task_id="start")


# Start task group definition
@task_group(
    group_id="group1"
)
def tg1():
    t1 = EmptyOperator(task_id="task1")
    t2 = EmptyOperator(task_id="task2")


    t1 >> t2
# End task group definition


t3 = EmptyOperator(task_id="end")


# Set task group's (tg1) dependencies
t0 >> tg1() >> t3
```

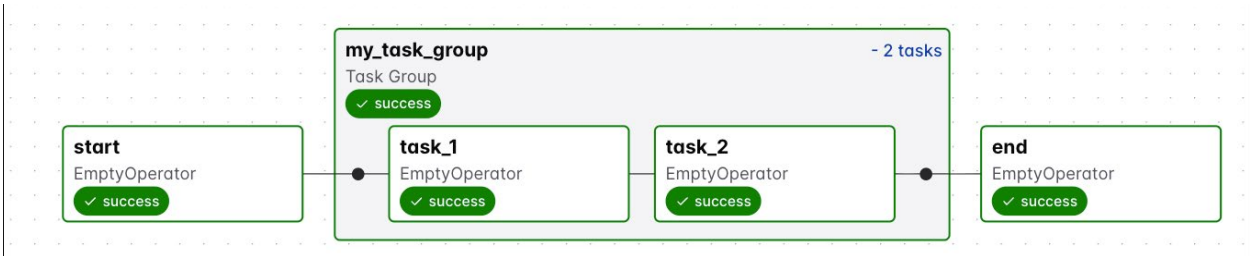

***Figure 14**: Screenshot of the Airflow Dag generated from the code above.*

You can also set dependencies between task groups, between tasks inside and out of task groups, and even between tasks in different (nested) task groups.

The image below shows types of dependencies that can be set between tasks and task groups. You can find the code that created this Dag in a GitHub repository both for the TaskFlow API and traditional version.

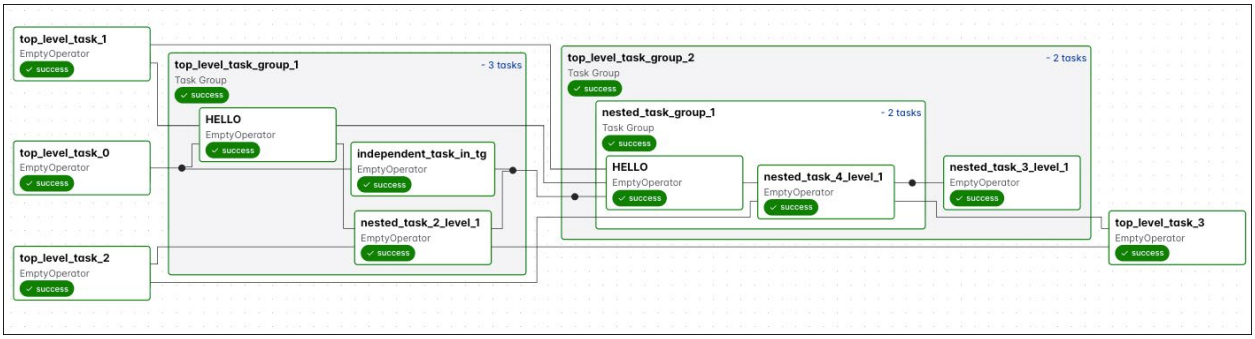

***Figure 15**: Screenshot of the Airflow Dag with complex dependencies between tasks inside and outside of task groups.*

Modularized task groups

If you have a common pattern of tasks, consider creating a Python class representing that pattern inside a task group that can be instantiated with just a few parameters.

For example, imagine you have the same extract, transform, load pattern as a part of many of your Dags, and only a few parameters like a URL or key of interest change. You can standardize the pattern in a custom class like the `MyETLTaskGroup` shown in the code below.

```
from airflow.sdk import TaskGroup, task


class MyETLTaskGroup(TaskGroup):
    """A task group containing a simple ETL pattern."""

    def __init__(self, group_id, url: str, key_of_interest: str, **kwargs):
        super().__init__(group_id=group_id, **kwargs)

        self.url = url

        @task(task_group=self)
        def extract(url: str) -> dict:
            """Extracts data from a URL.
            Args:
```

```
            url (str): The URL to extract data from."""
        import requests

        response = requests.get(url)
        return response.json()

    @task(task_group=self)
    def transform(api_response: dict, key: str) -> str:
        """Transforms the api_response.
        Args:
            api_response (dict): The number to transform."""

        return api_response[key]

    @task(task_group=self)
    def load(transformed_data: str):
        """Prints the transformed data (simulating loading).
        Args:
            transformed_data (dict): The transformed data."""
        print(transformed_data)

    load(transform(api_response=extract(self.url), key=key_of_interest))
```

This class can be imported and used in several Dags to create a three task pattern in just one line of code.

```
from airflow.sdk import dag, task
from include.custom_task_group.etl_task_group import MyETLTaskGroup


@dag
def simple_etl_task_group():
    @task
    def get_url():
        return "https://catfact.ninja/fact"

    MyETLTaskGroup(group_id="my_task_group", url=get_url(), key_of_interest="fact")


simple_etl_task_group()
```

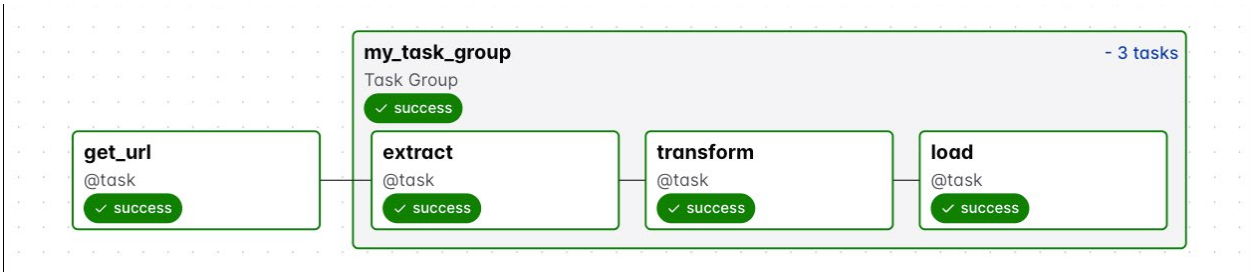

***Figure 16**: Screenshot of the Dag graph created using the modularized task group.*

You can learn more about task groups, including custom modularized task groups in this guide and this webinar.

XCom

XCom stands for "cross-communication" and is Airflow's default way of passing data between tasks. XComs are defined by a key, value, and timestamp.

XComs can be "pushed", meaning sent by a task, or "pulled", meaning received by a task. When an XCom is pushed, it is stored in the Airflow metadata database and made available to all other tasks. Any time a task returns a value (for example, when your Python callable for your PythonOperator has a `return` value), that value is automatically pushed to XCom. Tasks can also be configured to push XComs by calling the `xcom_push()` method. Similarly, `xcom_pull()` can be used in a task to receive an XCom.

You can view your XComs in the Airflow UI by going to **Browse > XComs**.

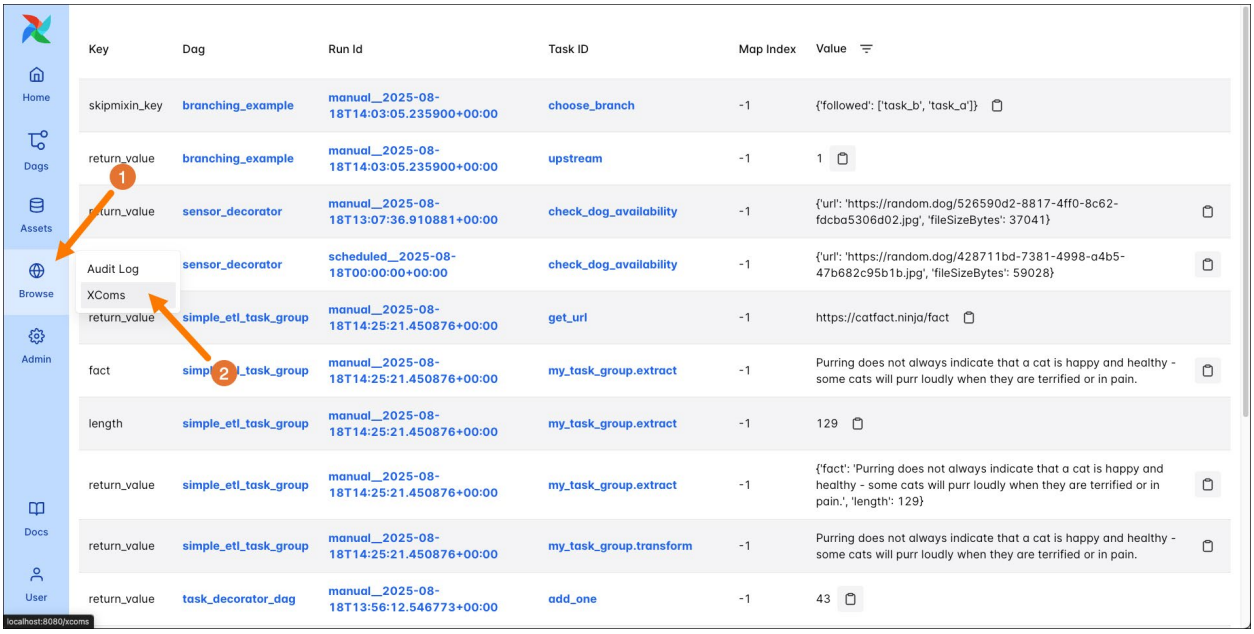

***Figure 17**: Screenshot of the Airflow UI showing the XCom List.*

Standard XCom backend

Standard XComs should only be used to pass small amounts of data between tasks. For example, task metadata, dates, model accuracy, or single value query results are all ideal data to use with XCom. If you want to pass larger amounts of data with XCom you should use a custom XCom backend and make sure to scale your Airflow resources accordingly.

The standard XCom backend stores all data that is being passed between tasks in a table in the Airflow metadata database.

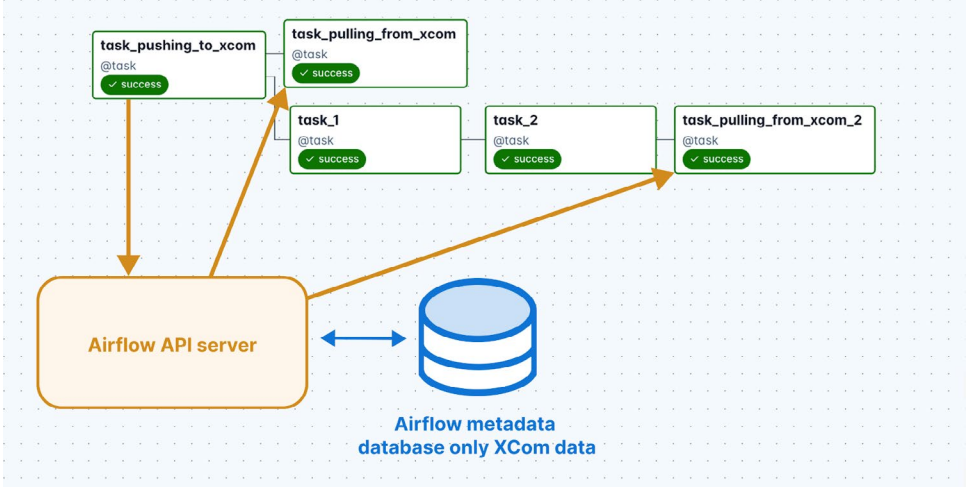

***Figure 18**: Screenshot of a Dag pushing data to XCom and pulling data from XCom using a standard XCom backend.*

When you use the standard XCom backend, the size-limit for an XCom is determined by your metadata database. Common sizes are:

- Postgres: 1 Gb
- SQLite: 2 Gb
- MySQL: 64 Mb

The second limitation in using the standard XCom backend is that only certain types of data can be serialized.

By default, Airflow supports serializations for:

- JSON
- pandas DataFrame (Airflow version 2.6+)
- Delta Lake tables (Airflow version 2.8+)
- Apache Iceberg tables (Airflow version 2.8+)

If you need to serialize other data types you can do so using a custom XCom backend.

Custom XCom backends

As Airflow evolved, the need to pass large amounts of data from task to task arose and with it came *custom XCom backends*.

When using a custom XCom backend, the data is stored in a system external to Airflow, typically an object storage solution like Amazon S3, GCS, or Azure blob storage. The Airflow metadata database only saves a string referencing the data location, typically a URI.

Custom XCom backends allow data to be saved between tasks in any location and format, removing limitations on data size and type.

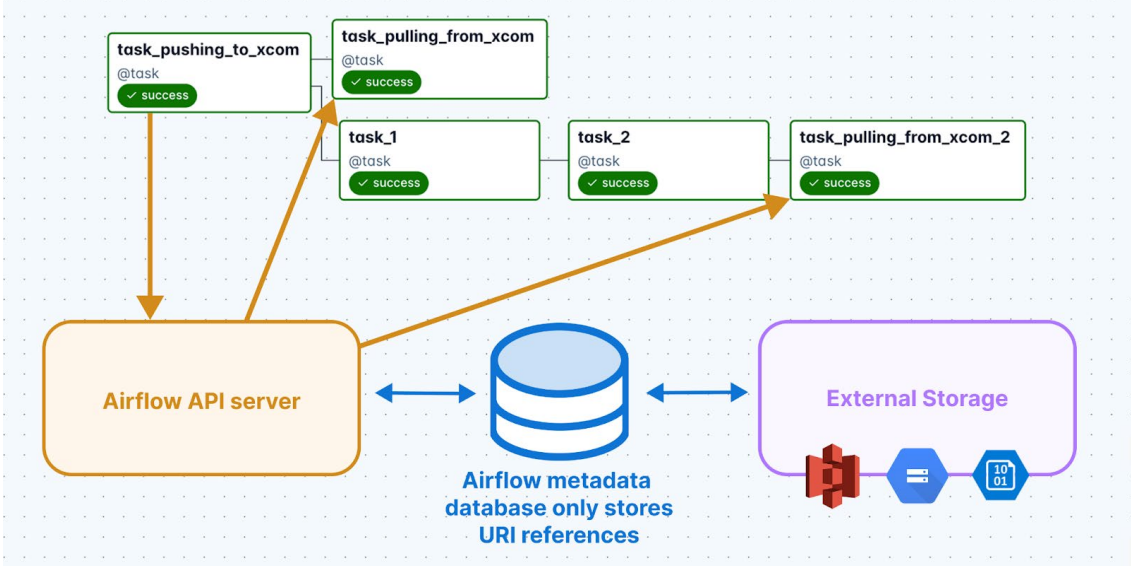

***Figure 19**: Custom XCom backend. Data is serialized and written to external storage, the Airflow metadata database only stores a reference to the data location.*

There are two main ways you can create a custom XCom backend:

- **Create your own custom XCom backend class**, see our documentation for a code example.
- **Use the Object Storage XCom backend,** the easy way to set up a custom XCom backend using only environment variables.

The other common pattern is to explicitly write your data to external storage from within an Airflow task using any tool to interact with the external storage solution. This pattern gives you complete flexibility over what data is saved where and in which format from each task.

It is a best practice to use a custom XCom backend in production environments if any significant amounts of data is passed through XCom, which has become simple to set up with the Object Storage XCom backend.

Additionally, the Airflow Object Storage feature allows you to create a custom XCom backend only by setting a few environment variables. You can find an example in the code snippet below that will create a custom XCom backend in the my-bucket bucket in S3 where any XCom larger than 1000 bytes will be stored, compressed in zip format. Note that you need to have the Common IO provider installed to use this feature.

```
AIRFLOW_CONN_AWS_DEFAULT='{
   "conn_type":"aws",
   "login":"<YOUR AWS KEY ID>",
   "password":"<YOUR AWS SECRET KEY>"
}'

AIRFLOW__CORE__XCOM_BACKEND="airflow.providers.common.io.xcom.backend.XComObjectStorageBackend"
AIRFLOW__COMMON_IO__XCOM_OBJECTSTORAGE_PATH="s3://my_aws_conn@my-bucket"
AIRFLOW__COMMON_IO__XCOM_OBJECTSTORAGE_THRESHOLD="1000"
AIRFLOW__COMMON_IO__XCOM_OBJECTSTORAGE_COMPRESSION="zip"
```

You can learn more about the Airflow Object Storage feature in this tutorial and about the Object Storage custom XCom backend in this tutorial.

Passing data: @task vs traditional operators

PASSING THE OUTPUT OF A TRADITIONAL OPERATOR

Pass the `.output` values of traditional Operator-based tasks to the callable of TaskFlow tasks to automatically create a relationship between the two tasks. For example:

```
# from airflow.sdk import task
# from airflow.providers.standard.operators.python import PythonOperator

def first_task_callable():
    return "hello world"

first_task = PythonOperator(
    task_id="first_task",
```

```
    python_callable=first_task_callable
)

first_task_result = first_task.output

@task
def second_task(first_task_result_value):
    return f"{first_task_result_value} and hello again"

# Providing first_task_result (traditional operator result) as an argument to a second_task
# (a TaskFlow function) automatically registers the dependency between the tasks.
second_task(first_task_result)
```

PASSING THE OUTPUT OF A @TASK DECORATOR

TaskFlow tasks, when called, return a reference (referred to internally as an `XComArg`) that can be passed into templateable fields of traditional operators to automatically create a relationship between the two tasks.

The list of templateable fields varies by operator. Astronomer maintains a searchable registry of operators at the Astronomer Registry that details which fields are templateable by default and users can control which fields are templateable.

Here's how you could pass the result of a TaskFlow function to a traditional PythonOperator's callable via an argument:

```
@task
def first_task():
    return "hello"

first_task_result = first_task()

def second_task_callable(x):
    uppercase_text = x.upper()
    return f"Upper cased version of prior task result: {uppercase_text}"

# when first_task_result (an XComArg) is provided as an argument to a templated function (op_
args)
# Airflow automatically registers that second_task depends on first_task
second_task = PythonOperator(
    task_id="second_task",
```

```
    python_callable=second_task_callable,
    op_args=[ first_task_result ] # note op_args requires a LIST of XComArgs
)

# Astronomer recommends against explicitly defining this redundant-reference
# first_task_result >> second_task
```

You can find more examples of passing data between tasks here.

Want to know more?

- Airflow: XComs 101 Astronomer Academy Module
- Airflow: Branching Astronomer Academy Module
- Airflow: Task Groups Astronomer Academy Module
- How to write better Dags webinar
- How to expertly organize your Dags with task groups webinar
- Dag writing best practices in Apache Airflow guide
- Manage task and task group dependencies in Airflow guide
- Airflow trigger rules guide
- Branching in Airflow guide
- Airflow task groups guide
- Pass data between tasks
- Strategies for custom XCom backends in Airflow
- Mixing TaskFlow decorators with traditional operators guide

Scheduling Dags

5

KEY CONCEPTS

- **Schedule**: The schedule of a Dag determines when it runs.
- Cron: is a command line utility for simple time-based schedules. Airflow offers the option to use cron syntax to define time-based schedules for Dags.
- Asset: an asset is an Airflow feature that allows you to define data-driven schedules by making a Dag run as soon as the data it operates on is available.

Scheduling in Airflow

One of the fundamental features of Apache Airflow is the ability to schedule jobs. Historically, Airflow users scheduled their Dags by specifying a schedule with a cron expression, a timedelta object, or a preset Airflow schedule. Recent versions of Airflow have added new ways to schedule Dags, including data-aware scheduling with assets to event-driven scheduling based on messages in a queue.

To gain a better understanding of Dag scheduling, it's important that you become familiar with the following timestamps:

- **Logical Date**: The point in time after which a specific Dag run can start. This timestamp is displayed prominently in the Airflow UI as the main date the Dag run is associated with. The logical date can be set to `None` by the user for Dags that are triggered using the Airflow REST API or Airflow UI.
- **Run after**: The point in time after which a specific Dag run can start. If a logical date is supplied, the run after date is set to the logical date. If the logical date is `None` the run after date is set to the current time as soon as the Dag run is triggered.
- **Start and Start Date**: The time the Dag run actually started. This timestamp is unrelated to the Dag parameter `start_date`.
- **End and End Date**: The time the Dag run finished. This timestamp is unrelated to the Dag parameter `end_date`.
- **Duration and Run Duration**: The time it took for the Dag run to complete, difference between the start and end date timestamps.
- **Run ID**: The unique identifier for the Dag run. The run ID is a combination of the type of Dag run, for example scheduled and the logical date. If the logical date is `None`, the run after date is used with an added random string suffix to ensure uniqueness. The run ID is used to identify the Dag run in the Airflow metadata database.

There are two additional timestamps that are only meaningful when using the `CronDataIntervalTimetable`.

- **Data Interval Start**: When the `CronDataIntervalTimetable` is used, the data interval start timestamp of a Dag run is equivalent to the run after date of the previous scheduled Dag run of the same Dag. When using other schedules, the data interval start timestamp is equivalent to the run after date of the current Dag run. If the Dag run is triggered with the logical date set to `None`, the data interval start timestamp is also `None`.
- **Data Interval End**: When the `CronDataIntervalTimetable` is used, the data interval start timestamp of a Dag run is equivalent to the run after date of the current scheduled Dag run. If the logical date is set to `None`, the data interval end timestamp is also `None`.

For more information on the data intervals and the differences between the `CronDataIntervalTimetable` and the `CronTriggerTimetable`, see the Airflow documentation.

Parameters

The following parameters ensure your Dags run at the correct time:

- `start_date`: The timestamp after which this Dag can start running. When using the `CronDataIntervalTimetable`, the `start_date` is the moment in time after which the first data interval can start. Default: `None`.
- `schedule`: Defines the rules according to which Dag runs are scheduled. This parameter accepts cron expressions, timedelta objects, timetables, and lists of assets. Default: `None`.
- `end_date`: The date beyond which the Dag won't run. Default: `None`.
- `catchup`: A boolean that determines whether the Dag should automatically fill all runs between its `start_date` and the current date. Default: `False`. Aside from this automatic behavior, you can also manually trigger Dag runs for any date in the past. See Backfills for more information.

Cron-based schedules

For pipelines with straightforward scheduling needs, you can define a schedule in your Dag using:

- A cron expression.
- A cron preset.
- A timedelta object.

Setting a cron-based schedule

CRON EXPRESSIONS

You can pass any cron expression as a string to the schedule parameter in your Dag. For example, if you want to schedule your Dag at 4:05 AM every day, you would use `schedule='5 4 * * *'`.

If you need help creating the correct cron expression, see crontab guru.

CRON PRESETS

Airflow can utilize cron presets for common, basic schedules. For example, `schedule='@hourly'` will schedule the Dag to run at the beginning of every hour. For the full list of presets, see Cron Presets. If your Dag does not need to run on a schedule and will only be triggered manually or externally triggered by another process, you can omit the schedule parameter and it will default to `None`.

TIMEDELTA OBJECTS

If you want to schedule your Dag on a particular cadence (hourly, every 5 minutes, etc.) rather than at a specific time, you can pass a timedelta object imported from the datetime package to the schedule parameter. For example, `schedule=timedelta(minutes=30)` will run the Dag every thirty minutes, and `schedule=timedelta(days=1)` will run the Dag every day.

Do not make your Dag's schedule dynamic (e.g. `datetime.now()`)! This will cause an error in the Scheduler.

Limitations of cron-based schedules

Some time-based schedules cannot be expressed as either, as cron strings or timedelta objects. The following are examples of the limitations of a cron-based schedule:

- Schedule a Dag at different times on different days. For example, 2:00 PM on Thursdays and 4:00 PM on Saturdays.
- Schedule a Dag daily except for holidays.
- Schedule a Dag at multiple times daily with uneven intervals. For example, 1:00 PM and 4:30 PM.

If you want to implement more complex time-based schedules like these, you can use a Timetable.

Data-driven Scheduling

With Assets, Dags that access the same data can have explicit, visible relationships, and Dags can be scheduled based on updates to these assets. This feature helps make Airflow data-aware and expands Airflow scheduling capabilities beyond time-based methods such as cron.

Assets can help resolve common issues. For example, consider a data engineering team with a Dag that creates an asset and a machine learning team with a Dag that trains a model on the asset. Using assets, the machine learning team's Dag runs only when the data engineering team's Dag has produced an update to the asset.

Why use Airflow assets

Assets allow you to define explicit dependencies between Dags and updates to your data. This helps you to:

- Standardize communication between teams. Assets can function like an API to communicate when data in a specific location has been updated and is ready for use.

- Reduce the amount of code necessary to implement cross-Dag dependencies. Even if your Dags don't depend on data updates, you can create a dependency that triggers a Dag after a task in another Dag updates an asset.
- Get better visibility into how your Dags are connected and how they depend on data. The Assets graphs in the Airflow UI display how assets and Dags depend on each other and can be used to navigate between them.
- Reduce costs, because assets do not use a worker slot in contrast to sensors or other implementations of cross-Dag dependencies.
- Create cross-deployment dependencies using the Airflow REST API. Astronomer customers can use the Cross-deployment dependencies best practices documentation for guidance.
- Create complex data-driven schedules using Conditional Asset Scheduling and Combined Asset and Time-based Scheduling.
- Schedule a Dag based on messages in a message queue with event-driven scheduling.

Asset Concepts

You can define assets in your Dag code and use them to create cross-Dag dependencies. Airflow uses the following terms related to the assets feature:

- **Asset**: an object in Airflow that represents a concrete or abstract data entity and is defined by a unique name. Optionally, a URI can be attached to the asset, when it represents a concrete data entity, like a file in object storage or a table in a relational database.
- **@asset**: a decorator that can be used to write a Dag with the asset-oriented approach, directly declaring the desired asset in Python with less boilerplate code. Using @asset creates a Dag with a single task that updates an asset.
- **Asset event**: an event that is attached to an asset and created whenever a producer task updates that particular asset. An asset event is defined by being attached to a specific asset plus the timestamp of when a producer task updated the asset. Optionally, an asset event can contain an `extra` dictionary with additional information about the asset or asset event.
- **Asset schedule**: the schedule of a Dag that is triggered as soon as asset events for one or more assets are created. All assets a Dag is scheduled on are shown in the Dag graph in the Airflow UI, as well as reflected in the dependency graph of the Assets tab.
- **Producer task**: a task that produces updates to one or more assets provided to its `outlets` parameter, creating asset events when it completes successfully.
- **Asset expression**: a logical expression using AND (`&`) and OR (`|`) operators to define the schedule of a Dag scheduled on updates to several assets.

- **Queued asset event**: It is common to have Dags scheduled to run as soon as a set of assets have received at least one update each. While there are still asset events missing to trigger the Dag, all asset events for other assets the Dag is scheduled on are queued asset events. A queued asset event is defined by its asset, timestamp and the Dag it is queuing for. One asset event can create a queued asset event for several Dags. You can access queued Asset events for a specific Dag or a specific asset programmatically, using the Airflow REST API.
- **AssetAlias**: an object that can be associated to one or more assets and used to create schedules based on assets created at runtime, see Use asset aliases. An asset alias is defined by a unique name.
- **Metadata**: a class to attach `extra` information to an asset from within the producer task. This functionality can be used to pass asset-related metadata between tasks, see Attaching information to an asset event.
- **AssetWatcher**: a class that is used in event-driven scheduling to watch for a `TriggerEvent` caused by a message in a message queue.

Two parameters relating to Airflow assets exist in all Airflow operators and decorators:

- `Outlets`: a task parameter that contains the list of assets a specific tasks produces updates to, as soon as it completes successfully. All outlets of a task are shown in the Dag graph in the Airflow UI, as well as reflected in the dependency graph of the Assets tab as soon as the Dag code is parsed, i.e. independently of whether or not any asset events have occurred. Note that Airflow is not yet aware of the underlying data. It is up to you to determine which tasks should be considered producer tasks for an asset. As long as a task has an outlet asset, Airflow considers it a producer task even if that task doesn't operate on the referenced asset.
- `Inlets`: a task parameter that contains the list of assets a specific task has access to, typically to access extra information from related asset events. Defining inlets for a task does not affect the schedule of the Dag containing the task and the relationship is not reflected in the Airflow UI.

To summarize, tasks produce updates to assets given to their `outlets` parameter, and this action creates asset events. Dags can be scheduled based on asset events created for one or more assets, and tasks can be given access to all events attached to an asset by defining the asset as one of their `inlets`. An asset is defined as an object in the Airflow metadata database as soon as it is referenced in either the `outlets` parameter of a task or the `schedule` of a Dag.

Using assets

When you work with assets, keep the following considerations in mind:

- Assets events are only registered by Dags or listeners in the same Airflow environment. If you want to create cross-Deployment dependencies with Assets you will need to use the Airflow REST API to create an asset event in the Airflow environment where your downstream Dag is located. See the Cross-deployment dependencies for an example implementation on Astro.
- Airflow monitors assets only within the context of Dags and tasks. It does not monitor updates to assets that occur outside of Airflow. I.e. Airflow will not notice if you manually add a file to an S3 bucket referenced by an asset. To create Airflow dependencies based on outside events, use Airflow sensors, deferrable operators or consider using a message queue as an intermediary and implement event-driven scheduling.
- The Assets tab in the Airflow UI provides a list of active assets in your Airflow environment with an asset graph for each asset showing its dependencies to Dags and other assets.

Asset definition

An asset is defined as an object in the Airflow metadata database as soon as it is referenced in either the `outlets` parameter of a task or the schedule of a Dag. For information on how to create an asset alias, see Use Asset Aliases.

BASIC ASSET DEFINITION

The simplest asset schedule is one Dag scheduled based on updates to one asset which is produced to by one task. In this example we define that the `my_producer_task` task in the `my_producer_dag` Dag produces updates to the `my_asset asset`, creating attached asset events, and schedule the `my_consumer_dag` Dag to run once for every asset event created.

First, provide the asset to the outlets parameter of the producer task.

```
from airflow.sdk import Asset, dag, task

@dag
def my_producer_dag():

    @task(outlets=[Asset("my_asset")])
    def my_producer_task():
        pass
```

```
    my_producer_task()

my_producer_dag()
```

You can see the relationship between the Dag containing the producing task (`my_producer_dag`) and the asset in the Asset Graph located in the Assets tab of the Airflow UI.

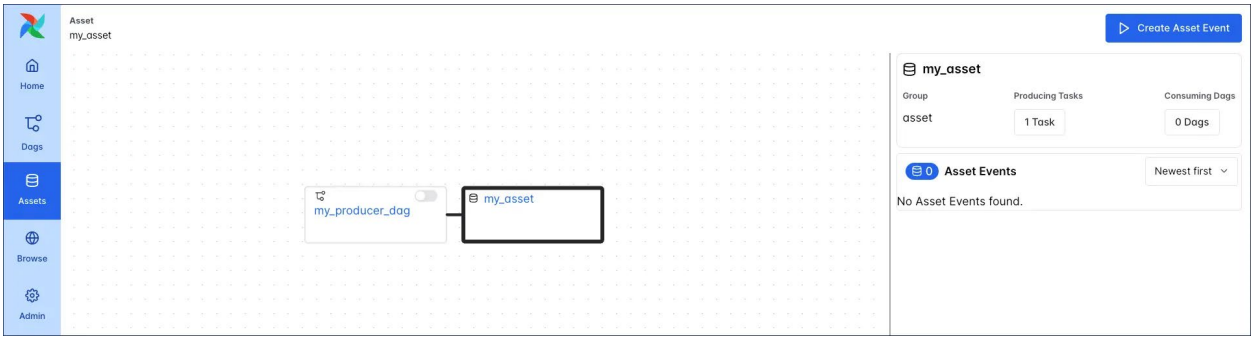

***Figure 1**: Screenshot of the Airflow UI Assets view showing the Dag containing the producing task being connected to the asset that is being produced to.*

The graph view of the `my_producer_dag` shows the asset as well, if external conditions or all Dag dependencies are selected in the graph options Options.

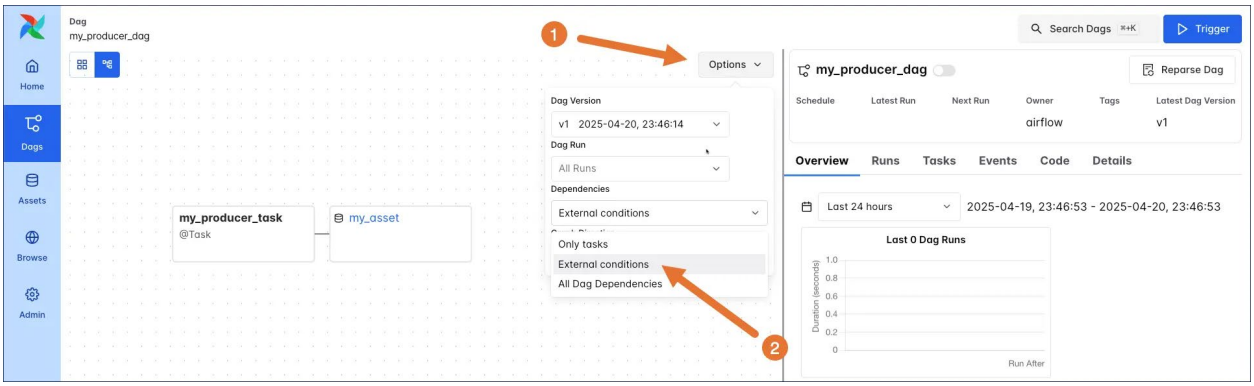

***Figure 2**: Screenshot of the Airflow UI showing a Dag graph with a producing task being connected to the asset that is being produced to.*

Next, schedule the `my_consumer_dag` to run as soon as a new asset event is produced to the `my_asset` asset.

```python
from airflow.sdk import Asset, dag
from airflow.providers.standard.operators.empty import EmptyOperator

@dag(
    schedule=[Asset("my_asset")],
)
def my_consumer_dag():

    EmptyOperator(task_id="empty_task")

my_consumer_dag()
```

You can see the relationship between the Dag containing the producing task (`my_producer_dag`), the consuming Dag `my_consumer_dag` and the asset in the asset graph located in the Assets tab of the Airflow UI.

***Figure 3**: Screenshot of the Airflow UI Assets view showing an asset dependency between two Dags.*

When external conditions or all Dag dependencies are selected, the `my_consumer_dag` graph shows the asset as well.

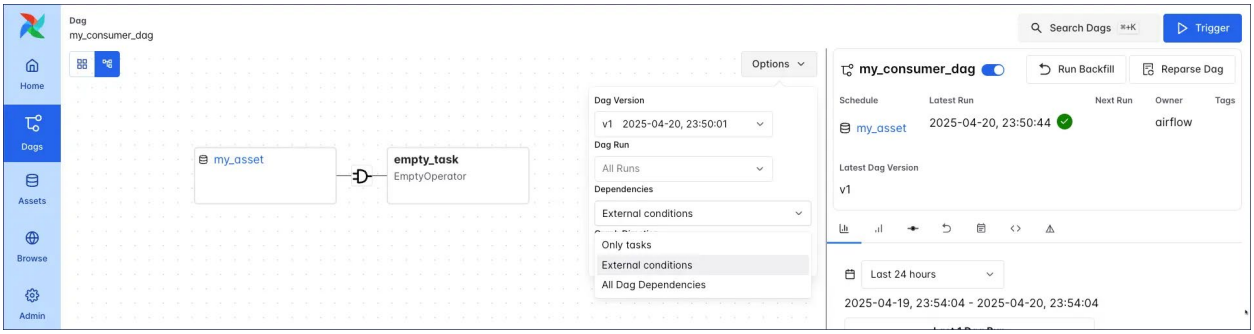

***Figure 4**: Screenshot of the Airflow UI showing the Dag graph of a Dag scheduled on one asset.*

After unpausing the `my_consumer_dag`, every successful completion of the `my_producer_task` task triggers a run of the `my_consumer_dag`.

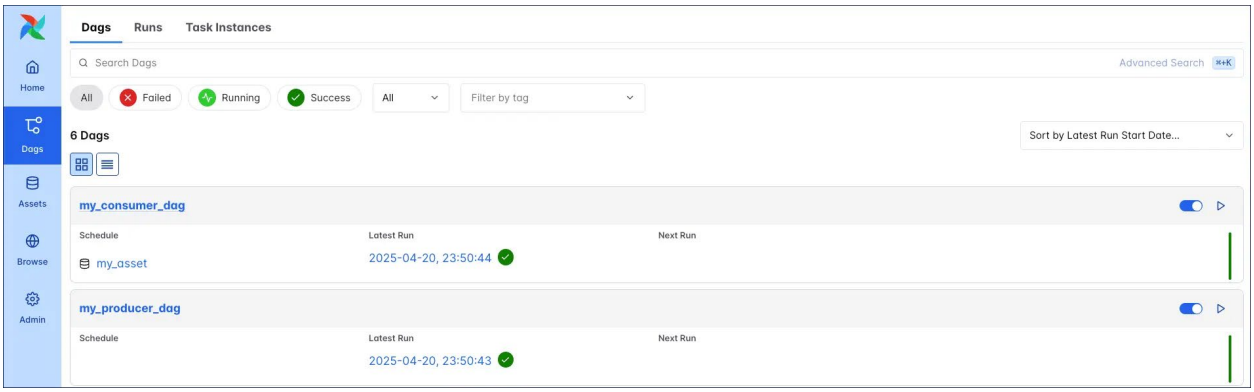

***Figure 5**: Screenshot of the Airflow UI showing the two Dags `my_consumer_dag` and `my_producer_dag` after one successful run each.*

USE ASSET ALIAS

You have the option to create asset aliases to schedule Dags based on assets with names generated at runtime. An asset alias is defined by a unique name string and can be used in place of a regular asset in outlets and schedules. Any number of asset events updating different assets can be attached to an asset alias.

There are two ways to add an asset event to an asset alias:

- Using the `Metadata` class.
- Using `outlet_events` pulled from the Airflow context.

See the code below for examples, note how the name of the asset is determined at runtime inside the producing task.

```
# from airflow.sdk import Asset, AssetAlias, Metadata, task


my_alias_name = “my_alias”


@task(outlets=[AssetAlias(my_alias_name)])
def attach_event_to_alias_metadata():
    bucket_name = “my-bucket”  # determined at runtime, for example based on upstream input
    yield Metadata(
        asset=Asset(f”updated_{bucket_name}”),
        extra={“k”: “v”},  # extra has to be provided, can be {}
        alias=AssetAlias(my_alias_name),
    )
```

```
attach_event_to_alias_metadata()


# from airflow.sdk import Asset, AssetAlias, Metadata, task


my_alias_name = “my_alias”


@task(outlets=[AssetAlias(my_alias_name)])
def attach_event_to_alias_context(outlet_events): # outlet_events is pulled out of the Context
    bucket_name = “my-other-bucket”   # determined at runtime, for example based on upstream
input
    outlet_events[AssetAlias(my_alias_name)].add(
        Asset(f”updated_{bucket_name}”), extra={“k”: “v”}
    )  # extra is optional


attach_event_to_alias_context()
```

In the consuming Dag you can use an asset alias in place of a regular asset.

```
from airflow.sdk import AssetAlias, dag
from airflow.providers.standard.operators.empty import EmptyOperator


my_alias_name = “my_alias”


@dag(schedule=[AssetAlias(my_alias_name)])
def my_consumer_dag():


    EmptyOperator(task_id=”empty_task”)


my_consumer_dag()
```

Once the `my_producer_dag` containing the `attach_event_to_alias_metadata` task completes successfully, reparsing of all Dags scheduled on the asset alias `my_alias` is automatically triggered. This reparsing step attaches the `updated_{bucket_name}` asset to the `my_alias` asset alias and the schedule resolves, triggering one run of the `my_consumer_dag`.

Any further asset event for the `updated_{bucket_name}` asset will now trigger the `my_consumer_dag`. If you attach asset events for several assets to the same asset alias, a Dag scheduled on that asset alias will run as soon as any of the assets that were ever attached to the asset alias receive an update.

See Dynamic data events emitting and asset creation through AssetAlias for more information and examples of using asset aliases.

To use Asset Aliases with traditional operators, you need to attach the asset event to the alias inside the operator logic. If you are using operators besides the `PythonOperator`, you can either do so in a custom operator's `.execute` method or by passing a `post_execute` callable to existing operators (experimental). Use `outlet_events` when attaching asset events to aliases in traditional or custom operators. Note that for deferrable operators, attaching an asset event to an alias is only supported in the `execute_complete` or `post_execute` method.

```
    def _attach_event_to_alias(context, result):  # result = the return value of the execute
method
        # use any logic to determine the URI
        uri = "s3://my-bucket/my_file.txt"
        context["outlet_events"][AssetAlias(my_alias_name)].add(Asset(uri))


    BashOperator(
        task_id="t2",
        bash_command="echo hi",
        outlets=[AssetAlias(my_alias_name)],
        post_execute=_attach_event_to_alias,  # using the post_execute parameter is experimental
    )
```

Update an asset

There are five ways to update an asset:

- A Dag defined using `@asset` completes successfully. Under the hood, `@asset` creates a Dag with one task which produces the asset.
- A task with an `outlet` parameter that references the asset completes successfully.
- A POST request to the assets endpoint of the Airflow REST API.
- An `AssetWatcher` that listens for a `TriggerEvent` caused by a message in a message queue. See event-driven scheduling for more information.

- A manual update in the Airflow UI by using the **Create Asset Event** button on the asset graph. There are two options when creating an asset event in the UI:
    - **Materialize**: This option runs the full Dag which contains the task that produces the asset event.
    - **Manual**: This option directly creates a new asset event without running any task that would normally produce the asset event. This option is useful for testing or when you want to create an asset event for an asset that is not updated from within a Dag in this Airflow instance.

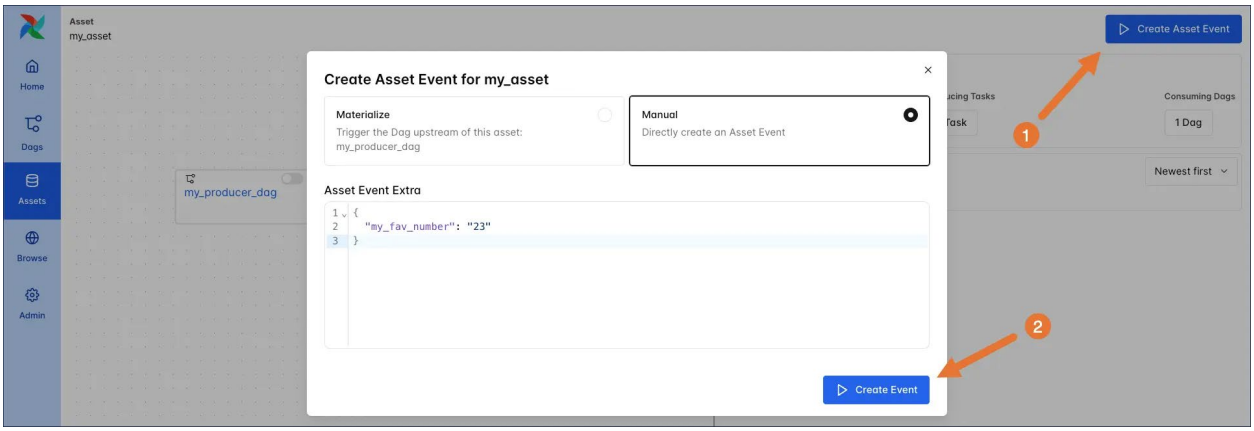

***Figure 6**: Screenshot of the Airflow UI showing the button to manually update an Asset.*

You can attach and retrieve information from assets' `extra` dictionary, see:

- Attaching information to a asset event and
- Retrieving asset information in a downstream task

for more information and example code.

Schedule a Dag on Assets

Any number of assets can be provided to the schedule parameter. There are 3 types of asset schedules:

- `schedule=[Asset("a"), Asset("b")]`: Providing one or more Assets as a list. The Dag is scheduled to run after all Assets in the list have received at least one update.
- `schedule=(Asset("a") | Asset("b"))`: Using AND (`&`) and OR (`|`) operators to create a conditional asset expression. Note that asset expressions are enclosed in smooth brackets `()`.
- `AssetOrTimeSchedule`: Combining time based scheduling with asset expressions, see combined asset and time-based scheduling.

When scheduling Dags based on assets, keep the following in mind:

- Consumer Dags that are scheduled on an asset are triggered every time a task that updates that asset completes successfully. For example, if `task1` and `task2` both produce `asset_a`, a consumer Dag of `asset_a` runs twice - first when `task1` completes, and again when `task2` completes.
- Consumer Dags scheduled on an asset are triggered as soon as the first task with that asset as an outlet finishes, even if there are downstream producer tasks that also operate on the asset.
- Consumer Dags scheduled on multiple assets run as soon as their expression is fulfilled by at least one asset event per asset in the expression. This means that it does not matter to the consuming Dag whether an asset received additional updates in the meantime, it consumes all queued events for one asset as one input. See Multiple Assets for more information.
- A consumer Dag that is paused will ignore all updates to assets that occurred while it was paused. Meaning, it starts with a blank slate upon being unpaused.
- Dags that are triggered by assets do not have the concept of a data interval. If you need information about the triggering event in your downstream Dag, you can use the parameter `triggering_asset_events` from the context. This parameter provides a list of all the triggering asset events with the parameters `[timestamp, source_dag_id, source_task_id, source_run_id, source_map_index]`. See Retrieving information in a downstream task for an example.

Conditional asset scheduling

You can use logical operators to combine any number of assets provided to the `schedule` parameter. The logical operators supported are `|` for OR and `&` for AND.

For example, to schedule a Dag on an update to either `asset1`, `asset2`, `asset3`, or `asset4`, you can use the following syntax. Note that the full statement is wrapped in `()`.

```
from airflow.sdk import Asset, dag


@dag(
    schedule=(
        Asset("asset1")
        | Asset("asset2")
        | Asset("asset3")
        | Asset("asset4")
    ),  # Use () instead of [] to be able to use conditional asset scheduling!
)
def downstream1_on_any():
```

```
    # your tasks here

downstream1_on_any()
```

Combined asset and time-based scheduling

You can combine asset-based scheduling with time-based scheduling with the `AssetOrTimeSchedule` timetable. A Dag scheduled with this timetable will run either when its `timetable` condition is met or when its `assets` condition is met.

The Dag shown below runs on a time-based schedule defined by the `0 0 * * *` cron expression, which is every day at midnight. The Dag also runs when either `asset3` or `asset4` is updated.

```
from airflow.sdk import Asset, dag, task
from pendulum import datetime
from airflow.timetables.assets import AssetOrTimeSchedule
from airflow.timetables.trigger import CronTriggerTimetable

@dag(
    start_date=datetime(2025, 8, 1),
    schedule=AssetOrTimeSchedule(
        timetable=CronTriggerTimetable("0 0 * * *", timezone="UTC"),
        assets=(Asset("asset3") | Asset("asset4")),
        # Use () instead of [] to be able to use conditional asset scheduling!
    )
)
def toy_downstream3_asset_and_time_schedule():

    # your tasks here

toy_downstream3_asset_and_time_schedule()
```

Event-driven scheduling

Event-driven scheduling is a sub-type of data-aware scheduling where a Dag is triggered when messages are posted to a message queue. This is useful for scenarios where you want to trigger a Dag based on events that occur outside of Airflow, such as data delivery to an external system or IoT sensor events, and is key to inference execution pipelines. At the time of writing, event-driven scheduling is available for Amazon SQS and Apache Kafka.

Concepts

There are a number of concepts that are important to understand when using event-driven scheduling.

- **Data-aware scheduling**: Data-aware or data-driven scheduling refers to all ways you can schedule a Dag based on updates to assets. Updates to assets outside of event-driven scheduling occur by tasks in the same Airflow instance completing successfully, manually through the Airflow UI, or through a call to the Airflow REST API.
- **Event-driven scheduling**: Event-driven scheduling is a sub-type of data-aware scheduling where a Dag is run based on messages posted to a message queue. This message in the queue is triggered by an event that occurs outside of Airflow.
- **Message queue**: A message queue is a service that allows you to send and receive messages between different systems. Examples of message queues include Amazon SQS, RabbitMQ, and Apache Kafka. Event-driven scheduling in Airflow 3.0 is supported for Amazon SQS and Apache Kafka with support for other message queues planned for future releases.
- **Trigger**: A trigger is an asynchronous Python function running in the Airflow triggerer component. Triggers that inherit from `BaseEventTrigger` can be used in AssetWatchers for event-driven scheduling. The trigger is responsible for polling the message queue for new messages, when a new message is found, a `TriggerEvent` is created. The message is deleted from the queue.
- **AssetWatcher**: An AssetWatcher is a class in Airflow that watches one or more triggers for events. When a trigger fires a `TriggerEvent`, the `AssetWatcher` updates the asset it is associated with, creating an `AssetEvent`. The payload of the trigger is attached to the `AssetEvent` in its `extra` dictionary.
- **AssetEvent**: An `Asset` is an object in Airflow that represents a concrete or abstract data entity. For example, an asset can be a file, table in a database, or not tied to any specific data. An `AssetEvent` represents one update to an asset. In the context of event-driven scheduling, the `AssetEvent` represents one message having been detected in the message queue.

When to use event-driven scheduling

Basic and advanced data-aware scheduling are great for use cases where updates to assets occur within Airflow or can be accomplished via a call to the Airflow REST API. However, there are scenarios where you need a Dag to run based on events in external systems. Two common patterns exist for event-driven scheduling:

1. **Data delivery to an external system**: Data is delivered to an external system, such as manually by a domain expert, and a data-ready event is sent to a message queue. The Dag in Airflow is scheduled based on this message event and runs an extract-transform-load (ETL) pipeline that processes the data in the external system.

2. **IoT sensor events**: An Internet of Things (IoT) device sends a sensor event to a message queue. A Dag in Airflow is scheduled based on this message event and consumes the message to evaluate the sensor value. If the evaluation determines that an alert is warranted, an alert event is published to another message queue.

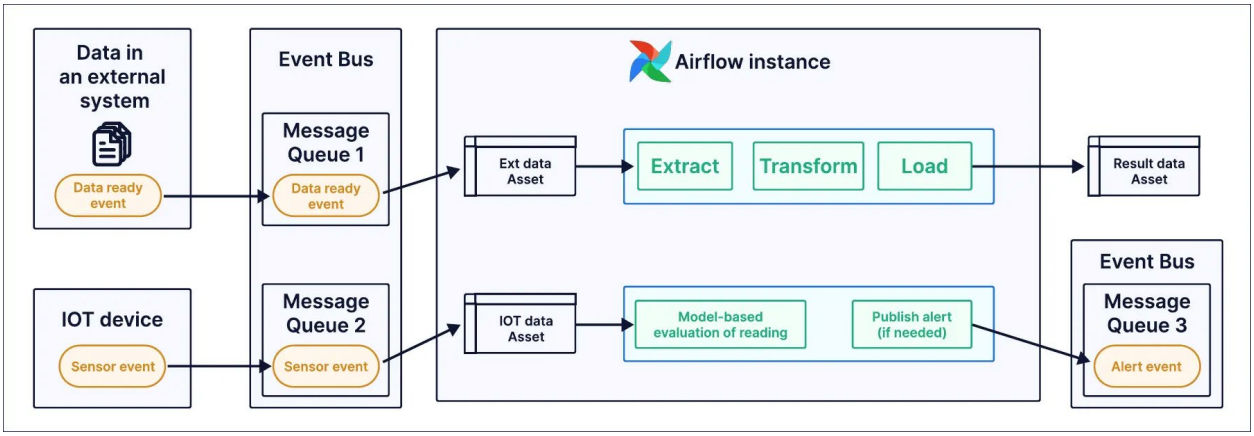

***Figure 7**: Architecture diagram showing the two common patterns for event-driven scheduling described above.*

One common use case for event-driven scheduling is inference execution, where the Dag that is triggered involves a call to a machine learning model. Airflow can be used to orchestrate inference execution pipelines of all types, including in Generative AI applications. A key change enabling inference execution in Airflow 3.0 is that a Dag can be triggered with `None` provided as the `logical_date`, meaning simultaneous triggering of multiple Dag runs is possible.

Currently, the `MessageQueueTrigger` works with Amazon SQS and Apache Kafka out-of-the-box with support for other message queues planned for future releases. You can create your own triggers to use with AssetWatchers by inheriting from the `BaseEventTrigger` class. See the Airflow documentation for more information on supported triggers for event-driven scheduling.

Example: Apache Kafka

To use Apache Kafka as the message queue for event-driven scheduling, follow these steps:

1. Create an Apache Kafka topic. See the Apache Kafka documentation for instructions.
2. Add the Airflow Common Messaging provider and Airflow Apache Kafka provider to your Airflow instance. When using the Astro CLI, you can add the providers to your `requirements.txt` file:

```
apache-airflow-providers-apache-kafka>=1.9.0

apache-airflow-providers-common-messaging>=1.0.2
```

3. Set the connection to your Apache Kafka topic in your Airflow instance. Below is an example connection json, depending on your Kafka setup you may need to provide additional values in the `extra` dictionary.

```
AIRFLOW_CONN_KAFKA_DEFAULT='{
    "conn_type": "general",
    "extra": {
        "bootstrap.servers": "<your host>:<your port>",
        "group.id": "<your group id>",
        "security.protocol": "<your security protocol>",
        "enable.auto.commit": false,
        "auto.offset.reset": "beginning"
    }
}'
```

4. Create a new file in the `dags` folder of your Airflow project and add the following code. Replace the `KAFKA_TOPIC` with the name of your Kafka topic.

```python
import json

from airflow.providers.common.messaging.triggers.msg_queue import MessageQueueTrigger
from airflow.providers.standard.operators.empty import EmptyOperator
from airflow.sdk import dag, Asset, AssetWatcher, task

# Define the Kafka queue URL
# Replace <your_kafka_host>, <port>, and <your_topic> with your Kafka
KAFKA_QUEUE = "kafka://<your_kafka_host>:<port>/<your_topic>"


# Define a function to apply when a message is received
# This function will be called with the message as an argument
def apply_function(*args, **kwargs):
    message = args[-1]
    val = json.loads(message.value())
    print(f"Value in message is {val}")
    return val


# Define a trigger that listens to an external message queue (Kafka in this case)
trigger = MessageQueueTrigger(
    queue=KAFKA_QUEUE,
    apply_function="dags.event_driven_dag.apply_function",
)

# Define an asset that watches for messages on the Kafka topic
kafka_topic_asset = Asset(
    "kafka_topic_asset", watchers=[AssetWatcher(name="kafka_watcher", trigger=trigger)]
)


@dag(schedule=[kafka_topic_asset])
def event_driven_dag():
    @task
    def process_message(**context):
        # Extract the triggering asset events from the context
```

```
        triggering_asset_events = context["triggering_asset_events"]
        for event in triggering_asset_events[kafka_topic_asset]:
            # Get the message from the TriggerEvent
            print(f"Processing message: {event}")

    process_message()


event_driven_dag()
```

This Dag is scheduled to run as soon as the `kafka_topic_asset` asset is updated. This asset uses one `AssetWatcher` with the name `kafka_watcher` that watches one `MessageQueueTrigger`. This trigger is polling for new messages in the provided Kafka topic.

The `process_message` task gets the triggering asset events from the Airflow context and prints the event information from the trigger event that includes the message. The `process_message` task is a placeholder for your own task that processes the message.

5. Create a new message in the Kafka topic. It will trigger the Dag to run and the `process_message` task will print the event information.

For an example using Amazon SQS see our documentation.

Timetables

Each time-based schedule in Airflow is implemented using a timetable under the hood. There are a couple of built-in timetables, including the `CronTriggerTimetable` and `CronDataIntervalTimetable`. If no timetbale exists for your use case, you have the option to create your own custom one.

Continuous timetable

You can run a Dag continuously with a pre-defined timetable. To use the ContinuousTimetable, set the schedule of your Dag to `"@continuous"` and set `max_active_runs` to 1.

```
@dag(
    start_date=datetime(2023, 4, 18),
    schedule="@continuous",
    max_active_runs=1,
)
```

This schedule will create one continuous Dag run, with a new run starting as soon as the previous run has completed, regardless of whether the previous run succeeded or failed. Using a `ContinuousTimetable` is especially useful when sensors or deferrable operators are used to wait for highly irregular events in external data tools.

Warning: Airflow is designed to handle orchestration of data pipelines in batches, and this feature is not intended for streaming or low-latency processes. If you need to run pipelines more frequently than every minute, consider using Airflow in combination with tools designed specifically for that purpose like Apache Kafka for example with an event-driven schedule.

Want to know more?

- Airflow: Dag Scheduling Astronomer Academy Module
- Airflow: Assets Astronomer Academy Module
- Scheduling in Airflow: A Comprehensive Introduction
- Schedule Dags in Airflow guide
- Assets and data-aware scheduling in Airflow guide
- Event-driven scheduling guide

Notifications and Alerts

6

KEY CONCEPTS

- Callback: Dags and tasks in Airflow allow you to provide functions to different callback parameters that are executed when the Dag or task enters a specific state. This is the most common way to create notifications and alerts in open source Airflow.
- Astro alerts: Advanced alerts including for Task Duration limits and Timeliness that Astro customers can define in the Astro UI across Dags and Deployments.
- **SLA**: Service-level agreement. In data engineering, an SLA generally refers to a time by which a specific task or Dag has to be completed. Note that the Airflow 2 SLA feature has been removed in Airflow 3. We recommend using Astro alerts or Astro Observe instead. In Airflow 3.1 the experimental Deadline Alerts were added for OSS users.
- Astro Observe: Astronomer product that allows users to define and track data products across Dags in different deployments. Business-level SLAs can be set on data products to alert users when the timeliness or freshness of your data products are at risk.

Overview

Using the guidance in this book will help you write robust Dags, but sometimes Dags fail due to circumstances outside the data engineer's control. In these events it is crucial that you have notifications and alerts in place to be apprised of Dag and task-level events you really care about.

Open-source Airflow offers the option to define Airflow callbacks which are functions that execute when a Dag or task enters a specific state, for example when it fails (`on_failure_callback`) or completes successfully (`on_success_callback`).

Astro customers have the option to define complex alerting rules on all their Dags across deployments using Astro alerts and the Astro Observe feature.

Airflow Callbacks

In Airflow you can define actions to be taken due to different Dag or task states by providing functions to `*_callback` parameters:

- `on_success_callback`: Invoked when a task or Dag succeeds.
- `on_failure_callback`: Invoked when a task or Dag fails.
- `on_skipped_callback` : Invoked when a task is skipped. This callback only exists at the task level, and is only invoked when an `AiflowSkipException` is raised, not when a task is skipped due to other reasons, like a trigger rule. See Callback Types.
- `on_execute_callback`: Invoked right before a task begins executing. This callback only exists at the task level.
- `on_retry_callback`: Invoked when a task is retried. This callback only exists at the task level.

The Dag code below shows a simple function, `my_callback_function` being executed for different callback parameters at the Dag and task level. Note that you can set callbacks for all tasks in a Dag by using the Dag-level parameter `default_args`.

```
from airflow.sdk import dag, task
from pendulum import datetime, duration
from airflow.exceptions import AirflowSkipException



def my_callback_function(context):
    t_id = context["ti"].task_id
```

```
    t_state = context[“ti”].state
    print(f”Hi from my_callback_function! {t_id} finished as: {t_state}”)
    # add logic to send notifications such as Slack or email here


@dag(
    # Dag level callbacks depend on events happening to the Dag itself
    on_success_callback=[my_callback_function],
    on_failure_callback=[my_callback_function],
    # callbacks provided in the default_args are giving to all tasks in the Dag
    default_args={
        “on_execute_callback”: [my_callback_function],
        “on_retry_callback”: [my_callback_function],
        “on_success_callback”: [my_callback_function],
        “on_failure_callback”: [my_callback_function],
        “on_skipped_callback”: [my_callback_function],
        “retries”: 2,
        “retry_delay”: duration(seconds=5),
    },
)
def callbacks_overview():
    @task(
        # you can override default_args on the task level
        on_execute_callback=[my_callback_function],
        on_retry_callback=[my_callback_function],
        on_success_callback=[my_callback_function],
        on_failure_callback=[my_callback_function],
        on_skipped_callback=[my_callback_function],
    )
    def task_success():
        return 10

    @task
    def task_failing():
        return 10 / 0

    @task
    def task_skip():
        raise AirflowSkipException(“Skipping task”)
```

```
    task_success()
    task_failing()
    task_skip()


callbacks_overview()
```

You can provide any Python callable to the `*_callback` parameters or use Airflow notifiers. Notifiers are pre-built classes to send alerts to various tools, for example to Slack using the SlackNotifier.

To execute multiple functions when a Dag or task reaches a certain state, you can provide several callback items to the same callback parameter in a list.

The OSS notification library Apprise contains modules to send notifications to many services. You can use Apprise with Airflow by installing the Apprise Airflow provider which contains the AppriseNotifier. See the Apprise Airflow provider documentation and the Apprise example Dag for more information and examples.

Want to know more?

- Introduction to Observability for Data Pipelines webinar
- How to monitor your pipelines with Airflow and Astro alerts webinar
- Create and use data products with Astro Observe
- Manage Airflow Dag notifications guide

Write dynamic and adaptable Dags

7

KEY CONCEPTS

- **Dynamic Task Mapping**: An Airflow feature allowing you to change the number of copies of a task or task group at runtime based on data provided by upstream tasks in your Dag without any changes to the Dag code. Dynamic tasks are highly scalable and observable. Consider using dynamic tasks whenever you need “one task (group) per X”.
- Dynamic Dags: refers to the possibility to programmatically create Dag code using custom Python scripts or tools like dag-factory. The number of Dags change following code updates in the Dags folder. Consider programmatically creating Dags when you need “one Dag per X”. Dynamic Dags are covered in chapter 8 of this eBook.

Note that you can use dynamic tasks within dynamic Dags.

Dynamic Task concepts

The Airflow dynamic task mapping feature is based on the MapReduce programming model. Dynamic task mapping creates a single task for each input. The reduce procedure, which is optional, allows a task to operate on the collected output of a mapped task. In practice, this means that your Dag can create an arbitrary number of parallel tasks at runtime based on some input parameter (the map), and then if needed, have a single task downstream of your parallel mapped tasks that depends on their output (the reduce).

Airflow tasks have two methods available to implement the map portion of dynamic task mapping. For the task you want to map, you must pass all operator parameters through one of the following functions.

- `.expand()`: This method passes the parameters that you want to map. A separate parallel task is created for each input. For some instances of mapping over multiple parameters, `.expand_kwargs()` is used instead.
- `.partial()`: This method passes any parameters that remain constant across all mapped tasks which are generated by `.expand()`.

In the following example, the `add` task uses both, `.partial()` and `.expand()`, to dynamically generate three task runs.

Example using the `@task` decorator:

```
# from airflow.sdk import task


@task(
    # optionally, you can set a custom index to display in the UI
    map_index_template="{{ my_custom_map_index }}"
)
def add(x: int, y: int):

    # get the current context and define the custom map index variable
    from airflow.sdk import get_current_context

    context = get_current_context()
    context["my_custom_map_index"] = "Input x=" + str(x)

    return x + y

added_values = add.partial(y=10).expand(x=[1, 2, 3])
```

Example using a traditional operator (`PythonOperator`):

```
# from airflow.providers.standard.operators.python import PythonOperator

def add_function(x: int, y: int):
    return x + y

added_values = PythonOperator.partial(
    task_id="add",
    python_callable=add_function,
    op_kwargs={"y": 10},
    # optionally, you can set a custom index to display in the UI (Airflow 2.9+)
    map_index_template="Input x={{ task.op_args[0] }}",
).expand(op_args=[[1], [2], [3]])
```

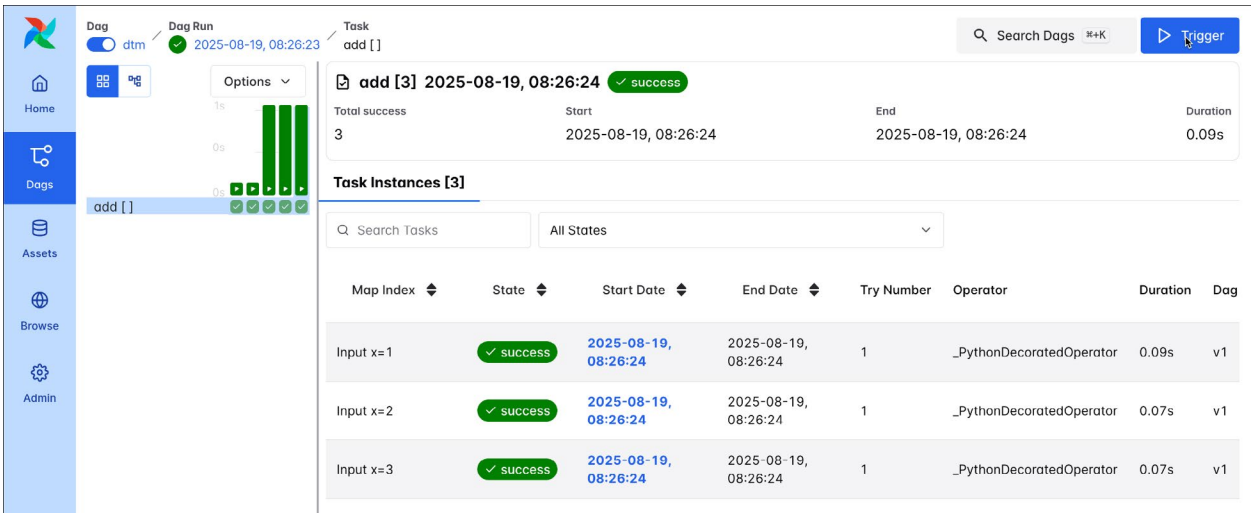

***Figure 1**: Screenshot of the **Task Instances []** tab for the dynamic task `add` showing 3 dynamically mapped task instances generated by the code snippets above.*

Defining the `map_index_template` parameter is optional. If you don't set it, the default map index is used, which is an integer index starting from 0. It is a best practice to set a custom index to make it easier to identify the mapped task instances in the Airflow UI. For example, if you are mapping over a list of files, you can display the name of the file as the **Map Index** in the Airflow UI.

The `.expand` method creates three mapped `add` tasks, one for each entry in the `x` input list. The `. .partial` method specifies a value for `y` that remains constant in each task.

When you work with mapped tasks, keep the following in mind:

- You can use the results of an upstream task as the input to a mapped task. The upstream task must push a value in a dict or list form to XCom using the `return_value` key, which is the default key for values returned from a `@task` decorated task.
- You can map over multiple parameters.
- You can use the results of a mapped task as input to a downstream mapped task.
- You can have a mapped task that results in no task instances. For example, when your upstream task that generates the mapping values returns an empty list. In this case, the mapped task is marked skipped, and downstream tasks are run according to the trigger rules you set. By default, downstream tasks are also skipped.
- Some parameters can't be mapped. For example, `task_id`, `pool`, and many other BaseOperator arguments.
- `.expand()` only accepts keyword arguments, i.e. you need to specify the parameter you are mapping over.
- The maximum amount of mapped task instances is determined by the `max_map_length` parameter in the Airflow configuration. By default it is set to `1024`.
- You can limit the number of mapped task instances for a particular task that run in parallel by setting the following parameters in your dynamically mapped task:
    - Set a limit for parallel runs of a task across **all** Dag runs with the `max_active_tis_per_dag` parameter.
    - Set a limit for parallel runs of a task within a **single** Dag run with the `max_active_tis_per_dagrun` parameter.
- XComs created by mapped task instances are stored in a list and can be accessed by using the map index of a specific mapped task instance. For example, to access the XComs created by the third mapped task instance (map index of 2) of `my_mapped_task`, use `ti.xcom_pull(task_ids=['my_mapped_task'])[2]`. The map_indexes parameter in the `.xcom_pull()` method allows you to specify a list of map indexes of interest (`ti.xcom_pull(task_ids=['my_mapped_task'], map_indexes=[2])`).

For additional examples of how to apply dynamic task mapping functions, see Dynamic Task Mapping in the official Airflow documentation.

In the Airflow UI, dynamic tasks are identified with a set of brackets [ ] following the task ID. All mapped task instances are combined into one row on the grid view and one node in the graph view.

The number in the brackets shown in the Dag run graph is updated for each Dag run to reflect how many mapped instances were created. The following screenshot shows a Dag run graph with two tasks, the latter having 49 dynamically mapped task instances.

***Figure 2**: Screenshot of the graph of a Dag run with two tasks, the upstream `get_nums` task as well as the dynamic task `add` with `[49]` dynamically mapped task instances for this Dag run.*

To see the logs for, and XCom pushed by each dynamically mapped task instance, click on the grid square of a dynamically mapped task and then on an individual task instance.

Mapping over sets of keyword arguments

To map over sets of inputs to two or more keyword arguments (kwargs), you can use the `.expand_kwargs()` method. You can provide sets of parameters as a list containing a dictionary or as an `XComArg`. The operator gets 3 sets of commands, resulting in 3 mapped task instances.

```
# from airflow.providers.standard.operators.bash import BashOperator
# input sets of kwargs provided directly as a list[dict]
t1 = BashOperator.partial(task_id="t1").expand_kwargs(
    [
        {"bash_command": "echo $WORD", "env" : {"WORD": "hello"}},
        {"bash_command": "echo `expr length $WORD`", "env" : {"WORD": "tea"}},
        {"bash_command": "echo ${WORD//e/X}", "env" : {"WORD": "goodbye"}}
    ]
)
```

The task `t1` will have three mapped task instances printing their results into the logs:

- Map index 0: `hello`
- Map index 1: `3`
- Map index 2: `goodbyX`

Dynamically map task groups

You can use dynamic task mapping with the `@task_group` decorator to dynamically map over task groups. The following Dag shows how you can dynamically map over a task group with different inputs for a given parameter.

```
# from airflow.sdk import task_group


# creating a task group using the decorator with the dynamic input my_num
@task_group(group_id="group1")
def tg1(my_num):
    @task
    def print_num(num):
        return num


    @task
    def add_42(num):
        return num + 42


    print_num(my_num) >> add_42(my_num)


# creating 6 mapped task group instances of the task group group1
tg1_object = tg1.expand(my_num=[19, 23, 42, 8, 7, 108])
```

This Dag dynamically maps over the task group `group1` with different inputs for the `my_num` parameter. 6 mapped task group instances are created, one for each input. Within each mapped task group instance two tasks will run using that instances' value for `my_num` as an input.

Want to know more?

- Dynamic tasks in Airflow webinar
- Dynamic Task Mapping on Multiple Parameters webinar
- Airflow: Dynamic Task Mapping Academy Module
- Create dynamic Airflow tasks guide

Programmatically generate Dags

8

KEY CONCEPTS

- **Dynamic Dags:** refers to Dag code that is created programmatically using custom Python scripts or tools like dag-factory. The number of Dags change based on code changes in the Dags folder. Consider programmatically creating Dags when you need “one Dag per X”. If you find yourself writing the same code over and over, it might be time to create your Dag code programmatically instead.
- dag-facory: is a Python package that allows you to programmatically create Airflow Dags based on YAML configuration files.

The dynamic task mapping Airflow feature allows you to change the number of copies of a task or task group at runtime based on data provided by upstream tasks in your Dag without any changes to the Dag code. Dynamic tasks are highly scalable and observable. Consider using dynamic tasks whenever you need “one task (group) per X”.

dag-factory

A popular way to generate Dag code is using dag-factory. This package allows you to simplify code and reduce redundancy by allowing Dag definitions in YAML format, as shown in the example below.

```
my_dag:
  schedule: '0 0 * * *'
  default_args:
    owner: 'Astro'
    start_date: '2024-10-01'
  tasks:
    extract:
      operator: airflow.providers.standard.operators.python.PythonOperator
      python_callable_name: extract
      python_callable_file: /usr/local/airflow/include/python_func.py
    transform:
      operator: airflow.providers.standard.operators.python.PythonOperator
      python_callable_name: transform
      python_callable_file: /usr/local/airflow/include/python_func.py
      dependencies: [extract]
    load:
      operator: airflow.providers.standard.operators.python.PythonOperator
      python_callable_name: load
      python_callable_file: /usr/local/airflow/include/python_func.py
      dependencies: [transform]
```

When dag-factory is installed in an Airflow environment, you can generate Dags from yaml files like this with the following Dag generation methods:

```
import os
from pathlib import Path
from dagfactory import load_yaml_dags

DEFAULT_CONFIG_ROOT_DIR = "/usr/local/airflow/dags/"
CONFIG_ROOT_DIR = Path(os.getenv("CONFIG_ROOT_DIR", DEFAULT_CONFIG_ROOT_DIR))
config_file = str(CONFIG_ROOT_DIR / "basic_example.yml")

load_yaml_dags(
    globals_dict=globals(),
    config_filepath=config_file,
)
```

See this tutorial for in-depth instructions on how to use dag-factory.

There are other methods to programmatically create Dag code that allow for more customization than dag-factory, see this guide for more information.

Want to know more?

- Airflow: Dynamic Dags Academy Module
- Dynamically generate Dags in Airflow guide

9

Dag versioning and Dag bundles

KEY CONCEPTS

- **Dag versioning**: Airflow keeps track of changes to your Dags. This is automatic and happens no matter which Dag bundle is used.
- **Dag bundle**: A collection of files containing Dag code and supporting files.
- **Dag bundle backend**: The location a Dag bundle is stored. Dag bundles are named after the backend they use to store the Dag code. For example, the `LocalDagBundle` uses the local file system to store Dag code, while the `GitDagBundle` uses a Git repository. If a Dag bundle backend supports *code versioning* the Dag bundle stored in it is called a versioned Dag bundle.

Importance of Dag Versioning

In Airflow 2, both the Airflow UI and Dag execution always used the latest Dag code. This led to two major constraints:

- **No observability of previous Dag versions**: If you changed a Dag and then, for example, removed a task, all history for previous runs of that task disappeared in the grid and graph view of the Airflow UI.
- **Version collisions during code pushes**: If the code of a Dag changed while a Dag was still running, some tasks of the same run might have been executed using the older version while others used the newer version. This situation carried a significant risk of unintended consequences.

  For example: The older Dag version might use task A to retrieve the name of table X in a relational database for data insertion and task B to insert data into that table. If the Dag was updated to change the table from X to Y in the middle of a run, task A (from the old version) might pass the table name X while the updated task B inserted data intended for table Y into table X.

Dag bundles and Dag versioning were introduced in Airflow 3 to address these issues.

Dag Versioning vs Dag Bundles

Airflow 3 introduces two new concepts:

- **Dag versioning**: Airflow now keeps track of changes to your Dags. This is automatic and happens no matter which Dag bundle is used.
  - *A new Dag version is created every time a Dag run is created for a Dag that has undergone a structural change since the last run.* A structural change is any change that affects `serdag`, this includes changes to Dag or task parameters, task dependencies, task IDs or adding or removing tasks.
  - Each Dag run is associated with a Dag version that is visible in the Airflow UI.
  - Whenever a new Dag run is initiated, the scheduler uses the latest version of the Dag to create a run.
- **Dag bundle**: A collection of files containing Dag code and supporting files. Dag bundles are named after the backend they use to store the Dag code. For example, the `LocalDagBundle` uses the local file system to store Dag code, while the `GitDagBundle` uses a Git repository.
  - Some Dag bundles are versioned, such as the `GitDagBundle`. A version of a Dag bundle

is created by versioning the underlying backend. For example, a new version of the `GitDagBundle` is created by every Git commit, whether or not any Dags change.

- The default `LocalDagBundle` is not versioned.

Dag versioning is automatic in Airflow 3 and does not require any setup. Using a Dag bundle other than `LocalDagBundle` requires changes to your Airflow configuration.

Dag versioning

You can view Dag versions in several places in the Airflow UI. In the Options menu of the Dag graph, you can select which version of the Dag graph you want to display. The Dag details page also shows the latest available version of the Dag, which is used to create new Dag runs.

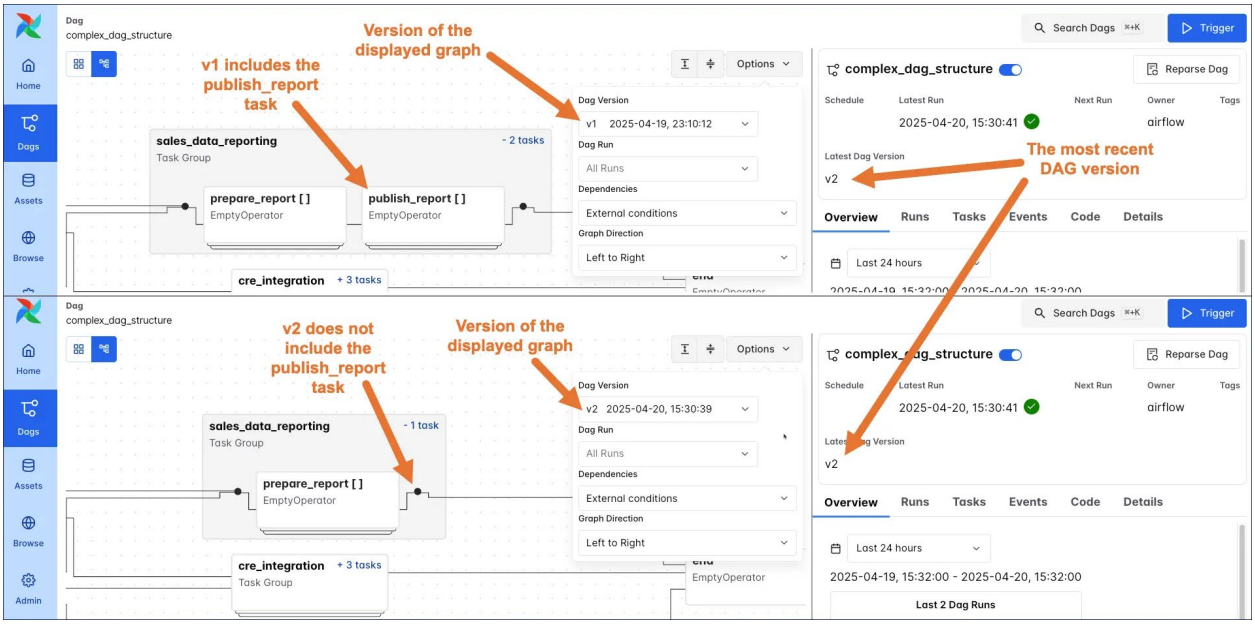

***Figure 1**: Screenshot a Dag called `complex_dag_structure` showing two different versions of the same Dag. Version v1 on top includes the `publish_report` task. Version v2 on the bottom does not include that task.*

The Dag grid now retains the history for all tasks, even if they were removed in the latest version of the Dag. You can also select which version of the Dag code you want to display in the code tab.

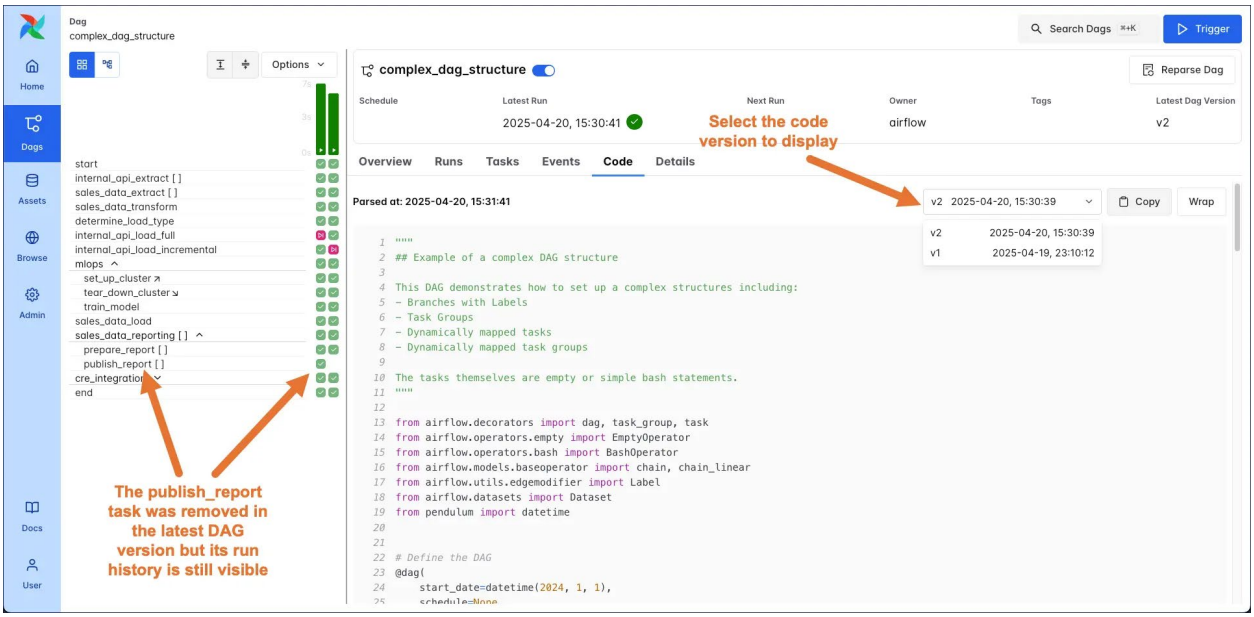

***Figure 2**: Screenshot a Dag called `complex_dag_structure` showing two different versions of the same Dag in the grid view. The `publish_report` task that was removed in v2 of the Dag still shows up in the task list with its historical runs.*

Dag bundles

For Dags running on Astro using Hosted execution mode a specialized versioned Dag bundle is configured automatically, without any need for additional setup. Configuring custom Dag bundles and using several Dag bundles in the same Astro Deployment is only supported when using Remote Execution mode. For more information, see Configure Dag bundles for Remote Execution.

Dag bundles contain Dag code and supporting files. There are versioned and unversioned Dag bundles; the default Dag bundle (`LocalDagBundle`) is not versioned, while the `GitDagBundle` is versioned. Support for other Dag bundle backends is planned for future releases.

Versioned and unversioned Dag bundles behave differently in the following situations:

- **Clearing and rerunning a previous Dag run**:
    - Unversioned Dag bundle: Airflow uses the current Dag code, i.e., the latest version of the Dag.
    - Versioned Dag bundle: The scheduler uses the Dag version that existed at the time of the Dag run to determine which task instances to create. The workers use the code contained in the Dag bundle version that existed at the time of the original Dag run to execute their tasks.

- **Rerunning individual tasks of a previous Dag run**:
    - Unversioned Dag bundle: Airflow uses the latest version of the Dag for tasks that are rerun.
    - Versioned Dag bundle: Airflow uses the code of the task contained in the Dag bundle version at the time of the original Dag run.
- **Changing code while a Dag is running**:
    - Unversioned Dag bundle: The Dag always uses the current Dag code at the time it starts a task, as in Airflow 2.
    - Versioned Dag bundle: The Dag run finishes using the bundle version it started with.
- **Making code changes**:
    - Unversioned Dag bundle: Every structural change to the Dag creates a new Dag version.
    - Versioned Dag bundle: Every committed or saved structural change to a Dag creates a new version of that Dag. This means with every new bundle version, all Dags that have had structural changes will also have a new Dag version.

See the Airflow documentation on Dag bundles for more information, including how to create a custom Dag bundle.

Set up a GitDagBundle

To directly fetch your Dag code from a GitHub repository, you can use the `GitDagBundle`. This bundle is versioned. To configure a `GitDagBundle` for an Astro CLI project, follow these steps:

1. Push your Dag code to a GitHub repository.
2. Install the `git` package in your Astro project by adding it to your `packages.txt` file.
3. Install the Airflow Git provider by adding the following to your `requirements.txt` file. Replace `<version>` with the latest version of the provider package.

```
apache-airflow-providers-git==<version>
```

4. Define a Git connection using an environment variable in your `.env` file. Replace `<account>` and `<repo>` with the name of your GitHub account and repository, respectively. Replace `github_pat_<your-token>` with your GitHub personal access token. Note that the token needs to have read and write access to the code of the repository.

```
AIRFLOW_CONN_MY_GIT_CONN='{
    "conn_type": "git",
    "host": "https://github.com/<account>/<repo>.git",
    "password": "github_pat_<your-token>"
}'
```

5. Change the `[dag_processor].dag_bundle_config_list` configuration to use a `GitDagBundle` by setting the associated environment variable in your `.env` file. Replace `your-bundle-name` with the name you want to give to your Dag bundle. The `subdir` should point to the directory in your GitHub repository where your Dag code is stored. The `tracking_ref` should point to the branch you want to use.

```
AIRFLOW__Dag_PROCESSOR__Dag_BUNDLE_CONFIG_LIST='[
    {
        "name": "your-bundle-name",
        "classpath": "airflow.providers.git.bundles.git.GitDagBundle",
        "kwargs": {
            "git_conn_id": "my_git_conn",
            "subdir": "dags",
            "tracking_ref": "main"
        }
    }
]'
```

6. Restart your project using `astro dev restart` to apply the changes.

Programmatic Dags and Dag bundles

If you are creating your Dags programmatically, i.e. you are using Python code to generate your Dag code and want to use a versioned Dag bundle, you need to ensure that there are no Dag structure changes without a Dag bundle change.

The reason is, that when clearing a Dag run, the scheduler uses the Dag bundle version that existed at the time of the Dag run to determine which task instances to create. The workers use the code contained in the Dag bundle version that existed at the time of the original Dag run to execute their tasks. In the rare case where programmatic Dag creation leads to a Dag structure, and therefore Dag version change without a Dag bundle change, the scheduler and workers will use different Dag versions to create and execute the tasks. This can lead to unexpected behavior.

An example for programmatic Dag creation that is safe to use with a versioned Dag bundle is usage of the dag-factory or to create tasks in a loop that only changes when the code changes:

```
# this list only changes when the code changes

my_tables = [“TABLE_A”, “TABLE_B”, “TABLE_C”]


for my_table in my_tables:

    @task(

        task_id=f”modify_{i}”,

    )

    def modify_table(my_table):

        # do something with the table

        pass


    modify_table(my_table=my_table)
```

If you are using top-level code that connects to an external system (a practice that we caution against, see Avoid top-level code), you might have a change in Dag structure without a change in the Dag bundle. An example would be if the list `my_tables` from the above example is created by querying a database.

Testing Airflow Dags

10

KEY CONCEPTS

- **Testing:** Testing in Airflow refers to testing the Dag code itself. It is a best practice to test Airflow Dags like any software code, using unit tests, integration tests, and Dag validation tests.
- **CICD**: Continuous integration and continuous delivery. CICD is a software development best practice that automates testing and deploying code in a version control system to protect production environments. Astro customers can use the GitHub integration feature to quickly implement CICD for their Astro deployments.
- **Data quality checks**: refers to the practice of testing the data flowing through an Airflow pipeline. These checks are separate from Dag code testing and can be implemented as part of a data pipeline, see Data quality and Airflow.

Types of Tests

In Airflow you have different layers of code for which you can write tests:

- **Unit Tests**: All custom Python code should be unit tested as you would in any software application code.
- **Dag validation tests**: You can define Airflow-specific tests to ensure your Dags meet organizational standards, such as requiring tags on Dags or restricting the use of certain operators.
- **Integration Tests**: Since Airflow interacts with many tools it is wise to have end-to-end integration tests, these typically run as pipelines in a staging environment.

Note that all these tests are about the Dag *code* itself, the data flowing through your pipeline is tested by specialized tasks in your pipeline using Data Quality Checks.

If you use the Astro CLI to run your Dags, you can add all of your tests to the tests folder and run them with the command astro dev pytest (no matter which Python framework you are using for your tests).

Unit Tests

In Airflow you can create unit tests the same way as you would for any Python code with any Python testing framework.

You should test all your custom operators, as well as all Python functions used in your tasks. If you use operators from provider packages you don't need to write unit tests for them, since these operators are already tested by the community.

This is an example of a unit test for a custom operator doing basic math created with the unittest framework. The test makes sure that if given the inputs 2 and 3 as well as the operation +, the operator will return 5. You can find the full test suite for this basic custom operator in the corresponding testing file here.

```
import unittest
from include.custom_operators import MyBasicMathOperator



class TestMyBasicMathOperator(unittest.TestCase):


    def test_addition(self):
        operator = MyBasicMathOperator(
            task_id="basic_math_op", first_number=2, second_number=3, operation="+"
        )
```

```
        result = operator.execute(None)
        self.assertEqual(result, 5)
```

Dag validation tests

Dag validation tests examine your Dags according to your defined standards. You import the Dags using the Airflow DagBag class and create one test for each property you want to enforce, using any Python testing framework.

The following is an example of using pytest to only allow only one decorator (@task) and one operator (SQLExecuteQueryOperator).

```
import logging
import os
from contextlib import contextmanager

import pytest
from airflow.models import DagBag



@contextmanager
def suppress_logging(namespace):
    logger = logging.getLogger(namespace)
    old_value = logger.disabled
    logger.disabled = True
    try:
        yield
    finally:
        logger.disabled = old_value



def get_dags():
    """
    Generate a tuple of dag_id, <Dag objects> in the DagBag
    """
    with suppress_logging("airflow"):
        dag_bag = DagBag(include_examples=False)
```

```
    def strip_path_prefix(path):
        return os.path.relpath(path, os.environ.get("AIRFLOW_HOME"))

    return [(k, v, strip_path_prefix(v.fileloc)) for k, v in dag_bag.dags.items()]


ALLOWED_OPERATORS = [
    "_PythonDecoratedOperator",  # this allows the @task decorator
    "SQLExecuteQueryOperator",
]


@pytest.mark.parametrize(
    "dag_id, dag, fileloc", get_dags(), ids=[x[0] for x in get_dags()]
)
def test_dag_uses_allowed_operators_only(dag_id, dag, fileloc):
    """
    Test if a Dag uses only allowed operators.
    """
    for task in dag.tasks:
        assert any(
            task.task_type == allowed_op for allowed_op in ALLOWED_OPERATORS
        ), f"{task.task_id} in {dag_id} ({fileloc}) uses {task.task_type}, which is not in the list of allowed operators."
```

Want to know more?

- Airflow best practices: debugging and testing webinar
- Astro: CI/CD Academy Module
- How to easily test your Airflow Dags with the new dag.test() function webinar
- Use frameworks to test Apache Airflow data pipelines | Conf42 Python 2024
- Test Airflow Dags guide

Scaling Airflow Dags

11

KEY CONCEPTS

As soon as you start moving from local development to production, you need to make sure your environment scales with your pipelines.

On the Dag authoring side this means you need to be aware of scaling configurations and parameters. There are ways to adjust how Airflow scales at three different levels:

- Environment-level settings: These configuration parameters apply to your whole Airflow instance. A common value you might need to adjust here is the AIRFLOW__CORE__DAGBAG_IMPORT_TIMEOUT. By default Dag processing will time out after 30 seconds, causing dags to not appear in the Airflow UI in larger environments. Increasing the value of this environment variable can prevent the Dag import process from timing out. You can find all available Airflow configurations in the Configuration Reference.
- Dag-level settings: Some Dag parameters allow you to control how much a specific Dag can scale. For example if you have a long running Dag that should only ever have one concurrent active Dag run, you can set max_active_runs=1. See Dag-level parameters for a comprehensive list of Dag-level parameters.
- Task-level settings: For individual tasks it is possible to control their execution at the task-level using task parameters. A common pattern is to assign a set of tasks to an Airflow pool, which limits how many tasks in this set can run concurrently.

Environment-level settings

Environment-level settings are those that impact your entire Airflow environment (all Dags). They all have default values that can be overridden by setting the appropriate environment variable or modifying your `airflow.cfg` file. Generally, all default values can be found in the Airflow Configuration Reference. To check current values for an existing Airflow environment, go to **Admin > Config** in the Airflow UI. For more information, see Setting Configuration Options in the Apache Airflow documentation.

When using the Astro CLI locally you can define environment variables in the `.env` file of your project.If you're running Airflow on Astro, you can modify most of these parameters with environment variables. For more information on how to set those environment variables for your Astro Deployment, see Environment Variables in the Astronomer documentation.

You should modify environment-level settings if you want to tune performance across all of the Dags in your Airflow environment.

Core Settings

Core settings control the number of processes running concurrently and how long processes run across an entire Airflow environment. The associated environment variables for all parameters in this section are formatted as `AIRFLOW__CORE__PARAMETER_NAME`.

- `parallelism`: The maximum number of tasks that can run concurrently on each scheduler within a single Airflow environment. For example, if this setting is set to 32, and there are two schedulers, then no more than 64 tasks can be in a running or queued state at once across all Dags. If your tasks remain in a scheduled state for an extended period, you might want to increase this value. The default value is 32.
  On Astro, this value is set automatically based on your maximum worker count, meaning that you don't have to configure it.
- `max_active_tasks_per_dag`: The maximum number of tasks that can be scheduled at the same time across all runs of a Dag. Use this setting to prevent any one Dag from taking up too many of the available slots from parallelism or your pools. The default value is 16.
  If you increase the amount of resources available to Airflow (such as Celery workers or Kubernetes resources) and notice that tasks are still not running as expected, you might have to increase the values of both parallelism and `max_active_tasks_per_dag`.
- `max_active_runs_per_dag`: Determines the maximum number of active Dag runs (per Dag) that the Airflow scheduler can create at a time. In Airflow, a Dag run represents an instantiation of a Dag in time, much like a task instance represents an instantiation of a task. This parameter is most relevant if Airflow needs to backfill missed Dag runs. Consider how you want to handle these scenarios when setting this parameter. The default value is 16.

- `dagbag_import_timeout`: How long the `dagbag` can import Dag objects before timing out in seconds, which must be lower than the value set for `dag_file_processor_timeout`. If your Dag processing logs show timeouts, or if your Dag is not showing in the Dags list or the import errors, try increasing this value. You can also try increasing this value if your tasks aren't executing, since workers need to fill up the `dagbag` when tasks execute. The default value is 30 seconds.

Scheduler settings

Scheduler config settings control how the scheduler parses Dag files and creates Dag runs. The associated environment variables for all parameters in this section are formatted as `AIRFLOW__SCHEDULER__PARAMETER_NAME`.

- `min_file_process_interval`: The frequency that each Dag file is parsed, in seconds. Updates to Dags are reflected after this interval. A low number increases scheduler CPU usage. If you have dynamic Dags created by complex code, you can increase this value to improve scheduler performance. The default value is 30 seconds.

If you have fewer than 200 Dags in a Deployment on Astro, it's safe to set `AIRFLOW__SCHEDULER__DAG_DIR_LIST_INTERVAL=30` (30 seconds) as a Deployment-level environment variable.

- `scheduler_heartbeat_sec`: Defines how often the scheduler should run (in seconds) to trigger new tasks. The default value is 5 seconds.
- `max_dagruns_to_create_per_loop`: The maximum number of Dags to create Dag runs for per scheduler loop. Decrease the value to free resources for scheduling tasks. The default value is 10.
- `max_tis_per_query`: Changes the batch size of queries to the metastore in the main scheduling loop. A higher value allows more task instances to be processed per query, but your query may become too complex and cause performance issues. The default value is 16 queries. Note that the `scheduler.max_tis_per_query` value needs to be lower than the `core.parallelism` value.

For more information on scheduler parameters see also Fine-tuning your Scheduler performance in the Airflow documentation.

Dag processor settings

Dag processor config settings tune the Dag processor component which reads your Dag files to create the serialized version in the Airflow metadata database. The associated environment variables for all parameters in this section are formatted as `AIRFLOW__DAG_PROCESSOR__PARAMETER_NAME`.

- `dag_file_processor_timeout`: How long a DagFileProcessor, which processes a Dag file, can run before timing out. The default value is 50 seconds.
- `refresh_interval`: The frequency that the Dags directory is scanned for new files, in seconds. The lower the value, the faster new Dags are processed and the higher the CPU usage. The default value is 300 seconds (5 minutes).
  It's helpful to know how long it takes to parse your Dags (`dag_processing.total_parse_time`) to know what values to choose for `min_file_process_interval` and `refresh_interval`. If your `refresh_interval` is less than the amount of time it takes to parse each Dag, performance issues can occur.
- `parsing_processes`: How many processes the scheduler can run in parallel to parse Dags. Astronomer recommends setting a value that is twice your available vCPUs. Increasing this value can help serialize a large number of Dags more efficiently. If you are running multiple schedulers, this value applies to each of them.
- `file_parsing_sort_mode`: Determines how the scheduler lists and sorts Dag files to determine the parsing order. Set to one of: `modified_time`, `random_seeded_by_host` and `alphabetical`. The default value is `modified_time`.

Dag-level settings

Dag-level settings apply only to specific Dags and are defined in your Dag code. You should modify Dag-level settings if you want to performance tune a particular Dag, especially in cases where that Dag is hitting an external system such as an API or a database that might cause performance issues if hit too frequently. When a setting exists at both the Dag-level and environment-level, the Dag-level setting takes precedence.

There are three primary Dag-level Airflow settings that you can define in code:

- `max_active_runs`: The maximum number of active Dag runs allowed for the Dag. When this limit is exceeded, the scheduler won't create new active Dag runs. If this setting is not defined, the value of the environment-level setting `max_active_runs_per_dag` is assumed.
  If you're utilizing catchup or backfill for your Dag, consider defining this parameter to ensure that you don't accidentally trigger a high number of Dag runs.
- `max_active_tasks`: The total number of tasks that can run at the same time for a given Dag run. It essentially controls the parallelism within your Dag.

- `max_consecutive_failed_dag_runs`: The maximum number of consecutive failed Dag runs, after which the scheduler will disable this Dag.

You can define any Dag-level settings within your Dag definition. For example:

```
# Allow a maximum of 3 active Dag runs
@dag("my_dag_id", max_active_runs=3)
def my_dag():
```

Task-level settings

Task-level settings are defined by task operators that you can use to implement additional performance adjustments. Modify task-level settings when specific types of tasks are causing performance issues.

There are two primary task-level Airflow settings users can define in code:

- `max_active_tis_per_dag`: The maximum number of times that the same task can run concurrently across all Dag runs. For instance, if a task pulls from an external resource, such as a data table, that should not be modified by multiple tasks at once, then you can set this value to 1.
- `pool`: Defines the amount of pools available for a task. Pools are a way to limit the number of concurrent instances of an arbitrary group of tasks. This setting is useful if you have a lot of workers or Dag runs in parallel, but you want to avoid an API rate limit or otherwise don't want to overwhelm a data source or destination. For more information, see the Airflow Pools Guide.

The parameters above are inherited from the BaseOperator, so you can set them in any operator definition. For example:

```
@task(
    pool="my_custom_pool",
    max_active_tis_per_dag=14
)
def t1():
    pass
```

Depending on how you run Airflow, you may also need to adjust your executor settings when scaling up your environment. See:

- Executors and scaling
- Manage Airflow executors on Astro

Want to know more?

- Best practices for managing Airflow across teams webinar
- Scaling Airflow to optimize performance guide

Debugging Airflow Dags

12

KEY CONCEPTS

- **Structured Debugging:** When debugging any software it helps to have a structured approach, systematically excluding possible points of failure. See: General Airflow debugging approach.
- **Debugging Log**: When debugging more complex issues, consider keeping a log of error messages, code changes and other relevant events to refer back to or use as the basis for asking questions.
- **Ask questions**: Airflow is an open-source tool used by a large community of data engineers who are eager to help and share their knowledge. The Airflow Slack is the best place to ask your Airflow questions. See: I need more help.

General Airflow debugging approach

To give yourself the best possible chance of fixing a bug in Airflow, contextualize the issue by asking yourself the following questions:

- Is the problem with Airflow, or is it with an external system connected to Airflow? Test if the action can be completed in the external system without using Airflow.
- What is the state of your Airflow components? Inspect the logs of each component and restart your Airflow environment if necessary.
- Does Airflow have access to all relevant files? This is especially relevant when running Airflow in Docker or when using the Astro CLI.
- Are your Airflow connections set up correctly with correct credentials?
  See Troubleshooting connections.
- Is the issue with all Dags, or is it isolated to one Dag?
- Can you collect the relevant logs? For more information on log location and configuration, see the Airflow logging guide.
- Which versions of Airflow and Airflow providers are you using? Make sure that you're using the correct version of the Airflow documentation.
- Can you reproduce the problem in a new local Airflow instance using the Astro CLI?

Answering these questions will help you narrow down what kind of issue you're dealing with and inform your next steps.

You can debug your Dag code with IDE debugging tools using the dag.test() method. See Debug interactively with dag.test().

Issues running Airflow locally

The 3 most common ways to run Airflow locally are using the Astro CLI, running a standalone instance, or running Airflow in Docker. This guide focuses on troubleshooting the Astro CLI, which is an open source tool for quickly running Airflow on a local machine.

The most common issues related to the Astro CLI are:

- The Astro CLI was not correctly installed. Run `astro version` to confirm that you can successfully run Astro CLI commands. If a newer version is available, consider upgrading. Note that you need to be at least on version 1.34.0 to run Airflow 3.
- There are errors caused by custom commands in the Dockerfile, or dependency conflicts with the packages in `packages.txt` and `requirements.txt`.
- Airflow components are in a crash-loop because of errors in custom plugins or XCom backends. View scheduler logs using `astro dev logs -s` to troubleshoot.

To troubleshoot infrastructure issues when running Airflow on other platforms, for example in Docker, on Kubernetes using the Helm Chart or on managed services, please refer to the relevant documentation and customer support.

You can learn more about testing and troubleshooting locally with the Astro CLI in the Astro documentation.

Common Dag issues

Dags don't appear in the Airflow UI

If a Dag isn't appearing in the Airflow UI, it's typically because Airflow is unable to parse the Dag. If this is the case, you'll see an Import Error in the Airflow UI.

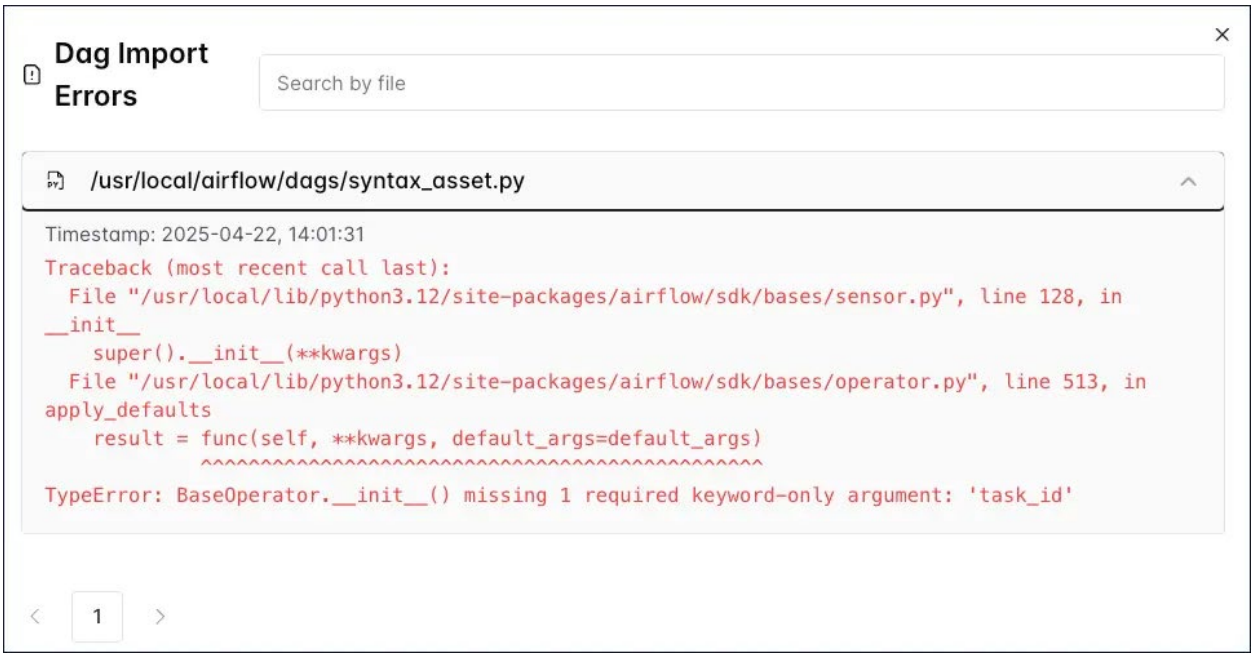

***Figure 1**: Screenshot of the Airflow UI showing a Dag import error due to a missing `task_id`.*

The message in the import error can help you troubleshoot and resolve the issue.

To view import errors in your terminal, run `astro dev run dags list-import-errors` with the Astro CLI, or run `airflow dags list-import-errors` with the Airflow CLI.

If you don't see an import error message but your Dags still don't appear in the UI, try these debugging steps:

- Make sure all of your Dag files are located in the dags folder.
- Airflow scans the dags folder for new Dags every `AIRFLOW__DAG_PROCESSOR__REFRESH_INTERVAL`, which defaults to 5 minutes but can be modified. You might have to wait until this interval has passed before a new Dag appears in the Airflow UI or restart your Airflow environment.
- Ensure that you have permission to see the Dags, and that the permissions on the Dag file are correct.
- Run `astro dev run dags list` with the Astro CLI or `airflow dags list` with the Airflow CLI to make sure that Airflow has registered the Dag in the metadata database. If the Dag appears in the list but not in the UI, try restarting the Airflow API server.

- Try restarting the Airflow scheduler with `astro dev restart`.
- If you see an error in the Airflow UI indicating that the scheduler is not running, check the scheduler logs to see if an error in a Dag file is causing the scheduler to crash. If you are using the Astro CLI, run `astro dev logs -s` and then try restarting.

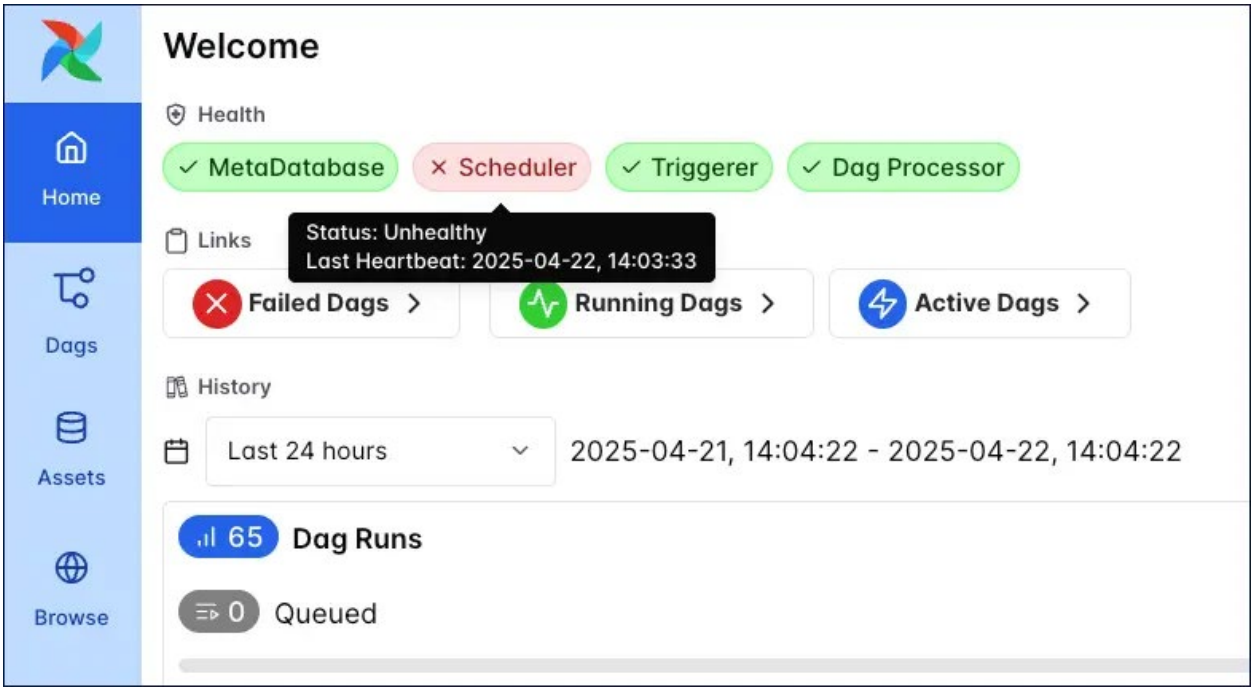

***Figure 2**: Screenshot of the Airflow UI showing a message indication that the scheduler has been down.*

At the code level, ensure that each Dag:

- Has a unique `dag_id`.
- Has its dag function called when defined with the `@dag` decorator. See Writing a simple Dag.

You can also configure an Airflow listener as a plugin to run any Python code, either when a new import error appears (`on_new_dag_import_error`) or when the Dag processor finds a known import error (`on_existing_dag_import_error`). See Airflow listeners for more information.

Dags not running correctly

If your Dags are either not running or running differently than you intended, consider checking the following common causes:

- Dags need to be unpaused in order to run on their schedule. You can unpause a Dag by clicking the toggle on the right side of the Airflow UI or by using the Airflow CLI. Creating a manual Dag run on a paused Dag with the play button by default will unpause the Dag.

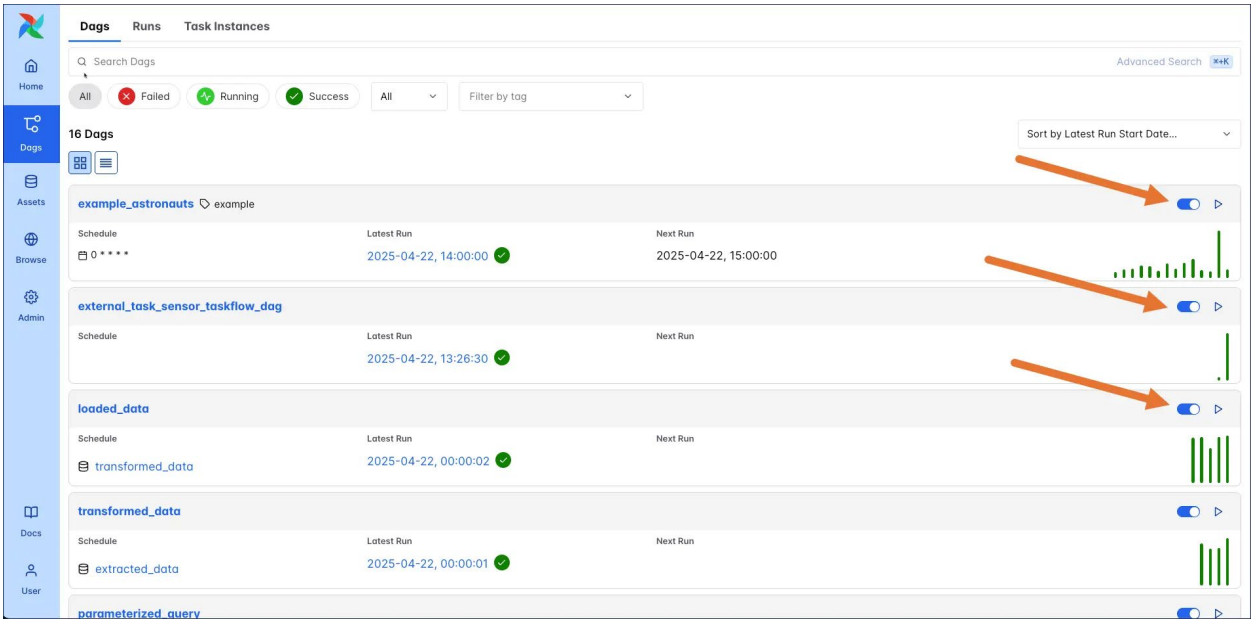

***Figure 3**: Screenshot of the Airflow UI Dags view with the toggle to pause/unpause Dags highlighted.*

- If you want all Dags unpaused by default, you can set dags_are_paused_at_creation=False in your Airflow config.
- Double check that each Dag has a unique `dag_id`. If two Dags with the same id are present in one Airflow instance the scheduler will pick one at random every 30 seconds to display.
- Make sure your Dag has a `start_date` in the past. A Dag with a `start_date` in the future will result in a successful Dag run with no task runs. Do not use `datetime.now()` as a `start_date`.
- Test the Dag using `astro dev dags test <dag_id>`. With the Airflow CLI, run `airflow dags test <dag_id>`.
- If no Dags are running, check the state of your scheduler using `astro dev logs -s`.

If your Dag is running, but not on the schedule you expected, review the Scheduling Dags section of this eBook. If you are using a custom timetable, ensure that the data interval for your Dag run does not precede the Dag start date.

Common Task issues

This section covers common issues related to individual tasks you might encounter. If your entire Dag is not working, see the Dags are not running correctly section above.

Tasks are not running correctly

It is possible for a Dag to start but its tasks to be stuck in various states or to not run in the desired order. If your tasks are not running as intended, try the following debugging methods:

- Double check that your Dag's `start_date` is in the past. A future `start_date` will result in a successful Dag run even though no tasks ran.
- If your tasks stay in a scheduled or queued state, ensure your scheduler is running properly. If needed, restart the scheduler or increase scheduler resources in your Airflow infrastructure.
- If your tasks have the `depends_on_past` parameter set to `True`, those newly added tasks won't run until you set the state of prior task runs.
- When running many instances of a task or Dag, be mindful of scaling parameters and configurations. Airflow has default settings that limit the amount of concurrently running Dags and tasks. See Scaling Dags to learn more.
- If you are using task decorators and your tasks are not showing up in the Graph and Grid view, make sure you are calling your tasks inside your Dag function.
- Check your task dependencies and trigger rules. Consider recreating your Dag structure with EmptyOperators to ensure that your dependencies are structured as expected.
- The task_queued_timeout configuration controls how long tasks can be in queued state before they are either retried or marked as failed. The default is 600 seconds.

If tasks are failing, the task logs will provide you with more information to help troubleshoot. Most task failure issues fall into one of 3 categories:

- **Issues with operator parameter inputs**. For example passing a string to a parameter that only accepts lists.
- **Issues within the operator**. For example exceptions caused by outdated logic in a custom operator, like using deprecated methods of an SDK when connecting to an external system.
- **Issues in an external system**. For example an API or database being down.

Missing Logs

When you check your task logs to debug a failure, you may not see any logs. On the log page in the Airflow UI you may just see a blank file.

Generally, logs fail to appear when a process dies in your scheduler or worker and communication is lost. The following are some debugging steps you can try:

- Try rerunning the task by clearing the task instance to see if the logs appear during the rerun.
- Increase your log_fetch_timeout_sec configuration to greater than the 5 second default. This parameter controls how long the API server waits for the initial handshake when fetching logs from the worker machines, and having extra time here can sometimes resolve issues.
- Increase the resources available to your workers (if using the CeleryExecutor) or scheduler (if using the LocalExecutor).
- If you're using the KubernetesExecutor and a task fails very quickly (in less than 15 seconds), the pod running the task spins down before the API server has a chance to collect the logs from the pod. If possible, try building in some wait time to your task depending on which operator you're using. If that isn't possible, try to diagnose what could be causing a near-immediate failure in your task. This is often related to either lack of resources or an error in the task configuration.
- Increase the CPU or memory for the task.
- Ensure that your logs are retained until you need to access them.
- Check your scheduler and API server logs for any errors that might indicate why your task logs aren't appearing.

Astronomer customers can check the View Airflow component and task logs for a Deployment page of our documentation to see more information on how to find logs for Deployments running on Astro. To evaluate potential resource bottlenecks, Astro allows you to view detailed metrics about your Airflow components in the Astro UI.

Troubleshooting connections

Typically, Airflow connections are needed to allow Airflow to communicate with external systems. Most hooks and operators expect a defined connection parameter. Because of this, improperly defined connections are one of the most common issues Airflow users have to debug when first working with their Dags.

While the specific error associated with a poorly defined connection can vary widely, you will typically see a message with "connection" in your task logs. If you haven't defined a connection, you'll see a message such as 'connection_abc' is not defined.

The following are some debugging steps you can try:

- Review Airflow connections to learn how connections work.
- Make sure you have the necessary provider packages installed to be able to use a specific connection type.
- Change the <external tool>_default connection to use your connection details or define a new connection with a different name and pass the new name to the hook or operator.
- Define connections using Airflow environment variables instead of adding them in the Airflow UI. Make sure you're not defining the same connection in multiple places. If you do, check the Connection options section to review the order of precedence for connections.
- Test if your credentials work when used in a direct API call to the external tool.

To find information about what parameters are required for a specific connection:

- Read provider documentation in the Astronomer Registry to access the Apache Airflow documentation for the provider. Most commonly used providers will have documentation on each of their associated connection types. For example, you can find information on how to set up different connections to Azure in the Azure provider docs.
- Check the documentation of the external tool you are connecting to and see if it offers guidance on how to authenticate.
- View the source code of the hook that is being used by your operator.

You can also test connections from within your IDE by using the `dag.test()` method. See Debug interactively with dag.test() and How to test and debug your Airflow connections.

I need more help

The information provided here should help you resolve the most common issues. If your issue was not covered in this guide, try the following resources:

- Join the Apache Airflow Slack and open a thread in #user-troubleshooting. The Airflow slack is the best place to get answers to more complex Airflow specific questions.
- If you are an Astronomer customer contact our customer support.
- Post your question to Stack Overflow, tagged with airflow and other relevant tools you are using. Using Stack Overflow is ideal when you are unsure which tool is causing the error, since experts for different tools will be able to see your question.
- If you found a bug in Airflow or one of its core providers, please open an issue in the Airflow GitHub repository. For bugs in Astronomer open source tools like Cosmos please open an issue in the relevant Astronomer repository.

To get more specific answers to your question, include the following information in your question or issue:

- Your method for running Airflow (Astro CLI, standalone, Docker, managed services).
- Your Airflow version and the version of relevant providers.
- The full error with the error trace if applicable.
- The full code of the Dag causing the error if applicable.
- What you are trying to accomplish in as much detail as possible.
- What you changed in your environment when the problem started.

Want to know more?

- Airflow: Airflow: Debug Dags Academy module
- Airflow best practices: debugging and testing webinar

Conclusion

Congratulations! You've learned all you need to know about how to write Airflow Dags and are now officially an Airflow magician, welcome to the coven!

What's next? Beyond using your magic to write amazing Dags delivering the world's data, you could consider deploying your Dags to a free Astro trial.

If you're looking to prove your Airflow expertise on your resume, sign up to the Astronomer Academy to go through a structured Airflow course and get Airflow certified!

Thank you for reading, and have a lovely day!