UPDATED FOR AIRFLOW 2.10

Apache Airflow® 101

Essential Concepts and Tips for Beginners

ASTRONOMER

Table of Contents

Editor's Note

Welcome to The Apache Airflow® 101 eBook, brought to you by Astronomer.

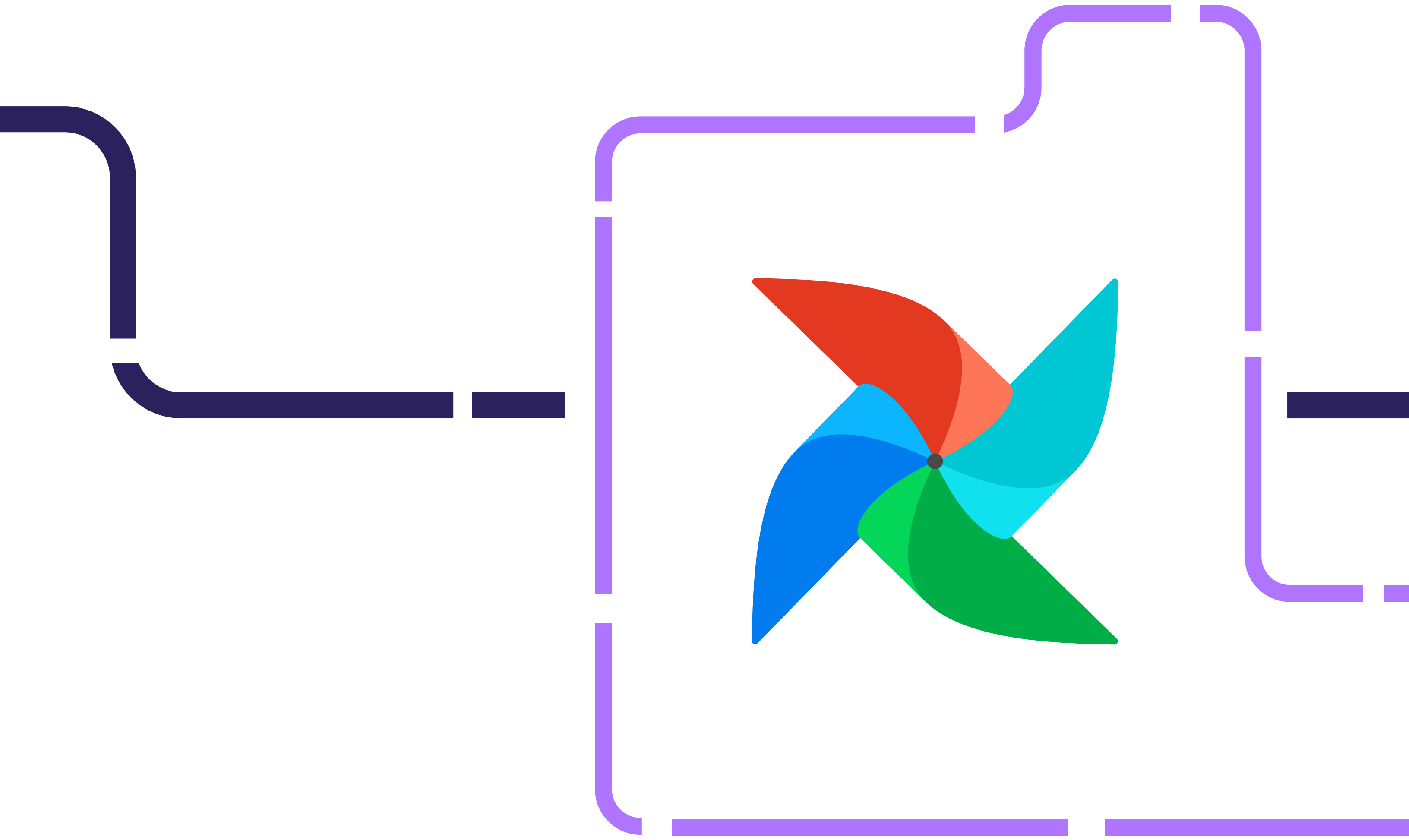
Apache Airflow® is the open-source standard for data orchestration, and getting proficient in Airflow is essential for a successful career in data engineering.

In this eBook, you'll learn everything you need to know to write your first Airflow pipelines, from how to run Airflow, basic concepts to major features and where you can find more information.

We wish you all the best on your Airflow journey!

Make sure to also check out:

- Astronomer webinars: 1h deep-dives into Airflow and Astro related topics.
- Airflow Slack: The best place to ask questions about Airflow (#user-troubleshooting).
- 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.



1

What is Apache Airflow®?

Apache Airflow® is the world's most popular open-source orchestration tool, a framework for programmatically authoring, scheduling, and monitoring data pipelines and workflows. It was created as a data orchestration tool at Airbnb and later brought into the Incubator Program of the Apache Software Foundation®, which named it a Top-Level Apache Project in 2019. Since then, it has evolved into an advanced data workflow orchestration tool used by organizations spanning many industries, including software, healthcare, retail, banking, and fintech.

Today, with a thriving community of more than 3k contributors, 37k+ stars on Github, and tens of thousands of users—including organizations ranging from early-stage startups to Fortune 100s and tech giants—Airflow has become the standard data orchestration solution.

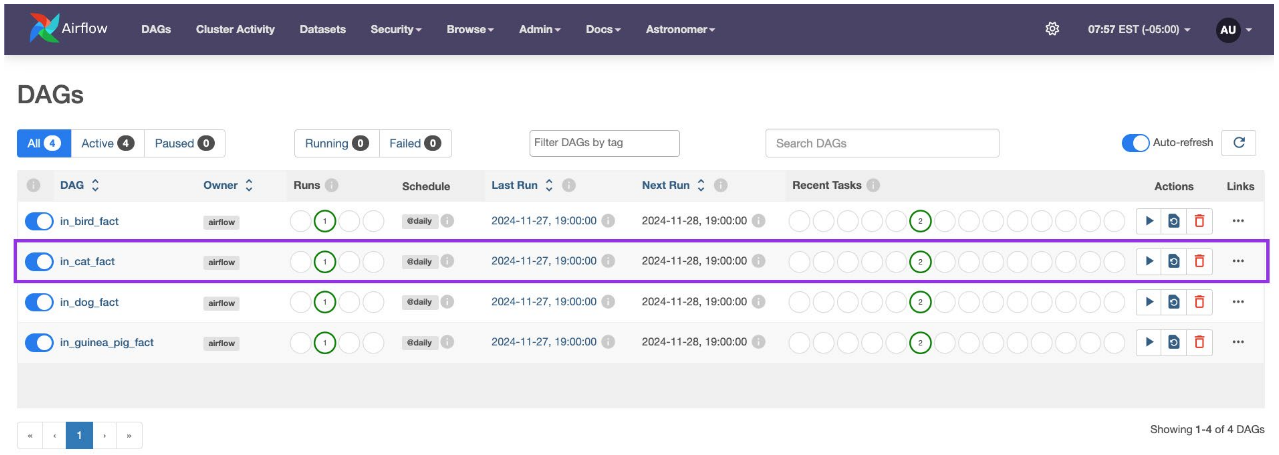
Airflow allows you to express your workflows in Python code, giving you full control and flexibility over how you define your data pipelines and the possibility to connect to any tool that has an API.

For example, the following code defines a simple two-task pipeline, that can be observed in the Airflow UI:

```python
from airflow.decorators import dag, task
from airflow.models.baseoperator import chain
from airflow.operators.bash import BashOperator
from pendulum import datetime
import requests


@dag(
    start_date=datetime(2024, 11, 6),
    schedule="@daily",
    catchup=False,
)
def in_cat_fact():
    @task
    def get_cat_fact():
        r = requests.get("https://catfact.ninja/fact")
        return r.json()["fact"]

    get_cat_fact_obj = get_cat_fact()

    print_cat_fact = BashOperator(
        task_id="print_cat_fact",
        bash_command=f"echo '{get_cat_fact_obj}'",
    )

    chain(get_cat_fact_obj, print_cat_fact)


in_cat_fact()
```

Code block 1: *Example code for a simple Airflow pipeline (in_cat_fact) with two sequential tasks (get_cat_fact, print_cat_fact), the first one retrieving information from an API, the second one printing that information in a bash statement.*

The Airflow UI offers extensive observability of all your data pipelines in the overview:



DAGs

All 4 | Active 4 | Paused 0 | Running 0 | Failed 0 | Filter DAGs by tag | Search DAGs | Auto-refresh

DAG	Owner	Runs	Schedule	Last Run	Next Run	Recent Tasks	Actions	Links
in_bird_fact	airflow	1	@daily	2024-11-27, 19:00:00	2024-11-28, 19:00:00	2		
in_cat_fact	airflow	1	@daily	2024-11-27, 19:00:00	2024-11-28, 19:00:00	2		
in_dog_fact	airflow	1	@daily	2024-11-27, 19:00:00	2024-11-28, 19:00:00	2		
in_guinea_pig_fact	airflow	1	@daily	2024-11-27, 19:00:00	2024-11-28, 19:00:00	2		

Showing 1-4 of 4 DAGs

Figure 1: *Screenshot of the Airflow UI DAGs view showing 4 active pipelines, including `in_cat_fact`, highlighted in a purple box.*

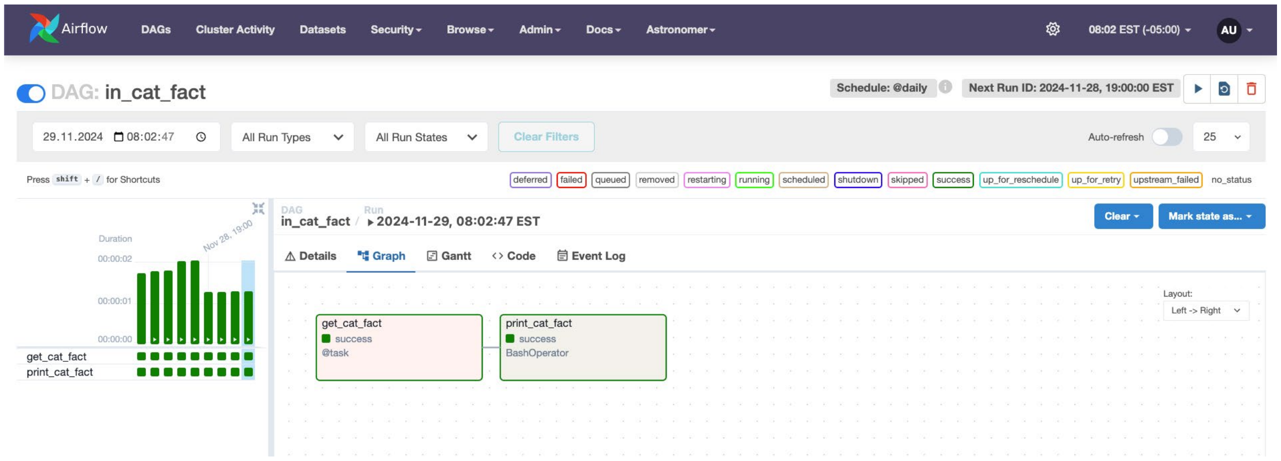


Figure 2: *Screenshot of the Airflow UI showing the graph of one run of the `in_cat_fact` pipeline.*

2

Why Apache Airflow®?

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:

- **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. Solutions like Astro Observe complement Airflow by providing DAG and task-level visibility for complete understanding of your pipelines and their dependencies.

Airflow Use Cases

Many data professionals at companies of all sizes and types use Airflow. Data engineers, data scientists, ML engineers, and data analysts all need to perform actions on data in a complex web of dependencies. With Airflow, you can orchestrate these actions and dependencies in a single platform, no matter which tools you are using and how complex your pipelines are.

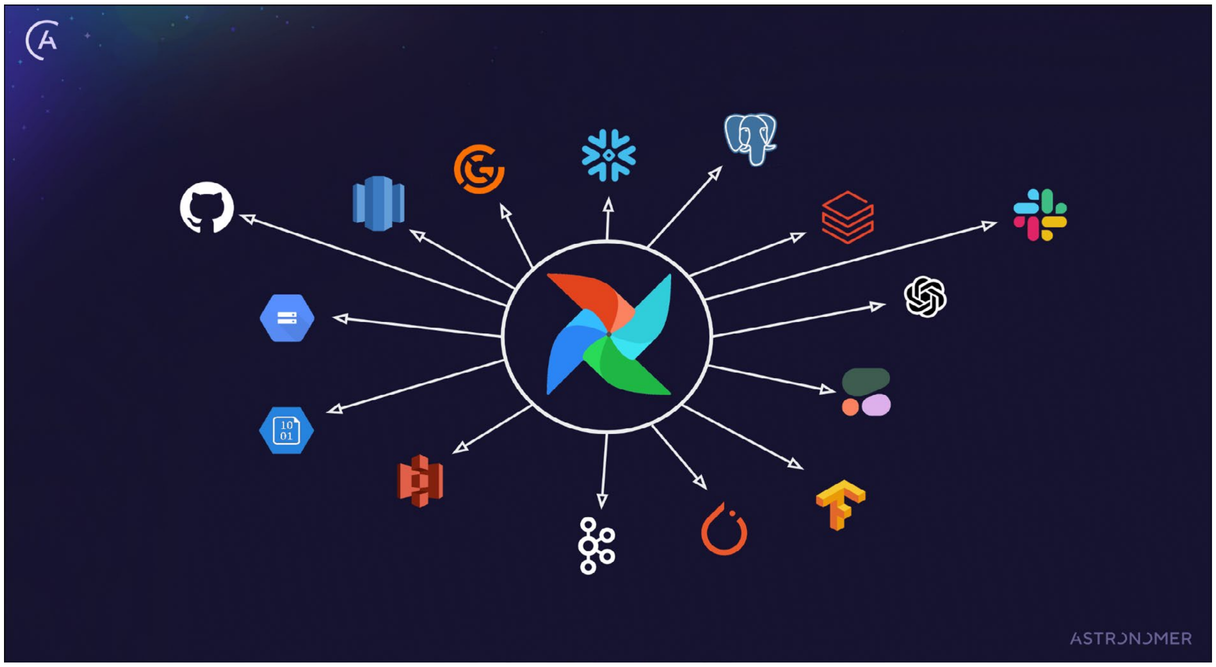


Figure 3: *A few of the many tools in which Apache Airflow can orchestrate actions.*

Some common use cases of Airflow include:

- **ETL/ELT**: 90% of Airflow users use it for Extract-Transform-Load (ETL) and Extract-Load-Transfrom (ELT) patterns. Often, these pipelines support critical operational processes. See our documentation for code examples.
- **Business operations**: 68% of Airflow users have used Airflow to orchestrate data supporting their business directly, creating data-powered applications and products, often in combination with MLOps pipelines. For an example use case, watch the The Laurel Algorithm: MLOps, AI, and Airflow for Perfect Timekeeping webinar.
- **MLOps (incl. GenAI)**: 28% of Airflow users are already orchestrating Machine Learning Operations (MLOps) with Apache Airflow. See Best practices for orchestrating MLOps pipelines with Airflow for more information. For full code examples of GenAI and MLOps use cases see our documentation.
- **Managing infrastructure**: Airflow can be used to spin up and tear down infrastructure. For example, to create and delete temporary tables in a database or spin up and down a Spark cluster.

Of course, these are just a few examples; you can orchestrate almost any kind of batch workflows with Airflow.

If you are looking to learn more about using Airflow for ETL and ELT use cases, check out the **Apache Airflow® Best Practices for ETL and ELT Pipelines** eBook

3

How to run Apache Airflow®

There are many ways to run Airflow, some of which are easier than others.
Astronomer recommends:

- Using the open-source Astro CLI to run Airflow locally. The Astro CLI is the easiest way to create a local Airflow instance running in containers and is free to use for everyone.
- Using Astro to run Airflow in production. Astro is a fully-managed SaaS application for data orchestration that helps teams write and run data pipelines with Apache Airflow at any level of scale. A free trial is available.

All Airflow installations include the mandatory Airflow components as part of their infrastructure: the webserver, scheduler, and database. For other ways to run Airflow see the Airflow documentation.

Local Airflow setup in 5min

To run Airflow locally on your machine, follow these steps to install and use the Astro CLI. If you are on a Mac (for other operating systems see the Astro CLI documentation):

1. Make sure you have Homebrew installed.
2. Run `brew install astro` in a terminal to install the Astro CLI package.
3. Navigate to an empty directory and run `astro dev init` to create an Astro project. This project contains all the files necessary to run Airflow locally, including an example pipeline.
4. Run `astro dev start` to run the project.
5. After the project builds successfully it will automatically open the Airflow UI in your web browser at `http://localhost:8080`.
6. Log in with `admin` as the username and `admin` as the password.

To install additional python packages, including Airflow providers in your project, add them to the `requirements.txt` file. For example to install version 5.8.0 of the Snowflake provider package, add `apache-airflow-providers-snowflake==5.8.0` to the file.

If you can't install new software on your machine: Some users might encounter barriers when installing the Astro CLI. If this is you, don't worry, you can use the Astro CLI without installing anything through GitHub Codespaces.

The only prerequisite is having a GitHub account. Fork this repository and follow the steps in the README to create a fully functional dev environment in a GitHub Codespace.

Run Airflow in the Cloud in 5min

To run Airflow in a production-ready cloud environment, follow the steps below to create an Astro trial. The only requirement is a GitHub account:

1. Sign up for a free trial.
2. Follow the signup flow to create an organization and workspace on Astro.
3. Select any of the template projects for your first Deployment.
4. Connect your GitHub account to Astro to map the Deployment to a new repository in your GitHub account. While this option is part of the free trial signup flow, you can map any Deployment to a GitHub repository any time, see: Deploy code with the Astro GitHub integration.
5. Finish the setup to create your Deployment.

Any changes made to the mapped GitHub repository will automatically be deployed to your Astro Deployment. See Deploy code with the Astro GitHub integration.

You can still create an Astro account and deployment if you do not have a GitHub account. Just click **Skip this Step** instead of selecting a template to create an empty Deployment and check out our documentation for instructions on how to deploy your code.

4

Airflow 101 Core Concepts

The only pre-requisite for writing Airflow pipelines is basic knowledge of Python. Beyond that, you can get started developing and deploying your first pipelines just knowing about a few core concepts.

Terminology

Airflow uses a few key terms to describe data pipelines:

- **DAG**: An Airflow DAG is a workflow defined as a graph, where all dependencies between nodes are directed and nodes do not self-reference, meaning there are no circular dependencies.
- **DAG run**: The execution of a DAG at a specific point in time. A DAG run can be one of four different types: scheduled, manual, dataset_triggered or backfill.
- **Task**: A step in a DAG describing a single unit of work. Tasks are defined using Airflow building blocks such as operators and decorators.
- **Task instance**: The execution of a task at a specific point in time.
- **Dynamic task**: An Airflow task that serves as a blueprint for a variable number of dynamically mapped tasks created at runtime.

DAGs

The following screenshot shows one simple DAG, called `example_astronauts`, with two tasks, `get_astronauts` and `print_astronaut_craft`. The `get_astronauts` task is a regular task, while the `print_astronaut_craft` task is a dynamic task, as indicated by the `[ ]` behind its name. The grid view of the Airflow UI shows individual DAG runs and task instances, while the graph displays the structure of the DAG.

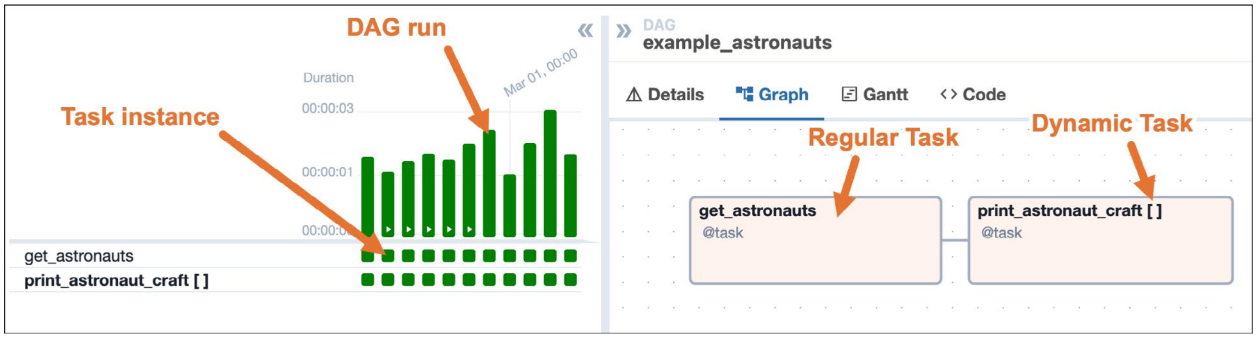


Figure 4: *Screenshot of the Airflow UI showing a DAG graph with a regular and a dynamic task as well as several DAG runs and task instances in the Airflow grid.*

You can define an Airflow DAG using the @dag decorator. The three parameters you should always give to your DAG are:

- start_date: The date and time after which the DAG starts being scheduled. Note that the first actual run of the DAG may be later than this date depending on how you define the schedule. See DAG scheduling and timetables in Airflow for more information. This parameter may be required depending on your Airflow version and schedule.
- schedule: The schedule of the DAG.
- 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 True. It is a best practice to always set it to False unless you specifically want to backfill missed DAG runs, see Catchup for more information.

Any task instantiated in the decorated function will be part of the DAG. For more information on Airflow DAGs, see Introduction to Airflow DAGs.

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


@dag(
    start_date=datetime(2024, 11, 6),
    schedule="@daily",
    catchup=False,
)
def my_dag():
    @task
    def my_task():
        # any Python code
        pass

    my_task()


my_dag()
```

Code block 2: *An Airflow DAG defined using the @dag decorator containing one task. Note that the decorated function needs to be called!*

Schedule

Airflow DAGs can be scheduled to run automatically. There are different kinds of schedules:

- Cron-based schedules:
    - **A cron expression**. For example: `schedule="5 4 * * *"` to run a DAG every day at 4:05am.
    - **A cron preset**. For example: `schedule="@daily"` to run a DAG every day at midnight.
    - **A timedelta object**. For example: `schedule=timedelta(minutes=30)` to run a DAG every thirty minutes.
- Data-aware scheduling. Schedule DAGs to run based on updates to Airflow datasets.
- Timetables. If your time-based schedule can't be expressed using cron you can write a custom timetable to schedule your DAG.

See Schedule DAGs in Airflow to learn more about scheduling.

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

Building Blocks

Airflow tasks are defined in Python code. You can define tasks using:

- **Decorators** (`@task`): The TaskFlow API allows you to define tasks by using a set of decorators that wrap Python functions. This is the easiest way to create tasks from existing Python scripts. Each call to a decorated function becomes one task in your DAG.
- **Operators** (`XYZOperator`): Operators are classes abstracting over Python code designed to perform a specific action. You can instantiate an operator by providing the necessary parameters to the class. Each instantiated operator becomes one task in your DAG.

```
# from airflow.decorators import task

    @task
    def my_task():
        # any Python code
        pass

    my_task()
```

Code block 3: *An Airflow task defined using the `@task` decorator. You can use this decorator to turn any Python function into an Airflow task. Don't forget to call the function.*

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

    my_task = BashOperator(
        task_id="my_task",
        bash_command="echo hello"  # execute any bash command
    )
```

Code block 4: *An Airflow task defined using the `BashOperator`. This operator executes any `bash_command` it is given.*

There are a couple of special types of operators that are worth mentioning:

- Sensors: operators that keep running until a certain condition is fulfilled. For example, the S3KeySensor waits for one or more keys (files) to be present in an S3 bucket.
- Deferrable Operators: operators that use the Python asyncio library to run tasks asynchronously, which can help save resources and costs if you run many tasks in Airflow. Note that your Airflow environment needs to run a triggerer component to use deferrable operators.

Some commonly used building blocks, like the `BashOperator`, the `@task` decorator, or the `PythonOperator`, are part of core Airflow and automatically installed in all Airflow instances. Other operators are maintained separately to Airflow in Airflow provider packages, which group modules interacting with a specific service into a package.

You can browse all available operators and find detailed information about their parameters in the Astronomer Registry. For many common data tools, there are integration tutorials available, showing a simple implementation of the provider package.

Task Dependencies

In Airflow you can define any DAG structure, as long as the graph is directed and does not contain any circles. You can define the dependency between two tasks by using the `chain` function. The tasks in the `chain` function need to execute in the given order. In the code example below, `my_upstream_task` needs to complete successfully before `my_downstream_task` can run.

```
chain(my_upstream_task, my_downstream_task)
```

Code block 5: *Code example defining the dependency between `my_upstream_task` and `my_downstream_task`.*

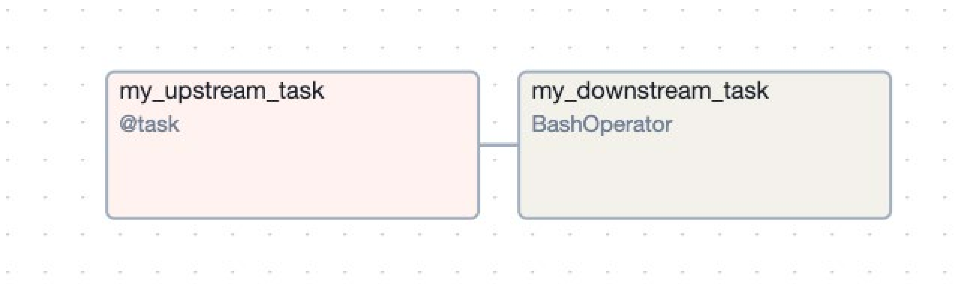


Figure 5: *The resulting DAG graph showing `my_upstream_task` to the left and the dependent `my_downstream_task` to the left.*

You can learn more about setting dependencies in Airflow in the Manage task and task group dependencies in Airflow guide.

> By default, a task waits until all its upstream tasks have completed successfully. You can modify this behavior using **Airflow trigger rules**.

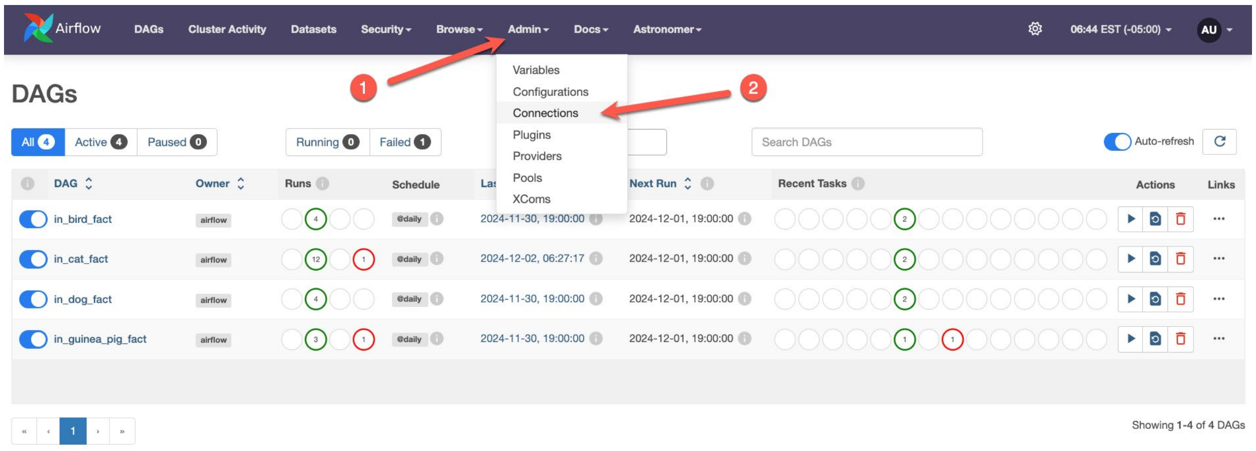
Connecting Airflow to other tools

Apache Airflow sits at the center of your data ecosystem. Airflow operators connect to other data tools using Airflow connections, a secure way to store your credentials when using Airflow.

An Airflow connection is an object that stores all connection information to connect to a specific tool, for example AWS or Snowflake. Each Airflow connection is identified by a unique string, the Connection ID. A Connection ID can be given to many Airflow operators, using a `conn_id` or similarly named parameter.

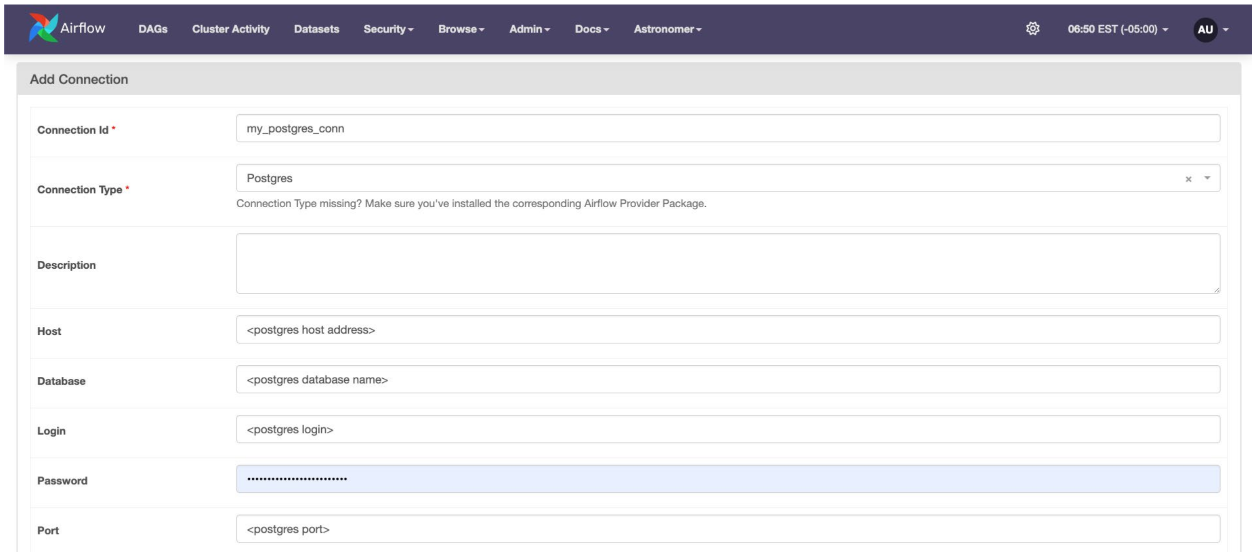
You can define Airflow connections in many different ways. A common way to get started is by using the Airflow UI.

To define a connection in the Airflow UI, navigate to the Connections view by clicking on the **Admin** tab in the top bar and then on **Connections**. Click the **+** sign to create a new connection and fill out your connection details. In our example the connection is to a Postgres database.



DAG	Owner	Runs	Schedule	Las	Next Run	Recent Tasks
in_bird_fact	airflow	4	@daily	2024-11-30, 19:00:00	2024-12-01, 19:00:00	2
in_cat_fact	airflow	12, 1	@daily	2024-12-02, 06:27:17	2024-12-01, 19:00:00	2
in_dog_fact	airflow	4	@daily	2024-11-30, 19:00:00	2024-12-01, 19:00:00	2
in_guinea_pig_fact	airflow	3, 1	@daily	2024-11-30, 19:00:00	2024-12-01, 19:00:00	1, 1

Figure 6: *Instructions for navigating to the Connections page in the Airflow UI (Admin → Connections).*



Add Connection

Connection Id *	my_postgres_conn
Connection Type *	Postgres Connection Type missing? Make sure you've installed the corresponding Airflow Provider Package.
Description	
Host	<postgres host address>
Database	<postgres database name>
Login	<postgres login>
Password	••••••••••••••••••••••••••••
Port	<postgres port>

Figure 7: *Example for a Postgres connection created in the Airflow UI with the Connection ID* `my_postgres_conn`.

If you cannot find your data tool in the **Connection Type** dropdown, make sure you have the relevant Airflow provider package installed in your environment. If you are using the Astro CLI just add the provider package to the `requirements.txt` file.

After saving the connection it can be referenced in operators accepting a connection to Postgres by using the Connection ID. For example to run a SQL statement in the Postgres database with the SQLExecuteQueryOperator.

```
# from airflow.providers.common.sql.operators.sql import SQLExecuteQueryOperator

    my_task = SQLExecuteQueryOperator(
        task_id="my_task",
        conn_id="my_postgres_conn",
        sql="SELECT * FROM MY_DB.MY_SCHEMA.MY_TABLE"
    )
```

Code block 6: *Code example using the SQLExecuteQueryOperator to run a SQL statement in the Postgres database with credentials defined in the* `my_postgres_conn` *connection.*

You can learn more about Airflow connections in the Manage connections in Apache Airflow guide.

Airflow Components

Airflow consists of a few core components. When you are running Airflow using the Astro CLI and an Astro Deployment, you do not need to worry about spinning up individual components, they are automatically set up when running `astro dev start` and when creating an Astro Deployment.

The Airflow components are:

- **Webserver**: A Flask server running with Gunicorn that serves the Airflow UI.
- Scheduler: A Daemon responsible for scheduling jobs. This is a multi-threaded Python process that determines what tasks need to be run, when they need to be run, and where they are run. Within the scheduler, the Executor configuration property determines where and how tasks are run. Astro supports the CeleryExecutor and the KubernetesExecutor in production.
- Database: A database where all DAG and task metadata are stored. This is typically a Postgres database.
- **Worker**: The process that executes tasks, as defined by the executor. Depending on which executor you choose, you may or may not have workers as part of your Airflow infrastructure.
- Triggerer: A separate process which supports deferrable operators.

On Astro you can hibernate a Deployment when you have longer periods without any DAGs running to save cost. See **Hibernate a development Deployment.**

5

Major Airflow Features

The previous chapter taught you everything you need to know to start writing Airflow pipelines. In addition to the basic features, Airflow offers a variety of intermediate and advanced options to optimize your DAGs. This chapter offers an introduction to two key intermediate Airflow features: Data-driven scheduling and dynamic task mapping.

Data-driven scheduling

Airflow offers extensive scheduling capabilities, with one of the most versatile being data-driven scheduling using Airflow datasets. Data-driven scheduling allows a DAG to run automatically, whenever a supporting dataset is updated by any task within your Airflow environment.

To use data-driven scheduling you first need to define a `Dataset`. It is common to use the URI of the underlying data as its identifier, but you can also use any other string. If you aren't using a URI it is a best practice convention to prefix your string with `x-`.

```
# from airflow.datasets import Dataset


my_dataset = Dataset("gs://my-bucket")
```

Code block 7: *Definition of an Airflow dataset with the URI* `gs://my-bucket`.

In any task updating this dataset, for example by uploading a new file to a bucket, you can set the `outlets` parameter to tell Airflow about the task-dataset relationship. Note that this functions as a flag defined by you, the DAG author, Airflow does not interact with the underlying dataset.

```
@task(outlets=[Dataset(my_dataset)])

def my_task():
    # any Python code
    pass

my_task()
```

Code block 8: *Setting an Airflow task created with the `@task` decorator as a producer tasks to the Dataset `my_dataset`.*

```
# from airflow.providers.google.cloud.transfers.gcs_to_gcs import GCSToGCSOperator

my_task = GCSToGCSOperator(
    task_id="my_task",
    ...
    outlets=[Dataset(my_dataset)]
)
```

Code block 9: *Setting an Airflow task created with the `GCSToGCSOperator` decorator as a producer tasks to the Dataset `my_dataset`.*

To schedule a downstream DAG based on an update to one or more datasets, set its `schedule` parameter to the Dataset identified by the same URI string.

```
@dag(

    start_date=datetime(2024, 11, 6),

    schedule=[Dataset(my_dataset)],

    catchup=False,

)
```

Code block 10: *Scheduling a DAG on updates to the `my_dataset` dataset.*

As of Airflow 2.9 you can use conditional logic (AND (&) and OR (|)) to combine datasets in a DAG schedule, for example to run a DAG if either of two datasets have been updated.

```
@dag(

    start_date=datetime(2024, 10, 18),
    schedule=(my_dataset | my_other_dataset),
    # Use () instead of [] to be able to use conditional dataset scheduling!
    catchup=False
)
```

Code block 11: *Scheduling a DAG based on updates to either `my_dataset` or `my_other_dataset`.*

You can even combine dataset scheduling with time-based scheduling, creating a DAG that will run both, at a set time and additionally whenever its dataset condition is fulfilled. The DAG below runs every week, 2am UTC on Monday, as well as whenever one of two dataset has received an update.

```
# from airflow.timetables.datasets import DatasetOrTimeSchedule

# from airflow.timetables.trigger import CronTriggerTimetable


@dag(

    start_date=datetime(2024, 10, 18),

    schedule=DatasetOrTimeSchedule(

        timetable=CronTriggerTimetable(“0 0 * * 1”, timezone=”UTC”),

        datasets=(my_dataset | my_other_dataset),

    ),

    catchup=False,

)
```

Code block 12: *Scheduling a DAG based on updates to either `my_dataset` or `my_other_dataset` as well as running every Monday at midnight.*

Dependencies between DAGs and Datasets can be explored in the Airflow UI in the Datasets tab's Dependency Graph.

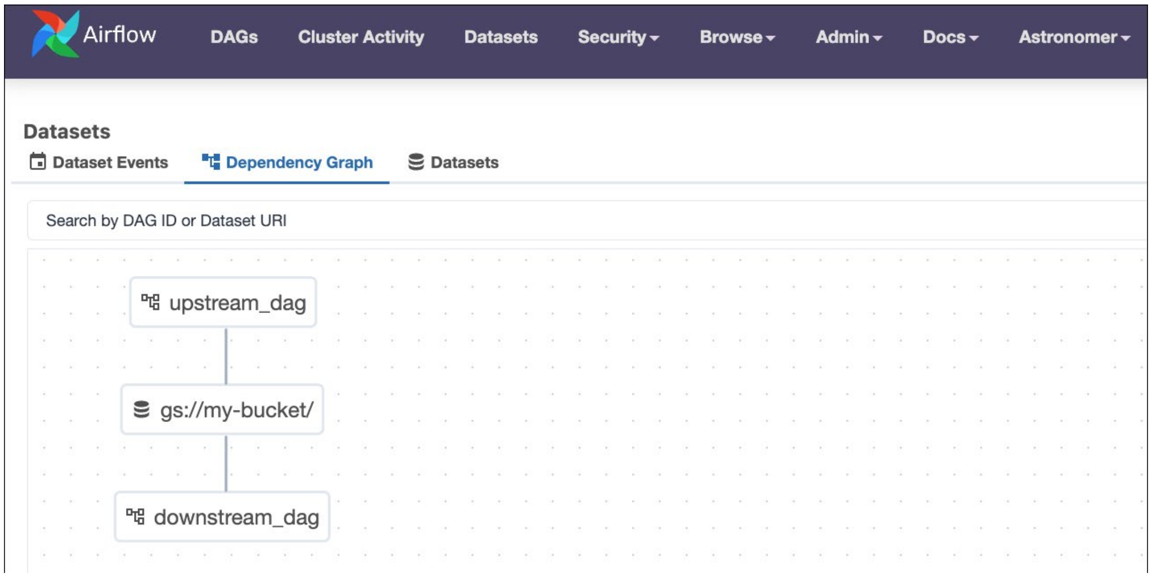


Figure 8. *Screenshot of the Dependency Graph of the Datasets tab showing one dataset with an upstream and downstream DAG dependency.*

To learn more about dataset-driven scheduling including advanced features, see this guide.

Dynamic Task Mapping

Static Airflow pipelines can handle many use cases, but often, you'll reach a point where you need a more fine-grained, parallelized structure in your DAG.

A common scenario is loading data from multiple locations in object storage to a target database. In a static pipeline, you might have one task that extracts all the data, one that transforms it, and one that loads it. However, if just one of the source files is corrupted, the entire task fails and must be rerun.

A better approach is to create a separate task for each storage location. This way, if a failure occurs, you only need to rerun the pipeline for the specific location that contained the corrupted file.

But how do you design such a pipeline when the number of locations or their names change frequently?

This is where dynamic task mapping can help you.

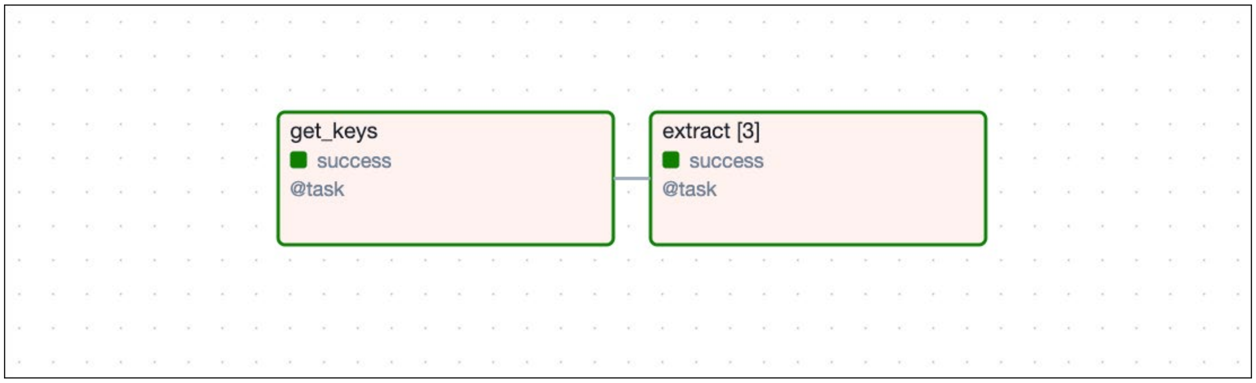


***Figure 9**. A DAG graph containing a dynamically mapped task called `extract` with three parallel dynamically mapped task instances (`[3]`).*

Dynamic task mapping allows you to define a task as a placeholder to create a changing number of dynamically mapped task instances for every DAG run, based on input received at runtime.

In our example you have one upstream task (`get_keys`) pulling the list of this DAG run's extraction locations and then passing that list to the dynamically mapped task to create one dynamically mapped task instance per element in the list, in Figure 10, 3 copies of the `extract` task.

In its most basic form, dynamic task mapping uses two methods:

- **.expand()**: This method passes the parameter 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().

The code snippet below shows how the `extract` task can be given a value for one of its parameters `object_storage_type` that stays constant across all mapped task instances, and a changing value for its other parameter `key` that is derived from the upstream task `get_keys`.

This simple example hardcodes the return values from get_keys but in a real life scenario it could change with every DAG run, for example at one run at 1am returning 20 keys, at the next run at 2am returning 1000 keys and at its next run at 3am returning no keys at all.

No matter how many elements the `get_keys` task returns, the `extract` task can adapt.

```
@task
def get_keys():
    # Code to fetch object storage locations
    return ["bucket1", "bucket2", "bucket3"]


@task
def extract(object_storage_type, key):
    # Code to extract data from object storage locations
    return f"{object_storage_type}://{key}"

_get_keys = get_keys()
_extract = extract.partial(object_storage_type="s3").expand(key=_get_keys)
```

There are many variations to dynamic task mapping, including mapping over multiple outputs or whole task groups. For more information see the Create dynamic Airflow tasks guide.

Other features

Airflow has many more awesome features that can make your DAGs more efficient and easier to manage. A few commonly used features include:

- Airflow retries: In Airflow, you can configure individual tasks to retry automatically in case of a failure. It is a best practice to set retries on all your tasks, unless you have a specific reason not to.
- XCom: XCom stands for cross-communication and is the mechanism by which tasks can exchange data. The standard XCom backend can only handle small amounts of data, but this limitation can be overcome by using a Custom XCom Backend.
- Airflow UI: We've only scratched the surface in exploring the Airflow UI. Check out this guide for an extensive walkthrough.

Some additional features for advanced DAG writing include:

- Hooks: Hooks are Python classes you can use *inside* `@task` decorated functions to use Airflow connections and simplify connecting to other data tools in custom code.
- Airflow context: All Airflow tasks have access to a dictionary containing information about a running DAG and its environment, elements of this dictionary can be used in your Airflow tasks, for example to retrieve the exact timestamp of the current DAG run to be used in a filename.
- Jinja templates: Airflow allows you to use Jinja templating on parameters in many operators.
- Custom hooks and operators: If no operator exists for your use case but you'd still like to standardize your Airflow code you can create your own custom hooks and operators. See the guide for template code.
- Deferrable operators: Deferrable operators use asynchronous processes running in the Airflow Triggerer component, this frees up a worker slot while waiting for long running actions in external systems to complete. Many deferrable operators perform similar actions to sensors. You can use pre-existing deferrable operators or create your own.
- Airflow variables**:** Airflow variables are key-value pairs that you can use to store instance-level information like paths to configuration files.
- Isolated environments: It is very common to run a task with different dependencies than your Airflow environment. Your task might need a different Python version than core Airflow, or it has packages that conflict with your other tasks. In these cases, running tasks in an isolated environment can help manage dependency conflicts and enable compatibility with your execution environments.

Some additional features allowing for advanced DAG structures include:

- Task groups: Airflow task groups are a tool to organize tasks into groups within your DAGs. You can also dynamically map over task groups.
- Branching: Branching in Airflow offers the possibility to decide which task(s) to run next based on conditions at runtime.
- DAG parameters: In this eBook you learned about the `start_date`, `schedule` and `catchup` DAG parameters but there are many more optional parameters with which you can influence aspects of your DAG.
- Airflow params: Params are arguments which you can pass to an Airflow DAG or task at runtime and are stored in the Airflow context dictionary for each DAG run. You can pass DAG and task-level params by using the params parameter and a form will be rendered in the UI upon manual runs.

Some additional features relating to running DAGs in production include:

- The Airflow API: Airflow comes with a full REST API to interact with your Airflow instance from within other tools.
- Airflow notifications: Airflow includes the option to run any Python function upon specific tasks or DAG entering a state. This is commonly used to create task or DAG failure notifications. Note that on Astro you additionally can set Astro alerts in the Astro UI.
- Testing: Airflow DAGs are just Python code, so you can use software development best practices like version control, CICD, and testing when writing Airflow DAGs.
- Plugins: The Airflow UI can be extended with custom plugins.

6

Resources to Get Started with Airflow

Congratulations! You now know all you need to know to write your first Airflow DAGs.

Make sure to also check out:

- The Ultimate Guide to Apache Airflow® DAGs: A 130+ page eBook diving deeper into many best DAG writing best practices at an intermediate to advanced level.
- Astronomer webinars: 1h deep-dives into Airflow and Astro related topics.
- Airflow Slack: The best place to ask questions about Airflow (#user-troubleshooting).
- 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.
- Ask Astro: Conversational chat-bot trained with access to the newest Airflow documentation and resources.
- Free trial of Astro: The best place to run Airflow in production.