Sponsored by ASTRONOMER

Practical Guide to Apache Airflow® 3

T.J. Fingerlin

MANNING

Practical Guide to Apache Airflow® 3

Practical Guide to Apache Airflow® 3

T.J. Fingerlin

MANNING
SHELTER ISLAND

For online information and ordering of this and other Manning books, please visit www.manning.com. The publisher offers discounts on this book when ordered in quantity.

For more information, please contact

Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964
Email: orders@manning.com


Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964

Development editor: Frances Lefkowitz
Copyeditor: Keir Simpson
Proofreader: Melody Dolab
Typesetter: Tamara Švelić Sabljić
Production editor: Aleksandar Dragosavljević

ISBN: 9781633434622
Printed in the United States of America

brief contents

contents

acknowledgments

This book was started during active development of Airflow 3.0, a time when features were still being finalized and polished. Writing in parallel with the evolution of the project required the active involvement of the Airflow developers, and I'm deeply grateful to all the Airflow PMCs and contributors, as well as members of the Astronomer team, who took time out of their already-packed days and nights to answer questions, clarify implementation details, and steer me in the right direction.

I especially want to thank Ash Berlin-Taylor, Vikram Koka, Jed Cunningham, Brent Bovenzi, Kaxil Naik, Daniel Standish, Piotr Chomiak, T.P. Chung, Rahul Vats, Jeremy Beard, Neel Dalsania, Mehul Goyal, Wei Lee, Amogh Rajesh Desai, Ankit Chaurasia, and Jens Scheffler for their help, insights, and reviews, sometimes provided directly and sometimes indirectly by posting exactly what I was looking for in one of the many Airflow 3 channels. I'd also like to thank Abhishek Bhakat for helping me make my own small contribution to Airflow 3.

Thanks to Naveen Sukumar, Steven Hillion, Laura Zdanski, Yan Mastin, and Marc Lamberti for their expert reviews and suggestions. Constance Martineau, thank you for reviewing many chapters, helping me make sense of changes and new features in Airflow 3, from assets to logical dates, and being my sounding board as I figured out how to contextualize and present them.

To the Manning team—Frances Lefkowitz, Keir Simpson, Azra Dedic, Melody Dolab, Nathaniel Coon, Aleksandar Dragosavljević, Tamara Švelić Sabljić, and Debbie Holmgren—your support made this book possible. Thank you for polishing every chapter, bringing clarity to the content and bearing with my last-minute changes.

Thanks to the entire Astronomer marketing team for always having my back, giving me the opportunity and space to write, and for their ongoing efforts to ensure that as many Airflow users as possible will read this book.

Finally, I'd like to thank Kenten Danas for her support from the very first moment I floated the idea of writing an Airflow 3 book to coincide with the release. From creating the book's strategy and content plan to carefully reviewing every chapter, she was there every step of the way. I couldn't have done it without you, Kenten, and I'm deeply grateful for your help and friendship.

1 Welcome to Airflow 3

This chapter covers

- How to use this book
- Apache Airflow 3's exciting new features
- The role of Astronomer in the Airflow project and community
- Who uses Airflow today and what they use it for

Apache Airflow (https://airflow.apache.org) is an open source tool used to write, schedule, and manage workflows as code. Whenever you have actions that depend on one another and must be performed in a specific order, you can define them as a workflow in Airflow. A common example is a data pipeline that extracts data from various sources, combines it, and loads it to a destination. With Airflow, you can define this pipeline in a Python file and schedule it to run automatically. In the Airflow UI, you can observe and interact with the pipeline, including by viewing detailed information about past runs.

Because your pipeline is defined in code, you can apply software best practices such as version control and continuous integration/continuous development (CI/CD) and connect to any tool that has an API, giving you full flexibility for

defining and managing your workflows. Additionally, Airflow contains a large set of easy-to-implement features that make your workflows more robust and even adapt to your data automatically.

Over the past few years, Airflow has been widely adopted as the industry standard for workflow orchestration and the central element for modern DataOps. Whether the workflows enable operational analytics, customer-facing data products, machine learning (ML) operations, or generative AI (GenAI) workflows, Airflow sits at the center, ensuring that the actions making up these workflows occur in the correct order and with the right dependencies.

In April 2025, Apache Airflow Project Management Committee (PMC) released the third major version of Airflow, including long-anticipated features such as dag versioning, expanded data awareness, and improved task isolation. This book provides an introduction to Airflow 3, highlighting the most significant features and changes in this version. You do not need Airflow experience or knowledge of Airflow 2 to read this book; it contains everything you need to write your first pipeline. The only prerequisite is a basic understanding of Python (https://wiki.python.org/moin/BeginnersGuide). In fact, throughout this book, you will follow Chris, a data engineer working at an online news publication. Chris is using Airflow for the first time to automate an up-and-coming newsletter that delivers a selection of inspiring quotes.

Whether you are new to Airflow or you've been writing dags long enough to remember the Tree View, we are excited to share this book with you and hope that it serves as your trusted companion while you explore Airflow 3.

NOTE You can find the code used for all figures and code listings in the companion GitHub repository (https://github.com/astronomer/practical-guide-to-apache-airflow-3).

1.1 How to use this book

This book is structured to introduce the features and changes in Airflow 3 through distinct chapters that can be read independently, but you can also read it cover to cover while you enjoy a nice cup of tea. We recommend the following starting points, depending on your situation:

- *If you are new to Airflow*, you should start by reading this chapter for a general overview and then continue to chapter 2, which covers setting up a local Airflow environment and writing your first pipeline.
- *If you are an experienced Airflow user* looking to upgrade your existing dags as fast as possible, we recommend jumping to chapter 8, which covers upgrading utilities and breaking changes.
- *If you are a platform engineer or Airflow administrator*, you are likely to be most interested in the contents of chapter 5, which covers Airflow architecture changes, and chapter 6, which illustrates how to deploy pipelines to production using Astro, the DataOps platform built on Airflow by Astronomer.

- *If you are looking for a list of Airflow-related resources,* chapter 10 provides a collection of links to documentation, educational materials, and the Airflow community.

1.2 What is Apache Airflow?

Apache Airflow empowers you to automate workflows using code. An Airflow environment contains a workflow defined using one or more dags. *Dag* is an Airflow term derived from the acronym for Directed Acyclic Graph. A dag can be created using a task-oriented approach (section 1.2.1) or an asset-oriented approach (section 1.2.2). The term *pipeline* can refer to one or more dags in an Airflow environment plus their associated actions and data objects in other tools.

Dags can be scheduled to run automatically based on time and/or data-specific requirements (chapter 3), performing actions in different tools and executing code with clearly defined dependencies. When you are writing a dag to train an ML model, for example, the training data must be ready before model training can begin. This dependency is enforced by the Airflow dag definition; the dag's schedule determines exactly when the model is trained.

The origin of the term *dag*

A *dag* in Airflow is a structure representing a workflow that consists of tasks and their dependencies. Each task in the workflow is visualized as a node in a graph, with the edges between nodes denoting task dependencies. In a dag, all dependencies between nodes are directed, and nodes do not self-reference, meaning that there are no circular dependencies (so the dag is acyclic). The mathematical concept of a Directed Acyclic Graph (DAG) has been widely used in graph theory and computer science, particularly in scheduling systems for tasks that must be executed in a specific order.

In the context of Airflow and data engineering, dags commonly represent data pipelines, with the tasks of a dag performing the actions that form the pipeline. Using a DAG structure ensures that tasks execute only after all their dependencies are met and that each task's dependency requirements are clearly defined.

Although dags can define any type of workflow, from managing infrastructure to fine-tuning generative AI models, the most common use case is an extract-transform-load (ETL) or extract-load-transform (ELT) pipeline. An ETL pipeline extracts data from a source system like an API, transforms the data, and loads it to a target system, such as a relational database.

1.2.1 The task-oriented approach

A common way to create a dag in Airflow is to define individual tasks, each of which performs an action in the pipeline, and then define their dependencies. Before Airflow 3, this *task-oriented approach* was the only way to create an Airflow dag.

Figure 1.1 is an architecture diagram which includes a dag that representing a simple ETL pipeline, showing how individual tasks interact with other tools in the system.

By default, dags are read from left to right, so in the example dag, the extract task runs first, followed by the transform task, which depends on successful completion of the extract task. When the transform task completes successfully, the load task can run.

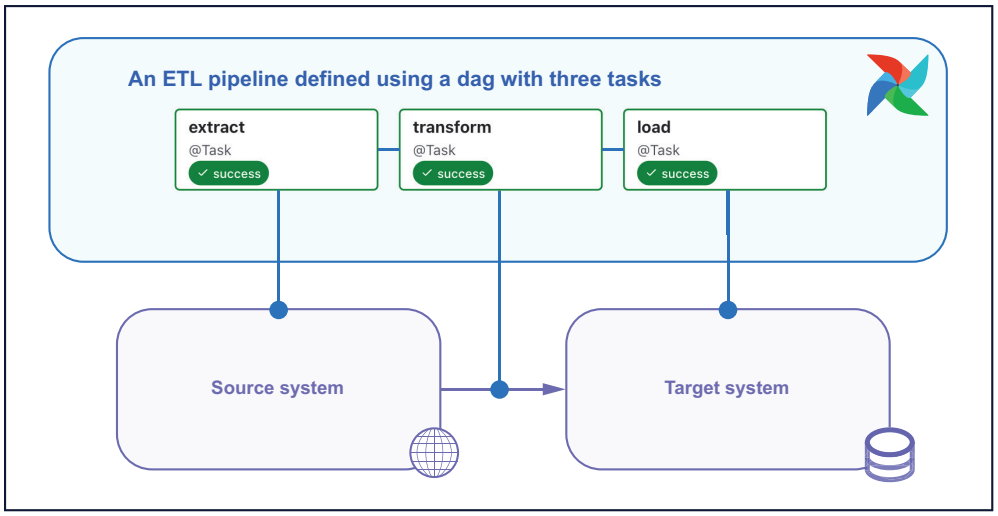


Figure 1.1 A simple ETL pipeline created using one Airflow dag. This dag consists of three sequential tasks: extract, transform, and load.

In chapter 2, you will learn how to write and schedule dags using the task-oriented approach. Here, we'll look at the code used to create the dag in figure 1.1. Note that the syntax example in the following listing does not pass data between tasks. See the companion repository (https://github.com/astronomer/practical-guide-to-apache-airflow-3) for more code examples, including a version that passes data between tasks.

Listing 1.1 Code structure of the task-oriented approach

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


@dag
def my_etl_pipeline():
    @task
    def extract():
        pass

    @task
    def transform():
```

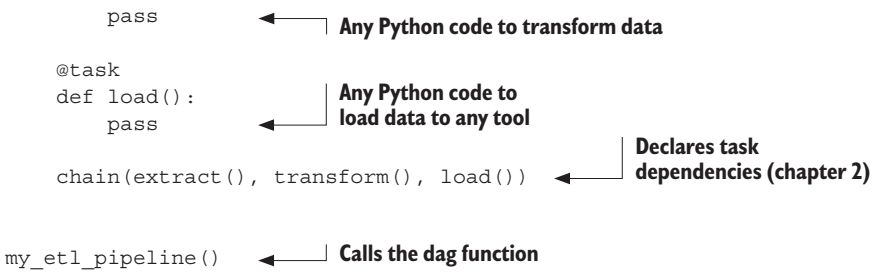
Defines a dag with default parameters

By default, the @dag decorated function's name is the ID of the dag.

Defines a Python task with default parameters

By default, the @task decorated function name is the ID of the task.

Any Python code to extract data from any tool



Task decorators vs. traditional Airflow operators

The dags in this book use the `@task` decorator to define Airflow tasks. A second way to define Airflow tasks is to instantiate operator classes such as `BashOperator`, `PythonOperator`, or `SQLExecuteQueryOperator`, which is still valid in Airflow 3.

Many of these traditional operators are contained in additional Airflow provider packages (see the Astronomer Registry at https://registry.astronomer.io for a comprehensive collection) and are useful for performing specific tasks. `S3ListOperator`, for example, lists all files at a provided prefix in an AWS S3 bucket.

When you create Python-based tasks, your decision about whether to use `@task` decorator or `PythonOperator` is based on your preferences. You can find examples that use traditional Airflow operators in the companion GitHub repository (https://github.com/astronomer/practical-guide-to-apache-airflow-3).

1.2.2 The asset-oriented approach

Assets are objects within Airflow that represent real or abstract data structures in other systems. An asset is identified by a unique name and can have a Uniform Resource Identifier (URI; https://www.w3.org/TR/webarch/#identification) string attached when representing a real data structure.

Examples of data that might be represented as assets are a file in blob storage such as S3, a table in a relational database like MySQL, or the set of parameters for a trained ML model stored in MLflow. It is also possible to define an asset that represents an abstract concept without referencing a concrete object.

Assets in Airflow 3 are an expansion of the dataset feature in Airflow 2.4 and later. In addition to using assets in dags written with the task-oriented approach, you can use Airflow 3 to create a pipeline by defining the desired assets and their dependencies directly. This way of writing a pipeline in Airflow is called the *asset-oriented approach*.

You can choose to write an ETL pipeline as shown in figure 1.1 using assets that depend on one another instead of tasks, as shown in figure 1.2. Under the hood, each asset that is defined creates one dag containing one task that materializes the asset.

Writing a pipeline using the asset-oriented approach puts the focus on the object that is created, which is represented in the asset graph in the Airflow UI (see chapter

2) rather than the task that is performed to create said object. This concept is a mental shift in how to think about data pipelines, moving from a task-first to a data-first perspective. For many pipelines, user preference determines whether to take a task-oriented or asset-oriented approach in defining them. The Airflow developers are planning to greatly expand the capabilities of assets, including asset-level partitioning and validation (chapter 9).

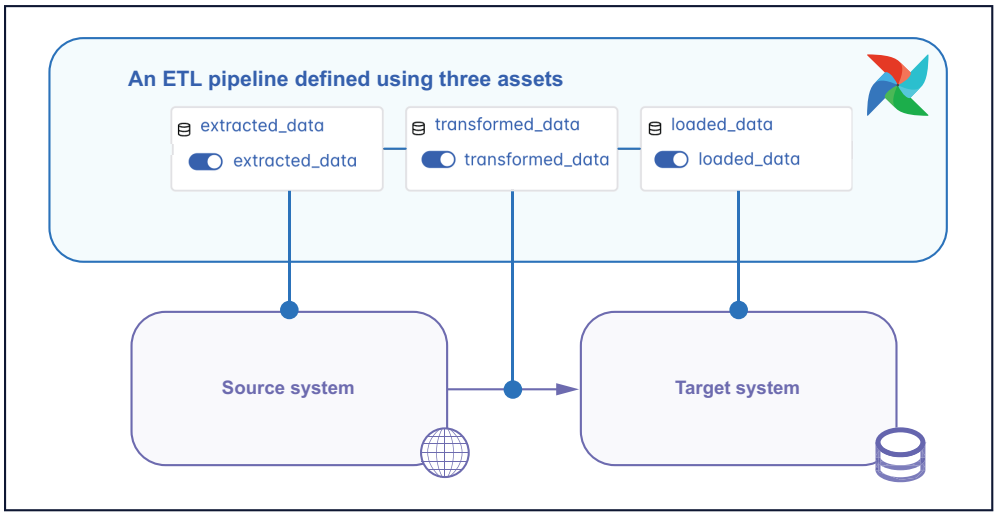


Figure 1.2 The same ETL pipeline as shown in figure 1.1, created using the asset-oriented approach

You will learn how to write (chapter 2) and schedule (chapter 3) assets later in this book. For now, see listing 1.2, which shows the code used to create the three assets in figure 1.2. Again, note that this syntax example does not pass data between assets. See the companion repository (https://github.com/astronomer/practical-guide-to-apache-airflow-3) for more code examples, including a version that passes data between assets.

Listing 1.2 Code structure of the asset-oriented approach

```
from airflow.sdk import asset


@asset(schedule="@daily")
def extracted_data():
    pass


@asset(schedule=[extracted_data])
def transformed_data():
```

Defines an asset

By default, the @asset decorated function's name is the name of the asset.

Any Python code to extract data from any tool

Scheduled to materialize as soon as the upstream asset receives an update

```
    pass                                    <-- Any Python code to transform data

@asset(schedule=[transformed_data])
def loaded_data():                          <-- Any Python code to load data to any tool
    pass
```

This listing is a simple example, of course. In real life, Airflow is used for a large variety of situations, coordinating actions in a multitude of tools and systems with complex dependencies. Because Airflow workflows are code-based, any action you can define in code, including any call to any API, can be part of an Airflow task or asset. A task might kick off a fine-tuning job for a large language model (LLM), run a SQL transformation in a data warehouse, or query a vector database. The only limit is your imagination!

> **NOTE** You may have read that Airflow is a pure orchestrator not designed to process data, which was true of early Airflow versions. Airflow has evolved over the past decade, however. In modern Airflow, you can process data inside Airflow workers, which run task and asset code (chapter 5), provided that you scale your environment appropriately. Even passing larger amounts of data between tasks and assets is possible now as long as you use an external system (referred to as a *custom XCom backend*) to persist the data between tasks; see chapter 6 for more information.

1.3 What's new in Airflow 3

Since the release of Airflow 2.0 in 2020, Airflow adoption has grown exponentially, making it the industry standard for programmatically defining workflows. The top-level Apache Software Foundation (ASF; https://www.apache.org) project is maintained by a vibrant community of engineers who have delivered user-requested new features with each minor version release. Airflow 3.0, released in April 2025, includes a significant architecture change and several of the most frequently requested features.

Table 1.1 provides an overview of significant features added in Airflow 3.0, centering on ease of use, stronger security, and capability to run tasks anywhere at any time. If you are new to Airflow, don't worry; the features are explained in more depth, with examples, in later chapters.

Table 1.1 Significant features added in Airflow 3.0

Feature	Description	See
Assets	Airflow 3 comes with expanded data awareness and a new way of defining pipelines using the `@asset` decorator.	Chapter 2
	The asset-oriented approach marks a paradigm shift in how Airflow pipelines can be defined, allowing for intuitive data-dependent workflows. This approach lays the foundation for upcoming features such as asset partitioning and asset-based data quality validations.	Chapter 9

Table 1.1 Significant features added in Airflow 3.0 (*continued*)

Feature	Description	See
Event-driven scheduling (run at any time)	In Airflow 3, you can schedule a workflow to start running as soon as a message appears in a message queue. This feature allows for fully data-aware dag execution, based on events in external systems without the need for sensors/deferrable operators at the start of a dag.	Chapter 3
UI modernization	Airflow 3 comes with a fully modernized, React-based user interface complete with new views, quick access to task errors, and dark mode.	Chapter 4
Dag versioning and dag bundles	The most frequently requested Airflow feature has arrived! Now Airflow fully retains previous versions of a dag and its associated run history, so even after tasks are removed from a dag, you can easily access their logs for past runs and view previous dag graphs in the Airflow UI. Additionally, Airflow 3 introduces versioned dag bundles to track code changes over time., which also allows Airflow to get dag code from different locations (such as GitHub repositories).	Chapter 4
Scheduler-managed backfills	Backfilling dags for dates in the past has long been common practice. In Airflow 3, backfills are treated as first-class citizens and can be run from the UI and API. Previously, backfilling was possible only via a command-line interface (CLI) command. In many production environments, users can't run CLI commands directly, requiring workarounds to create backfill jobs. These jobs were vulnerable to interruptions if the CLI process was interrupted. Now backfills are handled similarly to regular dag runs, with full observability in the UI.	Chapter 4
New task execution interface (Task SDK)	The new task execution interface provides full task isolation removing direct metadata database access from within Airflow workers. This architectural change greatly improves Airflow's security posture and enables features such as remote execution and multilanguage support.	Chapter 5
Remote execution (run anywhere)	Remote execution brings Airflow into today's distributed world. It allows users to run tasks in (almost) any environment: cloud, on premise, or hybrid. Executing certain tasks in dedicated, isolated environments is beneficial in many situations, such as when handling highly sensitive data that must remain on-premises or when workloads require specialized resources, such as training ML models on dedicated GPUs or TPUs.	Chapter 5
Inference execution	Airflow 3 enables users to run the same dag several times in the exact same moment (providing `None` as the logical date), enabling GenAI use cases such as using a dag as the backend for inference requests.	Chapter 7
Multilanguage support	Airflow 3 enables users to write SDKs allowing definition of Airflow tasks in languages other than Python. Multilanguage support is crucial for preventing language lock-in and enabling highly specialized, optimized tasks. Support for other languages also helps users migrate workflows from legacy tooling, where tasks are often written in a language other than Python, to Airflow without the need for expensive refactoring.	Learn guide (https://www.astronomer.io/docs/learn/airflow-multilanguage)

Many of these features lay the groundwork for improvements in versions beyond 3.0. See chapter 9 for a glimpse of Airflow's exciting future.

> **NOTE** Airflow 3 is a new major version that comes with removal of deprecated syntax and Airflow config changes. The Airflow developers worked diligently to make the upgrade process as easy as possible, including by creating specialized upgrade tooling. See chapter 8 for details on upgrading your existing Airflow 2 dags to Airflow 3.

1.4 Airflow 3 and Astronomer

Astronomer (https://www.astronomer.io) is the force behind open source Apache Airflow, employing many members of the Airflow Project Management Committee and Airflow committers and driving 100% of new Airflow version releases, as well as community events such as the Airflow podcast (https://www.astronomer.io/podcast) and the annual Airflow survey. Astronomer has been heavily involved in planning and developing Airflow 3, and it continues to be a major contributor to the project.

Astro (https://www.astronomer.io/product), Astronomer's unified data operations platform, is a fully managed Airflow offering that includes features for building, running, and observing data workflows. As a managed service, Astro makes it easy to create and operate several Airflow environments running on Kubernetes without managing cloud infrastructure (figure 1.3). A free trial of Astro is available (https://www.astronomer.io/trial-3). You'll learn more about using Astro and running Airflow in production in chapter 6.

1.5 Airflow adoption

You can use Airflow to create workflows that can be defined in code and interact with any tool that has an API. This versatility has led companies of all sizes and types to use Airflow for a large number of use cases.

1.5.1 Airflow use cases

Originally, Airflow was developed to orchestrate ETL and ELT data pipelines, often powering business intelligence dashboards. The latest Airflow survey shows that using Airflow for ETL/ELT related to analytics remains the most common use case, with more than 70% of Airflow users indicating that they have implemented such pipelines. Another widespread application for Airflow is to run business operations, such as creating external data products with the help of Airflow for customer-facing analytics.

Further, data professionals use Airflow to define ML operations (MLOps) pipelines, including GenAI workflows, and to manage infrastructure from within Airflow tasks. Figure 1.4 shows the breakdown of use case responses in the 2024 Airflow survey.

> **TIP** You can access more results of the survey in The State of Apache Airflow 2025 report (https://www.astronomer.io/airflow/state-of-airflow).

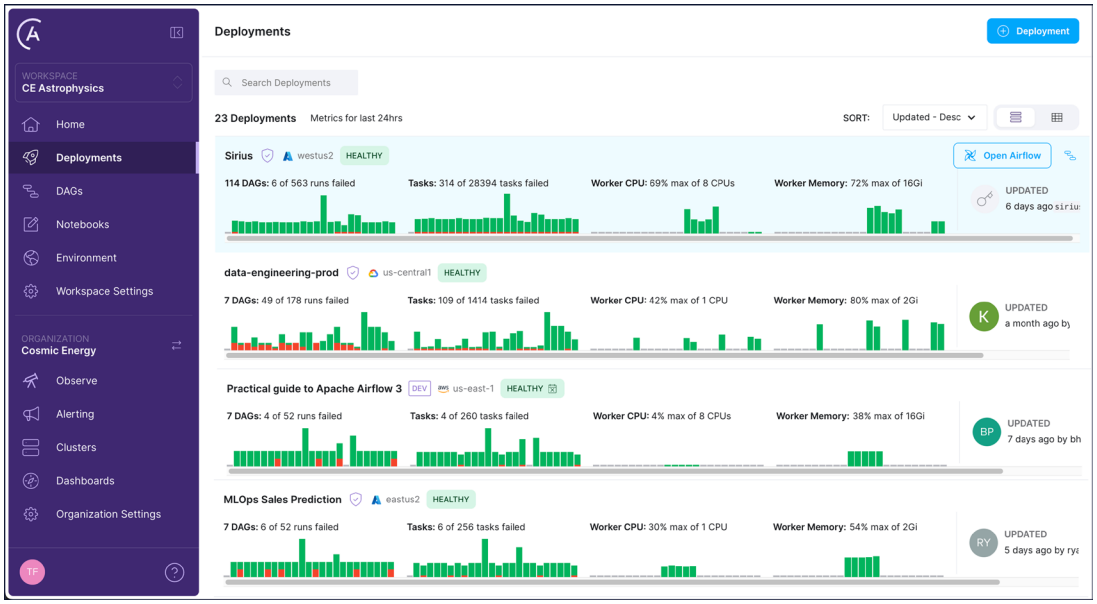


Figure 1.3 The Astro UI showing a Workspace with several Astro Deployments, each running a fully scalable Airflow environment

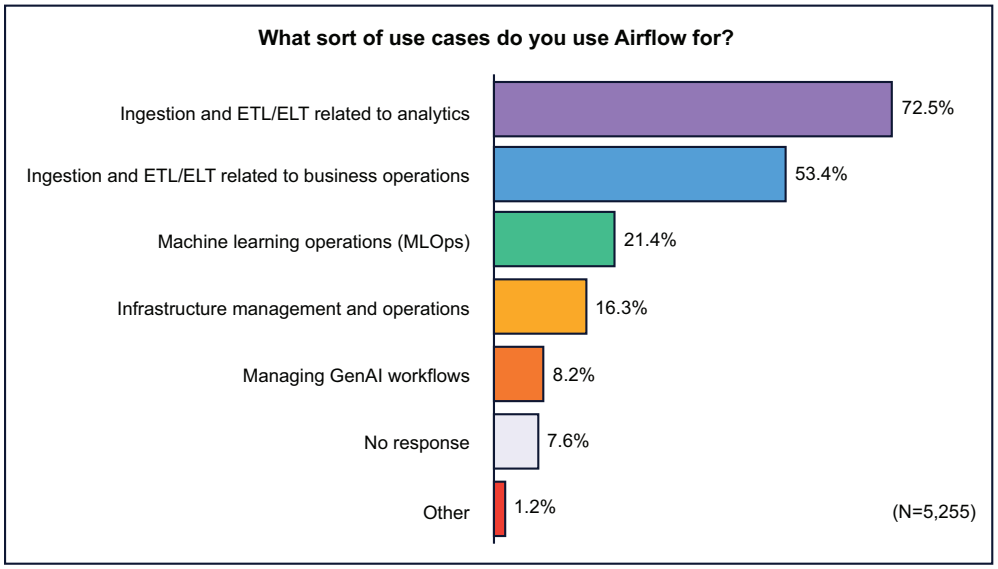


Figure 1.4 Responses of 5,255 Airflow users to the question "What sort of use cases do you use Airflow for?" Multiple answers were allowed. (Source: The State of Apache Airflow 2025 Report, https://www.astronomer.io/airflow/state-of-airflow)

1.5.2 Who uses Airflow

Companies of all sizes and in all industries use Airflow (https://github.com/apache/airflow/blob/main/INTHEWILD.md), from small agile startups to not-for-profit organizations to some of the world's largest enterprises. Working in the field of data, you will encounter Airflow everywhere, including unexpected situations such as powering postgame analytics for the Texas Rangers (https://www.youtube.com/watch?v=8ZW80xdx8vE).

Airflow users form a global open source community, with large groups in Brazil, India, the United States, and Europe (figure 1.5). See chapter 10 for information about participating in the Airflow community.

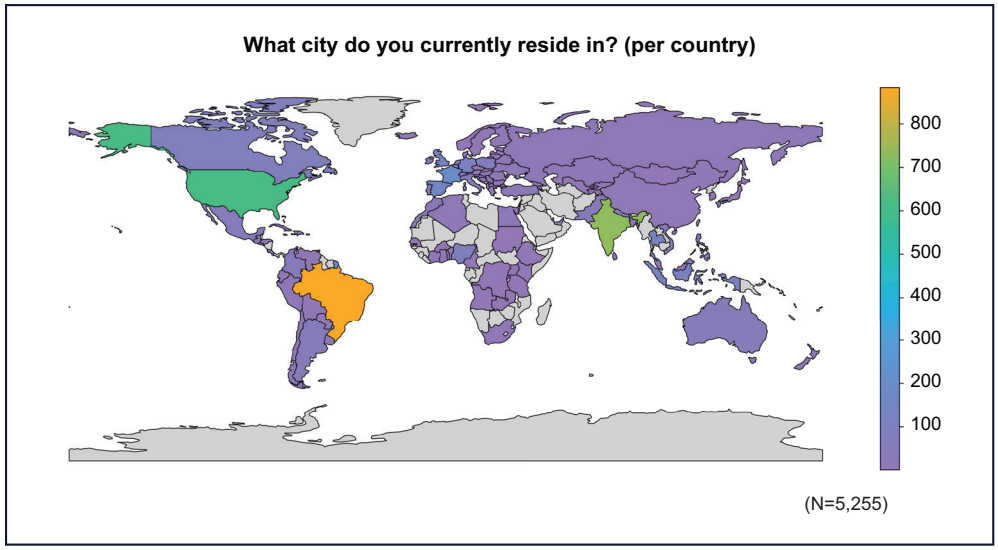


Figure 1.5 World map showing the global distribution of Airflow users according to responses to the 2024 Airflow survey, which included 5,255 users in 116 distinct countries. (Source: The State of Apache Airflow 2025 Report, https://www.astronomer.io/airflow/state-of-airflow)

Although most Airflow users are data engineers, about a third hold other job titles, such as analytics engineer, solutions architect, DevOps engineer, software engineer, or data scientist (figure 1.6), showing the wide range of applications for the tool.

Most Airflow users who responded to the 2024 survey (65%) had already experienced a positive effect on their career related to their Airflow expertise. Another 25% were pursuing or planning to pursue more education in Airflow for the purpose of career growth.

We hope that this book will be a valuable resource to you, helping you expand your existing skills and welcoming new users to the growing Airflow community.

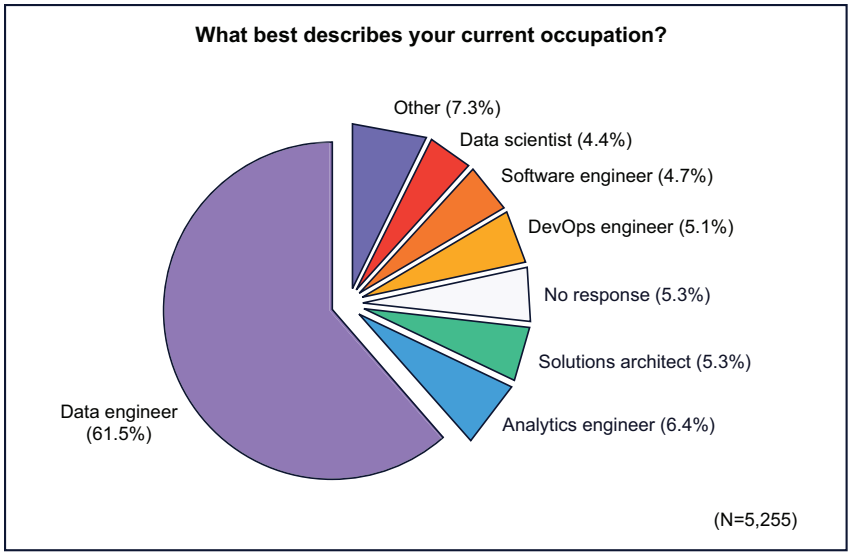


Figure 1.6 Job titles of Airflow users in the 2024 Airflow survey. (Source: The State of Apache Airflow 2025 Report, https://www.astronomer.io/airflow/state-of-airflow)

Summary

- Apache Airflow is an open source tool for creating workflows using code.
- Workflows in Airflow can be defined using dags that contain tasks. This allows you to enforce dependencies between your tasks. A computational resource like a Spark cluster needs to be provisioned by one task before the next task can use it to run a data transformation, for example.
- Airflow 3 expands on assets, which are objects that represent real or abstract data structures in other tools. Assets can be used within dags that are defined using the task-oriented approach or used to write dags in a new way, called the asset-oriented approach. The asset-oriented approach introduces a mental shift in defining an Airflow pipeline, putting the produced data object in the center both in the code and in the Airflow UI.
- Aside from assets, new features introduced in Airflow 3 include dag versioning, event-driven scheduling, UI modernization, improved security, and remote execution.
- Astronomer, a company that offers a DataOps platform built on Airflow called Astro, is the driving force behind the open source Airflow project.
- Airflow is used all over the world by companies of all sizes. While most users are data engineers, Airflow is also used by individuals in a wide range of other roles.

2 Write your first pipeline

This chapter covers

- Conceptually planning a data pipeline
- Setting up a local development environment for Airflow 3
- Writing your first Airflow 3 pipeline using both the asset-oriented and task-oriented approach to define dags

Now that you have a high-level view of Airflow, we'll jump into the action and show you how to write your first Airflow 3 pipeline. You'll follow Chris, the lone data engineer at the rapidly growing online publication *AllThingsLookingUp*, as he sketches out his newsletter pipeline, sets up Airflow locally, and defines his first dags. By the end of the chapter, Chris will have a first proof-of-concept version of his pipeline to improve on and schedule to run automatically in chapter 3.

2.1 Planning a pipeline

"Welcome to the team!" said Michelle, the publication's tech lead. "Happy to have you here fixing all of our data problems. We didn't have a data engineer on staff all of last year, so we are very excited to see what you can automate for us!"

Chris smiled and nodded. It was his first stand-up call, and already, he was starting to realize why the salary for this job was on the high end. He was the only data person working at *AllThingsLookingUp*, and all the data-related challenges were now his challenges, likely with some cryptic legacy SQL to decipher along the way—quite a step up from his last role as a junior member of a large team.

Later that day, after finally getting the right data warehouse credentials, Chris got a message from Jan, the publication's head of content (figure 2.1).

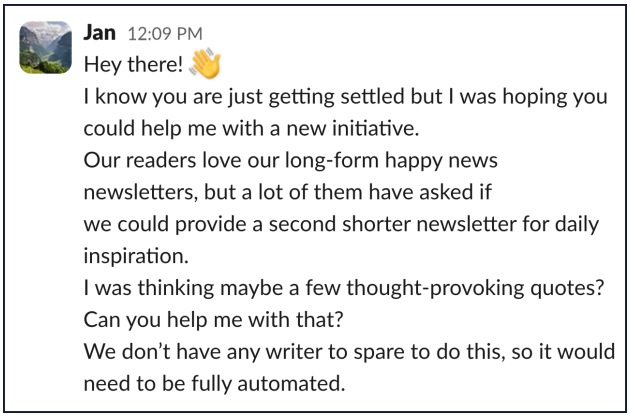


Figure 2.1 Screenshot of a message explaining the need for a fully automated newsletter pipeline sending out inspirational quotes

Ready to dive in, Chris analyzed the requirements for the data product he was asked to create: a text-based newsletter with motivational, philosophical, or inspirational quotes, created automatically every day. That sounded like a job for Airflow.

Luckily, Chris had heard of an API called ZenQuotes (https://zenquotes.io) that serves up inspirational quotes. He checked with Jan to ensure that these quotes were in the spirit of what the content team was looking for and that there were no additional stakeholder concerns to address. Then he could start drafting the pipeline architecture (figure 2.2).

In essence, the needed pipeline has an ETL structure:

- *Extract*—A set of quotes is extracted from the API.
- *Transform*—A selection process determines which quotes to use. It makes sense for Chris to separate this step so that he can easily adjust it down the line—perhaps to filter out quotes that are too long or too short or that have already been used.
- *Load*—The selected quotes are combined with a newsletter template file to create the formatted newsletter, ready to send. In the case of *AllThingsLookingUp*, newsletters were sent by an existing email service, so Chris simply needed to drop the file in the right location.

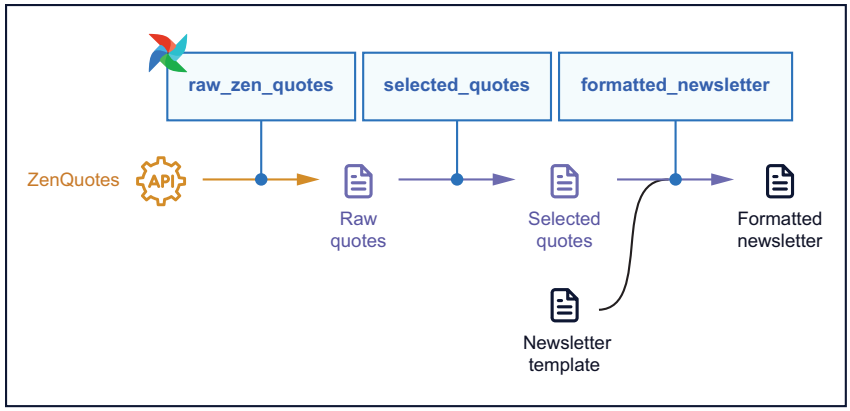


Figure 2.2 Architecture diagram of the newsletter pipeline: an extract-transform-load (ETL) pipeline extracting the quotes from an API, selecting some quotes (transformation), and loading the quotes into the formatted newsletter

Chris thought the pipeline looked simple enough. Wanting to make a good first impression, he was determined to get a prototype of the pipeline up in no time.

2.2 The local Airflow dev environment

To develop an Airflow pipeline, Chris first needed a local development setup, ideally running in a containerized environment. Running Airflow locally in containers matching the production environment is a good best practice because it ensures consistency across different stages of development, testing, and production and reduces the risk of "It works on my machine" problems due to environment mismatches.

After a quick search, Chris came across the Astro CLI (https://www.astronomer.io/docs/astro/cli/overview), a free tool provided by Astronomer that runs Airflow in containers. Setting up a local Airflow environment with the Astro CLI is simple. On a Mac, follow these steps:

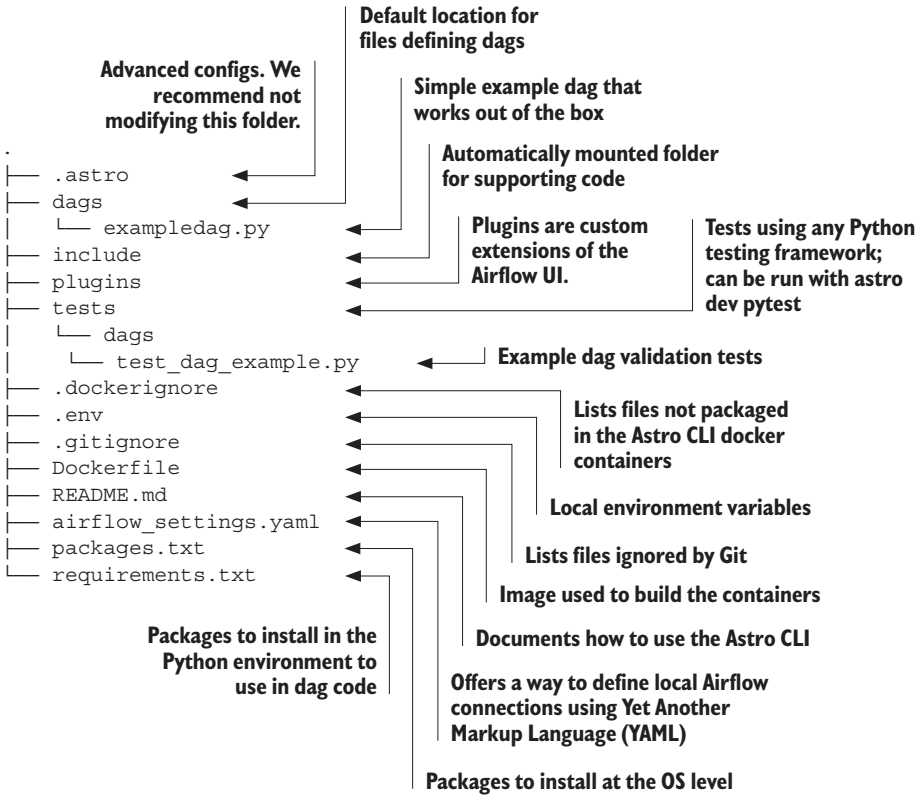
1. Make sure that you have Homebrew (https://brew.sh) installed.
2. Install the Astro CLI by running `brew install astro`.

For other operating systems, see detailed install instructions in the Astro CLI documentation (https://www.astronomer.io/docs/astro/cli/install-cli).

> **NOTE** If you already have the Astro CLI installed, double-check that you have at least version 1.34.x by running the command `astro version`. If you have an older version, you can upgrade with `brew upgrade astro`.

After installing the Astro CLI, you can go to any empty directory and run `astro dev init`. This command creates an Astro project containing all the files and folders needed to run Airflow, including an example dag, as shown in the following listing.

Listing 2.1 Files and folders in an Astro project



Astro projects are designed to work out of the box. Run `astro dev start` in your Astro project directory to spin up an Airflow environment. The Astro CLI uses the Astro Runtime image (https://mng.bz/9yzq), which contains open source Airflow with some additional packages. When the project is up, go to `localhost:8080` to access the Airflow UI and run the example dag (figure 2.3).

> **NOTE** By default, the Astro CLI uses port `8080` for the Airflow API server and port `5432` for the Airflow metadata database. If these ports are occupied on your machine, you can configure the behavior; see instructions in the Astro CLI documentation (https://www.astronomer.io/docs/astro/cli/configure-cli). To learn more about Airflow components such as the scheduler and API server, see chapter 5.

Airflow configuration

Much of Airflow's behavior can be configured at the environment level. Throughout this book, you'll see mentions of Airflow configurations used to modify default settings

in the format `[section].variable_name`. When using the Astro CLI, you can set these configurations by setting the corresponding environment variable `AIRFLOW__SECTION__VARIABLE_NAME=<new-value>` in the `.env` file. Note the double underscores surrounding the section! The default logging level in Airflow, for example, is `INFO`. You can set it to `DEBUG` by changing the `[logging].logging_level` config. To do so, set `AIRFLOW__LOGGING__LOGGING_LEVEL=DEBUG` in `.env`. You can find all Airflow configs in the Airflow documentation (https://mng.bz/jZr8).

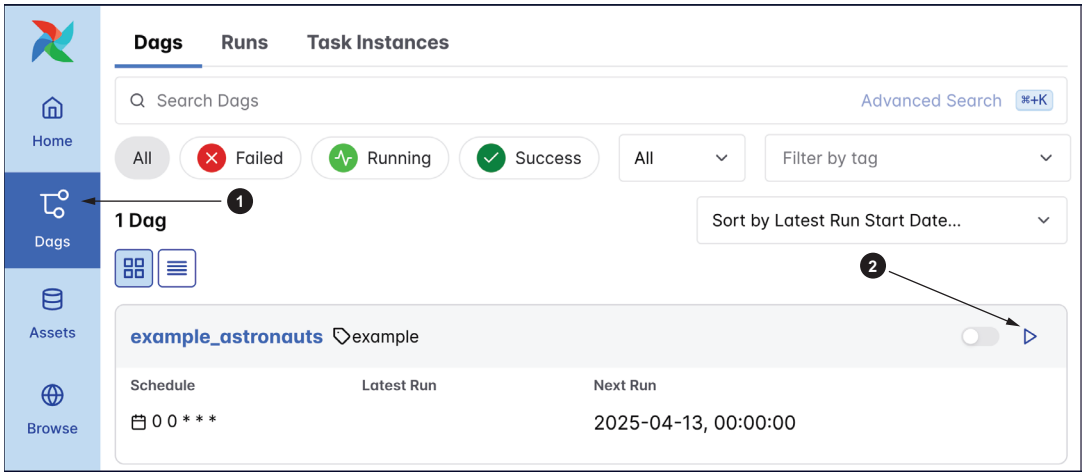


Figure 2.3 Screenshot of the Airflow UI showing the Dags overview page with one example dag. 1. Click the Dags tab to get to the Dags overview page. 2. Click the Play button to run a dag manually. You can learn more about the Airflow UI in chapter 4.

2.3 Writing a pipeline using assets

After installing the Astro CLI and feeling a nice rush of dopamine from seeing the example dag complete successfully, Chris was ready to write his first pipeline. Taking another look at his architecture draft (figure 2.4), he realized that each step in this particular data pipeline creates a new data object. Having recently attended a webinar in which assets were portrayed as a way to define data-aware Airflow pipelines, Chris decided to use the new asset-oriented approach (chapter 1) to write this pipeline.

2.3.1 Extracting data from an API

The first step of the pipeline fetches a random set of quotes from the ZenQuotes API (https://zenquotes.io/api/quotes/random). This set of quotes is a data object created by an `@asset` decorated function and represented in Airflow as an asset. Chris faced a choice about how to pass this data from one `@asset` decorated function to the next:

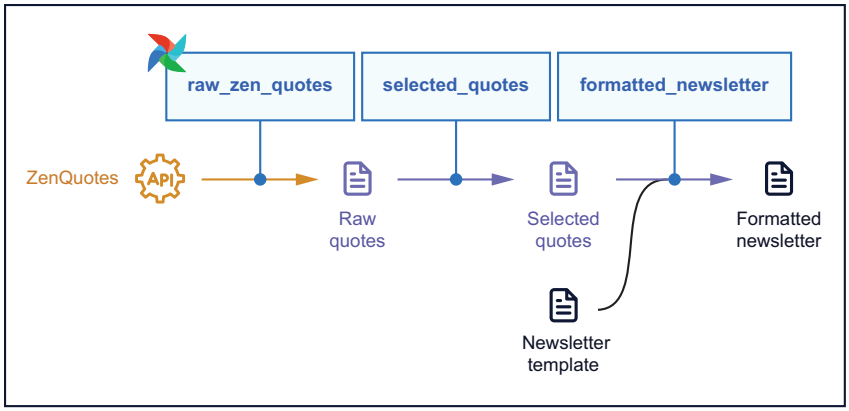


Figure 2.4 Architecture diagram of the newsletter pipeline: an ETL pipeline extracting the quotes from an API, selecting some quotes (transformation), and loading the quotes into the formatted newsletter

- Explicitly write the raw API response to a file, such as in an Amazon S3 bucket, from within the function

  or

- Return the API response from the function, which saves it in the cross-communication (XCom) table of the Airflow metadata database

Because he did not plan to use this data in any other context, Chris chose option 2, which can be easily adjusted for production use in newer Airflow versions (see the sidebar "Passing data between assets and tasks" later in this section). Following is the full code that defines the first asset.

Listing 2.2 The `raw_zen_quotes` asset defined using the `@asset` decorator

```
from airflow.sdk import asset


@asset(schedule="@daily")
def raw_zen_quotes() -> list[dict]:
    """
    Extracts a random set of quotes.
    """
    import requests

    r = requests.get(
        "https://zenquotes.io/api/quotes/random"
    )
    quotes = r.json()

    return quotes
```

Imports the @asset decorator

Schedules the asset to materialize once a day at midnight UTC

By default, the function name (raw_zen_quotes) is the name of the dag, task, and asset.

Modules imported within the function are imported only at run time.

Calls to the ZenQuotes API

Data returned from the function is saved using the XCom feature.

After putting the preceding code in a Python file inside the `dags` folder, Chris was able to view the result in the Airflow UI. Using the `@asset` decorator creates one dag, task, and asset using the name of the decorated function. This means that the code in listing 2.2 creates one dag with the dag ID `raw_zen_quotes`, containing one task with the task ID `raw_zen_quotes`, which materializes the asset using the decorated Python function. Every successful run of the task produces an update to the Airflow asset object of the same name (`raw_zen_quotes`).

The data that is returned from the function is saved in the XCom table of the Airflow metadata database and can be identified by the `dag_id`, `task_id`, and key `return_value` and retrieved from within any other asset or task in this Airflow instance.

> **TIP** By default, the dag-processor parses the `dags` folder every 30 seconds for any changes and every 5 minutes for new dags. If you don't want to wait, you can force a reparsing of the full `dags` folder by running `astro dev run dags reserialize`.

After unpausing the dag, the asset is scheduled to materialize once a day at midnight Coordinated Universal Time (UTC). Chris can also trigger a manual materialization by navigating to the Assets tab and clicking the Play button for the `raw_zen_quotes` asset or by running the `raw_zen_quotes` dag manually (figure 2.5).

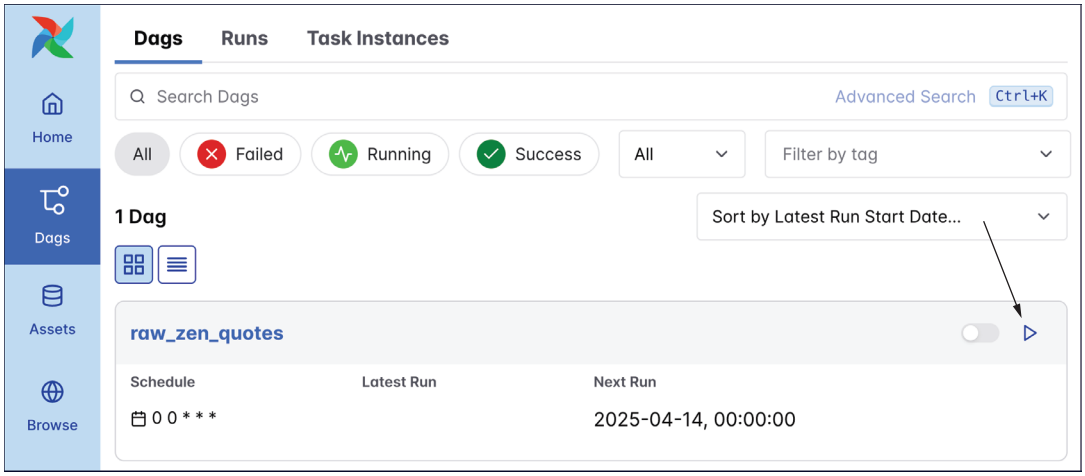


Figure 2.5 Screenshot of the Airflow UI showing how to run the `raw_zen_quotes` dag manually

Passing data between assets and tasks

Passing small amounts of data, such as short quotes, between tasks or assets using standard XCom is OK, especially during prototyping. In production, we strongly recommend using a custom XCom backend that stores the data passed between tasks in

(continued)

another location—commonly object storage such as a bucket in Google Cloud Storage. In Airflow, you can set up this backend using environment variables if you have the Airflow Common IO provider installed; see chapter 6 for more information.

2.3.2 Transforming the API response

So far, so good. The next asset retrieves the raw quotes from the XCom and selects three of them. In the first iteration of the pipeline, Chris used the character count of the quotes (which, conveniently, was already part of the API response under the `c` key) to get a median length, a short quote, and a longer quote. He added the code in listing 2.3 to the Python file.

NOTE Chris's transformation function uses the NumPy package, which is not installed in the Astro CLI environment by default. You can install Python packages hosted on PyPi (https://pypi.org) by listing them in the `requirements.txt` file. It is a best practice to pin the desired version—for example, `numpy==2.2.2`. Make sure to add only the name and version of the package without the `pip install` statement. For modifications to `requirements.txt` to take effect, restart the Astro project with `astro dev restart`.

Listing 2.3 Transforming the quotes and creating a second asset

```
from airflow.sdk import asset

@asset(schedule=[raw_zen_quotes])
def selected_quotes(context: dict) -> dict:
    """
    Transforms the extracted raw_zen_quotes.
    """

    import numpy as np

    raw_zen_quotes = context["ti"].xcom_pull(
        dag_id="raw_zen_quotes",
        task_ids=["raw_zen_quotes"],
        key="return_value",
        include_prior_dates=True,
    )

    quotes_character_counts = [
        int(quote["c"]) for quote in raw_zen_quotes
    ]
    median = np.median(quotes_character_counts)

    median_quote = min(
        raw_zen_quotes,
        key=lambda quote: abs(int(quote["c"]) - median),
    )
```

Materializes as soon as the raw_zen_quotes asset receives an update

The Airflow context contains dag run-specific information.

Imports numpy at run time to calculate the median length of quotes

You can retrieve any value from the XCom table.

The dag from which a value was pushed to XCom. @asset creates a one-task dag.

The task in the upstream dag that pushed the value to XCom

Ensures that the latest XCom matching the settings is fetched

Values returned by functions are saved with the key return_value.

```
    raw_zen_quotes.pop(raw_zen_quotes.index(median_quote))
    short_quote = [
        quote
        for quote in raw_zen_quotes
        if int(quote["c"]) < median
    ][0]
    long_quote = [
        quote
        for quote in raw_zen_quotes
        if int(quote["c"]) > median
    ][0]

    return {
        "median_q": median_quote,
        "short_q": short_quote,
        "long_q": long_quote,
    }
```

Multiple values can be returned in a dictionary to be stored as XComs.

2.3.3 Creating the formatted newsletter

After running the two-asset pipeline, Chris was ready to put everything together. First, he created the newsletter template shown in the following listing, with placeholders for the date and quotes; then he saved it in the `include` folder at `include/newsletter/newsletter_template.txt`.

Listing 2.4 The newsletter template text file

```
==========================================
      Daily Reality Tunnel {date}
==========================================

Hello Cosmic Traveler,

As you surf the probabilistic waves of existence today,
take these three quotes with you:

1. "{quote_text_1}" - {quote_author_1}
2. "{quote_text_2}" - {quote_author_2}
3. "{quote_text_3}" - {quote_author_3}

Have a fantastic journey!
AllThingsLookingUp Team

=====================================================
Consider also subscribing to our other newsletters:
"Weekly Dose of Happiness" and "Monthly Wonder".

"...reality is always plural and mutable." - R.A.W.
=====================================================
```

{date} will be templated by Airflow with the date of the pipeline run.

The quotes and authors will be templated by Airflow.

Now the only part left was to define the third asset in this asset pipeline, `formatted_newsletter`, to create the newsletter file in the same folder as the template (listing 2.5).

NOTE Chris uses the Airflow Object Storage abstraction to interact with files on his computer. This is a good choice ahead of moving this pipeline to production because he will be able to switch easily to cloud-based object storage (chapter 6). To use the Airflow Object Storage feature, install the Airflow Common IO provider (https://mng.bz/Wwel) by adding `apache-airflow-providers-common-io==<version>` to your `requirements.txt` file. Use the latest version available. Don't forget to restart your local Airflow environment with `astro dev restart` to make the changes take effect!

Listing 2.5 The asset populating the newsletter template

```
import os
from airflow.sdk import asset

OBJECT_STORAGE_SYSTEM = os.getenv(
    "OBJECT_STORAGE_SYSTEM", default="file"
)
OBJECT_STORAGE_CONN_ID = os.getenv(
    "OBJECT_STORAGE_CONN_ID", default=None
)
OBJECT_STORAGE_PATH_NEWSLETTER = os.getenv(
    "OBJECT_STORAGE_PATH_NEWSLETTER",
    default="include/newsletter",
)

@asset(
    schedule=[selected_quotes]
)
def formatted_newsletter(context: dict) -> None:
    """
    Formats the newsletter.
    """
    from airflow.sdk import ObjectStoragePath

    object_storage_path = ObjectStoragePath(
        f"{OBJECT_STORAGE_SYSTEM}://{OBJECT_STORAGE_PATH_NEWSLETTER}",
        conn_id=OBJECT_STORAGE_CONN_ID,
    )

    date = context["dag_run"].run_after.strftime(
        "%Y-%m-%d"
    )

    selected_quotes = context["ti"].xcom_pull(
        dag_id="selected_quotes",
        task_ids=["selected_quotes"],
        key="return_value",
        include_prior_dates=True,
    )

    newsletter_template_path = (
```

Imports the os package at the top level of the dag

Defines global variables at the top of the file

Materializes as soon as the selected_quotes asset receives an update

Passes the context to the function

Airflow Object Storage is used to interact with file storage.

Defines the object storage system and path to interact with

Fetches the run date of the dag run from the context; relevant for idempotency (see "Idempotency" sidebar)

Fetches the selected quotes returned by the upstream asset from the XCom table

```
    object_storage_path / "newsletter_template.txt"
)

newsletter_template = (
    newsletter_template_path.read_text()
)

newsletter = newsletter_template.format(
    quote_text_1=selected_quotes["short_q"]["q"],
    quote_author_1=selected_quotes["short_q"]["a"],
    quote_text_2=selected_quotes["median_q"]["q"],
    quote_author_2=selected_quotes["median_q"]["a"],
    quote_text_3=selected_quotes["long_q"]["q"],
    quote_author_3=selected_quotes["long_q"]["a"],
    date=date,
)

date_newsletter_path = (
    object_storage_path / f"{date}_newsletter.txt"
)

date_newsletter_path.write_text(newsletter)
```

Reads the text from the newsletter template

Formats the template text with today's quotes

Writes the full newsletter to a new file

This last asset in the pipeline introduces two new Airflow concepts: the Airflow context and Airflow Object Storage. The Airflow context (https://mng.bz/8Xy5) is a dictionary containing information about your Airflow environment and current dag run. One of the most common use cases is shown in listings 2.4 and 2.5, where the `ti` (task instance) is fetched from context to pull values from XCom. Another common use case is retrieving the date of the dag run from the context (listing 2.5) to be used in a filename, ensuring idempotency.

Idempotency

The concept of getting the same output for the same input is called *idempotency*, and making dags idempotent if possible is an Airflow best practice. The `formatted_newsletter` asset is idempotent because the date variable is tied to the `run_after` timestamp. It is important to use a date related to the context instead of `datetime.now()` because this way, if you rerun this specific asset materialization days or weeks later, the date of this specific newsletter stays the same. Defining Airflow workflows to be idempotent (same in = same out) is considered an Airflow best practice. The first asset that fetches the quotes (`raw_zen_quotes`) is not idempotent because each call to the ZenQuotes API serves a different list of quotes, making the pipeline as a whole not idempotent. This fact is a limitation of this example pipeline.

The second new concept is Airflow Object Storage (https://mng.bz/EwnX). This feature provides an abstraction to interact with different file storage systems.

When Chris moves this pipeline to production in chapter 6, he can easily switch the function to fetch the newsletter template from and save the templated newsletter in

cloud-based object storage. This task requires only providing different values for the environment variables: `OBJECT_STORAGE_SYSTEM`, `OBJECT_STORAGE_CONN_ID`, and `OBJECT_STORAGE_PATH_NEWSLETTER`; there's no need to change the function code. (See chapter 6 for more information on switching this asset to use Amazon S3 instead of local file storage.)

After finishing the third asset definition, Chris reran the first asset manually, watching all three assets materialize in order. The templated newsletter (listing 2.6) appeared in the `include/newsletter` folder.

NOTE The full code for Chris's pipeline is in the companion GitHub repository (https://github.com/astronomer/practical-guide-to-apache-airflow-3).

Listing 2.6 Example formatted newsletter

```
==========================================
      Daily Reality Tunnel 2025-04-22
==========================================

Hello Cosmic Traveler,

As you surf the probabilistic waves of existence today,
take these three quotes with you:

1. "Paths are made by walking." - Franz Kafka
2. "Nothing in life is to be feared,
it is only to be understood." - Marie Curie
3. "The future belongs to those who
believe in the beauty of their dreams." - Eleanor Roosevelt

Have a fantastic journey!
AllThingsLookingUp Team

========================================================
Consider also subscribing to our other newsletters:
"Weekly Dose of Happiness" and "Monthly Wonder".

"...reality is always plural and mutable." - R.A.W.
========================================================
```

Figure 2.6 shows the asset graph of the `formatted_newsletter` asset, showing its upstream dependencies and related dags, available on the Assets tab of the Airflow UI.

The pipeline was ready to run daily and create a newsletter automatically, fulfilling Jan's requirements. But Chris wasn't done yet. He wanted to show his new team the true power of Airflow, so he decided to take things to the next level and personalize the newsletter sent to each subscriber. Take a short break (you've earned it!), and then learn how Chris wrote a dag using the task-oriented approach to add personalization to the newsletter.

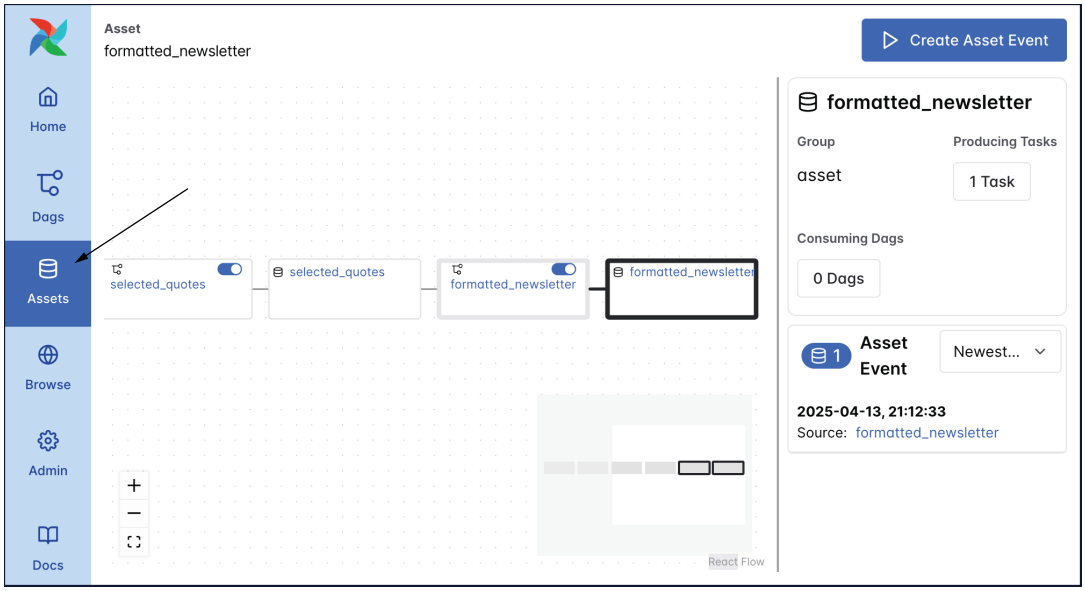


Figure 2.6 Screenshot of the Airflow UI showing the asset graph for the `formatted_newsletter` asset

2.4 Writing a task-oriented dag

On his second day, Chris logged on early, eager to improve his newsletter pipeline. The first step was checking what data *AllThingsLookingUp* had collected from loyal subscribers. Looking at the subscriber data (shown in the following listing), stored in one JSON file per subscriber, Chris was pleased to learn that in addition to the subscriber names, the data included their locations. This information could be used to create personalized weather reports.

Listing 2.7 Subscriber data example file

```
{
    "id": 2319,
    "name": "Heiner",
    "location": "Lipari",
    "motivation": "Seeing people finding their wings.",
    "favorite_sci_fi_character": "Janeway (Star Trek)"
}
```

Chris quickly added the necessary steps to the second part of the pipeline in his architecture diagram. He wanted this part to run as soon as the formatted newsletter (`formatted_newsletter` asset) was ready and for it to aggregate the information of each current subscriber, use their location to fetch the current weather from the Open-meteo API (https://open-meteo.com/en/docs), and then include that information

alongside the subscriber's name in the personalized newsletter. Figure 2.7 shows the updated pipeline diagram.

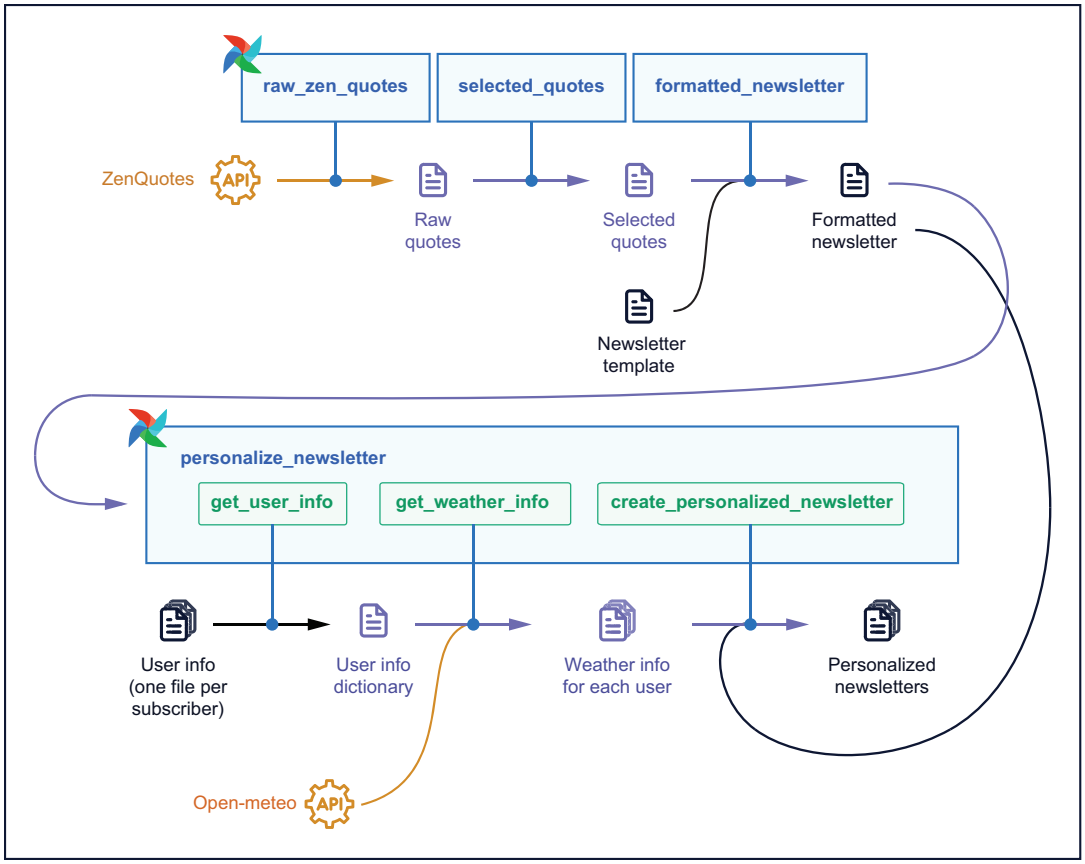


Figure 2.7 Architecture diagram of the newsletter pipeline. The added three-task dag runs after the `formatted_newsletter` asset is materialized, which adds the current weather for each user to personalize the newsletter.

2.4.1 Creating a dag

Although he could create this second part of the pipeline by using the asset-oriented way of writing dags, Chris wanted to sharpen his Airflow skills by learning how to write a dag using the task-oriented approach, especially because all Airflow 2 dags he might encounter in a future job would use the task-oriented approach. In a new Python file inside the `dags` folder, Chris added the following code to define an empty dag running with a data-aware schedule as soon as the newsletter template asset materializes.

Listing 2.8 Simple Airflow dag definition

```
from pendulum import datetime

from airflow.sdk import Asset, dag, task


@dag(
    start_date=datetime(2025, 3, 1),
    schedule=[Asset("formatted_newsletter")],
)
def personalize_newsletter():
    pass


personalize_newsletter()
```

Import of the dag and task decorators and the Asset class

The earliest point in time at which the dag can run

Runs as soon as the formatted_newsletter asset receives an update

pass will be replaced with task definitions in the next sections.

Don't forget to call the @dag decorated function.

> **NOTE** If you've used Airflow in the past, you're likely accustomed to setting `catchup=False` in the definition of each dag. In Airflow 3, the default for `catchup` was changed to `False` at the configuration level (`[scheduler].catchup_by_default`), which means you don't have to worry about the parameter anymore unless you explicitly want to enable catchup behavior. If your dags rely on catching up on runs by default, you can set the `[scheduler].catchup_by_default` configuration to `True` to replicate Airflow 2 behavior.

When using the task-oriented approach, you start by defining a dag with the `@dag` decorator and giving it a `start_date` (the earliest point in time the dag can run) and a `schedule`. You can schedule a dag in a large variety of ways, which we'll cover in chapter 3. This dag uses a data-aware schedule and runs whenever the `formatted_newsletter` asset receives an update. Next, we'll add several tasks to this dag.

> **NOTE** The `with DAG()` syntax is still valid, and you can run your existing dags using that syntax with Airflow 3. Whether you use `with DAG()` or `@dag` is a matter of personal preference.

2.4.2 Adding tasks to a dag

The dag in listing 2.8 is fully defined, but it does not contain any tasks yet. You can add tasks to dags by instantiating them within the `dag` function. Chris used the `@task` decorator to define the tasks in his dag to run custom Python code. The following listing shows the first task, `get_user_info`, which collects information about current newsletter subscribers from all JSON files in `include/user_data` and combines it into a list of dictionaries. As with `@asset` decorated functions, the value returned from the `@task` decorated function is pushed to XCom.

Listing 2.9 First task in the Airflow dag: Gathering user data

```
import os

from airflow.sdk import Asset, dag, task
from pendulum import datetime

OBJECT_STORAGE_SYSTEM = os.getenv(
    "OBJECT_STORAGE_SYSTEM", default="file"
)
OBJECT_STORAGE_CONN_ID = os.getenv(
    "OBJECT_STORAGE_CONN_ID", default=None
)
OBJECT_STORAGE_PATH_NEWSLETTER = os.getenv(
    "OBJECT_STORAGE_PATH_NEWSLETTER",
    default="include/newsletter",
)
OBJECT_STORAGE_PATH_USER_INFO = os.getenv(
    "OBJECT_STORAGE_PATH_USER_INFO",
    default="include/user_data",
)

@dag(
    start_date=datetime(2025, 3, 1),
    schedule=[Asset("formatted_newsletter")],
)
def personalize_newsletter():
    @task
    def get_user_info() -> list[dict]:
        import json

        from airflow.sdk import ObjectStoragePath

        object_storage_path = ObjectStoragePath(
            f"{OBJECT_STORAGE_SYSTEM}://"
            f"{OBJECT_STORAGE_PATH_USER_INFO}",
            conn_id=OBJECT_STORAGE_CONN_ID,
        )

        user_info = []
        for file in object_storage_path.iterdir():
            if file.is_file() and file.suffix == ".json":
                bytes = file.read_block(
                    offset=0, length=None
                )
                user_info.append(json.loads(bytes))

        return user_info

    _get_user_info = get_user_info()
```

Fetches environment variables about the object storage used

The @task decorator can turn any Python function into an Airflow task.

By default, the task ID is the name (get_user_info) of the decorated function.

Uses the Airflow Object Storage abstraction as in listing 2.5

Reads user info from JSON files

Pushes list of user info dictionaries to XCom

It is important to call the task function within the dag function to attach it to the dag. The output of the function can be assigned to a variable to be passed as input to downstream tasks.

After all current subscriber information was fetched, Chris needed a second task to make a call to the Open-meteo API to retrieve information about the current local weather for each subscriber. This action needed both the supporting code Chris added at the top of the dag file, namely the `_WEATHER_URL` and the `_get_lat_long` helper function, as well as a task looping over the list of user info dictionaries to gather information from the weather API (listing 2.10).

NOTE The `_get_lat_long` function uses the Nominatim API from the `geopy` package to get the latitude and longitude for the user-provided location. Install the package in your Airflow environment by adding `geopy==<version>` to the `requirements.txt` file. Don't forget to restart the project with `astro dev restart`. Note that in a production setting you would want to use an API that allows parallel calls instead.

Listing 2.10 Task to get weather information for each user

```
from airflow.sdk import task

_WEATHER_URL = (
    "https://api.open-meteo.com/v1/forecast?"
    "latitude={lat}&longitude={long}&current="
    "temperature_2m,relative_humidity_2m,"
    "apparent_temperature"
)

def _get_lat_long(location):
    import time

    from geopy.geocoders import Nominatim

    time.sleep(2)
    geolocator = Nominatim(
        user_agent="MyApp/1.0 (my_email@example.com)"
    )

    location = geolocator.geocode(location)

    return (
        float(location.latitude),
        float(location.longitude),
    )

# ... Dag definition and first task...

    @task
    def get_weather_info(
        users: list[dict],
    ) -> list[dict]:
        import requests

        user_info_plus_weather = []
```

The weather API URL with placeholders for the latitude and longitude values

Supporting function, typically defined in the include folder and imported

Delay to prevent rate limits using the free Nominatim API

Change to your email to comply with Nominatim's usage policy

Converts location string to lat/long

```
    for user in users:                                  # Loops through each dictionary in the users list
        lat, long = _get_lat_long(
            user["location"]                            # Calls the supporting function
        )
        r = requests.get(
            _WEATHER_URL.format(lat=lat, long=long)     # Calls the Open-meteo API
        )
        user["weather"] = r.json()
        user_info_plus_weather.append(user)             # Pushes the user and weather info to XCom

    return user_info_plus_weather

_get_weather_info = get_weather_info(                   # The get_weather_info task receives the user_info returned from the previous task via XCom.
    users=_get_user_info
)
```

For each user, the `get_weather_info` task in listing 2.10 converts the location string to a latitude and longitude value using the `_get_lat_long` helper function. Then the latitude and longitude values are templated into a simple call to the Open-meteo API to get the current temperature, relative humidity, and apparent temperature for the user's location.

> **NOTE** This task is another example of a task that is not idempotent; it always fetches the day's current weather, even when it is rerun a week later. To make this task idempotent, you'd need to implement logic to use the historical version of the API (https://mng.bz/Nwdv) for task reruns when the dag run's `run_after` time is in the past.

All the data is there. Only one more task is required to create a personalized newsletter for each subscriber.

Listing 2.11 Creating personalized newsletters

```
from airflow.sdk import Asset, task

# ... Dag definition and first two tasks...

    @task(
        outlets=[Asset("personalized_newsletters")]     # Updates the personalized_newsletters Airflow asset when the task completes successfully
    )
    def create_personalized_newsletter(
        user_info_plus_weather: list[dict], **context: dict
    ) -> None:
        from airflow.sdk import ObjectStoragePath

        date = context["dag_run"].run_after.strftime(   # Fetches the run date of the dag run from the context to match the newsletter template of the day
            "%Y-%m-%d"
        )

        for user in user_info_plus_weather:             # Iterates through the list with user info dictionaries
            id = user["id"]
```

```
        name = user["name"]
        location = user["location"]
        actual_temp = user["weather"]["current"][
            "temperature_2m"
        ]
        apparent_temp = user["weather"]["current"][
            "apparent_temperature"
        ]
        rel_humidity = user["weather"]["current"][
            "relative_humidity_2m"
        ]

        new_greeting = (
            f"Hi {name}! \n\nIf you venture outside right now "
            f"in {location}, you'll find the temperature to be "
            f"{actual_temp}°C, but it will feel more like "
            f"{apparent_temp}°C. The relative humidity is "
            f"{rel_humidity}%."
        )

        object_storage_path = ObjectStoragePath(
            f"{OBJECT_STORAGE_SYSTEM}://"
            f"{OBJECT_STORAGE_PATH_NEWSLETTER}",
            conn_id=OBJECT_STORAGE_CONN_ID,
        )

        daily_newsletter_path = (
            object_storage_path
            / f"{date}_newsletter.txt"
        )

        generic_content = (
            daily_newsletter_path.read_text()
        )

        personalized_content = (
            generic_content.replace(
                "Hello Cosmic Traveler,",
                new_greeting,
            )
        )

        personalized_newsletter_path = (
            object_storage_path
            / f"{date}_newsletter_userid_{id}.txt"
        )

        personalized_newsletter_path.write_text(
            personalized_content
        )

create_personalized_newsletter(
    user_info_plus_weather=_get_weather_info
)
```

Uses personalized information to create a new newsletter greeting

Gets the path to the newsletter folder

Reads the text from the newsletter template

Replaces the generic greeting with the personalized greeting

Writes the personalized newsletter to a new file

Calls the task function and provides the argument input to be pulled from XCom

This task iterates through the user information to create a personalized newsletter for each user. The generic greeting is replaced by a personalized one, including the name of the subscriber, as well as the current weather at their location. The task has its `outlets` parameter set to `Asset("personalized_newsletter")`. This means that every time this task completes successfully, the Airflow asset by that name receives an update. If he wanted to, Chris could schedule another pipeline to run as soon as this asset was updated.

> **NOTE** The full code for Chris's pipeline, including the newsletter dag he just built, is in the companion GitHub repository (https://github.com/astronomer/practical-guide-to-apache-airflow-3).

2.4.3 Setting task dependencies

Looking at the dag in the Airflow UI (figure 2.8), Chris was surprised to see that even though he hadn't set any dependencies between tasks, the dag graph reflected their correct order.

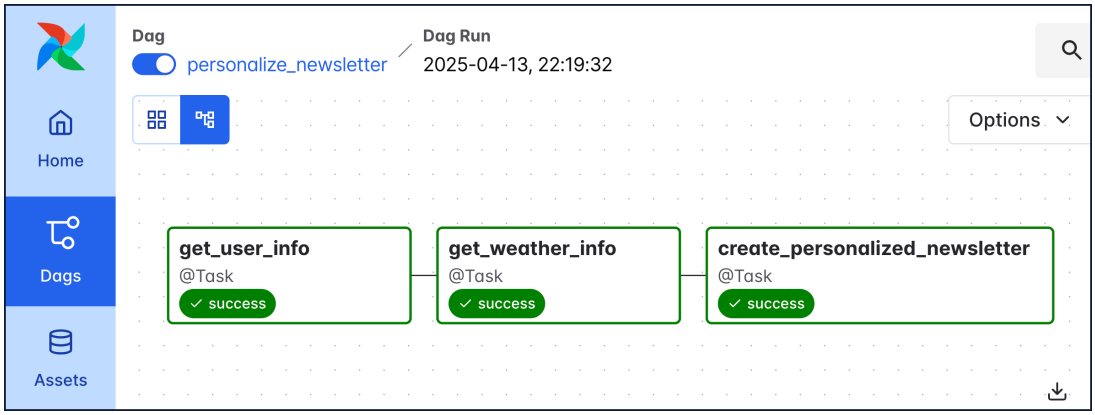


Figure 2.8 Graph of the `personalize_newsletter` dag in the Airflow UI

Taking a quick trip to the Astronomer Learn documentation (https://mng.bz/Dw8A), Chris found out why: when `@task` decorators are used, Airflow infers dependencies based on the task inputs. In Chris's pipeline, each task uses the output of its upstream task as an input, so he did not need to set any dependencies explicitly. Neat! In case he needed the information for the next pipeline, Chris made sure to jot down how to set explicit dependencies in Airflow in a small sample dag, shown in the following listing.

Listing 2.12 Setting explicit dependencies between tasks

```
from airflow.sdk import dag, task, chain
from airflow.providers.standard.operators.empty import (
    EmptyOperator,
)


@dag
def dependency_syntax():
    @task
    def my_task_1():
        pass

    _my_task_1 = my_task_1()

    @task
    def my_task_2():
        pass

    _my_task_2 = my_task_2()

    my_task_3 = EmptyOperator(task_id="my_task_3")
    my_task_4 = EmptyOperator(task_id="my_task_4")

    chain(
        _my_task_1,
        [_my_task_2, my_task_3],
        my_task_4,
    )

    chain(_my_task_1, my_task_4)


dependency_syntax()
```

A traditional operator that does nothing; used as a placeholder

Traditional operators need an explicit task_id.

Sets explicit dependencies

As long as you don't create a circular dependency, you can add as many chain functions as you like to structure your dag.

With the `chain` function, you can create dependencies between tasks and lists of tasks. Note that you cannot use `chain` to set a dependency between two lists of different lengths; `chain([t1,t2], [t3,t4,t5])` would cause an import error. Figure 2.9 shows the dag created by the code in listing 2.12.

> **TIP** If you are familiar with the bit-shift syntax of defining dependencies, you can think of `chain` as putting `>>` between its arguments. The bit-shift syntax is still valid in Airflow 3. Which dependency syntax you use is a matter of personal preference. You can learn more about different ways to set task dependencies in the Learn guide.

2.5 Running the pipeline

After spending a little too much time playing around building a complex-looking dag with `EmptyOperators` and `chain` functions, Chris turned his attention back to the

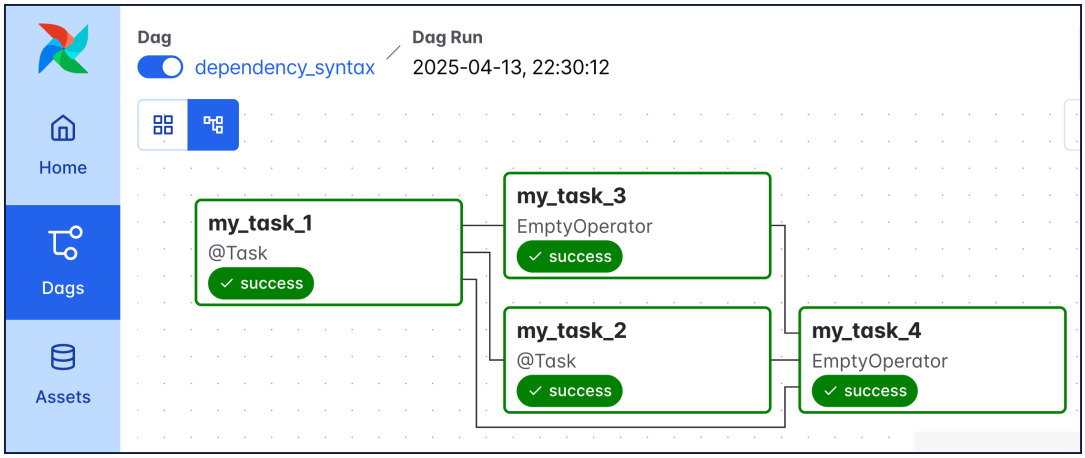


Figure 2.9 Dag graph of the `dependency_syntax` dag shown in listing 2.12

newsletter pipeline. He turned on the `personalize_newsletter` dag in the Airflow UI and started the pipeline manually from the first step by rematerializing the first asset of the asset-based workflow, `raw_zen_quotes`. Like dominos, all assets materialized in order, with the last asset materialization kicking off the dag. A few minutes later, Chris had the first batch of personalized newsletters ready to send to Jan.

Listing 2.13 Example personalized newsletter

```
==========================================
      Daily Reality Tunnel 2025-04-22
==========================================

Hi Heiner!

If you venture outside right now in Lipari,
you'll find the temperature to be 21.5°C,
but it will feel more like 22.0°C.
The relative humidity is 60%.

As you surf the probabilistic waves of existence today,
take these three quotes with you:

1. "Paths are made by walking." - Franz Kafka
2. "Nothing in life is to be feared,
it is only to be understood." - Marie Curie
3. "The future belongs to those who
believe in the beauty of their dreams." - Eleanor Roosevelt

Have a fantastic journey!
AllThingsLookingUp Team
```

```
=========================================================
Consider also subscribing to our other newsletters:
"Weekly Dose of Happiness" and "Monthly Wonder".

"...reality is always plural and mutable." - R.A.W.
=========================================================
```

Before sending the newsletter sample out for review, Chris took a moment to note what he'd learned about the Airflow building blocks (table 2.1).

Table 2.1 Comparison of Airflow building blocks

Building block	Function
`@asset`	The `@asset` decorator creates a dag with the decorated function name as the dag ID. The dag has one task with the ID name as the dag; it runs the decorated function and produces an update to the Airflow asset with the same name as the dag.
`@dag`	The `@dag` decorator defines a dag that you can schedule and fill with as many tasks as you want.
`@task`	The `@task` decorator turns any Python function into an Airflow task. Tasks can't be scheduled by themselves; they have to be part of a dag. Dependencies between tasks are defined using the `chain` function; in some cases, Airflow infers them automatically.
`Asset`	The `Asset` class defines an Airflow asset to be used in the schedule of a dag or in the outlets parameter of a task to turn any task into a producer task, producing updates to the asset whenever the task completes successfully.
Operators	There are many traditional Airflow operators, such as `EmptyOperator` (does nothing; it serves as a placeholder and structures dags), `PythonOperator` (traditional version of `@task`), `BashOperator` (runs a `bash_command`), and specialized operators like `SQLExecuteQueryOperator` (runs SQL statements). Many of these operators are part of Airflow provider packages (https://registry.astronomer.io).

2.6 Thinking about scaling

Looking at his work, Chris thought about how to modify and improve the pipeline when moving to production. All output files would need to be written to an accessible location—likely an object storage system like Amazon S3. Chris had already future-proofed his pipeline in this regard by using the Airflow Object Storage feature, so switching over to cloud-based storage would be easy (see chapter 6).

In terms of structural improvements, this first version of the pipeline fetches the latitude and longitude for each subscriber location every day. This information could be cached and refreshed only if the subscriber changed their location. Also, fetching the quotes is random, which is not idempotent (a rerun for the same day will likely return different quotes), and there is a risk of including the same quote two days in a row. Switching to an API that returns a static set of quotes for each date would address those problems. As mentioned earlier, the weather API call is not idempotent either. "Less perfectionism. There is such a thing as 'good enough' for now," Chris muttered, recalling performance reviews from his previous role as he put these items in his backlog.

He wanted to fix one aspect of the pipeline, though. The `get_weather_info` and `create_personalized_newsletter` tasks were processing data in a loop, meaning that if one iteration of the loop failed, the whole pipeline would stop. Realizing this, Chris decided that to be able to sleep at night, he needed to parallelize these tasks properly before deploying the pipeline. In short, pipeline resilience was not going to the backlog but directly to his to-do list for the next day.

Summary

- The Astro CLI is the easiest way to spin up a local, containerized Airflow development environment.
- In Airflow 3, you can write a pipeline using the `@asset` decorator. Each asset defined this way will create a dag under the hood with one task that runs the decorated function and updates an Airflow asset object. Dependencies between `@asset` defined dags are set using the asset's materialization schedule.
- Task-oriented workflows are written using the `@dag` decorator. A dag contains any number of tasks defined using `@task` and/or traditional Airflow operator classes.
- Dependencies between tasks are often implicit when passing data between `@task` decorated functions or can be defined explicitly by using the `chain` function.

3 Reliability and scheduling

This chapter covers

- Making your pipeline more observable and reliable with dynamic task mapping and retries
- Options for scheduling Airflow pipelines, including event-driven scheduling

"Loops are great for a prototype, but this first version is not reliable enough for production," Chris thought. Suppose that one subscriber's location had a typo, causing the API call fetching the coordinates to fail. The whole task would fail, and the pipeline would get stuck and have to be rerun manually, repeating the API calls for all subscribers. At scale, that could get expensive fast. Also, all newsletters would potentially be delayed instead of only the newsletter for the subscriber whose API call failed.

After brushing away the temptation to add `try/except` statements and watching a webinar on dag writing best practices, Chris had a plan: he would use dynamic task mapping instead of a loop to get the weather information for each subscriber and Airflow retries to protect his pipeline from transient API outages.

In this chapter, you'll learn to use the dynamic task mapping feature in Airflow to create a variable number of copies of one Airflow task, based on data determined at the run time of the dag. You'll also learn to set task retries, a core best practice for all Airflow pipelines.

After doing all that work to make the pipeline more robust, Chris sent a few sample newsletters to the content team for review, and while waiting for feedback, he read up on Airflow scheduling options. Although a combination of simple time-based and data-aware schedules worked well for this pipeline, Airflow has many more scheduling options to offer, and knowing them might come in handy down the road.

3.1 Dynamic task mapping

Dynamic task mapping is an Airflow feature that lets you write pipelines that adapt to your data at run time. A dynamically mapped task automatically creates a set of task copies to run in parallel based on a list of elements. One copy (a dynamically mapped task instance) is created per element in the list, even if the length of the list changes frequently.

This feature fits Chris's use case perfectly. He never knows how many current subscribers there are on any given day—only that he needs one task instance of the `get_weather_info` and `create_personalized_newsletter` tasks per subscriber. In the Astronomer Learn documentation (https://www.astronomer.io/docs/learn/dynamic-tasks), Chris found the blueprint code he needs, shown in the following listing.

Listing 3.1 Dynamic task mapping syntax

```
from airflow.sdk import dag, task


@dag
def simple_dynamic_task_mapping():
    @task
    def get_nums() -> list[int]:
        import random

        return [
            random.randint(1, 10)
            for _ in range(random.randint(3, 7))
        ]

    @task
    def times_a(a: int, num: int) -> int:
        return a * num

    @task
    def add_b(b: int, num: int) -> int:
        return b + num

    _get_nums = get_nums()

    _times_a = times_a.partial(
        a=2
```

Upstream task returns a list containing between three and seven integers

First downstream task multiplies each integer by a.

Second downstream task adds b to each multiplied integer.

Calls the upstream task and assigns the output to a variable

.partial includes the input that stays static for all dynamic task instances.

```
    ).expand(
        num=_get_nums
    )

    add_b.partial(
        b=10
    ).expand(
        num=_times_a
    )

simple_dynamic_task_mapping()
```

.expand includes the input that the times_a task dynamically maps over.

.partial includes the input that stays static for all dynamic task instances.

.expand includes the input that the add_b task dynamically maps over.

Listing 3.1 shows the syntax that dynamically maps two tasks in a sequential pattern. The upstream `get_nums` task returns a list of three to seven integers—`[3,7,8]`, for example. The length of this list varies with each dag run, which mimics Chris's pipeline that fetches a list of current subscribers every day. Ideally, the length of this list increases over time.

The second task (`times_a`) multiplies the integer by a constant (`a`). It is dynamically mapped over the `num` argument, which turns into one element in the list in the process of dynamic task mapping, creating three to seven copies (dynamically mapped task instances) of the `times_a` task.

The magical ingredient that turns a regular task into a dynamically mapped task is using the `.partial` and `.expand` methods when calling the task. `.partial` contains all arguments that stay the same for each dynamically mapped task instance and can contain as many arguments as you like. In this example, `.partial` sets the parameter `a` to `2` for each task instance. The `.expand` method contains the keyword argument that changes between tasks—in this example, the `num`.

For the input of `[3,7,8]`, this leads to three dynamically mapped task instances, each adding `2` to the input number. The first instance returns `5`; the second, `9`; and the third, `10`. In Chris's pipeline, this second task fetches the current weather for each subscriber; it should have one parallel task instance for each subscriber.

The third task, `add_b`, is dynamically mapped over the results of the `times_a` task. The return value of each dynamic task instance of the `times_a` task is put into a list (`[5,9,10]`), which is then passed to the `num` parameter of the `add_b` task. The parameter `b`, provided in the `.partial` method, stays the same for all dynamically mapped task instances—in our example, `b=10`. Meanwhile, the `num` parameter takes on a different value for each task instance, creating one dynamically mapped task instance of the `add_b` task per element in the list. The result, again, is three dynamically mapped task instances, this time with each multiplying the `num` by `10`. The first instance returns `50`; the second, `90`; and the third, `100`.

In Chris's newsletter pipeline, the second task that needs to be mapped dynamically is the downstream `create_personalized_newsletter` task, which runs one parallel task instance for each subscriber, using the output of the `get_weather_info` task. Figure 3.1 shows the result of running listing 3.1 in the Airflow UI.

Figure 3.1 Two dag runs of the dag shown in listing 3.1. The top screenshot shows a dag run in which the `times_a` and `add_b` tasks were mapped over a list containing six elements. The bottom screenshot shows a dag run where the list returned by `get_nums` contained three elements.

After experimenting with the template code for half an hour, Chris figured out how to apply dynamic task mapping to his dag, which didn't need many changes. For the `get_weather_info` task, he simply needed to change the code in the function to process one user info dictionary instead of looping over the full list of user info dictionaries. When calling the task function, he gave it its input through the `.expand` method. Because there was no static input to the task, Chris didn't need the `.partial` method.

The same was true of the `create_personalized_newsletter` task. The solution was a question of removing the loop and changing how the task was called.

Last, to respect the terms of use of the Nominatim API (https://mng.bz/64jZ), Chris specified that only one of the dynamically mapped task instances could run at any given time; he did so by setting `max_active_tis_per_dag` to `1`.

Listing 3.2 shows the adapted code for the `get_weather_info` task. You can find the fully adjusted dag, in which both the `get_weather_info` task and the `create`

`_personalized_newsletter` task are dynamically mapped, in the companion GitHub repository (https://github.com/astronomer/practical-guide-to-apache-airflow-3).

Listing 3.2 Dynamic task mapping of the `get_weather_info` task

```
@task(max_active_tis_per_dag=1)
def get_weather_info(user: dict) -> dict:
    import requests

    lat, long = _get_lat_long(user["location"])
    r = requests.get(
        _WEATHER_URL.format(lat=lat, long=long)
    )
    user["weather"] = r.json()

    return user

_get_weather_info = get_weather_info.expand(
    user=_get_user_info
)
```

Limits the number of parallel runs of the instances of this task

Inputs user:dict instead of users: list[dict]

No for loop is needed.

Returns the results for a single user

Dynamically maps over the list in _get_user_info

NOTE This section explains a simple implementation of dynamic task mapping. You can also map over sets of keyword arguments or even map whole groups of tasks over an input. Check out the Learn guide for a comprehensive explanation of dynamic task mapping (https://www.astronomer.io/docs/learn/dynamic-tasks).

TIP By default, the maximum number of dynamically mapped task instances of one task is `1024`. You can increase this number by setting the `[core].max_map_length` Airflow configuration.

3.2 Task retries

Chris remembered one more thing from the dag writing best practices webinar: you should always implement task retries (https://mng.bz/oZN2). Tasks that make calls to APIs are especially liable to fail occasionally, but often succeed when they try again after a couple of minutes. Chris decided to add two retries at three-minute intervals to all tasks in his dag, but he added four retries for each dynamically mapped task instance of the `get_weather_info` task. The next two listings show his code adjustments.

Listing 3.3 Setting retries for all tasks in a dag

```
from pendulum import datetime, duration
from airflow.sdk import dag

@dag(
```

```
    start_date=datetime(2025, 3, 1),
    schedule=[Asset("formatted_newsletter")],
    default_args={
        "retries": 2,
        "retry_delay": duration(minutes=3)
    }
)
```

default_args contains arguments applied to all tasks in a dag.

Gives each task in this dag two retries if it fails

Instructs each task to wait 3 minutes between retries

Listing 3.4 Overriding defaults for a specific task

```
from airflow.sdk import task

@task(
    retries=4
)
```

Gives each task instance of this task four retries if it fails

TIP Some users prefer to set a default number of retries at the configuration level for all tasks in their Airflow environment. You can do so by using the `[core].default_task_retries` config.

3.3 More scheduling options

While waiting for Jan to review a few sample personalized newsletters, Chris decided to look into other scheduling options. In the first part of his pipeline, he'd scheduled the first asset, `raw_zen_quotes`, using cron shorthand notation (`@daily`). The second asset, `selected_quotes`, was scheduled with data-aware scheduling to materialize as soon as the `raw_zen_quotes` asset was updated. The third asset, `formatted_newsletter`, used data-aware scheduling as well, so it would be materialized as soon as the `selected_quotes` asset was updated.

For the second part of the pipeline, Chris chose data-aware scheduling again, making the dag run as soon as the `formatted_newsletter` asset was updated by setting its schedule to `Asset("formatted_newsletter")`.

While reading the Astronomer scheduling guide (https://mng.bz/nZQe), Chris made a cheat sheet to reference later. This cheat sheet (table 3.1) contains all the options for scheduling Airflow pipelines.

Breaking change

In Airflow 2, providing a raw cron string like `"0 0 * * *"` or shorthand notation like `"@daily"` to the schedule parameter resulted in the schedule being instantiated using the `CronDataIntervalTimetable` under the hood. This timetable generates a `data_interval_start`, which used to be equivalent to the `logical_date` (previously `execution_date`), and `data_interval_end`, which is the point in time after which the dag runs (`run_after`).

Airflow 3 made a couple of changes to the default scheduling behavior (see chapter 8). Now the `logical_date` is always equal to the `run_after` date unless it is explicitly set to `None`.

Also, by default, raw cron strings are now interpreted using the `CronTriggerTimetable`. The `CronTriggerTimetable` does not include a data interval, which means that `data_interval_start`, `data_interval_end`, and `logical_date` are all the same—the point in time after which the dag is allowed to run (such as the `run_after` timestamp.)

If your dags depend on the data interval for their internal data partitioning logic, you can set the default timetable used by cron schedules back to `CronDataIntervalTimetable`, using the `[scheduler].create_cron_data_intervals` configuration (https://mng.bz/X7g1).

Table 3.1 Options for scheduling Airflow 3 dags

Name	Explanation	Example
Data-aware scheduling	*Data-aware scheduling* refers to scheduling a dag based on updates to one or more assets. You can use a data-aware schedule in dags written using both the task-oriented approach and the asset-oriented approach. In basic data-aware scheduling, a dag runs based on updates to one singular asset. This is the type of scheduling you used in chapter 2. *Advanced data-aware scheduling* refers to using conditional asset schedules and/or combining time and asset schedules. A special case of data-aware scheduling is event-driven scheduling (discussed later in this table). Data-aware scheduling enables users to create data-driven dependencies between tasks, assets, and dags. Every dag runs as soon as the data it depends on is ready, unlike a time-based schedule like cron. Use when your dag should run based on updates to a (set of) Airflow asset(s) in your Airflow environment(s). Updates to assets can occur when tasks complete successfully, when manual updates are created in the Airflow UI, or via a call to the Airflow `REST` API.	 See section 3.3.1. See section 3.3.1.
Event-driven scheduling	Event-driven scheduling, which is new in Airflow 3, allows users to schedule pipelines whenever a message appears in a message queue like Amazon SQS. It enables users to trigger a dag run at any time, dependent on events external to Airflow that post a message in a message queue. Note that the term *event-driven scheduling* refers to the use of an `AssetWatcher` when defining an `Asset`. You can view it as a subset of data-aware scheduling. Use when your dag should run based on detecting a message in a message queue (poll pattern).	See section 3.3.2.
Trigger via API	Airflow dags can be triggered from within any external system by sending a `POST` request to the Airflow `REST` API. Use when your dag should run based on an event in an external system using a push pattern.	See the Airflow `REST` API documentation (https://mng.bz/nZve).

Table 3.1 Options for scheduling Airflow 3 dags (*continued*)

Name	Explanation	Example

cron	Cron is a simple time-based job scheduling package for use directly in the command-line interface (CLI). A good resource for writing cron schedules is crontab guru (https://crontab.guru).
	
	For many common cron schedules, Airflow has built-in shorthand notation. To run a dag every day at midnight, for example, you can use `"@daily"` instead of `"0 0 * * *"`. For a full list, see the Airflow documentation (https://mng.bz/vZN1).
	
	Use when your dag should run regularly at the same point(s) in time. For more complex point-in-time schedules that can't be represented by a cron string, use timetables (discussed later in this table).
		`"0 0 * * *"`
		
		`"@daily"`

Timedelta	A timedelta schedule runs a dag on a regular cadence, such as every 35 minutes and 30 seconds, after it is first activated. You implement it by providing a `timedelta` object to the `schedule` parameter and specifying the desired cadence. For more information on the `timedelta` object, see the Python documentation (https://mng.bz/4nOD). You can also use the `duration` helper from the `pendulum` package (https://pendulum.eustace.io/docs/#duration). Use when your dag should run regularly every X time period after it is activated.	`timedelta(minutes=35, seconds=30)`
Timetable	*Timetables* are Airflow classes that can achieve more-complex time-based schedules. Airflow comes with a couple of built-in timetables, and users can define their own. Use when your dag should run on a time-based schedule that cannot be represented by a cron or timedelta schedule, such as running a dag every day at 6 a.m. and 4:30 p.m.	See the Airflow documentation (https://mng.bz/QwQR).
Run continuously	The `ContinuousTimetable` starts a new dag run as soon as the old one finishes. Aside from defining the schedule parameter, you need to set `max_active_runs=1` when using this timetable. Use when your dag should always have one active run. Caution: This feature is not intended for streaming processes.	`"@continuous"`

3.3.1 Data-aware scheduling

Chapter 2 contains several examples of simple data-aware scheduling in which one dag, be it written in a task-oriented or asset-oriented way, is scheduled based on updates to one Airflow asset. These updates can occur in different ways:

- A dag created with `@asset` runs successfully. By default, an asset with the same name as the dag is created and updated.
- Any Airflow task can update an asset by completing successfully if that asset is provided to the task's `outlets` parameter (see listing 3.5). Several tasks can produce updates to an asset of the same name.

- Any asset can be updated manually in the Airflow UI (see figure 4.7 in chapter 4).
- Any asset can be updated through a call to the Airflow `REST` API (https://mng.bz/yN1o).

When you update an asset through the Airflow UI or `REST` API, you have two options:

- *Creating an asset event*—This option produces an update to the Airflow asset object without running any underlying code. Choose this option if you care only about downstream dependencies, such as to test whether your data-aware schedule works as intended.
- *Materializing an asset*—This option runs the full dag containing the task that produces the update to the said asset. (Materializing assets is new in Airflow 3.)

Listing 3.5 shows the simplest implementation of a data-aware schedule. The upstream dag (`simple_asset_schedule_upstream`) contains one task, `my_producer_task`, which updates the asset named `simple_asset` whenever it completes successfully. The downstream dag uses data-aware scheduling to run as soon as the `simple_asset` receives an update, whether because the `my_producer_task` completed successfully or because the asset was updated in any of the preceding ways.

Listing 3.5 Simple asset schedule

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


@dag
def simple_asset_schedule_upstream():
    @task(outlets=[Asset("simple_asset")])
    def my_producer_task():
        pass

    my_producer_task()


simple_asset_schedule_upstream()


@dag(
    schedule=[Asset("simple_asset")]
)
def simple_asset_schedule_downstream():
    @task
    def my_task():
        pass

    my_task()


simple_asset_schedule_downstream()
```

Updates the asset

Runs as soon as the asset is updated

Often, a dag depends on more than one asset being updated. In these cases, you can use the conditional operators `&` (AND) and `|` (OR) to create a conditional asset schedule. The following listing shows a dag that is scheduled to run as soon as the asset `a` and either asset `b` or `c` have been updated.

Listing 3.6 Conditional asset schedule

```
from airflow.sdk import Asset, dag
from pendulum import datetime


@dag(
    start_date=datetime(2025, 1, 1),
    schedule=(
        Asset("a") & (
            Asset("b") | Asset("c")
        )
    )
)
```

Schedule based on Asset("a") being updated AND

on Asset("b") OR Asset("c") having received an update

Another common situation is wanting a dag to run on both a time-based and a data-aware schedule. Such a dag might need to run whenever two specific assets receive an update, but it needs to run at least once a day even if these two assets were not updated. You can achieve this result by using `AssetOrTimeSchedule`. The next listing shows a dag that runs every day at midnight, as well as whenever `Asset("a")` and `Asset("b")` have been updated.

Listing 3.7 `AssetOrTimeSchedule`

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

@dag(
    start_date=datetime(2025, 1, 1),
    schedule=AssetOrTimeSchedule(
        timetable=CronTriggerTimetable("0 0 * * *", timezone="UTC"),
        assets=(Asset("a") & Asset("b"))
    )
)
```

Provides the CronTriggerTimetable to the timetable argument

Provides a conditional asset expression to the assets argument

3.3.2 *Event-driven scheduling*

Basic and advanced data-aware scheduling are great for all use cases in which updates to assets occur from within Airflow, but sometimes, you need a dag to run based on events in external systems. Figure 3.2 shows two common situations:

- Data is being delivered to an external system, such as manually by a domain expert, and a data-ready event is sent to a message queue like Amazon SQS. The

dag in Airflow is scheduled based on this message event and runs an extract-transform-load (ETL) pipeline working on the data in the external system.

- 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. This evaluation can be conducted using a machine learning (ML) model. If the evaluation determines that an alert is warranted, an alert event is published to another message queue.

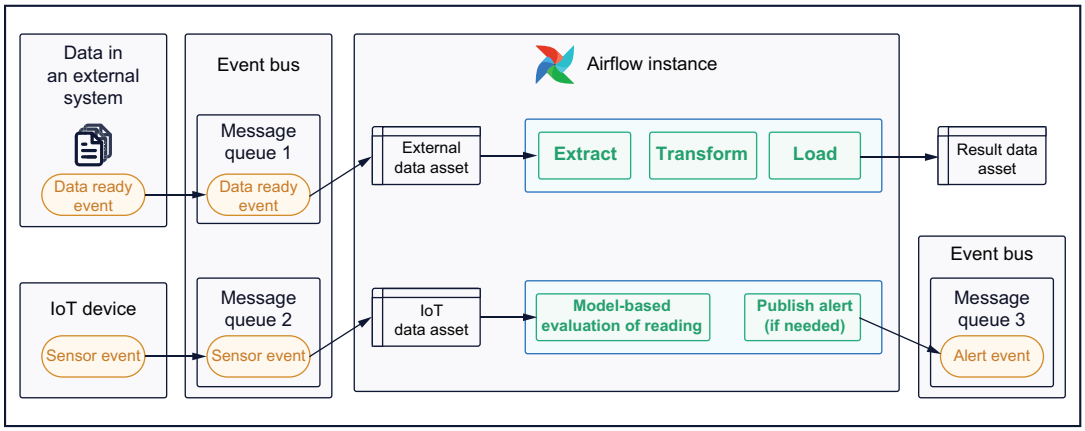


Figure 3.2 Diagram showing two common use cases of event-driven scheduling with a message queue

In Airflow 3, you can achieve these types of event-driven schedules by using assets in combination with `AssetWatcher`s. The `AssetWatcher` uses a trigger class that inherits from the `BaseEventTrigger` class to continuously check whether the message queue has received any new messages. Triggers run in the Airflow triggerer component (see chapter 5) as an asynchronous Python function, so they do not take up any worker slots. The following listing shows the syntax for using an `AssetWatcher` with a `MessageQueueTrigger`.

Listing 3.8 `AssetWatcher` syntax

```
from airflow.sdk import Asset, AssetWatcher, dag
from airflow.providers.common.messaging.triggers.msg_queue import (
    MessageQueueTrigger
)


trigger = MessageQueueTrigger(
    queue=<your-queue-url>,
    aws_conn_id="my_aws_conn",
    waiter_delay=20
```

The queue in which a message appears, triggering the dag

Specifies the connection ID used

Sets the wait time in between polls to 20 seconds

```
)

my_asset = Asset(
    "my_queue_asset",
    watchers=[
        AssetWatcher(
            name="my_queue_watcher",
            trigger=trigger
        )
    ]
)

@dag(
    schedule=[my_asset]
)
def my_dag():
```

Name of the asset in Airflow

Provides the trigger to the AssetWatcher

Schedules the dag based on this event-driven asset

TIP To interact with message queues using a common abstraction (similar to the way the Common IO provider in chapter 2 provides a common abstraction for object storage systems), a Common Messaging provider exists (https://mng.bz/vZl1). To use the code in listing 3.8, make sure to install the provider in your Airflow environment by adding `apache-airflow-providers-common-messaging==<version>` (insert the desired version) to your `requirement.txt` file. Make sure that you also provide the relevant Airflow connection (chapter 4). To use the provider with Amazon SQS, as shown in listing 3.8, provide a connection with the connection ID `my_aws_conn`. You will learn how to define Airflow connections in the next chapter.

You can find out more about event-driven scheduling in the Learn documentation (https://mng.bz/4neD).

Sensors vs. deferrable operators vs. AssetWatchers

Two options already present in Airflow poll for an event in an external system: sensors and deferrable operators.

A *sensor* is a task that polls for an external event using a synchronous Python function while occupying a worker slot. There are prebuilt sensors for many external systems, such as Amazon S3 and Amazon SageMaker, and users can create their own custom sensors using the `@task.sensor` decorator or `PythonSensor` traditional operator. See the Airflow sensors guide (https://mng.bz/MwxD) for more information and example code.

A *deferrable operator* uses an asynchronous trigger class to poll for an event in an external system. These triggers run in the triggerer component (see chapter 5) without taking up a worker slot. It is a best practice to use deferrable operators instead of regular sensors whenever possible for long-running tasks to maximize worker efficiency. Many sensors have a `deferrable` parameter that turns them into deferrable operators. See the deferrable operators guide (https://mng.bz/a979) for more information and example code.

`AssetWatcher`s are similar to deferrable operators insofar as they also use trigger classes, but only triggers inheriting from the `BaseEventTrigger` class are compatible with `AssetWatcher`s. The other notable difference is that `AssetWatcher`s always poll for their external event in the background of your Airflow environment without any dag or task running, unlike deferrable operators, which create regular tasks in a dag run.

To decide which method to use to wait for an event in an external system, evaluate whether you need a dag to run based on messages in a message queue. If so, choose `AssetWatcher`s. If not, use a deferrable operator/sensor in deferrable mode in a dag. Use regular sensors only if you expect the wait time to be short or if no deferrable option is available.

"Interesting," Chris thought, "I wonder whether I could use event-driven scheduling to turn this newsletter pipeline into an on-demand quote-and-weather service on the *AllThingsLookingUp* website." Then he reminded himself, "That is an item for the backlog. First, I need to finish the work I was tasked with."

DEFINITION A key use case in which event-driven scheduling plays a role is *inference execution*, which involves running a pipeline to make a call to an ML model to get a response. See chapter 7 for more information.

One more time, Chris double-checked whether his pipeline was ready to perform a scheduled dry run tonight. The dry-run newsletters would not be sent out to subscribers yet, of course, but an end-to-end test still seemed to be a good idea. At 6:45 p.m., just when Chris confirmed that all the relevant dags were activated, Häxli, his Appenzeller dog, decided that it was time to call it a day. "What a third day on the job!" Chris thought as he closed his laptop and got ready for a long, relaxing walk along the river.

Summary

- Dynamic task mapping is an Airflow feature that allows you to create a changing number of parallel runs of the same task with different input parameters based on a list of elements. This way, your dag can adapt to your data at run time. If you have a loop in any of your tasks, consider dynamically mapping the task instead.
- Task retries ensure that your tasks get another chance if they fail, perhaps because an API is down or a rate limit is reached. You can set them at the dag or the task level.
- You have many options for scheduling workflows in Airflow:
  - *Data-aware scheduling*—Run a dag based on updates to one or more assets. You can use a data-aware schedule in dags written using both the task-oriented and the asset-oriented approach. Aside from simple data-aware schedules involving only one asset, you can also schedule a dag based on conditional combinations of assets and combine time- and data-aware scheduling.

- *Event-driven scheduling*—Run dags based on detecting a message in a message queue service.
- API—Run dags using a `REST` API `POST` request.
- *Cron*—Run dags regularly at the same point(s) in time.
- *Timedelta*—Run dags regularly every X time periods.
- *Timetable*—Run dags on a time-based schedule that can't be represented by cron.
- *Continuous*—Ensure that a dag always has one active run.

4 UI and dag versioning

This chapter covers

- Navigating the Airflow 3 UI
- Using Airflow connections to connect Airflow to other tools in your data ecosystem
- Tracking edits to a dag using dag versioning and dag bundles
- Backfilling a dag from the Airflow UI

The next morning, Chris checked the dags and logs in the Airflow UI and confirmed that the first scheduled dry run of the *Daily Reality Tunnel* newsletter pipeline had been a success. Still waiting for feedback on the newsletters from Jan, the head of content, he had some time on his hands to explore the Airflow UI, including failing a task on purpose to find the quickest way to get to the logs of a failed task. (Planning never hurts.)

After seeing a companywide email encouraging responsible use of generative AI (GenAI) tools, Chris had an idea about how he could impress his new team (especially the CEO): he could add a task to the dag that used a large language model (LLM) to generate a unique personalized quote for each subscriber. Adding that

task would constitute a new version of this dag. Chris was pleased to find that he could see how his dag evolved over time in the Airflow UI—thanks to dag versioning, added in Airflow 3.

Finally, tech lead Michelle was curious about whether you could bulk-run a dag for several dates in the past, which led her and Chris to discover backfills.

In this chapter, you'll learn how to navigate the Airflow UI and connect Airflow to OpenAI to interact with an LLM. You'll see how dag versioning and dag bundles work in Airflow 3. Also, you'll backfill a dag directly from within the Airflow UI.

4.1 Exploring the UI

Still waiting for Jan's review, Chris set out to explore the Airflow 3 UI. Clicking all the buttons was his favorite way to discover hidden feature gems. Having seen a post on LinkedIn about dark mode in Airflow 3, that setting was the first one he wanted to check out. Figure 4.1 shows how you can switch between light and dark modes in the Airflow UI: click the User button, and then click Switch to Dark Mode (arrow).

> **NOTE** The Airflow UI receives frequent updates with minor releases. Depending on the version you're using when you read this book, your Airflow UI might look slightly different from what you see in the screenshots. Also, most screenshots in this book have been zoomed in for better readability.

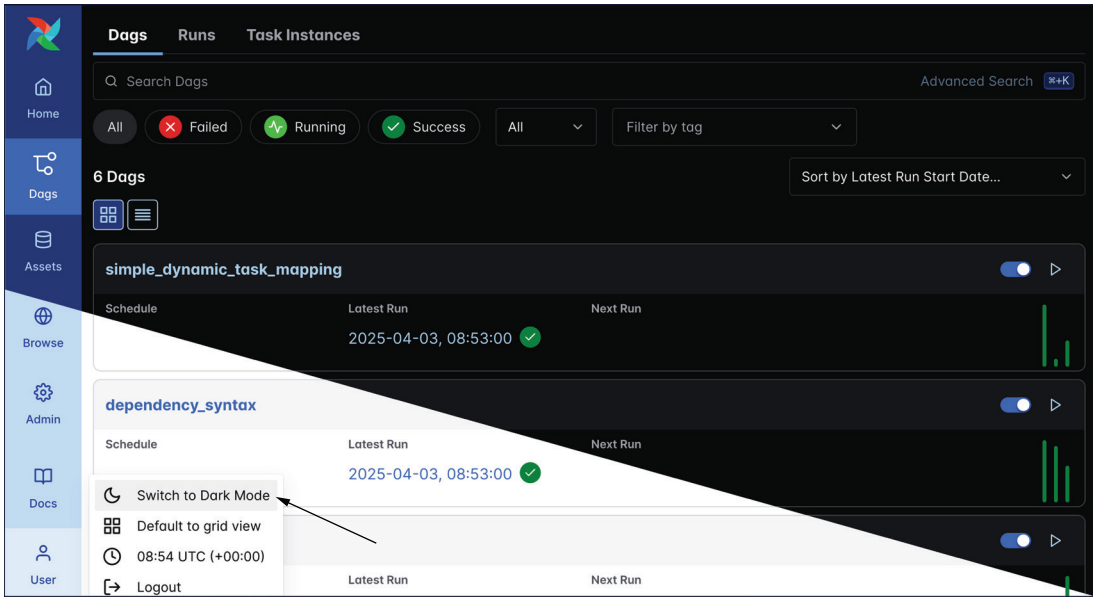


Figure 4.1 Screenshot of the Airflow UI with the top in dark mode and the bottom in light mode. The arrow shows how to switch between modes.

Dark mode looks awesome! But for this book, we'll stick with the easier-to-read light mode.

Next, Chris decided to explore the sidebar buttons from top to bottom, starting with Home. The Airflow UI home page (figure 4.2) provides an overview of Airflow component health (you'll learn more about Airflow components in chapter 5), recent dag runs, task runs, and asset updates. It also has quick links to lists of recently failed, currently running, and active dags in the Airflow instance.

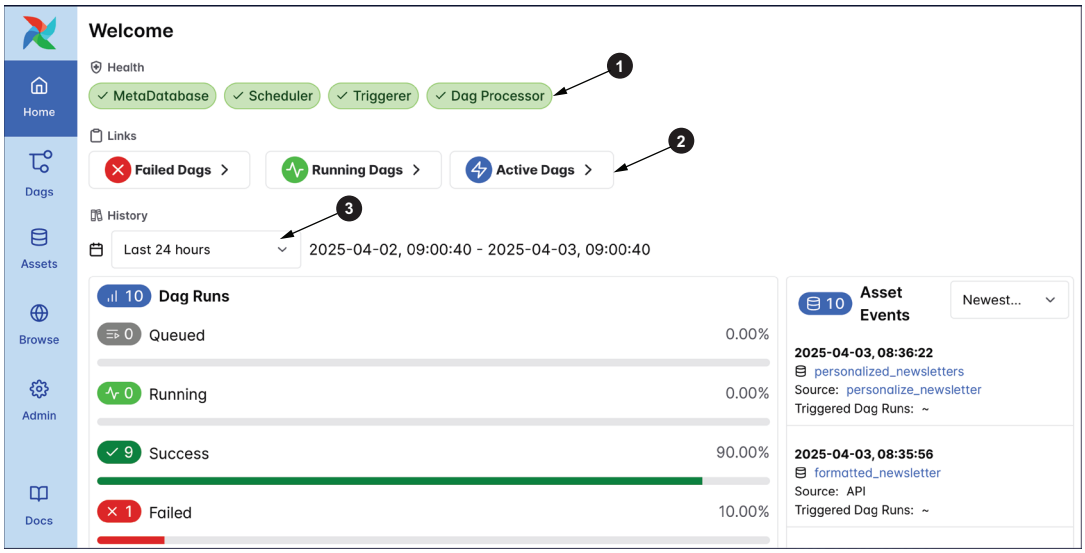


Figure 4.2 Home page. This page displays the health status of your Airflow components (1), quick links to lists of dags (2), and the recent history of dag runs, task instances, and asset events. You can adjust the time period for which to show history by making a choice from the drop-down History menu (3).

The next button on the sidebar opens the Dags page (figure 4.3), with which Chris became quite familiar while building his newsletter pipeline. This page displays an overview of dags in your Airflow environment, their schedule, and their latest runs. You can activate and deactivate dags by using the toggle switches and create manual runs as well. Note that you can use the search bar to search for any dag by name—a handy feature if you have hundreds of dags in an instance.

> **NOTE** Assets created with the `@asset` decorator create a dag with one task under the hood. These dags are listed on the Dags page as well. You can interact with them the same way you do with dags built using the task-oriented approach.

"All green. That is very satisfying," Chris thought. "But what happens if a task fails? It is bound to happen eventually. I should test this." Chris opened his dependency toy dag

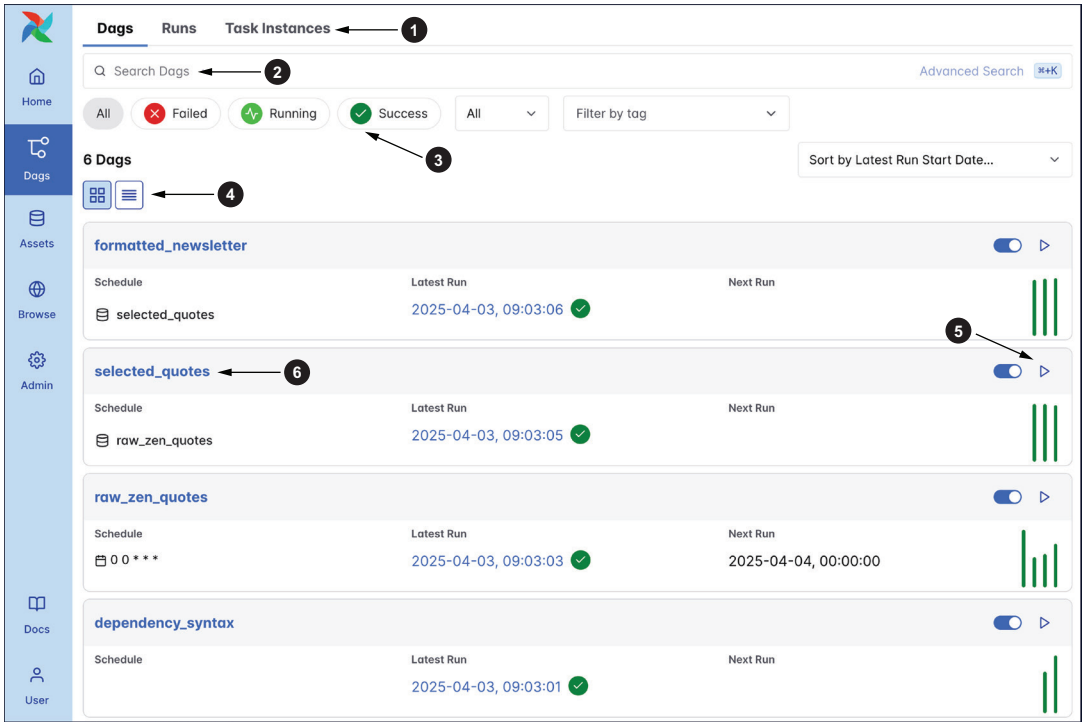


Figure 4.3 Dags page. You can search for dags by name (2), quickly filter for dags with a specific status or tag (3), toggle between expanded view (shown) and compact view (4), and toggle dag activation and create manual runs (5). The Dags page also has lists of dag runs and task instances (1) from which you can quickly rerun a dag run or task instance. Click any dag name to get to its overview page (6).

from chapter 2 and added one line to the `my_task_1` task: `print(10/0)`. That should cause the next run of this dag to fail.

He created a manual run of the dag, and as expected, the dag run failed. Figure 4.4 shows the failed run on the Dags page (shown in compact view).

"If this were a real failure, I'd want to find the logs of the failed task," Chris thought as he clicked the `dependency_syntax` dag. To his delight, error messages for recently failed tasks surfaced right on the dag overview page (figure 4.5).

The dag grid on the left side of the page is a great way to navigate to the full logs of previous task instances, regardless of their state. Click the relevant square, and the logs open on the right side of the screen (figure 4.6).

The third button on the sidebar, Assets, was one that Chris used to view the assets graph of the first part of his pipeline (chapter 2). Now, with a moment to spare, he took a look at an asset graph and a dag graph side by side.

When viewing an asset graph (figure 4.7), you can see dependencies between assets and dags. If a dag is scheduled based on updates to an asset, that asset is shown to the left of the dag. Assets that get updated by tasks in a dag are shown to the right of that dag.

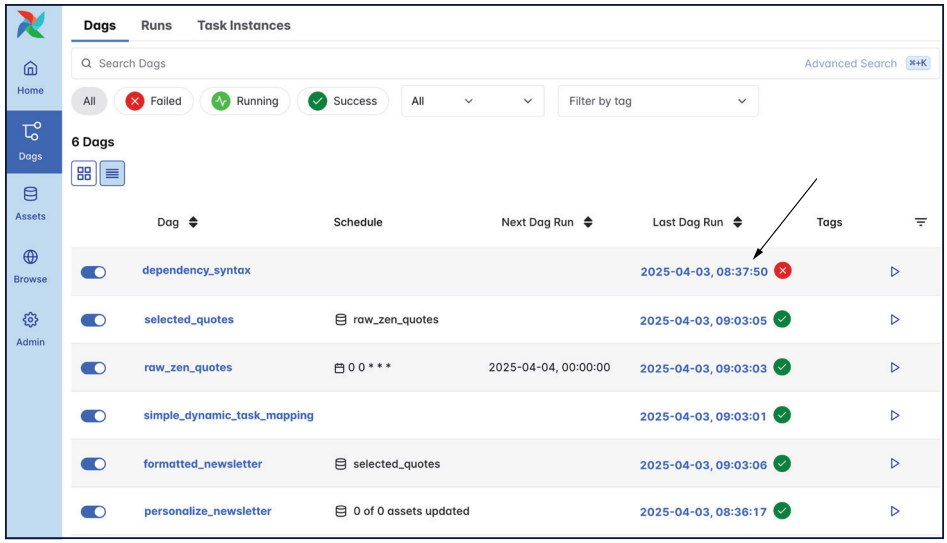


Figure 4.4 Dags page in compact view, showing one dag where the latest dag run failed

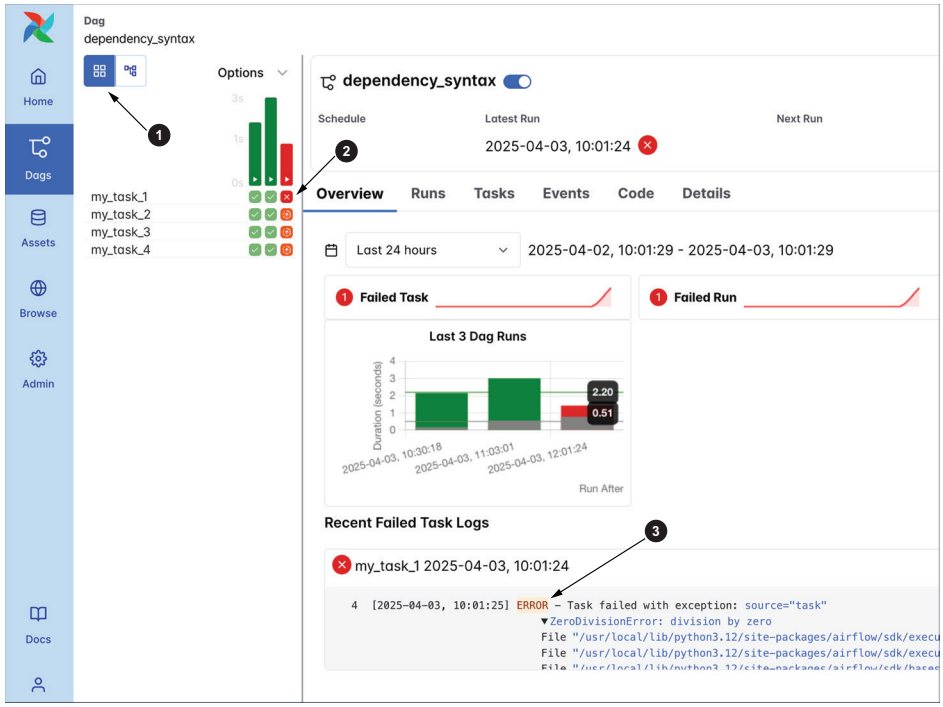


Figure 4.5 Dag overview page showing three dag runs, with the last run having failed its first task (2). The error lines of the logs of recently failed tasks are at the bottom of the screen (3). In this view, you can toggle between the dag graph and the dag grid (1).

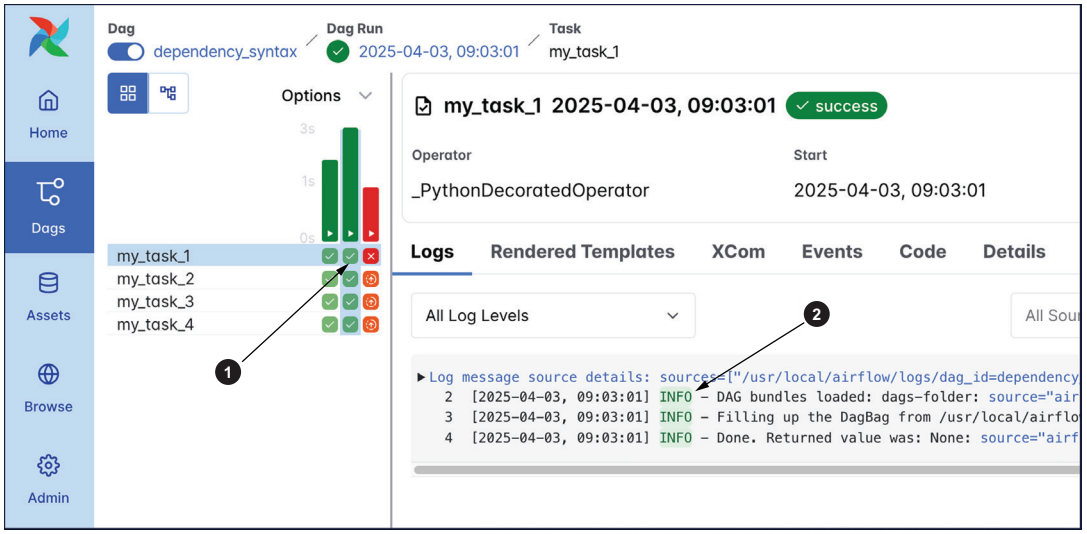


Figure 4.6 Task instance details. You can navigate to this view by clicking the square representing the task instance in the dag grid (1) to get to the logs quickly (2).

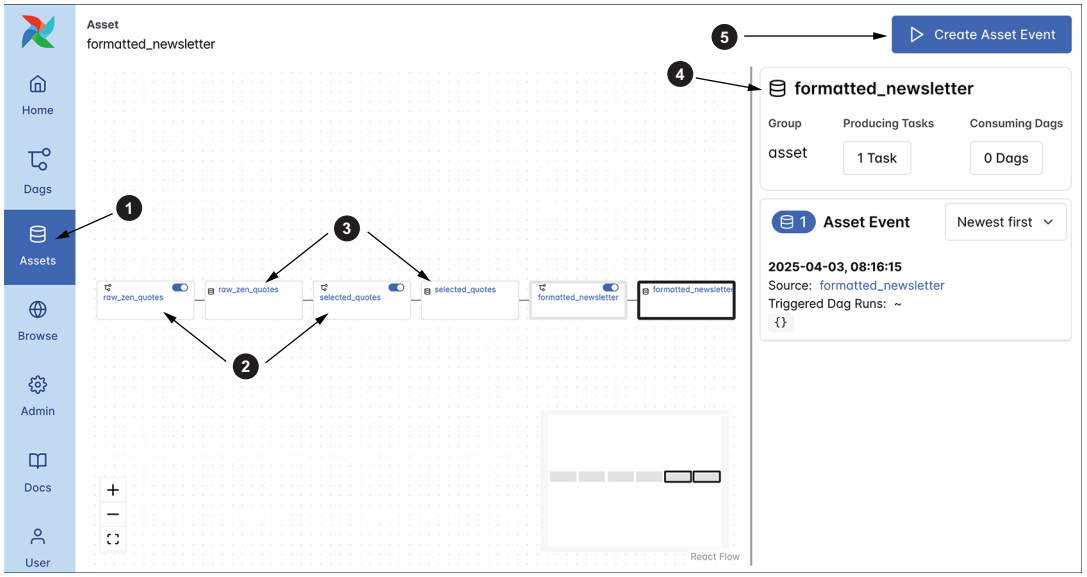


Figure 4.7 After clicking the Assets button (1) and the asset name, you see the graph showing dependencies between dags (2) and assets (3). On the right, you see details about the selected asset (4) and have the option to create asset events manually (5).

In the asset graph (figure 4.7), each dag is collapsed to one tile; you can't see or interact with individual tasks. By contrast, the dag graph shows individual tasks and their relationships, and you have the option to display related assets, selectable in the Options menu in the top-right corner of the dag graph (figure 4.8). The asset graph is a good place to get a high-level overview from a data-oriented mindset; the dag graph gives you more details about what is happening inside individual dags.

Figure 4.8 Dag graph, showing individual tasks in a dag. You can use the Options menu (1) to select which dag run to display (2), to determine which dependencies to include (3), and to switch the direction of the graph (4) to display tasks from top to bottom instead of left to right, for example.

4.2 Planning pipeline edits

Halfway through exploring the Airflow UI, Chris got an email from Ronald, the CEO of *AllThingsLookingUp*. After sharing the latest news about the rising numbers of subscribers and regular readers (and, of course, the resulting increase in ad revenue), he stressed the importance of smart adoption of GenAI. "Use it for first drafts," Ronald wrote, "but please always fact-check, and never enter sensitive information. Trust and integrity are our most valuable assets these days."

Chris was glad that he was unlikely to be asked to make a pipeline hallucinating news. But this email gave him an idea: he could use GenAI to create unique personalized quotes for each subscriber. To do this, he needed an input to the AI that was unique and personal to each subscriber.

Checking the subscriber data again, Chris saw that most subscribers submitted their greatest motivators and their favorite sci-fi characters with their entries to a raffle last year. In the following listing, you see how the available data is structured. In chapter 2, Chris used the ID, name, and location of each subscriber to personalize the newsletter. It was time to use the other two pieces of information.

Listing 4.1 Subscriber data example file

```
{
    "id": 2319,
    "name": "Heiner",
    "location": "Lipari",
    "motivation": "Helping people finding their wings.",
    "favorite_sci_fi_character": "Janeway (Star Trek)"
}
```

Chris thought he could work with that data. Using the pipeline architecture diagram he drafted in chapter 2, he easily figured out where to add a task that would use this information and the daily quotes to create a unique personalized quote for each user. Figure 4.9 shows the updated pipeline architecture diagram, now including a task that makes a call to the OpenAI API.

The first part of the pipeline could stay the same: three assets, dependent on one another, formed an extract-transform-load (ETL) pattern, extracting quotes from the ZenQuotes API, transforming them, and loading them into the newsletter template to create the generic newsletter of that day.

Chris decided to put the additional action—a call to an LLM hosted by OpenAI to generate a personalized quote—in the second part of the pipeline, the `personalize_newsletter` dag. He could use the same pattern for the personalized quote that he used to get the weather report for each user: dynamically map a task over the output of the `get_user_info` task. After that, he simply needed to add a helper task matching the two personalized elements—the weather information and personalized quote—based on the user ID. The last task in the pipeline needed only a small code adjustment to add the personalized quote to the newsletter.

This pipeline addition is very doable. Let's get started!

4.3 Using OpenAI to generate a quote

With the plan sketched out, Chris thought about the steps he would take to generate a personalized quote using OpenAI:

1. Create a connection between Airflow and OpenAI.
2. Craft the system prompt.
3. Add a task to the dag that uses the system prompt and the generic quotes of the day alongside the user's favorite sci-fi character and motivation to create a personalized quote.
4. Dynamically map this task to run one task instance per subscriber in parallel.

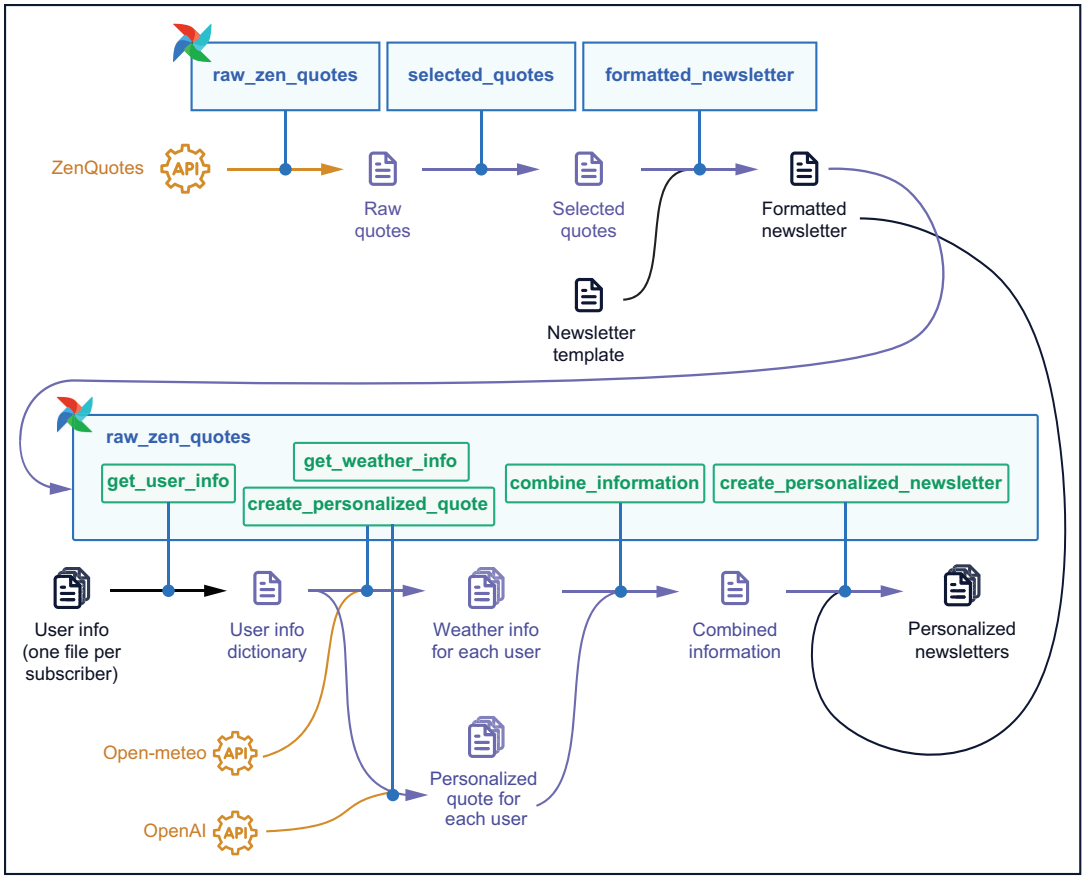


Figure 4.9 Pipeline architecture diagram, showing Chris's newsletter pipeline with an added task to create a personalized quote using GenAI

NOTE This section requires an OpenAI API key, which you can generate on the OpenAI developer platform (https://platform.openai.com). Depending on the current rate limits, you may need to pay a small fee to run the `create_personalized_quote` task. Alternatively, use any other LLM service to generate the personalized quote by adapting the code in the `create_personalized_quote` task to connect to your preferred service and model. This flexibility is one of the many reasons why Airflow is so powerful: you can easily switch out tools without having to change the rest of your pipeline.

4.3.1 Connect Airflow to OpenAI

To connect Airflow to OpenAI, first Chris needed to install the OpenAI provider package (https://airflow.apache.org/docs/apache-airflow-providers-openai/stable/index.html). You can install a provider package the way you install any Python package: by adding the package name to the `requirements.txt` file as shown in listing 4.2, using the latest available version.

> **NOTE** Always pin the version of your packages in your `requirements.txt` file; otherwise, Airflow will automatically install the latest version, which may one day include changes that are incompatible with your current code. We recommend that you deliberately upgrade any Python packages installed in your Airflow environment by changing the pinned version.

Listing 4.2 Installing the OpenAI provider package

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

Replace <version> with the version you want to install. Use the latest version available.

For any changes to the `requirements.txt` file to take effect, restart your Airflow environment. When you're using the Astro command-line interface (CLI), you can do this by running `astro dev restart` in the command line.

Now that the provider is installed, you can add an Airflow connection to OpenAI. Click the Admin button on the sidebar; then click Connections to view the list of connections defined for this Airflow instance. Click the Add Connection button to add a connection, as shown in figure 4.10.

In the connection form (figure 4.11), add the following values:

- (1) *Connection ID*—`my_openai_conn`. This is the string by which you will be able to refer to this connection in your code.
- (2) *Connection Type*—OpenAI. If you don't see OpenAI in the drop-down menu, double-check whether you installed the provider package correctly.
- (3) *API Key*—Your OpenAI API key (https://platform.openai.com).

In Airflow, you can use stored connections across your environment by referencing the connection ID string. Many traditional operators have a parameter named `conn_id` (or something similar) that accepts a connection string as an input. If you want to use a connection in a `@task` decorated function, you can use a hook class instead. A *hook* is a helper class in Airflow that connects to a specific external tool via an Airflow connection; most hooks have many methods that abstract calls to the API of the external tool. You can learn more about hooks in the Airflow hooks guide (https://www.astronomer.io/docs/learn/what-is-a-hook). Listing 4.3 shows how to use the `OpenAIHook` to connect to OpenAI using the `my_openai_conn` connection.

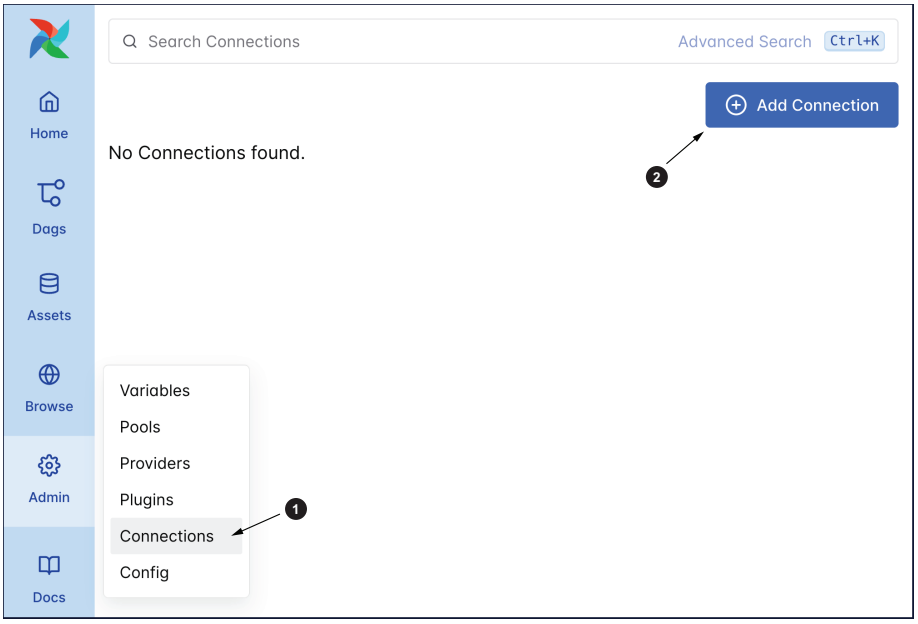


Figure 4.10 Empty connections list. You can view all connections defined for an Airflow instance by clicking the Admin button and then clicking Connections (1). The Add Connection button (2) opens a form where you can add a new connection.

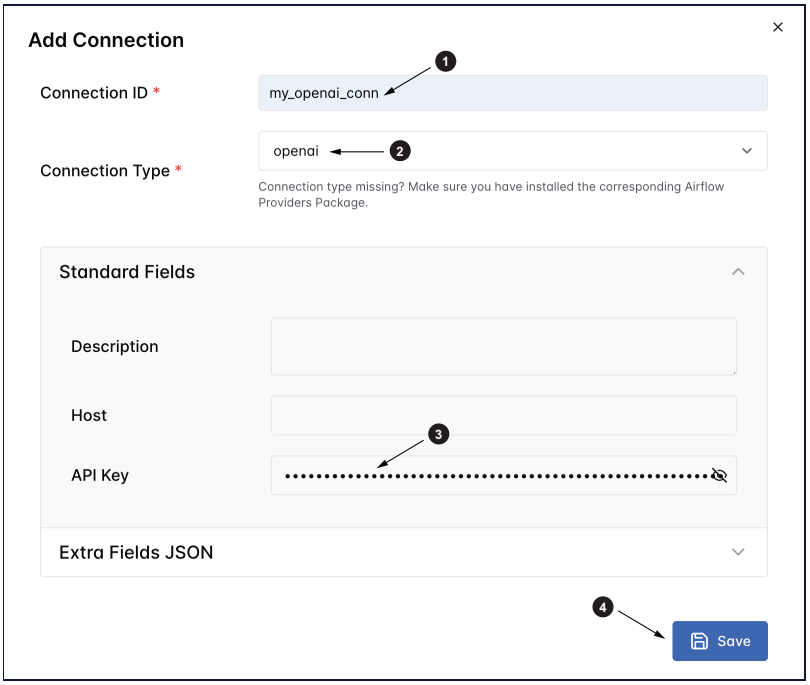


Figure 4.11 Connection form filled out for a connection to OpenAI, including Connection ID (1), Connection Type (2), and API Key (3). Don't forget to save (4) the connection!

Listing 4.3 Using the `OpenAIHook` in a task

```
@task
def my_genai_task():
    from airflow.providers.openai.hooks.openai import OpenAIHook

    my_openai_hook = OpenAIHook(
        conn_id="my_openai_conn"
    )
    client = my_openai_hook.get_conn()
```

Uses the credentials stored in the Airflow connection to connect to OpenAI

Retrieves the client object to interact with the OpenAI API

Other ways to connect Airflow to other tools

There are other ways to create an Airflow connection. You can also store connection credentials as an environment variable by using a URI or JSON format, or you can use a secrets backend. You can learn more about options for creating an Airflow connection in the related Learn guide (https://www.astronomer.io/docs/learn/connections).

Astro customers can use the Astro Environment Manager to reuse the same connection in several deployments and even locally. See chapter 6 for more information.

When using custom code in an `@task` or `@asset` decorated function, you can also directly retrieve connection credentials stored in the environment. In Chris's newsletter pipeline, storing the OpenAI API key as an environment variable is a valid alternative to using the OpenAI provider and hook. Which method you choose in custom code to connect to your external tools is a matter of preference.

Don't use connections in top-level code

One of the most common mistakes beginners make when writing dags is connecting to other tools outside their task code—in their top-level dag code, for example. Airflow files are parsed regularly, and on each parse, all top-level dag code is executed. This means that if you query a database in the top-level code, the query runs on every dag parse (by default, every 30 seconds), incurring cost.

Always make sure that your top-level code does not connect to other tools or take too long to execute. You can find more information, along with a code example showing a top-level connection and details on how to refactor the code to prevent this kind of mistake, in the best-practices Learn guide (https://mng.bz/Kw2O).

4.3.2 *Craft the system prompt*

Now that Airflow and OpenAI are connected, let's give the LLM some instructions on how to respond to our queries in the form of a system prompt. The following listing shows a possible system prompt to create a personalized motivating quote based on the individual subscriber's motivation and favorite sci-fi character.

Listing 4.4 System prompt

```
SYSTEM_PROMPT = (
    "You are {favorite_sci_fi_character} "
    "giving advice to your best friend {name}. "
    "{name} once said '{motivation}' and today "
    "they are especially in need of some encouragement. "
    "Please write a personalized quote for them "
    "based on the historic quotes provided, include "
    "an insider reference to {series} that only someone "
    "who has seen it would understand. "
    "Do NOT include the series name in the quote. "
    "Do NOT verbatim repeat any of the provided quotes. "
    "The quote should be between 200 and 500 characters long."
)
```

The user prompt is the current day's generic quotes. Both prompts will be provided to the LLM in an API call sent from an Airflow task.

4.3.3 Write the task

It was time to add the task (listing 4.5). The motivation and favorite sci-fi character information was already contained in the user dictionaries returned by the `get_user_info` task, which Chris could provide to the task as an argument called `user`. He also needed to provide the `system_prompt` template and the generic quotes from the day's formatted newsletter. The system prompt could be provided as an argument. As for the daily quotes, Chris found it easiest to retrieve them directly from the day's generic newsletter, taking advantage of the formatting and his hobby of regex-golfing. After getting all the elements of the prompts ready, he had only to pass them to the OpenAI API call and retrieve the text from the response.

Listing 4.5 Task creating the personalized quote

```
@task(max_active_tis_per_dag=16)
def create_personalized_quote(
    system_prompt,
    user,
    **context,
):
    import re

    from airflow.sdk import ObjectStoragePath
    from airflow.providers.openai.hooks.openai import (
        OpenAIHook,
    )

    my_openai_hook = OpenAIHook(
        conn_id="my_openai_conn"
    )
    client = my_openai_hook.get_conn()
```

Limits the maximum number of parallel API calls

Passes the system_prompt, subscriber information, and Airflow context to the function

Creates a connection to OpenAI

Creates an OpenAI client

```
id = user["id"]
name = user["name"]
motivation = user["motivation"]
favorite_sci_fi_character = user[
    "favorite_sci_fi_character"
]
series = favorite_sci_fi_character.split(" (")[
    1
].replace(")", "")
date = context["dag_run"].run_after.strftime(
    "%Y-%m-%d"
)

object_storage_path = ObjectStoragePath(
    f"{OBJECT_STORAGE_SYSTEM}://{OBJECT_STORAGE_PATH_NEWSLETTER}",
    conn_id=OBJECT_STORAGE_CONN_ID,
)

date_newsletter_path = (
    object_storage_path
    / f"{date}_newsletter.txt"
)

newsletter_content = (
    date_newsletter_path.read_text()
)

quotes = re.findall(
    r'\d+\.\s+"([^"]+)"', newsletter_content
)

system_prompt = system_prompt.format(
    favorite_sci_fi_character=favorite_sci_fi_character,
    motivation=motivation,
    name=name,
    series=series,
)
user_prompt = (
    "The quotes to modify are:\n"
    + "\n".join(quotes)
)

completion = client.chat.completions.create(
    model="gpt-4o",
    messages=[
        {
            "role": "system",
            "content": system_prompt,
        },
        {
            "role": "user",
            "content": user_prompt,
        },
    ],
)
```

Retrieves information from the user dictionary

Extracts the TV series from the favorite-character field

Connects to file storage based on environment variables

Uses a regex to retrieve the generic quotes in today's newsletter

Formats the system prompt from listing 4.4 with the subscriber information

Creates the user prompt

Makes a call to the OpenAI API

```
    generated_response = completion.choices[
        0
    ].message.content

    return {
        "user_id": id,
        "personalized_quote": generated_response,
    }
```

Gets the message content from the API response

```
_create_personalized_quote = (
    create_personalized_quote.partial(
        system_prompt=SYSTEM_PROMPT
    ).expand(user=_get_user_info)
)
```

Dynamically maps the task over the _get_user_info input while the system_prompt input stays the same for all dynamically mapped task instances

TIP By default, the maximum number of dynamically mapped task instances of one task is `1,024`. You can increase this number by setting the `[core].max_map_length` Airflow configuration.

After the `create_personalized_quote` task, Chris added a short helper task that takes the input from both—the task creating the personalized quote and the task getting the weather report for subscribers—and combines the information based on the subscriber's ID. The following listing shows how.

Listing 4.6 Task combining information from two upstream tasks

```
@task
def combine_information(
    user_info: list[dict],
    personalized_quotes: list[dict],
) -> None:
    user_info_dict = {
        user["id"]: user for user in user_info
    }
    for quote in personalized_quotes:
        user_id = quote["user_id"]
        user_info_dict[user_id][
            "personalized_quote"
        ] = quote["personalized_quote"]

    return list(user_info_dict.values())

_combine_information = combine_information(
    user_info=_get_weather_info,
    personalized_quotes=_create_personalized_quote,
)
```

Last, Chris updated the `create_personalized_newsletter` task to use `_combine_information` as an input to dynamically map over, and he added a couple of lines of code to include the personalized quote in the final personalized newsletter. You

can view the finished dag in the companion GitHub repository (https://github.com/astronomer/practical-guide-to-apache-airflow-3). The following listing shows an example of a final personalized newsletter, complete with a personalized quote in the style of Captain Kathryn Janeway!

Listing 4.7 Newsletter with a personally generated quote

```
==========================================
      Daily Reality Tunnel 2025-04-22
==========================================

Hi Heiner!

If you venture outside right now in Lipari,
you'll find the temperature to be 21.5°C,
but it will feel more like 22.0°C.
The relative humidity is 60%.


As you surf the probabilistic waves of existence today,
take these three quotes with you:

1. "Paths are made by walking" - Franz Kafka
2. "Nothing in life is to be feared,
it is only to be understood." - Marie Curie
3. "The future belongs to those who believe
in the beauty of their dreams." - Eleanor Roosevelt


-----------
This is what Janeway might say to you today:

Heiner, understanding where you've been gives
you the power to create your own path.
The past isn't a weight—it's a lesson, a map.
Turn fear into understanding, and you gain the ability to chart
a new course. Keep dreaming, and you'll help
others rise alongside you. After all, there
might be coffee in the next nebula!

-----------

Have a fantastic journey!
AllThingsLookingUp Team

========================================================
Consider also subscribing to our other newsletters:
"Weekly Dose of Happiness" and "Monthly Wonder".

"...reality is always plural and mutable." - R.A.W.
========================================================
```

4.4 Dag versioning and dag bundles

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

- *There was 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.
- *Pushing code during a dag run could lead to version collisions.* 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, of course, carried a significant risk of unintended consequences. 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 of the dag, 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.

Airflow 3 introduces two new concepts to address these constraints:

- *Dag bundle*—A collection of files containing dag code and supporting files:
  - Some dag bundles are versioned, such as the `GitDagBundle`. A version of a dag bundle is created by versioning the underlying backend. A new version of the `GitDagBundle`, for example, is created by every `git commit`, whether or not any dags change.
  - The default `LocalDagBundle` is not versioned.
- *Dag versioning*—Now Airflow 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—adding a new task, for example.
  - 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.

Airflow 3 supports configuring one or more dag bundles in the same Airflow environment. Those bundles can come from different systems. Some dags and their supporting code are stored in a GitHub repository, for example, and some are in blob storage. The Airflow 3.0 release provides a local dag bundle (`LocalDagBundle`), and a Git dag bundle class (`GitDagBundle`) with support for other backends is planned for the future.

NOTE For dags running on Astro, Astronomer's managed Airflow service, a specialized versioned dag bundle is configured automatically, without any

need for additional setup by the customer. See the Astronomer documentation (https://mng.bz/8Xlg) for more information.

Chris made several structural changes to the `personalize_newsletter` dag over the course of developing it. He also created dag runs during development, which led to several dag versions being stored. Chris found that he could use the Options menu in the top-right corner of the dag graph to see the dag graph of previous dag versions (figure 4.12).

Figure 4.12 Side-by-side comparison of v1 (1) and v2 (2) of the `personalize_newsletter` dag

"I wonder what would happen if I reran a dag run that originally ran using a different version from the latest code I wrote," Chris thought. In the dag versioning Learn guide (https://mng.bz/Ewxr), he found his answer: it depends on whether the dag bundle in which the dag is stored is versioned.

Table 4.1 breaks down the differences between using a dag bundle that supports versioning (such as `GitDagBundle`) and using a dag bundle that does not support versioning (such as `LocalDagBundle`). *Versioning* here refers to the versioning of the dag bundle itself. Dags are always versioned in Airflow 3, no matter which type of dag bundle is used.

You can control many aspects of how dag bundles and dag versioning behave. See the Airflow documentation for more information (https://mng.bz/NwxX). Note that rerunning programmatically created dags when using a versioned dag bundle can lead to problems if best practices are not followed; see the Learn guide for more information.

Table 4.1 Dag bundle type comparison

Scenario	Dag bundle that is not versioned	Versioned dag bundle
Viewing previous dag runs in the UI	The dag graph and code tab display the dag that existed at the time of the dag run.	The dag graph and code tab display the dag that existed at the time of the dag run.
Creating a new dag run	Uses the current dag code.	Uses the current dag code.
Rerunning a whole previous dag run	Uses the current dag code.	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 run their tasks.
Rerunning individual tasks of a previous dag run	Uses the latest version for tasks that are rerun.	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	The dag always uses the current dag code at the time it starts a task, as in Airflow 2.	The dag run finishes using the bundle version it started with.
Running a backfill (section 4.6)	Uses the current dag code.	Uses the current dag code.
Making code changes	Every structural change to the dag creates a new dag version.	Every committed/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.

4.5 Setting up a GitDagBundle

Wanting to be able to share his code with other team members, Chris decided to move his pipeline to GitHub. After creating a new repository and pushing his code to the main branch (https://docs.github.com/en/get-started), he decided to try out the `GitDagBundle` to see how the dag versioning behavior changed. The three ingredients necessary to configure a `GitDagBundle` are

- *Provider*—First, Chris had to install the Airflow Git provider (https://mng.bz/8XpD) by adding `apache-airflow-providers-git==<version>` to his `requirements.txt` file and `git` to his `packages.txt` file restarting the Airflow environment with `astro dev restart`.
- *Connection*— Next, Chris needed to define an Airflow connection to GitHub with the connection ID `git_default`. He set this connection using an environment variable (see listing 4.8). Environment variables starting with `AIRFLOW_CONN` can

be used to define Airflow connections. For more information, see the Learn guide on connections (https://www.astronomer.io/docs/learn/connections).

- *Configuration*—The `[dag_processor].dag_bundle_config_list` Airflow configuration needs to be changed to use the `GitDagBundle`, as shown in the following listing.

Listing 4.8 Configuration in `.env` to use a `GitDagBundle`

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

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

Determines which GitHub repository to pull dags from

Used to authenticate to GitHub

References the folder to fetch dags from

References the branch to fetch dags from

As shown in the listing, a GitHub personal access token is used to access the repository's code and metadata. See the GitHub documentation for details (https://mng.bz/EwER). Note that the token needs to have read and write access to the contents of your repository.

After restarting his environment using `astro dev restart` and committing a change to his repository, Chris observed that rerunning a previous run no longer used the latest version, as was the case with the `LocalDagBundle`. "Neat!" Chris thought. "This will be very useful when I lead a team of several data engineers (one day, I hope), being able to track dag changes like that directly in the Airflow UI."

4.6 Running a backfill

Michelle was quite impressed by Chris' impromptu demo of his new pipeline in their first one-to-one call. "This looks really cool," she said. "Now, I know this isn't relevant for this newsletter pipeline in particular, but I wonder if you could help me build out some pipelines analyzing our website metrics. The only problem is we often need to process a whole bunch of data for dates in the past at the same time. Can Airflow do that?"

"Of course," Chris replied, having recently seen a video about improvements to the backfill feature in Airflow 3. Now it's possible to create runs for dates in the past directly from the UI (figure 4.13) and the Airflow `REST` API (https://mng.bz/Nwnd),

and backfills are executed like other dag runs by the scheduler, fully observable in the Airflow UI.

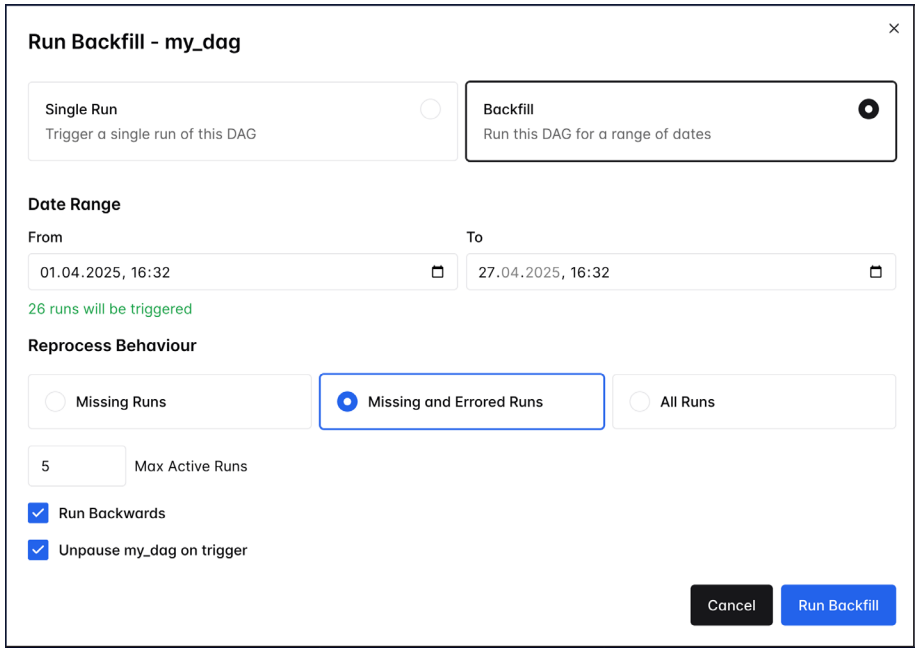


Figure 4.13 The form to run backfills from the Airflow UI. In this form, you can select the date range for your backfill, the type (missing, missing and errored, or all) and how many backfills are allowed to run concurrently.

Chris used the UI to backfill a week's worth of runs for a simple toy dag using a `@daily` schedule. And just like that, he already knew what his next big project would be at *AllThingsLookingUp*. First, though, he wanted to take the newsletter pipeline to production.

To do that, Chris still had to wait to hear back from Jan for final content approval. "This might take a bit," he thought to himself. "I think I can get away with reading some more about what is happening in my Airflow instance while running this pipeline."

Airflow plugins

Airflow plugins can extend the functionality of the Airflow UI. In Airflow 2, plugins were written using Flask or Flask-AppBuilder (FAB). Airflow 3.0 supports FastAPI apps, and in future Airflow versions, you'll be able to write plugins in React. For more information, see the Airflow plugins guide (https://mng.bz/DwzV).

Summary

- Airflow 3 comes with a completely redesigned UI, including dark mode, easier access to logs, and asset graphs.
- To connect Airflow to other tools in your data ecosystem, you can store connection credentials in an Airflow connection object. Airflow connections can be defined in several ways; the easiest is to add them in the Airflow UI. To use an Airflow connection, reference its connection ID string in a traditional operator or hook class.
- You can connect to OpenAI using the `OpenAIHook`, which is part of the OpenAI provider package for Airflow. The hook has many methods that make it easier to interact with the OpenAI API. For custom requirements like those in this pipeline, you can use the `.get_conn()` method to retrieve the client and use it directly.
- In Airflow 3, you can store your dags in different locations to be fetched by the dag processor in your Airflow environment. The retrieved sets of files are called dag bundles:
  - Airflow 3 ships with dag bundle options—`LocalDagBundle` (built-in) and `GitDagBundle` (part of the Airflow Git provider)—with support for other backends planned in the future.
  - There are differences in the behavior of versioned (such as `GitDagBundle`) and unversioned (such as the `LocalDagBundle`) dag bundles, especially when you're rerunning past dag runs.
- Dag versioning enables you to view structural changes to your dag over time in the Airflow UI. Dags are always versioned in Airflow 3, no matter which dag bundle is used:
  - You can view past versions of your dag graph by using the Options menu in the Airflow graph view.
  - Past versions of your dag code are visible on the Code tab of a dag run that used said dag code version.
  - Unrelated to your type of dag bundle, Airflow registers a new dag version every time a dag is run and changed its structure since the last dag run.
- You can create backfills in Airflow 3 from the Airflow UI and `REST` API, and they run using the Airflow scheduler..

5 Airflow 3 architecture

This chapter covers

- The components of Airflow 3 and how they interact
- The benefits of decoupling task execution from Airflow system components
- Options for running Airflow tasks on remote machines

While waiting for final approval to deploy his pipeline to production, Chris decided to dive deeper into the architecture of Apache Airflow to learn about the differences between Airflow 2 and Airflow 3.

Airflow 3 introduces a major architectural change: decoupling Airflow workers that execute task code from the Airflow metadata database. Instead, workers now use the Airflow API server to communicate with the metadata database. This change greatly improves Airflow's security posture and enables two new foundational features: Remote Execution of Airflow tasks and the ability to define tasks in languages other than Python.

In this chapter, you'll get an overview of the architectural differences between Airflow 2 and Airflow 3, as well as all Airflow components, old and new. You'll also get to

know two options for running Airflow tasks on remote machines: the `EdgeExecutor` for open source Airflow and the Remote Execution Agent for Astro.

> **NOTE** The redesign of the Airflow architecture keeps as much back compatibility as possible for dags written in Airflow 2. Notable exceptions are tasks and custom operators that access the Airflow metadata database directly. No direct access of tasks to the Airflow metadata database is possible in Airflow 3. See chapter 8 for more information on migrating from Airflow 2 to Airflow 3.

5.1 Airflow architecture

Airflow consists of several components that work together to run dags like the ones Chris wrote in the preceding chapters. Although you don't need an in-depth understanding of Airflow's architecture to write pipelines, having a general sense of how the components interact can be helpful in debugging.

Significant changes took place between the architectures of Airflow 2 and Airflow 3, as shown in figure 5.1. The most important difference is that in Airflow 2, all components interacted directly with the Airflow metadata database. In Airflow 3, task execution is decoupled from the other components. Now workers interact with a new component: the API server. This server serves the static assets for the Airflow UI alongside three APIs, one of which is a new minimal-privilege API acting as the task execution interface. This change embodies a major security enhancement, removing direct access of user code to the Airflow metadata database. Also, the change is fundamental to enabling running workers on remote machines and writing tasks in languages other than Python.

In the following sections, you'll learn more about the reasoning behind the architecture change; you'll also learn about the Airflow components and how they interact. To start, compare the two architectures in figure 5.1.

5.1.1 Limitations of the Airflow 2 architecture

The Airflow 2 architecture was designed to run all components within the same trusted network cluster. This means that both the user-provided task code (the code you write inside an Airflow task) and task execution code (the code within the Airflow package that turns your code into an Airflow task and runs it) ran in the same container and process.

This design gave Airflow workers direct access to the Airflow metadata database when retrieving connection credentials and environment information. This approach, however, had several limitations:

- *Security risks*—Because all tasks had full access to the Airflow metadata database, accidental or malicious destructive actions could be executed from within Airflow tasks, such as dropping the dag run history for a business-critical dag.
- *Package and version conflicts*—If multiple teams were running tasks in the same Airflow environment but needed different versions of Python or Python packages, they encountered package conflicts. Some of these conflicts could be addressed

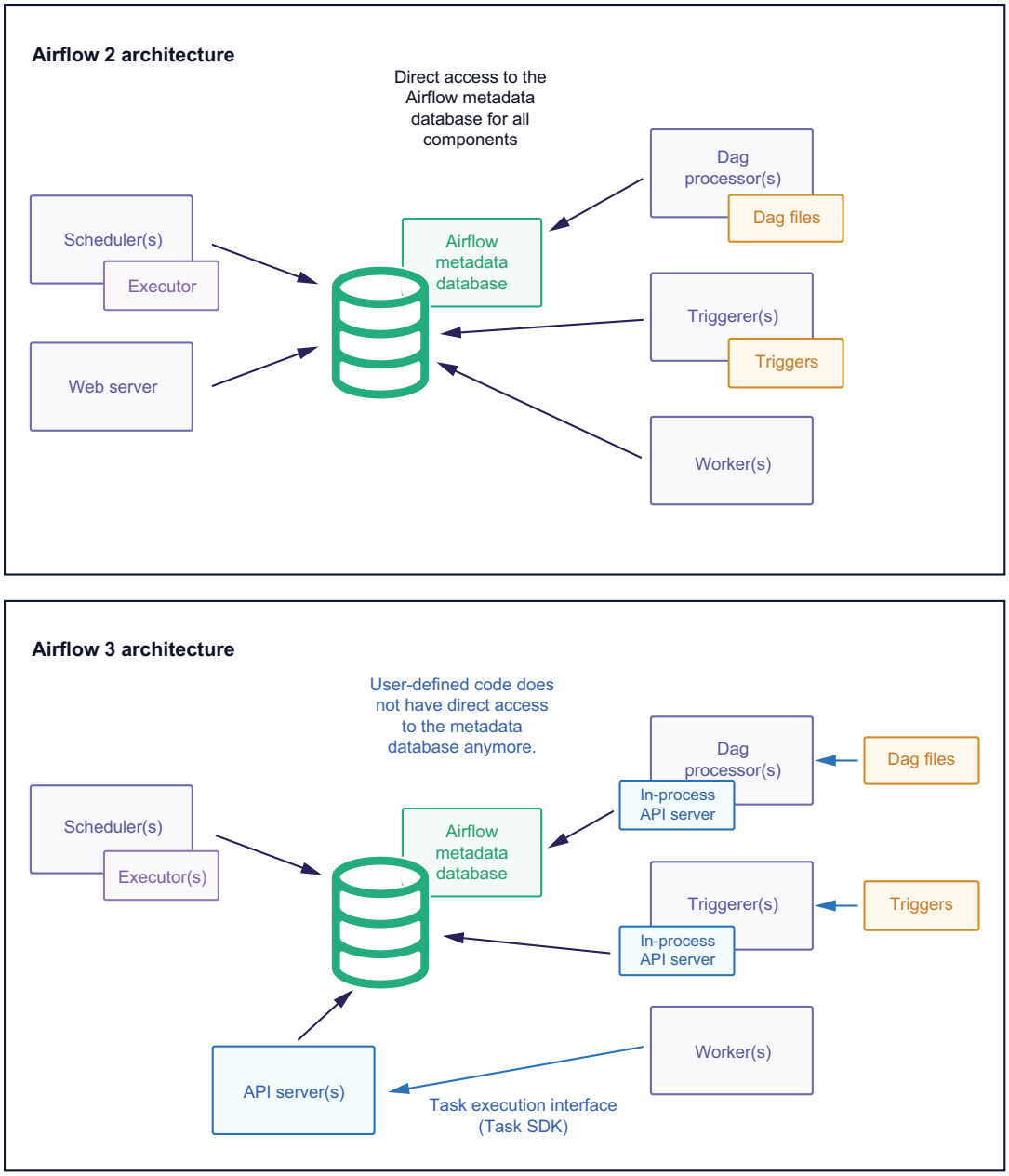


Figure 5.1 Overview of the Airflow 2 and Airflow 3 architectures. You can see that the Airflow 3 architecture uses the new API server component as the interface for task code running on workers. For a detailed explanation, see section 5.1.2.

using specialized operators, such as the `ExternalPythonOperator` or the `KubernetesPodOperator`; see the Learn guide on isolated environments (https://mng.bz/26Wa). Other conflicts, such as using operators from different versions of the same Airflow provider package, required users to maintain two separate Airflow environments.

- *Inability to separate component upgrades*—Airflow workers could not be upgraded independently of other Airflow components.
- *Limited scalability*—In Airflow 2, the number of concurrent connections to the Airflow metadata database limited scalability. Because all tasks connected directly to the database, the maximum number of allowed connections was reached more quickly.
- *Difficult feature implementation*—Constraints on feature implementation (for Remote Execution [section 5.2] and multilanguage support) proved to be difficult to implement in the old architecture.

These limitations led to the decision to decouple task execution from the Airflow metadata database in Airflow 3, with all interaction of user-defined task code being handled by the new API server.

5.1.2 Airflow components

Like many modern tools, Airflow consists of several components, shown in figure 5.2. Let's look at the roles of these components and how they interact to run your dags.

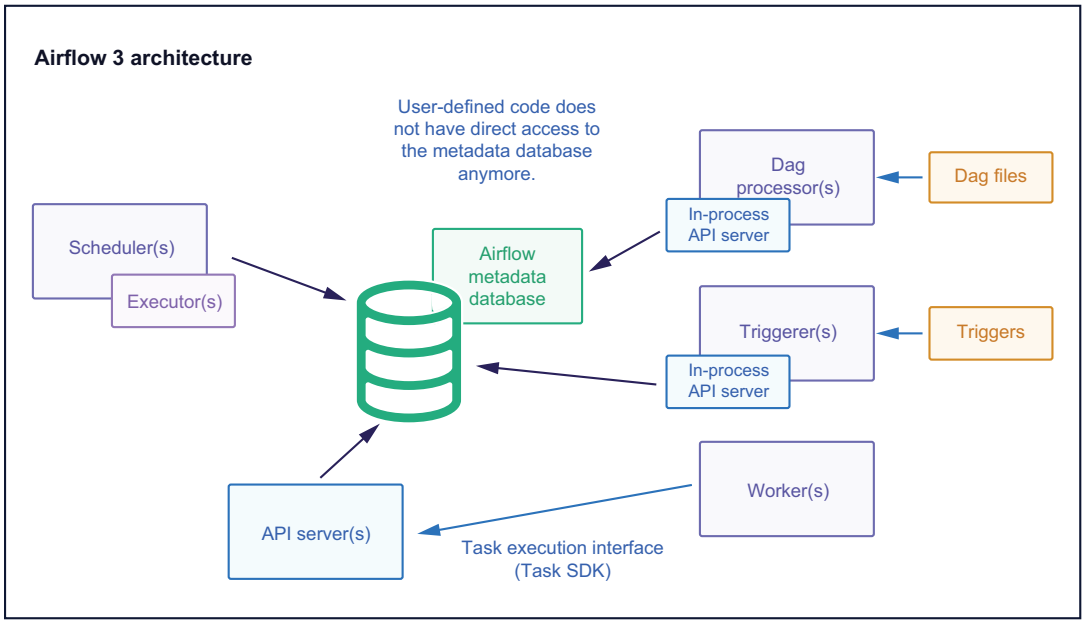


Figure 5.2 Overview of the Airflow 3 architecture

On a high level, this is what happens when you add a simple new dag to your Airflow environment:

1. The dag is parsed by the dag processor, which stores a serialized version of it in the Airflow metadata database.
2. The scheduler checks the serialized dags to determine whether any dag is eligible for execution based on its defined schedule. This process includes checking the schedules against the current time and checking for information such as updates to assets or events fired by AssetWatchers. (See chapter 3 for more information on scheduling dags.)
3. When the scheduler determines that a dag is ready for its next run, its configured executor decides how and where to run the first task instance(s) of the dag run. (See table 5.1 for details on some common executors.)
4. Next, the task instance(s) are scheduled and subsequently queued. The workers poll the queue for any queued task instances they can run.
5. The worker who picked up the task instance runs it, and metadata such as the task instance status or XCom is sent from the worker via the API server to be stored in the Airflow metadata database. If the task needs any information, such as an Airflow connection (chapter 4), the worker sends a request to the API server for this information. The API server retrieves the details from the Airflow metadata database and hands them back to the worker. As the task instance is running, the worker writes task instance logs directly to the defined log storage location.
6. Some of this information, such as the task instance status, is in turn important for the scheduler. It monitors all dags and, as soon as their dependencies are fulfilled, schedules task instances to run.

While this process is going on in the background, the Airflow UI, served by the API server, displays information about the current dag and task statuses that it retrieves from the Airflow metadata database.

There is more to each component (and we haven't even talked about the triggerer yet). So let's dive more deeply into individual components.

SCHEDULER(S) AND EXECUTOR(S)

The *scheduler* is the heart of Airflow. It monitors all tasks and dags and schedules task instances to run as soon as their dependencies are fulfilled. When creating a new dag run, the scheduler always picks the latest version of that dag. (See chapter 4 for more information on dag versioning.) When a task is ready to run, the scheduler uses its configured *executor* to run the task on a *worker*.

> **NOTE** In production environments, we recommend running several copies of the scheduler. In Astro, you can configure this when creating a new deployment by setting the High Availability toggle switch to On (UI) or setting `isHighAvailability` to `true` (API, Terraform).

You can choose among several executors. In a production environment, the ones most commonly used in open source Airflow are the `CeleryExecutor` and the `KubernetesExecutor`; for local development, we recommend the `LocalExecutor`. Table 5.1 provides an overview of frequently used executors. You can also create custom executors; see the Airflow documentation for instructions (https://mng.bz/15gq).

> **NOTE** Astro supports both the `CeleryExecutor` and the `KubernetesExecutor` in its cloud deployments. Airflow 3 deployments can also run with the new and optimized `AstroExecutor`, which enables Remote Execution (see section 5.2.1).

Table 5.1 Commonly used executors

Executor	Environment	Function
`AstroExecutor`	Production (Astro only)	Optimized executor to run Airflow 3 workloads on Astro. It offers better reliability than the `CeleryExecutor` and has less startup time for task instances than the `KubernetesExecutor`. This executor also allows Astro customers to execute their tasks remotely (see section 5.2.1), enabling them to use Remote Execution to run task instances in remote environments with full task instance isolation. This is ideal for task instances handling sensitive data that cannot leave a remote location. This executor is a good default choice for running Airflow 3 on Astro for both hosted and remote deployments.
`CeleryExecutor`	Production	The `CeleryExecutor` runs each task instance in a distributed worker process, enabling parallel execution across a fleet of worker nodes. Task instances are run in a process pool, minimizing startup time per task instance. This executor is a good default choice for production Airflow environments. It supports horizontal scaling and can distribute task execution efficiently across multiple workers, making it well suited to handle large dags or high task instance volumes.
`Kubernetes-Executor`	Production	The `KubernetesExecutor` runs each task in its own dedicated Kubernetes pod, enabling fine-grained resource and environmental control for every task. This executor is a good choice when you need precise control of each task's environment and resource requirements, such as specifying custom images, memory limits, or CPU requests.
`EdgeExecutor`	Production (open source Airflow only)	Specialized executor introduced in Airflow 3 to run task instances in remote environments. This executor is a good choice to run task instances with full task isolation for users running open source Airflow (see section 5.2.2).
`LocalExecutor`	Development and very lightweight production	The `LocalExecutor` runs each task instance in a separate process on the same machine as the scheduler, enabling parallel execution without requiring external workers or message brokers. This executor is a good choice for development environments and local testing; it can also be used for very lightweight production use cases.

Multiple-executor environments

You can use multiple executors in the same open source Airflow environment. This feature, introduced in Airflow 2.10 on an experimental basis, allows users running open source Airflow environments to specify executors at the task level using a task parameter. For more information, see the Airflow documentation on using multiple executors concurrently (https://mng.bz/Pw1R).

Astro customers running tasks that have significantly different resource needs can use the worker queues feature (https://mng.bz/JwxZ).

Reading up on executors, Chris realized that while developing his pipeline, he had never even thought about how his Airflow tasks were run using the local Airflow environment created by the Astro command-line interface (CLI). Checking the Astro documentation Chris learned that his local development environment was using the `LocalExecutor`. After he deployed his pipelines to Astro, he'd have a choice among the `AstroExecutor`, `CeleryExecutor`, and `KubernetesExecutor`.

NOTE In older versions of Airflow, the scheduler was responsible for parsing the `dags` folder to discover new dags and update existing dags after code changes. Airflow 2.3 introduced the option to offload this responsibility to a separate dag processor component. In Airflow 3, the dag processor always runs as its own dedicated process.

API SERVER(S)

The *API server* serves three APIs:

- An API for workers to interact with when running task instances using the task execution interface (section 5.1.3). Here, the API server acts as an intermediary between task instances and the Airflow metadata database.
- An internal API for the Airflow UI that provides updates on dynamic UI components such as the state of task instances and dag runs.
- The public Airflow `REST` API (https://mng.bz/Dwx0) that users can interact with.

A task instance might retrieve connection credentials, such as the OpenAI connection Chris used in chapter 4, by sending a `GET` request to the API server, which looks up the relevant information in the Airflow metadata database and provides it to the task. Similarly, if a task needs to store information in XCom, such as the `get_weather_info` task in Chris' `personalize_newsletter` dag, it would send a `POST` request to the API server.

Custom XCom backends

When using a custom XCom backend to store larger amounts of data passed between tasks, only a reference—typically, a Uniform Resource Identifier (URI)—is sent to the

(continued)

API server to be stored in the Airflow metadata database. The data itself remains in the external storage managed by the custom backend.

The API server also serves the elements of the Airflow UI you and Chris have been interacting with throughout the preceding chapters. In Airflow 3, the UI is fully rebuilt in React, and the internal API on which it relies is completely restructured. This new API replaces the previous reliance on Flask-AppBuilder (FAB), enabling more flexible future development of the UI.

AIRFLOW METADATA DATABASE

If the scheduler is the heart of Airflow, the *Airflow metadata database* is the brain. This database stores information vital to Airflow's functioning, such as Airflow connections, serialized dags, and XCom information. It also contains the history of previous dag runs and task instances alongside metadata about their states.

> **NOTE** Although you can store connection information safely in the Airflow metadata database, as you did in chapter 4 with the OpenAI credentials, it is common to want to use the same credentials across multiple Airflow environments. You can accomplish this by using a secrets backend. Astro offers a built-in secrets backend to store Airflow connections to be used across Airflow deployments and in local development (see chapter 6).

Although you can use many relational databases as Airflow metadata databases, PostgreSQL and MySQL are the two recommended options. The most common backend used for the Airflow metadata database is PostgreSQL. See the Airflow documentation for supported versions (https://mng.bz/qRNx).

Curious about which database the Astro CLI uses, Chris ran the listing command (`podman ps` or, if you are using Docker for your Astro CLI containers, `docker ps`) on his computer, which was still running the local Airflow environment. The output, shown in the following listing, returned the five containers running, including one named `postgres`.

Listing 5.1 List of running containers

```
CONTAINER ID   IMAGE               [...]    PORTS          NAMES
f565...    [...]/airflow:latest  [...]    ->8080/tcp     [...]-api-server-1
77e6…      [...]/airflow:latest  [...]                   [...]-dag-processor-1
88dc...    [...]/airflow:latest  [...]                   [...]-triggerer-1
aa5a...    [...]/airflow:latest  [...]                   [...]-scheduler-1
5f0e...    postgres              [...]    ->5432/tcp     [...]-postgres-1
```

DAG PROCESSOR(S)

The *dag processor* is responsible for retrieving and parsing the files from the location determined by the configured dag bundle(s). When you're using the Astro

CLI with default configuration, this location is your `dags` folder. The dag processor's job is to generate serialized dag objects that it writes to the Airflow metadata database.

Because this parsing process involves running user code, such as generating several similar tasks using a `for` loop, it can't interact directly with the metadata database for security reasons. That is why the dag processor uses the API server code in-process to talk to that database.

WORKER(S)

A *worker* in Airflow is the component responsible for executing the task assigned to it by the executor. The type of worker used depends on the executor(s) configured in Airflow:

- With the `CeleryExecutor`, task instances are executed by Celery workers running in distributed processes.
- With the `KubernetesExecutor`, each task instance runs in its own dedicated Kubernetes pod.
- With the `LocalExecutor`, the task instance runs in a separate Python process in the scheduler component.
- With the `EdgeExecutor`/Remote Execution Agent, workers can be located in remote environments. See section 5.2.

Many workers can run simultaneously in a single Airflow environment. Workers know which version of a dag to run through the executor, which tells each worker which version of a dag bundle to use when running a task. The worker retrieves that version from the dag bundle backend. Workers can cache the dag bundle in temporary storage, allowing them to reuse it for multiple tasks within the same dag run and reducing the need to fetch it repeatedly. The configuration settings control the number of versions retained and how long they are kept.

TRIGGERER

The *triggerer* is an Airflow component responsible for running asynchronous Python functions independently of workers. This capability is especially useful for long-running tasks. When a task needs to wait for a condition, such as the state of an external system (e.g., a specific file becoming available in blob storage), the worker hands the job of waiting to the triggerer. This handoff allows the worker to free resources and focus on other tasks in the meantime.

In Airflow 3, triggers use the same task execution interface to interact with the Airflow metadata database that workers do. This design is a necessary security measure because custom triggers can execute user-defined code. The triggerer component uses the API server code in-process to interact with the metadata database. Some trigger classes can also be used in `AssetWatchers` used for event-driven scheduling (see chapter 3).

5.1.3 New task execution interface (Task SDK)

As described in the preceding sections, Airflow 3 introduces the API server component. To interact with this component, a new *task execution interface* was defined, also referred to as the *Task SDK*. This change, which addresses the limitations mentioned at the beginning of this chapter, led to the improvements described in table 5.2.

Table 5.2 Task execution comparison for Airflow 2 and Airflow 3

Aspect	Airflow 2	Airflow 3
Airflow metadata database access	User-defined code in Airflow tasks can access the Airflow metadata database directly. This pattern is discouraged for the reasons mentioned in the preceding sections but can't be disabled.	Tasks cannot interact with the Airflow metadata database directly. Instead, they make calls to the API server to retrieve information necessary to run. This means that user-defined code in Airflow tasks cannot access the metadata database directly anymore. The results of this change are significantly improved security, task isolation, and reduced concurrent open connections to the Airflow metadata database.
Upgrading	System components and workers can be upgraded only together.	Airflow system components (scheduler, API server, and so on) can be upgraded to new Airflow versions separately from Airflow workers without the risk of incompatibility of user-provided dag code with the newest Airflow version. Individual workers can be on different Airflow versions and upgraded as needed by specific teams.
Remote Execution (run anywhere)	Workers need direct access to the Airflow metadata database to retrieve information such as connection credentials and information from the Airflow context and to report the success or failure of a task instance.	You can execute Airflow tasks in remote environments by using the `EdgeExecutor` in open source Airflow and the Remote Execution Agent with the `AstroExecutor` in Astro. Airflow tasks can get all the information they need to run from the API server.
Tasks in languages other than Python (run in any language)	Airflow tasks must be defined in Python. To execute task code written in other languages, the code must be baked into a Docker image and run with the `DockerOperator` or the `KubernetesPodOperator`.	Because the task execution interface is API-based, it is language-agnostic, so a Task SDK can be implemented in any language. Airflow 3 includes a Task SDK for Python and an experimental Task SDK for Golang. For more information, see the related Learn guide (https://mng.bz/7QGQ).

5.2 Run anywhere: Remote Execution

With task execution decoupled from Airflow system components, Airflow 3 opens the door to Remote Execution of tasks on any machine. Following are some situations in which you might want to use Remote Execution:

- Running tasks that need to access and/or use sensitive data that cannot leave a secure environment, such as an on-premises server. This requirement is common in highly regulated industries such as financial services and health care.
- Running tasks that require specialized compute, such as a GPU or TPU machine to train neural networks.

When you're using Remote Execution, only the information that's essential for running the task, such as scheduling details and heartbeat pings, is available to Airflow system components. Everything else stays within the remote environment

You can accomplish Remote Execution in two ways:

- When running Airflow in Astro (chapter 6), you can use the Remote Execution Agent with the `AstroExecutor`.
- When running open source Airflow, you can use the `EdgeExecutor`, which is part of the edge3 provider package (https://mng.bz/lZaz).

5.2.1 Remote Execution Agent

In Astro, you achieve Remote Execution by using the Remote Execution Agent, which can parse dag code, pull secrets from customer-managed secret stores, execute tasks, and log task results to a customer-defined location. This way of running tasks enables companies to keep their data, dag code, secrets, and task logs in their environment by default; only information crucial to scheduling, such as the task status, is communicated back to the orchestration plane. Enterprises in highly regulated industries such as health care and financial services can use the Remote Execution Agent to adhere to strict compliance and regulatory requirements.

Helm charts for Remote Execution Agents are available for all Kubernetes environments. The Remote Execution Agent connects to the API server on the orchestration plane using secure protocols, ensuring that only agents from authorized networks can connect to a Deployment's API server.

Figure 5.3 shows an example of an architecture using the Remote Execution Agent. When running Airflow in Astro, you interact with both the Astro Control plane—the home of additional Astro features such as alerting, observability, and role-based access control (RBAC)—and the Astro orchestration plane. The orchestration plane is where all Airflow Deployments on Astro live, using the same base architecture as open source Airflow with all the components explained in section 5.1.

The Remote Execution Agent has many additional features in Astro. See table 5.3 for an overview.

> **NOTE** Table 5.3 is a snapshot. Astronomer plans to expand features for Remote Execution in Astro in future releases. To learn more, visit https://mng.bz/Bzxq.

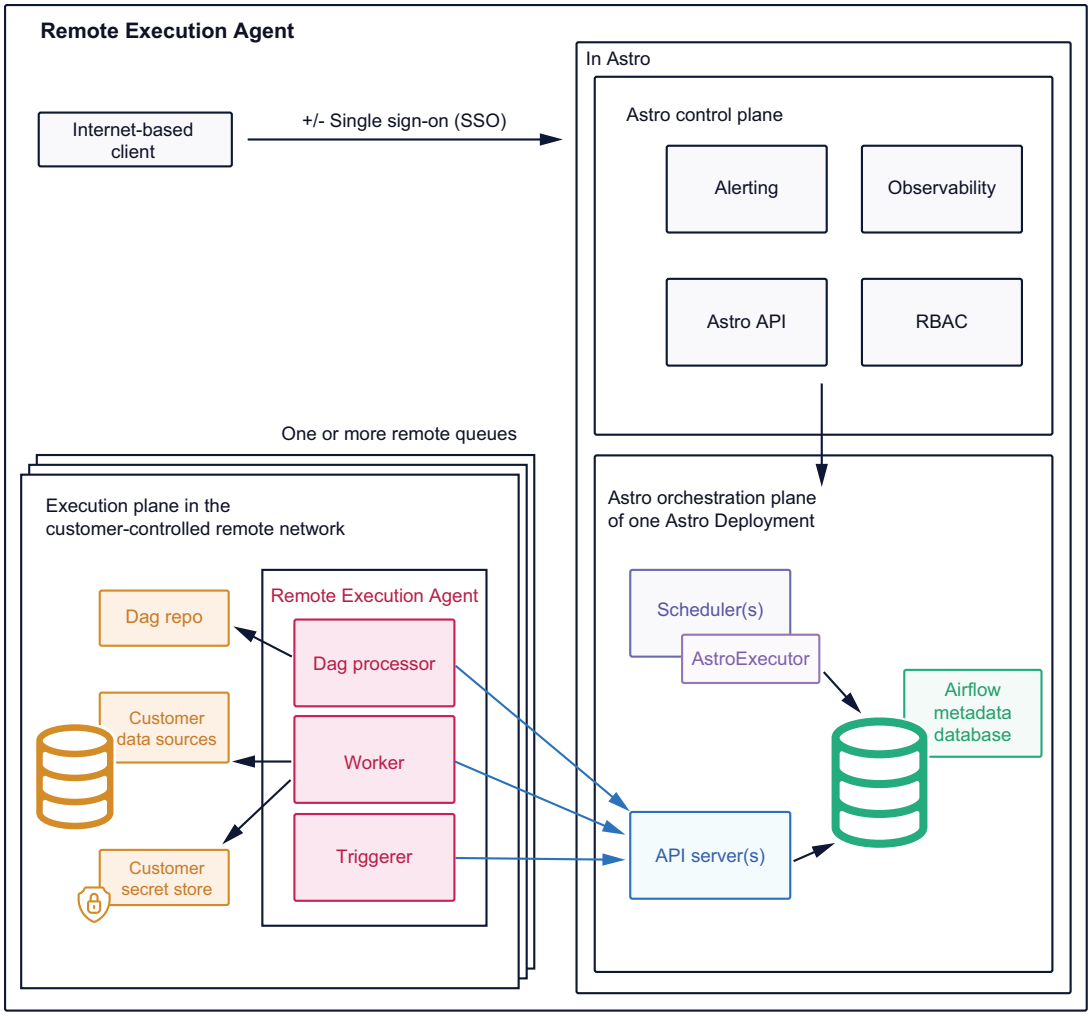


Figure 5.3 Architecture diagram showing the Remote Execution Agent

Table 5.3 Remote Execution Agent features

Feature	Explanation
Astro Remote Execution API	To interact with Remote Execution Agents, Astro built a proprietary Remote Execution API, which intelligently assigns tasks to workers and triggerers to maximize efficiency.
Astro Observe Support	Customers can collect and send lineage metadata from Remote Deployments and Agents to Astro to take advantage of Observe (https://www.astronomer.io/product/observe).

Table 5.3 Remote Execution Agent features (*continued*)

Feature	Explanation
Agent Observability	Astro displays information about Agents at a per-Deployment and per-worker queue level, including their health, status, task-slot availability, and last-heartbeat time.
Task-logging flexibility	Customers have full flexibility in how task logs are generated by tasks on remote workers. In the most high-security use cases, task logs are stored only locally in the customer's remote environment. You can configure task-log visibility in the Airflow UI for less-sensitive tasks.
Git and local bundle interfaces	The Remote Execution Agent supports Git and local dag bundles (see chapter 4). Dags are parsed and serialized locally, ensuring that customers' dag code is never stored in Astro.

5.2.2 EdgeExecutor

The `EdgeExecutor`, which is part of the edge3 provider package, enables open source Airflow users to spin up workers on remote sites via an HTTP(s) connection. You can use the `EdgeExecutor` in combination with other executors through a multiexecutor configuration (https://mng.bz/Pw1R). This configuration allows workloads that can run within the Airflow system component environment (e.g., using the `CeleryExecutor` or `KubernetesExecutor`) to be executed in the main Airflow environment; tasks that require isolation in a remote environment can be offloaded to remote workers.

Figure 5.4 illustrates an Airflow setup using the `EdgeExecutor` with the `CeleryExecutor`. Tasks are executed on workers inside the same environment and on remote workers.

In a setup using the `EdgeExecutor`, you can have additional edge workers running on any machine in your environment. These workers are fully decoupled from your central Airflow cluster and communicate only crucial information for scheduling to the API server, such as the status of their tasks. This allows you to keep your secrets and files on the remote machine without granting the central Airflow cluster any access.

The setup in figure 5.4 uses the `EdgeExecutor` alongside the `CeleryExecutor`. In an environment using multiple executors, tasks are assigned to their executor by the dag author. The tasks wait in their respective queue for an edge or Celery worker to pick them up and execute them. You can learn more about the `EdgeExecutor` in the documentation for the edge3 provider (https://mng.bz/lZaz).

"That was a lot," Chris thought, "but it's good to know about all these features, especially Remote Execution for highly sensitive workloads. I'll likely need it if I ever work at a financial institution again."

He decided to recap what he'd learned (see the Summary section). Just as he was about to head out for lunch, he got another message from Jan, the head of content, providing feedback on his newsletters (see chapter 6).

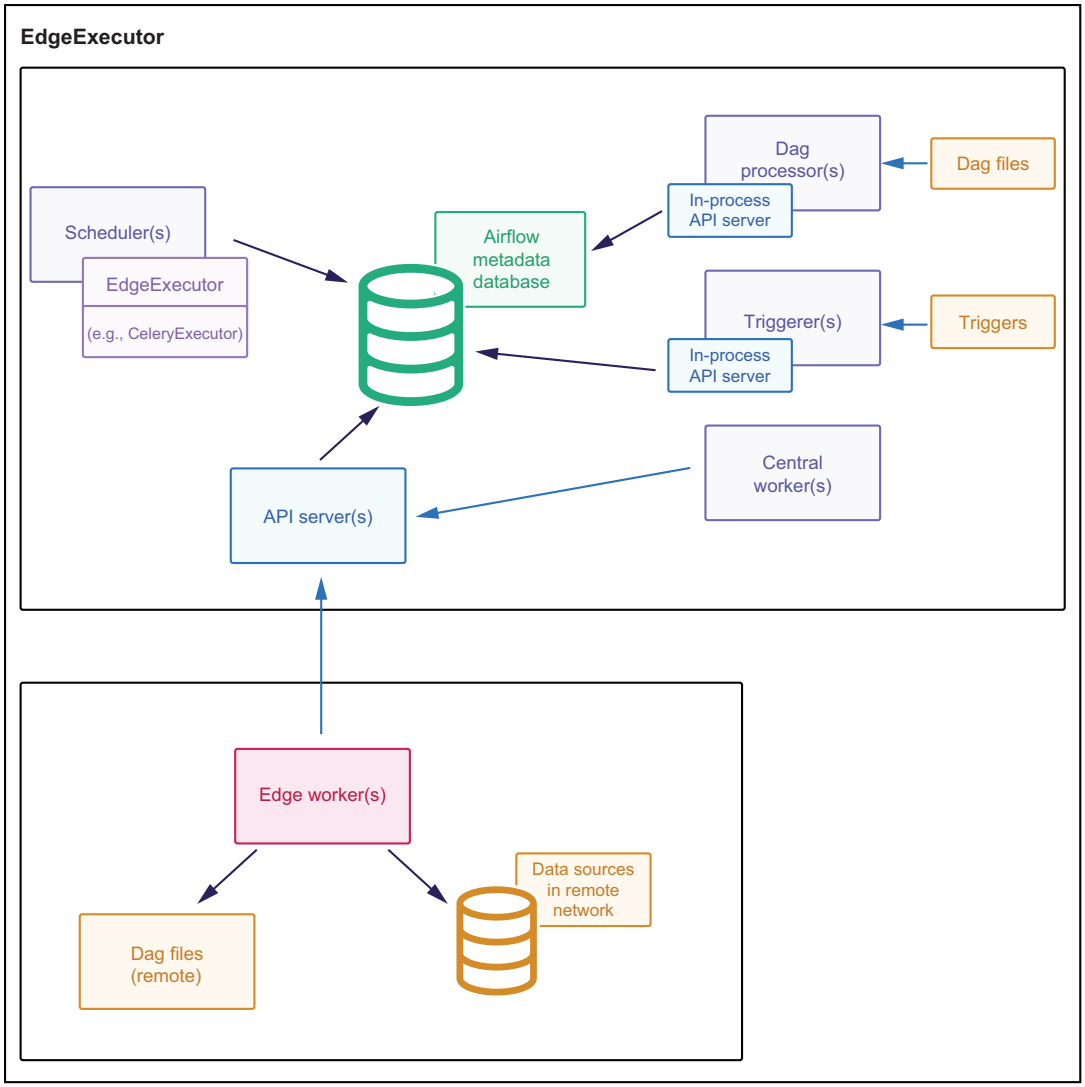


Figure 5.4 Architecture diagram of an Airflow environment using the `EdgeExecutor`

Summary

- The architecture of Airflow 3 differs significantly from that of Airflow 2. The most important change for data engineers writing pipelines is that tasks and custom operators cannot access the Airflow metadata database directly anymore. This architecture change makes Airflow more secure, enables features such as Remote Execution, and allows running tasks in languages other than Python.

- Airflow 3 introduces one new component, the API server, which replaces the web server and acts as an intermediary between the worker(s) and the Airflow metadata database.
- Workers interact with the Airflow metadata database using the new task execution interface, also known as Task SDK. Airflow 3 provides a Task SDK in Python and an experimental Task SDK in Golang. Support for additional languages is planned for future versions.
- The Airflow components are
  - The scheduler, which is the heart of Airflow and monitors dags and tasks. The scheduler creates dag runs and task instances, and its configured executors determine how and where task instances are run.
  - The API server, which provides the static and dynamic elements of the Airflow UI, the public Airflow `REST` API, and the API for workers to interact with via the task execution interface.
  - The Airflow metadata database, which stores vital information about the Airflow environment.
  - The dag processor, which parses user-provided files to generate serialized dag objects in the Airflow metadata database.
  - The workers, which are responsible for running the tasks they are assigned by the executor.
  - The triggerer component, which runs asynchronous Python functions, most commonly to free worker resources while waiting for a specific state in an external system to be reached or while using event-driven scheduling.
- Remote Execution allows you to run tasks on any machine. Open source Airflow users can use the `EdgeExecutor` from the edge3 provider package, and Astro customers can use the Remote Execution Agent.

6 Moving to production

This chapter covers

- Starting a free trial and creating your first production Deployment in Astro
- Using the GitHub integration to deploy code to an Astro Deployment
- Defining Airflow connections in the Astro Environment Manager
- Setting up a custom XCom backend using the Airflow Common IO provider
- Creating a Data Product with Astro Observe

Finally, Chris got the green light to deploy his pipeline. Let's follow along as he creates a free trial account for Astro, Astronomer's unified DataOps platform powered by Apache Airflow, and sets up his first production Deployment in the cloud.

Along the way, you'll learn how to deploy your local Airflow pipelines using Astro's GitHub integration, configure the necessary connections and environment variables for your Deployment, and find the quickest way to set up a custom XCom backend to store your XCom in cloud object storage.

Finally, when your pipeline is running in production, you'll see how Astro Observe helps keep everything on track by monitoring your data products and sending proactive service-level agreement (SLA) alerts. As Chris found out, these alerts notify him if there is a risk that the newsletter will be delayed, giving him a chance to fix the pipeline in time.

6.1 Planning a move to production

It looks as though the personalized newsletter was exactly what the content team wanted (figure 6.1). "Yes!" Chris thought. "Time to move to production." To do so, he needed to set up a production Airflow 3 environment. Because he was the first data engineer at *AllThingsLookingUp*, no Airflow infrastructure existed yet, meaning that he had to set up everything from scratch. That task is a lot for one engineer who's still new to Airflow, but luckily, managed services like Astro make things easy for engineers like Chris by handling all the infrastructure details.

Figure 6.1 Message from Jan, the head of content, telling Chris to move the pipeline to production

Here are the steps Chris had to take to move the project to production:

1. Create an Astro trial account (https://www.astronomer.io/trial-3), and follow the trial flow to create a first, empty Astro Deployment running an Airflow instance in the cloud.
2. Deploy code to the Astro Deployment.
3. Determine all configurations necessary for the dags to run in the cloud environment:
   a. Define a connection to OpenAI in the Astro cloud environment for the task that creates the personalized quote.
   b. Set environment variables so that the dags store the newsletters they create in Amazon S3 instead of local file storage.
   c. Configure a custom XCom backend to store the data passed between the tasks in S3 instead of the metadata database. This step is necessary because the metadata database, running the pipeline daily for weeks and months, would eventually fill up.

That is a plan. But first things first: lunch. Tomato-mozzarella salad. Data engineering is a marathon, not a sprint.

6.2 Getting a free Astro trial account

To move Airflow pipelines to the cloud, Chris needed a cloud Deployment. Although there are several options for setting up Airflow (https://mng.bz/oZ8N), he decided to use Astro by Astronomer, a managed Airflow service, so he could focus on building pipelines instead of managing infrastructure.

Chris started by creating a free Astro trial account. After he signed up with his email address, the trial flow guided him to create a new organization, which he named AllThingsLookingUp, and his first Astro Deployment. He had several options for creating a Deployment with example dags but chose to create an empty Deployment called Newsletter PROD.

After a couple of minutes, the new Deployment was ready in the Astro UI and accessible from the Deployments tab (figure 6.2). By clicking the blue Open Airflow button in the top-right corner, Chris opened the familiar Airflow UI, showing the empty Deployment running on Astro. Time to fill it with some dags!

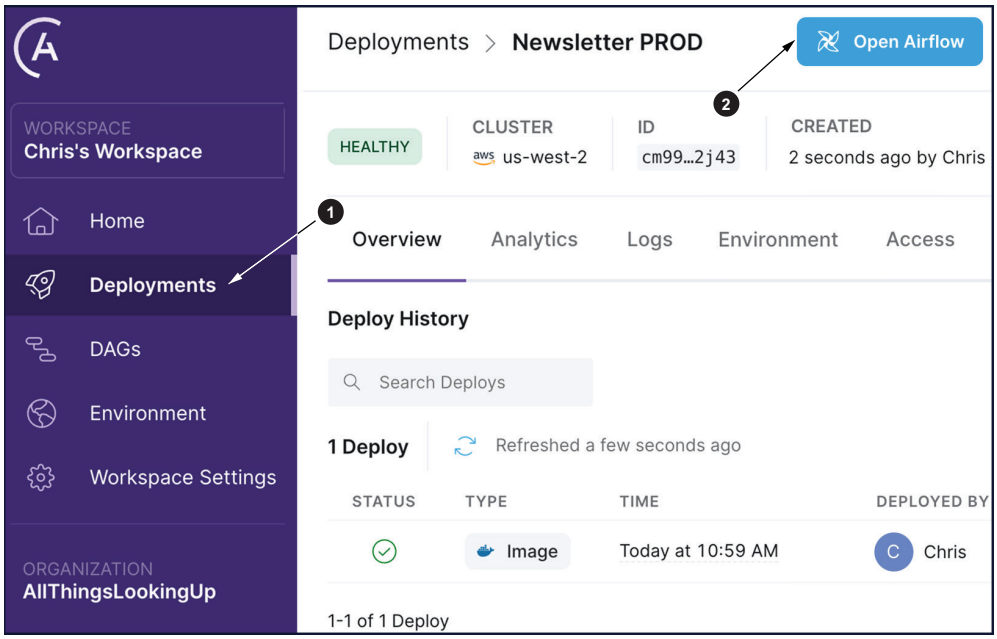


Figure 6.2 Screenshot of the Astro UI showing the Newsletter PROD Deployment. You can navigate to the Airflow UI by clicking first Deployments (1) and then the blue Open Airflow (2) button.

NOTE The Astro UI receives frequent updates. Depending on when you are reading this book, your Astro UI may look slightly different from the screenshots. Also, most screenshots in this book have been zoomed in for better readability.

Manage Astro Deployments at scale

Although the Astro UI is a great way to create your first Deployment, managing Deployments programmatically becomes essential as you scale. Programmatic Deployment management allows you to create multiple Deployments with consistent configurations, apply Infrastructure as Code (IaC) best practices by versioning your infrastructure configuration, and dynamically create or tear down resources as needed. Astro offers both a `REST` API (https://www.astronomer.io/docs/api) and a Terraform provider (https://mng.bz/nZrv) to manage resources programmatically.

6.3 Deploying code to Astro

You have three main options for deploying new dags to an Astro Deployment:

- *Astro CLI*—You can deploy any local project created with the Astro command-line interface to an Astro Deployment by running two Astro CLI commands: `astro login` and `astro deploy`.
- *GitHub integration*—Astro offers a native integration with GitHub, allowing you to map a branch in a GitHub repository to an Astro Deployment with a few clicks. Any changes to the code on the mapped branch in GitHub are automatically deployed to the Astro Deployment. This option is the easiest way to set up continuous integration/continuous development (CI/CD) for Astro.
- *Custom CI/CD workflow*—Astro customers can choose any CI/CD provider to push code to Astro Deployments. The Astro documentation provides template scripts for many popular CI/CD providers, such as GitHub Actions, Jenkins, and GitLab (https://mng.bz/vZ4x).

Having already stored his pipeline in a GitHub repository (https://docs.github.com/en/get-started; see chapter 4), Chris quickly decided to go with option 2, using the GitHub integration to deploy his code to his new Astro Deployment.

> **NOTE** You can find out more about deploying code to an Astro deployment in the Astronomer documentation (https://www.astronomer.io/docs/astro/deploy-code).

6.4 Deploying with the GitHub integration

After consulting the Astro documentation for the GitHub integration feature (https://mng.bz/4nKB), Chris knew exactly what to do. First, he clicked the Configure link in the Branch Mapping section of his Astro Deployment (figure 6.3); then he clicked Connect Repository.

The branch-mapping flow directed Chris to connect Astro to his GitHub repository. After establishing this connection, all he had left to do was map the main branch of his repo to the Newsletter PROD Astro Deployment (figure 6.4).

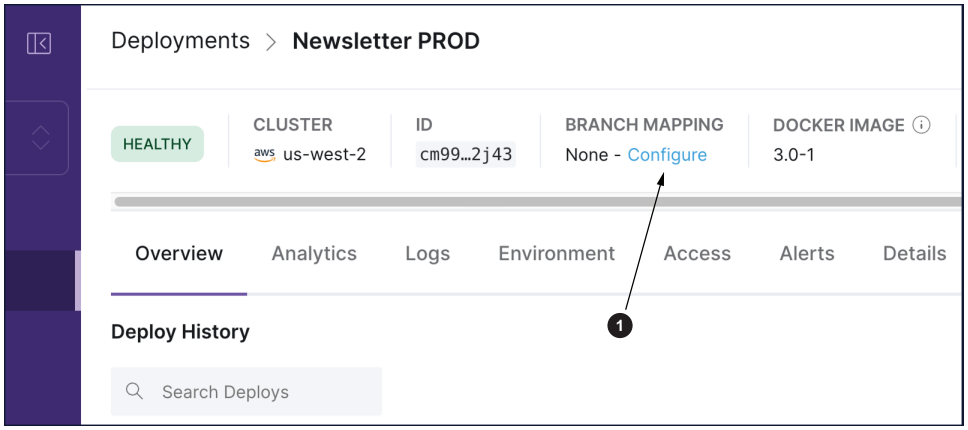


Figure 6.3 Screenshot of the Astro UI showing the Branch Mapping section, which contains the link to configure (1) the GitHub integration for an Astro Deployment

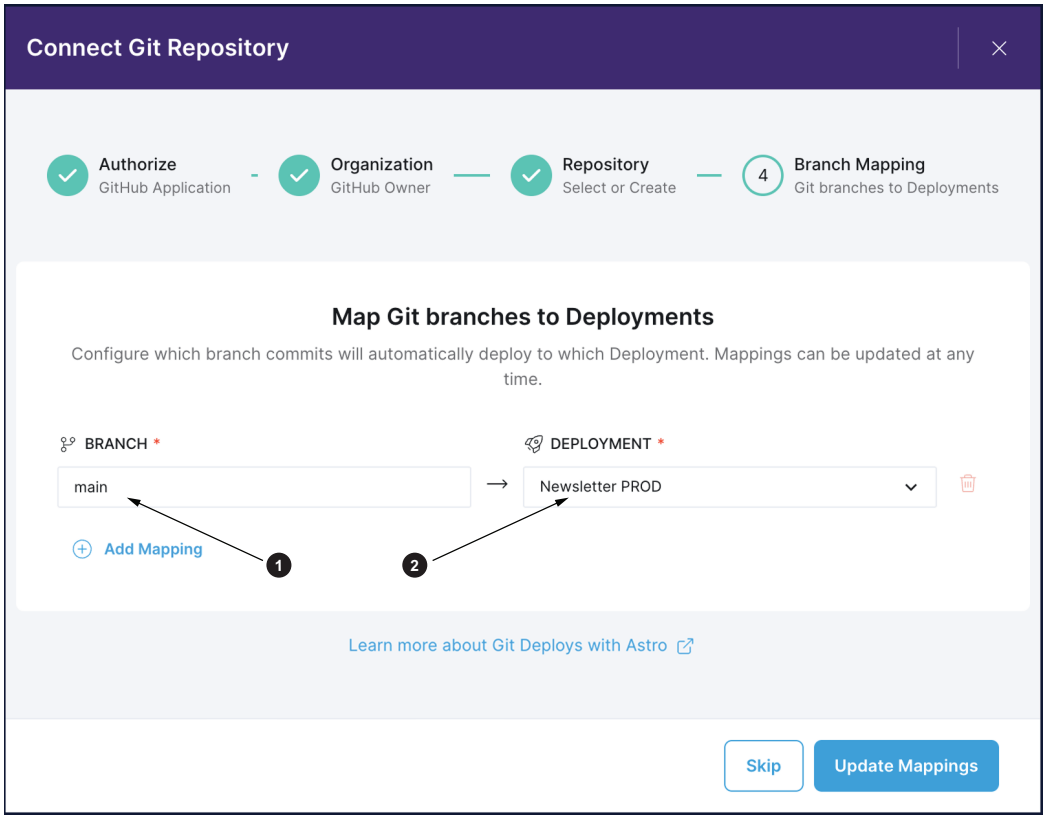


Figure 6.4 The last step of the process of setting up the GitHub integration for an Astro Deployment. Chris mapped the main branch (1) to his Newsletter PROD Deployment (2).

NOTE If you chose to create your first Deployment with example dags, you completed the mapping process in the Astro trial signup flow. In that case, simply replace the example dags in your mapped repository with the dags you want to deploy.

After the branch mapping was set up, Chris triggered the first code deploy from the Astro UI, as shown in figure 6.5. After the initial Deployment, every new commit pushed to the mapped branch of his GitHub repository automatically causes a code deploy to his Astro Deployment.

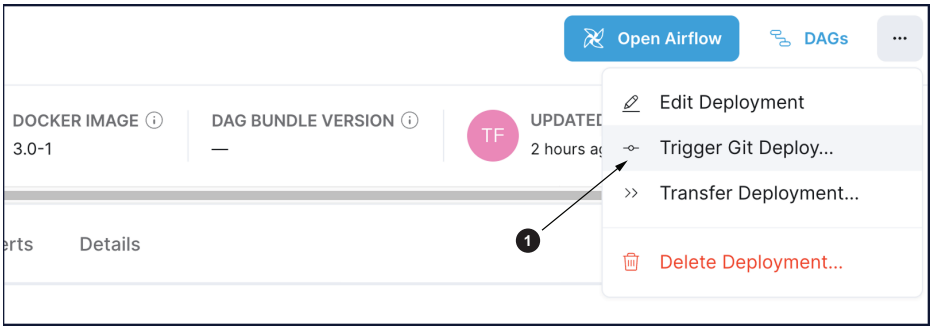


Figure 6.5 Screenshot of the Astro UI showing how to trigger the first Git deploy (1) after setting up the GitHub integration

"But what happens if I make a mistake?" Chris wondered, remembering how often he found himself searching online for ways to undo a commit. Would he be able to undo dag code changes easily in Astro?

Looking through the Astronomer docs, he found the answer. Yes, Astro Deployments can be rolled back to previous versions (https://mng.bz/Qwqm) directly in the Astro UI (figure 6.6).

Figure 6.6 Screenshot of the Astro UI showing how to roll back to a previous version of the Astro Deployment by clicking the drop-down button (1) and then clicking Rollback to this Deploy (2)

6.5 Configuring your Deployment

Having deployed his dags, Chris needed to configure the necessary connections and environment variables to run in the cloud. Luckily, this task involved only a short to-do list:

1. Define the Airflow connection to OpenAI in the Astro Environment Manager.
2. Enable the dags to use S3 instead of local storage to store the newsletters. Thanks to the Airflow Object Storage feature (chapter 2), changing the storage location took only a few steps:
   a. Move the `newsletter_template.txt` file from `include/newsletter` and the user information dictionaries from `include/user_data` to a bucket in Amazon S3.
   b. Add the Airflow Amazon provider with the `s3fs` extra to the Deployment.
   c. Create a connection to Amazon S3 with the connection ID `my_aws_conn`.
   d. Set `OBJECT_STORAGE_CONN_ID` to `my_aws_conn`.
   e. Set `OBJECT_STORAGE_SYSTEM` to `s3`.
   f. Set `OBJECT_STORAGE_PATH_NEWSLETTER` to the path in S3 containing the `newsletter_template.txt` file.
   g. Set `OBJECT_STORAGE_PATH_USER_INFO` to the path in S3 containing the user information dictionaries.

NOTE Although this example uses Amazon S3, you can use the Airflow Object Storage feature to connect to Google Cloud Storage or Azure Blob Storage instead (https://mng.bz/KwBn).

Anticipating an influx of subscribers leading to more data being passed between assets and tasks, Chris decided that it would be wise to set up a custom XCom backend for the Astro Deployment, storing data passed between tasks in S3 instead of the Airflow metadata database.

6.5.1 Install the Amazon provider

Installing additional Python packages, including Airflow providers, in Astro is simple: add them to your `requirements.txt` file, and deploy your change to your Astro Deployment. For Chris, who is using the GitHub integration, this task meant pushing the change to his GitHub repository; the integration took care of the rest.

To use the Airflow Object Storage feature with S3, install the provider with the `s3fs` extra. The following listing shows Chris's requirements file after the addition.

Listing 6.1 Chris's `requirements.txt` file

```
numpy==<version>
apache-airflow-providers-common-io==<version>
geopy==<version>
```

```
apache-airflow-providers-openai==<version>
apache-airflow-providers-amazon[s3fs]==<version>
```

Installs the Amazon Airflow provider with the s3fs extra. Replace <version> with the latest provider version.

NOTE Always pin your provider versions. You can find the latest version numbers for all providers maintained by the open source Airflow community in the Airflow documentation (https://airflow.apache.org/docs).

6.5.2 Define connections in Astro

In Astro, you can define Airflow connections the same way you did in chapter 4: by setting them in the Airflow UI. But you have a more convenient way.

Astro comes with a built-in secrets manager, the Astro Environment Manager. You can store connection credentials in the manager to be used in multiple Deployments in an Astro Workspace and override individual fields for each linked Deployment. You can even pull them down to work with a local Astro CLI development environment (https://mng.bz/X7rY).

Figure 6.7 shows how to navigate to the Connections list of the Astro Environment Manager. For many connections, the Astro Environment Manager contains prebuilt forms that make it easier to tell which values to provide. If no form is available for a specific connection, you can choose to create a generic connection. Make sure to link your connection with your Astro Deployment, either by toggling Automatically Link to All Deployments to On or by linking a connection to a specific Deployment after it is created.

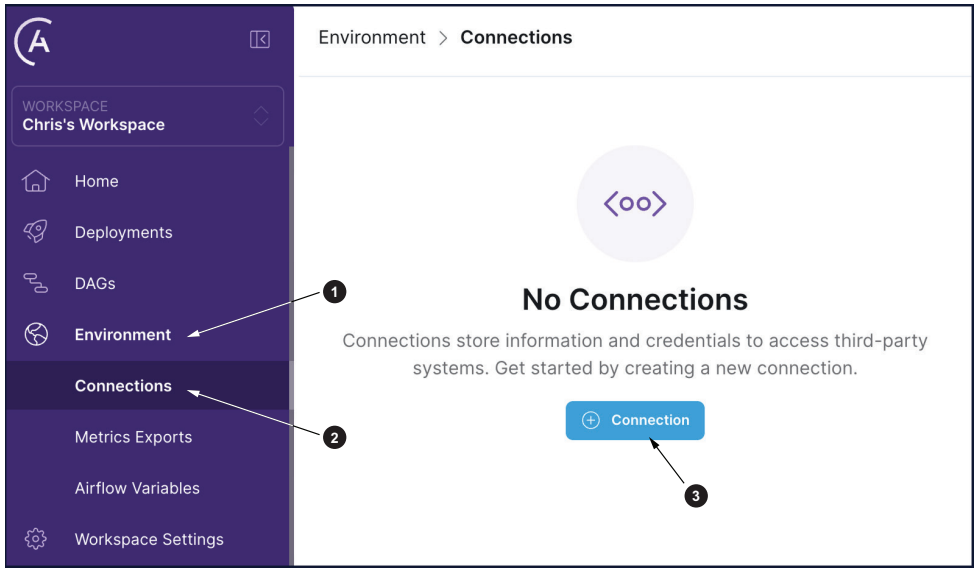


Figure 6.7 You can add connections to the Astro Environment Manager by clicking the Environment menu (1), the Connections (2) item, and the blue Connection button (3).

Chris added two connections: one to OpenAI and the other to Amazon S3. For the connection to OpenAI, he selected the generic connection form and entered the following values, leaving all other fields blank (figure 6.8):

- Connection ID—`my_openai_conn`
- Type—`openai`
- Password—OpenAI API key (https://platform.openai.com)

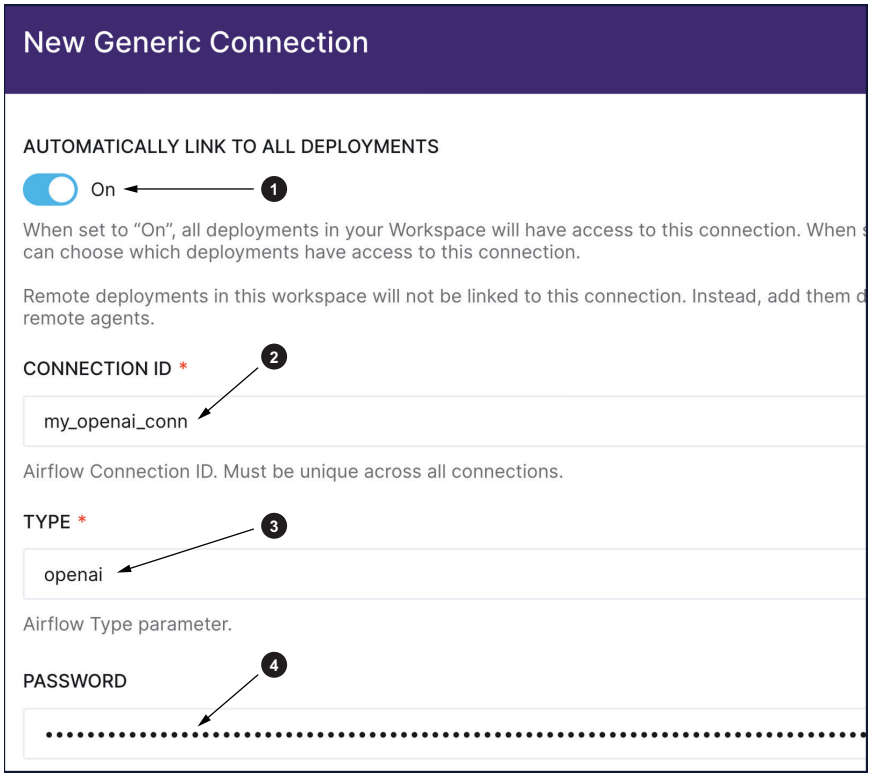


Figure 6.8 Connection form for the connection to OpenAI. The connection is set to be available to all Deployments in the Workspace (1). The connection ID (2), type (3), and OpenAI API key (4) are entered in the appropriate fields.

For the Amazon connection, Chris had two options: use a static access key or assume an identity and access management (IAM) role. For the time being, he went with a static key, but he made sure to put switching to a more secure authentication method on his to-do list. See the Amazon documentation for more information on authentication mechanisms (https://mng.bz/yNJJ).

For a connection based on an access key, Chris had to provide the following values:

- *Connection ID*—`my_aws_conn`
- *Access Key ID*—`<AWS Access Key ID>`
- *Secret Access Key*—`<AWS Secret Access Key>`

NOTE Astro has advanced features to authenticate to cloud resources. You can learn more in the Astro documentation on workload identities (https://mng.bz/Mwv2).

Determining connection credentials

When using Airflow connections, you often have multiple ways to connect with your tool of choice. A good strategy for learning about connection options, such as those for Amazon, is to check out the documentation on the relevant Airflow provider package that contains the hook and operators that will use the connection in question (https://mng.bz/a9ZJ).

Remember that even if no provider package exists for a specific tool, you can still use Airflow to orchestrate actions in it by connecting directly to the tool's API from within your task code. For that purpose, it is a common practice to store credentials such as API keys as environment variables (see section 6.5.3).

6.5.3 Define environment variables in Astro

With the connections to Amazon and to OpenAI defined, Chris needed to set his environment variables:

- `OBJECT_STORAGE_CONN_ID` to `my_aws_conn`
- `OBJECT_STORAGE_SYSTEM` to `s3`
- `OBJECT_STORAGE_PATH_NEWSLETTER` to `<bucket-name>/<path-to-newsletter-template>`
- `OBJECT_STORAGE_PATH_USER_INFO` to `<bucket-name>/<path-to-user-info-dictionaries>`

You can add environment variables directly to a specific Deployment in the Astro UI by following the instructions in figure 6.9. If you prefer to set your environment variables programmatically, you can add them in your Dockerfile (https://mng.bz/gm1R). If your environment variable is a secret, use the Astro UI and mark the environment variable as secret to store it securely.

6.5.4 Custom XCom backend

Although he was briefly tempted to call it a day, Chris knew that to make the pipeline truly scalable, he had one last task: saving the data being passed between assets and tasks (XComs) in object storage instead of the limited Airflow metadata database. To accomplish this, he needed to define a custom XCom backend. Without a custom

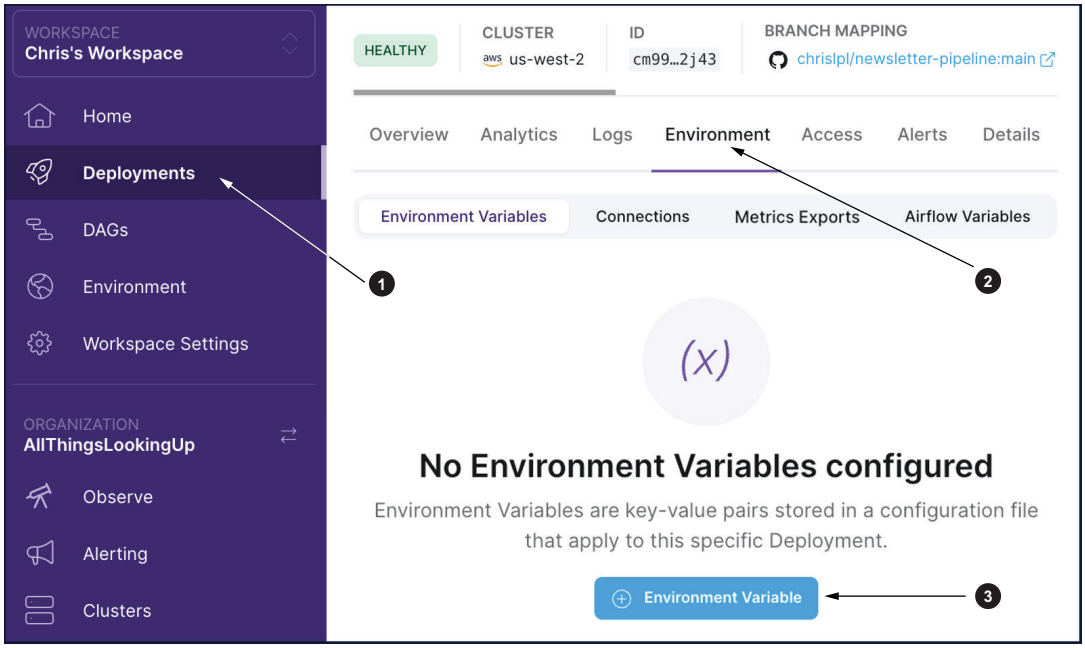


Figure 6.9 To add a new environment variable to a Deployment, click the Deployments tab (1) and select your Deployment. Then click the Environment tab (2) and the blue + Environment Variable button (3).

XCom backend, over time, the Airflow metadata database would fill with XCom and eventually wouldn't have space to store information important to Airflow's functioning, such as task statuses. Let's avoid that!

Following Astronomer's guide (https://mng.bz/eBPJ), Chris used a second S3 bucket in the same Amazon Web Services (AWS) account to store his XComs. Thanks to the `XComObjectStorageBackend`, that process was a simple matter of setting the right environment variables to the Astro Deployment:

- `AIRFLOW__CORE__XCOM_BACKEND` to `airflow.providers.common.io.xcom.backend.XComObjectStorageBackend`
- `AIRFLOW__COMMON_IO__XCOM_OBJECTSTORAGE_PATH` to `s3://my_aws_conn@<my-bucket>/<path>`
- `AIRFLOW__COMMON_IO__XCOM_OBJECTSTORAGE_THRESHOLD` to `64`
  (XCom larger than 64 bytes are stored in S3; smaller XCom are stored in the Airflow metadata database.)

Be aware that the Object Storage XCom backend needs the Common IO provider (https://mng.bz/pZ9P) to be installed in your Airflow environment.

NOTE You have other options for defining custom XCom backends for more fine-grained control of XCom, such as custom serialization. See the Learn documentation for more information (https://mng.bz/OwQP).

6.6 Observability with Astro Observe

Turning his pipeline on and getting a celebratory slice of black forest cake, Chris started to wonder how he could best make sure that the newsletter would never be late. He wanted to be alerted if any tasks were going to fail. So he quickly found the Alerting tab and set a failure alert for all his tasks (https://www.astronomer.io/docs/astro/alerts), including an AI-provided failure summary. But what would happen if a task was delayed instead of failing?

"This is like being back at Emile Wealth Nurture Services," Chris thought, remembering the challenges he encountered at his previous job; an SLA required that the compliance report be sent out before 9 a.m. He wondered whether there was a way to set and track such an SLA in Airflow. After some searching, he landed on the product page for Astro Observe (https://www.astronomer.io/product/observe), which promised timeliness and freshness SLAs on Airflow pipelines.

Already an Astro customer, Chris was able to get Astro Observe enabled for his organization quickly. Then he created his first Data Product, as shown in figure 6.10.

After creating the Data Product, Chris set a timeliness SLA on it (figure 6.11). Because the great majority of subscribers were based in the United States, he wanted to make sure that the newsletters were ready to be sent out by 5 a.m. U.S. Eastern Time so that subscribers would get their motivational quotes first thing in the morning.

Last, Chris added both an SLA violation and a proactive SLA alert. The violation alert would send him an email if the personalized newsletters were not ready by 5 a.m. Eastern Time. For the proactive SLA alert, Chris chose to be notified if the upstream `formatted_newsletter` asset had not materialized by 4 a.m. Eastern Time, which would give him enough time to debug (or find quotes by himself) in case of problems.

TIP To learn more about Astro Observe, contact Astronomer to experience a demo (https://www.astronomer.io/product/observe).

Having set all his various alerts, Chris knew he'd sleep much better that night because he didn't need to worry about his pipelines failing silently or being delayed.

More Astro features

This chapter covers the basics of running Airflow pipelines and creating Observe Data Products in Astro. Many more features are available, such as worker queues (which allow users to assign individual tasks to a worker of the right size) and detailed cost breakdown dashboards for customers at the Enterprise Tier. You can learn more about all Astro features in the Astro documentation (https://www.astronomer.io/docs/astro).

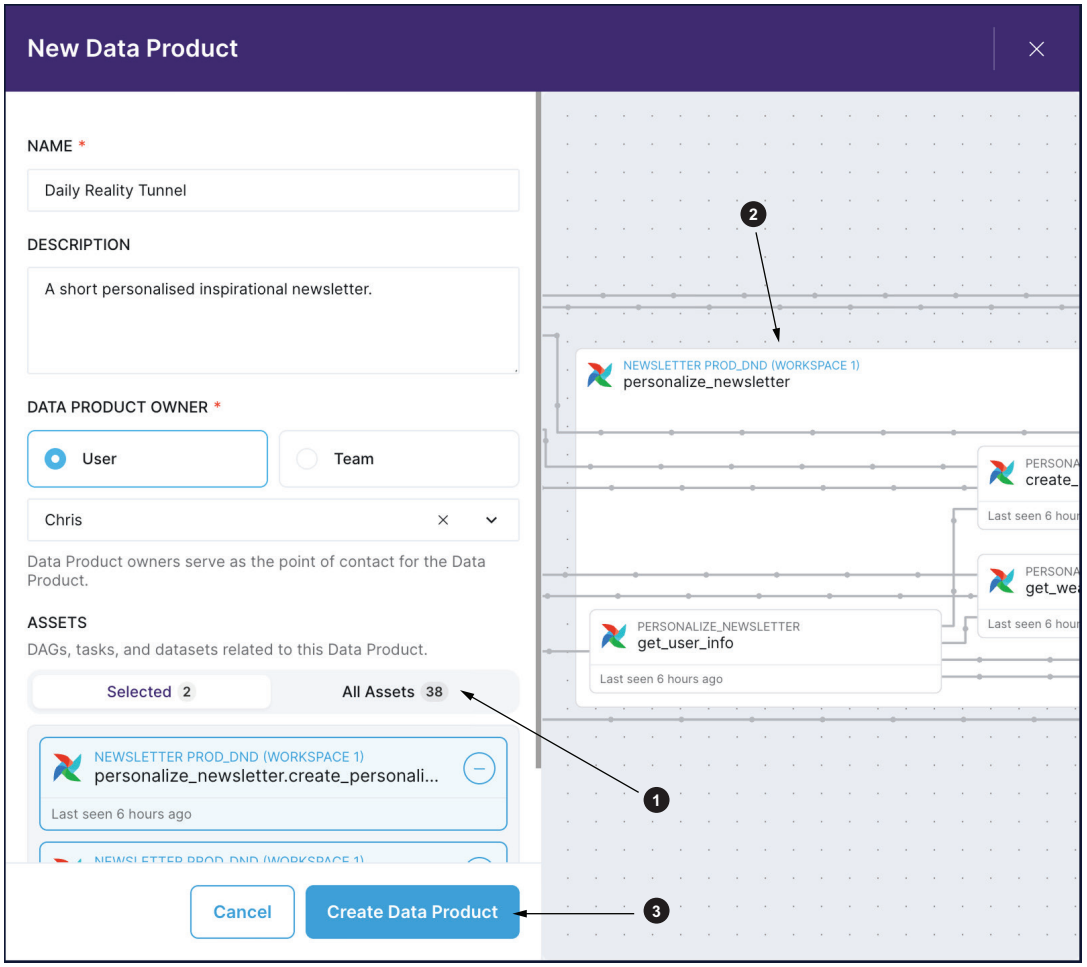


Figure 6.10 Astro Observe UI showing options for defining a Data Product. You can select assets from your Airflow pipelines (1) to be included in the Data Product; upstream dependencies are automatically inferred and added to the Observe graph (2). When your Data Product is ready, you can save it (3) to start defining SLAs related to it.

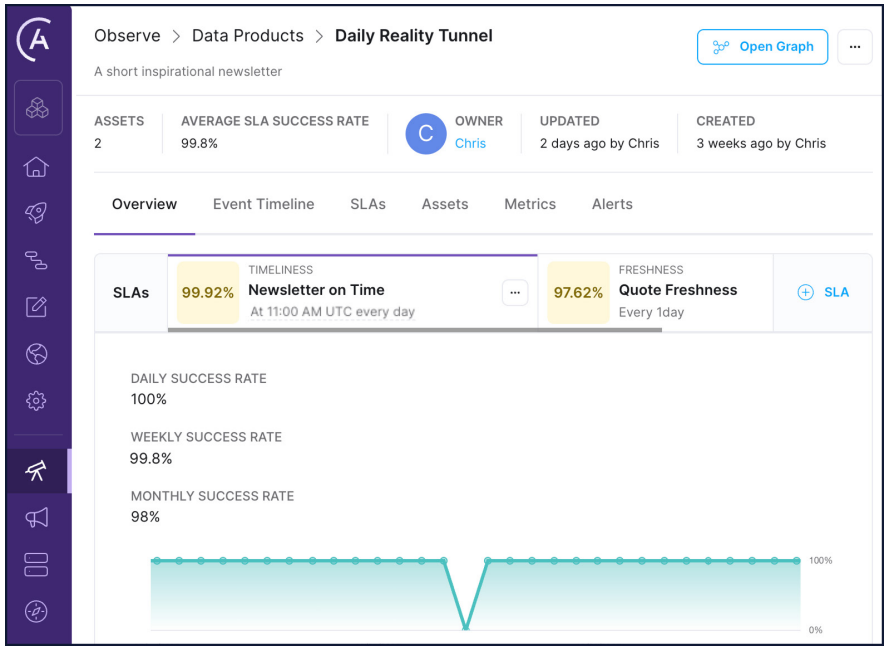


Figure 6.11 Screenshot of a timeliness SLA called Newsletter on Time set at 11 a.m. UTC (5 a.m. Eastern Time)

Summary

- Astro is a fully managed, unified DataOps platform powered by Apache Airflow. A free trial is available.
- You can manage Astro Deployments in the Astro UI or by using the Astro API or the Astro Terraform provider.
- You have three main ways to deploy code to an Astro Deployment: using the Astro CLI, using the GitHub integration, or using other CI/CD services with provided template scripts.
- Astro comes with a built-in secrets manager called the Astro Environment Manager, where you can define all connections for a Workspace and link them to as many Deployments as you want.
- You can define environment variables for Astro Deployments in the Astro UI or in your Dockerfile.
- You can create a custom XCom backend with environment variables by using the `XComObjectStorageBackend` of the Common IO provider.
- Astro Observe enables you to track your asset catalog, define business-level data products, track upstream dependencies, set and track freshness and timeliness SLAs, and configure alerts to detect when an SLA is at risk.

7 Inference execution

This chapter covers

- The concept of inference execution
- Airflow 3 dags used as a backend for GenAI applications in a push or pull pattern

"We got 150 new subscribers in a week, and we haven't even started promoting your new newsletter. Way to make a first impression!" Jan said during the content-sync call with Chris in his second week on the job. "I was wondering, because you are the Airflow expert, do you think it would be possible to add a form on our website where people can enter their information to get a sample newsletter? I think that would be the best promotion—getting a personalized preview like that."

Chris responded optimistically but cautiously, saying, "That is a great idea! Let me get back to you on that." He'd had a similar thought while reading about event-driven scheduling (chapter 3), but he wanted to be careful not to overpromise until he evaluated the feasibility of this request.

7.1 Drafting an inference execution dag

Jan was asking for inference execution for a generative AI (GenAI) application. The question was whether Airflow was up to the task. After reading recent blog posts about Airflow 3 and its applications, Chris knew the answer to the question: yes. As of Airflow 3, you can use a dag as the backend for a GenAI application.

The key change that enables inference execution use cases is allowing the logical date to be `None`. In Airflow 2, each dag run had to have a unique timestamp attached as a logical date (known as execution date in older Airflow versions). There couldn't be two runs of the same dag for the same point in time with the same inputs. This situation prevented inference execution use cases because two user requests could easily come in simultaneously. Airflow 3 solves this problem by no longer requiring dags to have a logical date; the logical date can be set to `None`.

Further, Airflow 2 was built primarily for running dags based on data intervals and often required workarounds to run other types of dags. Airflow 3 made key changes in the execution mechanism to better support other types of dags (chapter 8).

True simultaneous dag runs with the same inputs allow Chris to create a form on the *AllThingsLookingUp* website where users can request a test newsletter generated by an Airflow dag. The form would ask for the following user information:

- *Name*—To create a personal greeting.
- *Current location*—To create a weather report for the user (chapter 2).
- *Personal motivation and favorite sci-fi character*—To generate a personalized quote. This information is used in the call to the OpenAI API to ask the large language model (LLM) to generate a personalized quote related to what motivates this specific user, delivered in the voice of the user's favorite sci-fi character.

The only question is which pattern to choose to implement this pipeline. Airflow 3 has two patterns for implementing inference execution:

- *Push pattern*—Use the Airflow `REST` API to send information to trigger a dag run. In Chris's use case, the `REST` API call would be triggered by clicking the Submit Form button on the website.
- *Poll pattern*—Use event-driven scheduling (chapter 3) to poll for a message that is posted to a message queue service and trigger the dag based on the appearance of that message. In Chris's use case, clicking the Submit Form button on the website would post the message to the message queue.

For the first pattern, the information given by the user, such as their location and favorite sci-fi character, would be sent to the dag as part of the API call. Figure 7.1 shows a push-pattern implementation in the inference execution pipeline Chris was asked to create.

In this architecture, the user enters their information in the website form. Submitting the form sends a `POST` request to the Airflow environment using the Airflow `REST` API (https://mng.bz/dWww). The submitted information is included in this API call.

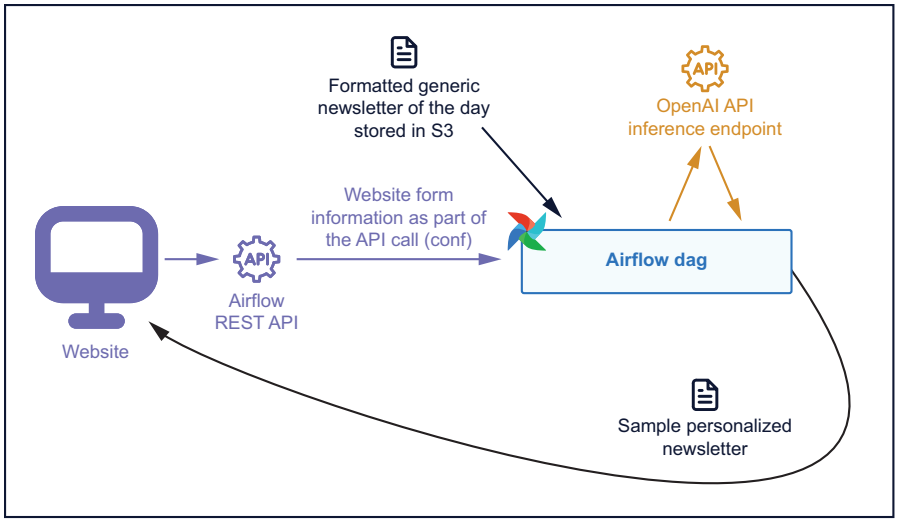


Figure 7.1 Architecture diagram showing the high-level architecture of an Airflow dag used for inference execution in a push pattern

With only a few modifications, Chris could use the same dag he had built in chapter 4: it uses the formatted generic newsletter of that day as a template and adds the personalized information, including the user's name, local weather (gathered from the Open-meteo API as in chapter 2), and the personalized quote, generated by sending the day's quotes alongside the user's motivation and favorite sci-fi character to the OpenAI API (chapter 4). When the personalized newsletter is ready, it's sent back to the website to be served to the user.

By contrast, the poll pattern uses event-driven scheduling instead of an API call. An `AssetWatcher` in the Airflow environment asynchronously polls for a new message posted to a message queue. The information that the user entered on the website is part of the message in the message queue; the rest of the pipeline stays the same (figure 7.2). You can find an example using the poll pattern in the companion GitHub repository (https://github.com/astronomer/practical-guide-to-apache-airflow-3).

Whichever pattern you use, the great advantage of using Airflow as the backend of a GenAI application is that you don't need to create additional backend components. You can reuse the batch inference pipeline you already have in your Airflow environment for near-real-time inference. You can also use the same dags as in your regular pipelines with all the advantages that Airflow brings to workflow orchestration, such as these:

- *Automatic task retries* (chapter 3)—Protects against transient API failures and rate limits.
- *Observability of the inference*—Runs in the Airflow UI (chapter 4).
- *Interconnectivity with the rest of your data environment*—The user input could be checked against specific criteria that cause additional dags to run. If a user

indicates that their favorite sci-fi character is from the TV series *Travelers*, for example, you might kick off a dag that sends them an email inviting them to subscribe to a newsletter that suggests additional great time-travel media to watch.

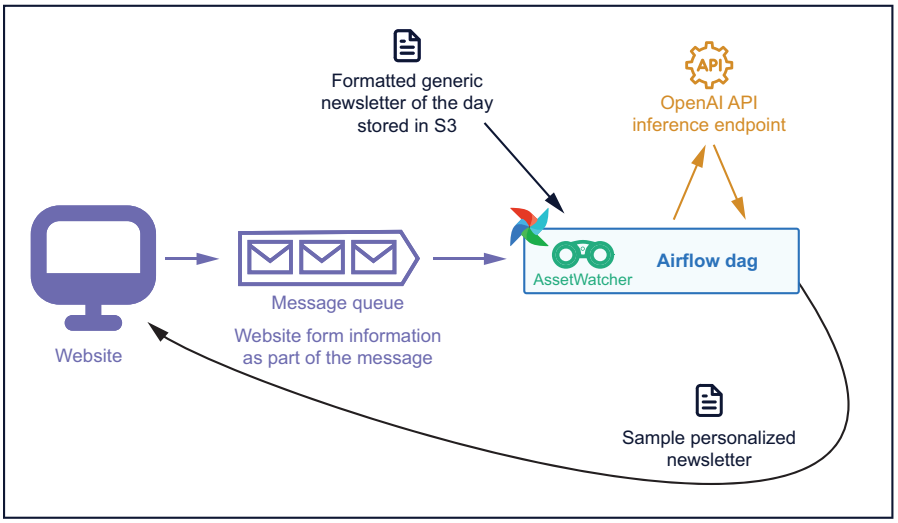


Figure 7.2 Architecture diagram showing the high-level architecture of using an Airflow dag for inference execution in a poll pattern

Knowing this, Chris was able to get back to Jan with two implementation options. He certainly wouldn't run out of projects on this job any time soon!

Real-life use cases: Airflow for AI and machine learning

This chapter describes only one way to use Airflow in AI applications. Many companies already use Airflow to orchestrate data processing for and training of machine learning (ML) models. Following are a few examples:

- *Ford Motor Company* (https://mng.bz/rZ2y) uses Airflow to orchestrate workflows training ML models for autonomous driving systems.
- *Laurel* (https://mng.bz/V98G), a company that offers automated timekeeping systems for lawyers, uses Airflow to train many ML models in parallel for individual customers.
- *ASAPP* (https://mng.bz/xZrX) processes large amounts of unstructured data, such as audio files containing speech, to offer AI solutions for customer support.

Summary

- Airflow 3 allows the logical date to be `None`, permitting you to trigger the same dag multiple times with the same parameters. This change, among others, enables inference execution pipelines, essentially using a dag as the backend of a GenAI application.
- You can create inference execution pipelines in a push pattern, using the Airflow `REST` API, or in a poll pattern, using event-driven scheduling.
- Inference execution is one of many situations in which Airflow is used to orchestrate ML- and AI-based applications.

8 Upgrading: Airflow 2 to 3

This chapter covers

- Learning about key breaking changes between Airflow 2 and Airflow 3
- Using `ruff` to check your Airflow 2 dags for any changed syntax or moved imports
- Using the `airflow config lint` command to prepare for an upgrade to Airflow 3

This chapter covers some of the most important breaking changes and upgrading utilities. For a full list of all changes between Airflow 2.10 and Airflow 3, see the Airflow release notes (https://mng.bz/mZB8).

8.1 Planning to upgrade

Another month had passed, and Chris's newsletter was a clear hit with a growing subscriber base. One subscriber sent this feedback to the newsletter: "Who doesn't want their favorite sci-fi character to motivate them each morning?"

After checking with the rest of the team to make sure he was allowed to share some of the magic from behind the scenes, Chris posted a screenshot of his newsletter

pipeline on a professional social media platform. Among many positive comments was one from Eléonore, Chris's manager at his previous job, asking whether he'd be willing to share his Airflow 3 knowledge to help her migrate some dags from Airflow 2 (figure 8.1).

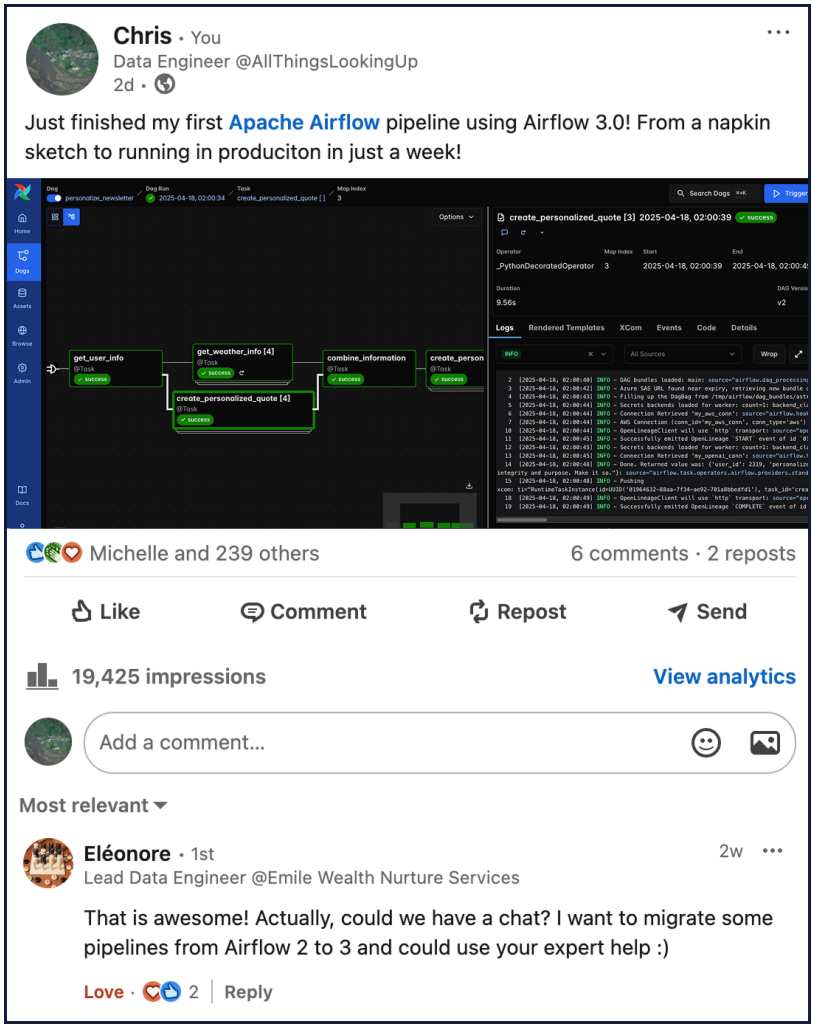


Figure 8.1 Screenshot of a social media post by Chris, showing his newsletter pipeline and Eléonore's request for migration help

Chris replied immediately that he was happy to help. When reading up on Airflow 3, he came across a migration tutorial (https://mng.bz/eBdG), and this request seemed like a great opportunity to test it.

In this chapter, you'll learn about the most important breaking changes between Airflow 2 and 3. You'll see how to use the Airflow migration tools to migrate your Airflow dag code as well as your Airflow configuration file. The Airflow developers took great care to keep as much backward compatibility as possible, making the upgrade process as efficient and smooth as it can be.

Minimum Airflow version to upgrade

To upgrade from Airflow 2 to Airflow 3, you need to have at least Airflow version 2.6.3 (Astro Runtime 8.7.0) due to database-migration constraints. We recommend upgrading to the most recent Airflow 2 version before your migration to Airflow 3 so that you can to benefit from and resolve all deprecation warnings that appear in your logs.

If you are still using Airflow 1, we highly recommend upgrading to Airflow 2 as soon as possible. Maintenance of Airflow 1 ended on June 17, 2021, so no further updates are being made, and potential security problems in Airflow 1 are not being addressed. After upgrading to Airflow 2, upgrade to Airflow 2.6.3+; then upgrade to Airflow 3 as explained in this chapter. For information on upgrading from Airflow 1 to Airflow 2, see the Airflow documentation (https://mng.bz/9yK8).

8.2 Important breaking changes

Many of the exciting new features in Airflow 3—particularly improved security through task isolation and remote execution (both covered in chapter 5)—necessitated significant changes to the Airflow architecture and codebase.

Knowing that most of these changes are internal to Airflow and do not break existing dag code, Chris started the "upgrading party" call by telling Eléonore, "Many Airflow 2 dags will run on Airflow 3 without any code change, so this is going to be quicker than you think!"

Some breaking changes were necessary, though, and it is good to be aware of them in case any dag or task must be updated as a result. This section explains some of the most important changes between Airflow 2 and 3 and offers recommendations for engineers who are looking to upgrade their Airflow 2 pipelines.

8.2.1 Removal of direct database access

In Airflow 2, all tasks had direct access to the Airflow metadata database. This access was removed in Airflow 3, greatly improving Airflow's security posture. If you are accessing the Airflow metadata database directly in any of your task or trigger code, such as by using the `SQLAlchemy` connection environment variable, that process will error in Airflow 3. This includes custom operators making such a connection.

Recommendation: Directly accessing the Airflow metadata database from within tasks is an antipattern because it could lead to accidental modifying or dropping of information that is vital to Airflow's functioning, up to and including corruption of your entire

Airflow instance. To interact with and retrieve information about your Airflow instance, use the Airflow REST API instead (https://mng.bz/Ow2j).

8.2.2 Changes related to scheduling

Airflow 3 introduces a few changes related to the scheduling of dags. Following are the most important:

- Changes to the scheduling defaults for a dag (for scheduling parameters that are not explicitly specified in the dag definition)
- Changes to the `logical_date` and `run_id` attributes of dag runs
- Changes to the default timetable used to interpret raw cron string schedules
- Removal of the deprecated dag parameters `schedule_interval` and `timetable` in favor of the `schedule` parameter
- Removal of the deprecated `days_ago` function in favor of `pendulum.today('UTC').add(days=-N, ...)`

This section explains these changes in detail and offers recommendations on adapting existing Airflow dags.

DEFAULT SCHEDULING PARAMETERS

In Airflow 2.10, the default scheduling values for a dag when no others are provided were

- `schedule=timedelta(days=1)`
- `catchup=True`

In Airflow 3, these defaults are

- `schedule=None`
- `catchup=False`

Recommendation: For any dag that you want to run with a `timedelta(days=1)` schedule, you have to set the `schedule` parameter in the dag definition accordingly. If your dags depend on catching up by default, you can enable Airflow 2 behavior by setting the Airflow config `[scheduler].catchup_by_default=True`.

LOGICAL DATE AND DAG RUN ID

In Airflow 3, the date attached to the dag run—referred to as the `logical_date` (`execution_date` in older Airflow versions)—has been made to behave . . . well, more logically.

In Airflow 2, the `logical_date` for scheduled runs (as opposed to manual runs) was equal to the `data_interval_start` date of the dag. For a dag that was running on a `@daily` schedule (every day at midnight), for example, the dag run that would run on January 29, 2025, at 00:00 Coordinated Universal Time (UTC) had the `logical_date 2025-01-28 00:00` UTC because the `data_interval_start` for the dag was when the *previous* dag run finished. The date displayed in the Airflow UI next to the dag run would be `2025-01-28 00:00` UTC, and the `run_id` of the dag would be

`scheduled__2025-01-28T00:00:00+00:00`, even though the dag ran at midnight January 29. This often led to confusion, especially with more irregular schedules.

It is much more intuitive to identify a dag run by the point in time after which it is allowed to run and subsequently starts its run. This is why in Airflow 3, the `logical_date` is equivalent to the `run_after` date, and the `run_id` takes its timestamp from the moment in time when the dag run is queued. Table 8.1 shows a side-by-side comparison of the timestamps associated with a dag run of a `@daily` dag in Airflow 2 and Airflow 3.

> **NOTE** Airflow 3 also allows users to explicitly pass `None` as the logical date of a dag. In this case, the `data_interval_start` and `data_interval_end` will be `None` as well.

Recommendation: This change affects mostly dags that use the `logical_date` or `run_id`, retrieved from the Airflow context, in their task logic. If you use the `logical_date` to segment data pulled from a database or use it as part of the name of a generated file, for example, check table 8.1 to determine whether you must make any changes in your dags.

If you use the `data_interval_start` or `data_interval_end` date in your dags, make sure that you also read the section "Timetable to interpret raw cron strings" later in this chapter. Also note that there have been changes to the Airflow context, including to timestamp keys. The location and existence of some keys might depend on how you trigger your dag. See the Learn guide on the Airflow context (https://mng.bz/64QR) for more information.

Table 8.1 Comparisons of timestamps for a scheduled 5-minute run of a `@daily` dag

Element	Airflow 2	Airflow 3
`run_after` The moment a dag gets scheduled; it cannot run *before* this point in time	`2025-01-29 00:00` UTC	`2025-01-29 00:00` UTC
`logical_date` If not explicitly set to `None`, the `logical_date` has to be unique per dag.	`2025-01-28 00:00` UTC	`2025-01-29 00:00` UTC Now equivalent to the `run_after` date unless explicitly set to `None`
`queued_at`	`2025-01-29 00:01` UTC It can take a moment for a dag to get queued after it has been scheduled.	`2025-01-29 00:01` UTC It can take a moment for a dag to get queued after it has been scheduled.
The moment the dag starts running	`2025-01-29 00:02` UTC *Started* in the Airflow UI. It can take a moment for a dag to go from queued state to running.	`2025-01-29 00:02` UTC *Start* in the Airflow UI. It can take a moment for a dag to go from queued state to running.

Table 8.1 Comparisons of timestamps for a scheduled 5-minute run of a `@daily` dag (*continued*)

Element	Airflow 2	Airflow 3
The moment the dag finishes	`2025-01-29 05:00` UTC *Ended* in the Airflow UI	`2025-01-29 05:00` UTC *End* in the Airflow UI
`data_interval_start`	`2025-01-28 00:00` UTC	`2025-01-29 00:00` UTC with the `CronTriggerTimetable` (default) `2025-01-28 00:00` UTC with the `CronDataIntervalTimetable` Will be set to the `run_after` date for runs that are not scheduled, such as asset-triggered or manual runs. If the `logical_date` is explicitly set to `None`, the `data_interval_start` is also `None`.
`data_interval_end`	`2025-01-29 00:00` UTC	`2025-01-29 00:00` UTC Will be set to the `run_after` date for runs that are not scheduled, such as asset-triggered or manual runs. If the `logical_date` is explicitly set to `None`, the `data_interval_start` is also `None`.
`run_id` of the dag run (`dag_run_id`)	`scheduled__2025-01-28T00:00:00+00:00`	`scheduled__2025-01-29T00:00:00+00:00` with the `CronTriggerTimetable` (default) `scheduled__2025-01-28T00:00:00+00:00` with the `CronDataIntervalTimetable` For runs that are not scheduled (e.g., manual or asset-triggered runs), the `run_id` uses the `run_after` timestamp. For runs where the `logical_date` is explicitly set to `None`, a random string is appended to the `run_id` to ensure uniqueness.

TIMETABLE TO INTERPRET RAW CRON STRINGS

In Airflow 2, providing a raw cron string such as `"0 0 * * *"` to the `schedule` parameter led to the instantiation of a schedule using the `CronDataIntervalTimetable` under the hood. This timetable generates a `data_interval_start` date that is equivalent to the previous dag run's `data_interval_end` date. This `data_interval_start` date is used as the `logical_date`, whereas the point in time after which the dag can run is the `data_interval_end` timestamp.

In Airflow 3, this default has changed to the `CronTriggerTimetable`. The `CronTriggerTimetable` does not include a data interval, which means that `data_interval_start`, `data_interval_end`, and `logical_date` are the same: the point in time after which the dag can run.

Recommendation: If your dags that are scheduled on raw cron strings depend on the data interval for their internal data partitioning logic, you can set the default timetable used by cron schedules back to the Airflow 2 default (`CronDataIntervalTimetable`) using the `[scheduler].create_cron_data_intervals` configuration (https://mng.bz/5va7).

You can also be explicit about which of the two timetables to use directly in your dag's schedule by setting `schedule=CronDataIntervalTimetable("0 0 * * *", timezone="UTC")` or `schedule=CronTriggerTimetable("0 0 * * *", timezone="UTC")` respectively.

TIP For more information, see the Airflow documentation on trigger versus data interval timetables (https://mng.bz/YZPj).

8.2.3 Other important changes

Airflow 3 introduces other improvements and changes that may affect you if you use any of the related features. See table 8.2 for an overview of changes and related migration recommendations.

Table 8.2 Other important changes

Aspect	What changed	Recommendation
`REST` API	Airflow 3 comes with a new version of the Airflow `REST` API (version 2) to match new features. The long-deprecated experimental API used in Airflow 1 has been removed.	Refer to the updated Airflow `REST` API documentation (https://mng.bz/0w2j) to adjust your endpoints.
Airflow context	There were changes to a few elements of the Airflow context, including these: ■ `conf`: was removed. ■ `dag_run.is_backfill` and `.external_trigger` were removed. Additionally, there were several changes to context keys relating to dag run timestamps, and all context keys referencing the deprecated `execution_date` were removed.	If you use these context parameters, make the following adjustments: ■ Use `airflow.configuration.conf` in your task code instead. ■ Use `dag_run.run_type` instead. ■ Use alternative keys such as `prev_start_date_success` or `prev_data_interval_end_success` instead of `prev_execution_date`, depending on your use case. See related documentation for more information about the Airflow context (https://mng.bz/64QR) and templates in Airflow (https://mng.bz/GwWA).
Subdags	The long-deprecated subdags feature has been removed in Airflow 3.	Use task groups instead (https://www.astronomer.io/docs/learn/task-groups) and/or consider splitting larger dags into smaller ones connected by Airflow assets.

Table 8.2 Other important changes (*continued*)

Aspect	What changed	Recommendation
Service-level agreements (SLAs)	The Airflow SLA feature has been removed in Airflow 3.	Astronomer customers can use Astro Alerts (https://www.astronomer.io/docs/astro/alerts) or the SLA feature in Astro Observe (https://www.astronomer.io/docs/astro/astro-observe) instead. For OSS Airflow, a new feature called Deadline Alerts (https://mng.bz/z2M6) is planned for a release after 3.0.
Flask-AppBuilder (FAB)	In Airflow 2, `FabAuthManager` was the default auth manager, and the Airflow UI was built using FAB. In Airflow 3, reliance on FAB for core Airflow components was removed to enhance security and make Airflow easier to maintain. In Airflow 3, the default auth manager is `SimpleAuthManager`, and the UI is React-based.	If you need FAB integration, install the FAB provider (https://mng.bz/oZPD). For more information on auth managers, see the Airflow documentation (https://mng.bz/nZPd).
XCom pickling	XCom pickling is no longer allowed when using the default XCom backend in Airflow 3 for security reasons. The `[core].enable_xcom_pickling` configuration was removed. Existing pickled XCom are moved to an archive table.	Use a custom XCom backend with custom serialization to pass data between tasks that Airflow cannot serialize by default. Refer to the Python docs for an explanation of the security implications of pickling (https://docs.python.org/3/library/pickle.html). See the Custom XCom backend Learn guide for instructions on setting up a custom XCom backend (https://mng.bz/0zWE).
Email/SMTP	The email/SMTP integration in Airflow core was deprecated and will be removed in Airflow 4. The `airflow.operators.email.EmailOperator` was removed in Airflow 3.	For new dags, use the SMTP provider with the `SmtpNotifier` instead. See the notification Learn guide (https://mng.bz/Kw1g) for more information. Use the `EmailOperator` from the SMTP provider package (https://mng.bz/9yOq) instead.
`max_active_tasks`	The `max_active_tasks` dag parameter now sets the maximum active task instances per dag run instead of across all dag runs.	Consider whether you want to adjust the value for this parameter if you use it.
Trigger rules	The deprecated trigger rule `dummy` was removed. The deprecated trigger rule `none_failed_or_skipped` was removed.	Use `always` instead. Use `none_failed_min_one_success` instead. See the trigger rules Learn guide for an up-to-date list of available trigger rules (https://mng.bz/jZO8).
`.airflowignore`	The default syntax for the `.airflowignore` file changed from `regexp` to `glob`.	To revert to the Airflow 2 behavior, set `[core].dag_ignore_file_syntax` to `regexp`.

Table 8.2 Other important changes (*continued*)

Aspect	What changed	Recommendation
Airflow providers	Depending on how you run Airflow, you may find that some Airflow providers (such as FTP, HTTP, and IMAP) that used to be preinstalled in your image/package for Airflow 2 are not preinstalled in Airflow 3.	Pip-install the needed providers in your Airflow environment. See the Airflow documentation for a list of officially supported providers (https://airflow.apache.org/docs).
Python	Support for Python 3.8, which reached its end of life in October 2024, was removed in Airflow 3.	Upgrade to Python 3.9+.
Postgres	Support for Postgres 12, which reached its end of life in November 2024, was removed in Airflow 3.	Upgrade to Postgres 13+.

8.3 How to upgrade

"All right, that wasn't too bad," Eléonore said after she and Chris went through the changes in section 8.2 one by one. "We'll have to update our `REST` API calls, but other than that, we've kept up with deprecation warnings, so no major changes are needed based on that list. I like that the `logical_date` in the Airflow UI makes more sense now."

"Then let's start upgrading one of the dev environments as a test run step by step," Chris responded. Together, he and Eléonore drafted a to-do list for upgrading:

1. Make sure that your current Airflow environment is at least version 2.6.3 (Astro Runtime 8.7.0). This version is the oldest one from which upgrading to Airflow 3 is possible, due to changes in the Airflow metadata database starting with this version. We recommend upgrading to the latest Airflow 2 version before you migrate to Airflow 3 so that you can benefit from and resolve all deprecation warnings that appear in your logs.
2. Use the Airflow ruff rules to check your Airflow dag code (section 8.3.1), and make all necessary changes.
3. Use the Airflow config linter to check your Airflow config (section 8.3.2), and make all necessary changes.
4. Assess whether you must make additional changes to the dags based on the list in section 8.2 and the Airflow release notes.
5. Upgrade your local Airflow environment to Airflow 3.
6. Run your updated dags locally to test them.
7. Update your cloud environment accordingly.

NOTE Astronomer customers should consult the Astronomer documentation (https://mng.bz/jZy9) for comprehensive upgrading instructions for Astro environments.

8.3.1 Check your Airflow dag code

To help with the migration from Airflow 2 to Airflow 3, the developers created `ruff` rules. `AIR30` alerts you to any required changes to your dag code; the rules under `AIR31` contain additional recommended changes, such as alerts about newly deprecated syntax.

Ruff is a code linter that allows you to check your Python code against different rules. You can install ruff by running `pip install ruff` (make sure to use the latest version) in your Python environment and adding a `ruff.toml` file specifying the use of the `AIR30` rule in the root directory of your Airflow project, as shown in the following listing.

Listing 8.1 `ruff.toml` using the Airflow migration linter

```
[lint]
select = ["AIR30"]
preview = true
```

After adding this rule, run `ruff check` to check all dag files in your environment for any necessary changes. The following listing shows a sample response in which one syntax change was found: the `fail_stop` parameter was renamed `fail_fast`. You can adjust the dag manually or use `ruff check --fix` to adjust the dag code automatically.

Listing 8.2 Sample output from running `ruff check`

```
23:19:42 my_directory  % ruff check
dags/my_dag.py:19:5: AIR301 [*] `fail_stop` is removed in Airflow 3.0
   |
16 |     doc_md=__doc__,
17 |     default_args={"owner": "Astro", "retries": 3},
18 |     fail_stop=True,
   |     ^^^^^^^^^ AIR302
19 | )
20 | def my_dag():
   |
   = help: Use `fail_fast` instead

Found 1 error.
[*] 1 fixable with the `--fix` option.
```

The `AIR30` ruff rule covers most of the code changes necessary to migrate Airflow 2 dags to Airflow 3 that are not related to the more conceptual changes mentioned in this chapter. Note that there might be additional deprecations not covered yet in the AIR31 ruff rule; the developers are adding more rules as Airflow evolves.

> **TIP** You can check your dag code in one command without the need for a `ruff.toml` file by running `ruff check --preview --select AIR30 --fix`.

8.3.2 Check your Airflow configuration

Now that your dag code is ready, the only thing left to do is update the Airflow configuration `airflow.cfg` file. There are four possible changes to Airflow configuration parameters:

- *The default value for a configuration changed.* The default for `[scheduler].catchup_by_default`, for example, has changed from `True` to `False`.
- *A configuration was renamed and/or moved to another section.* `[webserver].web_server_host`, for example, has been renamed and moved to `[api].host` .
- *A configuration was removed.* `[webserver].error_logfile`, for example, has been removed.
- *A previously valid configuration value is now invalid.* `0` used to be a valid input for `[core].parallelism` to set it to unlimited, for example; now a positive integer is required.

You can check your `airflow.cfg` file against these changes by running the `airflow config lint` command. Listing 8.3 shows example output with a moved and renamed configuration. You can make the changes manually or apply an automated script using `airflow config update`. See the Airflow command-line interface (CLI) documentation (https://mng.bz/8XJ5) for more information.

NOTE Astronomer customers take a different path to upgrade their Airflow configuration. See the Astronomer documentation for guidance.

Listing 8.3 Sample output from running `airflow config lint`

```
Found issues in your airflow.cfg:
  - `stalled_task_timeout` configuration parameter
moved from the `celery` section to the `scheduler` section
as `task_queued_timeout`.

Please update your configuration file accordingly.
```

"Just two commands! And here I was worried I'd have to spend weeks refactoring code," Eléonore said to Chris after the first environment was updated much faster than she anticipated. Now she had the playbook to start upgrading all her team's Airflow environments, getting them ready for the future of Airflow (chapter 9).

Summary

- The Airflow developers took great care to keep as much backward compatibility between Airflow 2 and 3 as possible to simplify the migration process. Wherever possible, outdated dag code was deprecated, not removed, so users can upgrade to Airflow 3 and have plenty of time to modernize their codebase.

- Important breaking changes include the removal of direct database access for task and trigger code and changes to scheduling parameters.
- You can check dag code for breaking changes by using the ruff rule `AIR30`.
- You can check the Airflow config for changes by using the command `airflow config lint`.

9 The future of Airflow

This chapter covers

- Airflow milestones and priorities for versions beyond 3.0
- Airflow Improvement Proposals (AIPs) that are accepted and planned for version 3.1+
- The Airflow development process and how you can get involved
- The future of Astro by Astronomer as a unified DataOps platform

The release of Airflow 3 represents a significant milestone in the development of Apache Airflow. Alongside long-anticipated user-facing features, architectural changes in version 3 lay the groundwork for additional major features planned for future releases.

This chapter explores key high-level themes and highlights features outlined in AIPs (https://cwiki.apache.org/confluence/display/AIRFLOW). It concludes with a preview of how Astronomer will continue to improve its unified DataOps platform on top of Apache Airflow.

Given that Airflow is a community-driven project that adapts to evolving data practices, the features outlined in this chapter may change, and new features are likely to be added to coming minor releases, maybe even by you.

With that in mind, let's hop into our TARDIS and explore the future of Airflow.

9.1 Beyond 3.0

Apache Airflow has come a long way since it was created by Maxime Beauchemin at Airbnb in 2014. Airflow became enterprise-ready in 2020, with the version 2 release focusing on reliability features such as a highly available (HA) scheduler. Since then, significant new features have been added in every minor release, making Airflow more efficient and easier to use and unlocking new use cases. Figure 9.1 shows the timeline of Airflow's evolution.

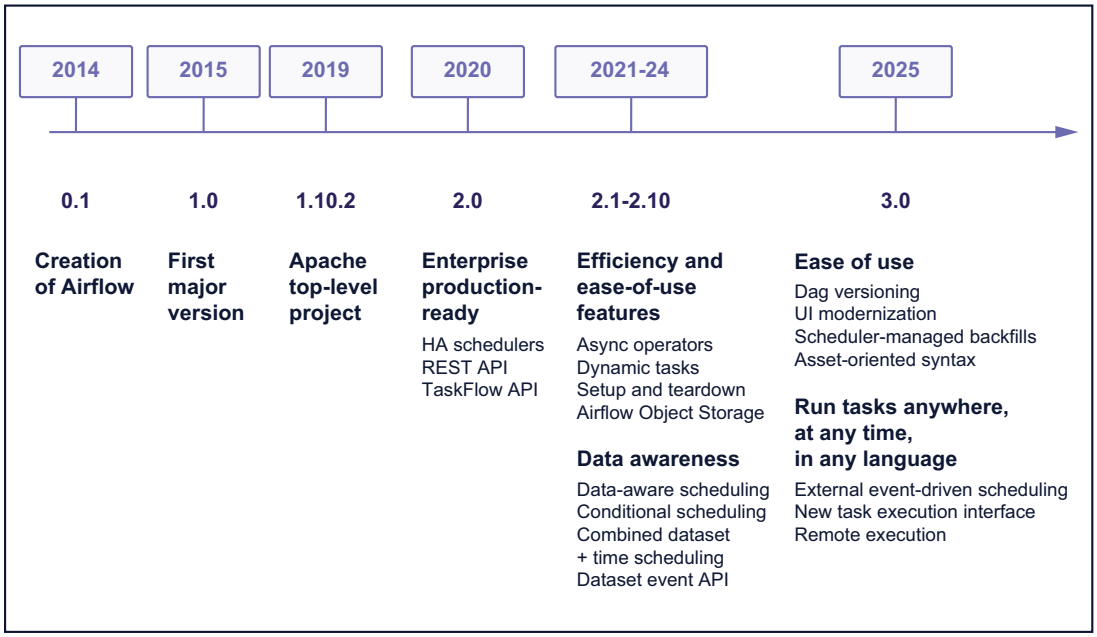


Figure 9.1 Airflow milestones and their headline features over time

The version 3 release ushers in a new area for Airflow, with two fundamental shifts:

- The expansion of data awareness through the asset feature and the asset-oriented approach to writing pipelines (see chapter 2) allows you to intuitively define data-dependent pipelines and form the basis for several more planned data-awareness features.

- The new task execution interface (see chapter 5) greatly improves Airflow's security posture and builds the foundation to allow tasks to run anywhere and in any language.

Already, several AIPs have been voted on and accepted by the Airflow community for specific features to be added in future minor releases.

9.1.1 Airflow Improvement Proposals

Larger Airflow features are written up in AIPs that are put to a vote by the community before they are implemented. Table 9.1 provides an overview of AIPs for version 3.1+ that have already been accepted.

> **NOTE** Any information in AIPs is subject to change. In addition to the AIPs listed in table 9.1, there are plans to implement support for writing tasks in more languages.

Table 9.1 Accepted AIPs for Airflow 3.1+ as of April 2025

AIP	Description
AIP-76 Asset Partitions (https://mng.bz/NwAv)	Many data pipelines process data incrementally based on partitions. Asset-level partitions allow you to define which parts of your data are processed independently of the asset's schedule. Such granular partitions can enhance performance when processing large datasets and make it easier to track data lineage across your pipelines.
AIP-67 Multi-team deployment of Airflow components (https://mng.bz/DwVA)	This proposed multiteam mode allows several teams to deploy dags to the same deployment to run in isolated environments while retaining the ability to define dependencies between dags using data-aware scheduling. Multiteam deployments reduce the need to create a separate deployment per team, increasing Airflow infrastructure efficiency and making cross-team dag dependencies more easily visible.
AIP-86 Deadline Alerts (Formerly SLA) (https://mng.bz/lZpd)	In the context of open source Airflow, service-level agreements (SLAs) refer to alerts that send a notification when a task has not finished within a time period defined from the start of the dag run. The Airflow 2 implementation of this feature, which was removed in Airflow 3, has known limitations and can be counterintuitive. This AIP proposes a complete refactor to allow for different types of intuitive SLAs.
AIP-80 Explicit Template Fields in Operator Arguments (https://mng.bz/BzDw)	Changes to the templating feature are planned to remove the need for an additional space when templating the name of a file, such as in the `BashOperator`'s `bash_command` parameter, making templates easier to use.
AIP-68 Extended Plugin Interface for React Views (https://mng.bz/dWPg)	Following the modernization of the Airflow UI, this AIP will add the possibility of user-defined panels and other custom elements in the Airflow UI by extending the Airflow plugin Interface.

9.1.2 Asset partitions

Partitioning data and data processing are ubiquitous needs in data engineering. As soon as a data object, such as a table in a data warehouse, reaches a size threshold, it becomes inefficient to rematerialize the whole object every time it needs to be updated. Instead, you want to be able to make incremental changes to your data so that each run changes only certain parts of the data. These parts, called *partitions*, can be time-based, segment-based, or based on a combination of the two.

Before this feature, partitions had to be tied to the dag's logical date, meaning that partitions were directly coupled to the schedule of the dag. In other words, a task that was supposed to update the data based on hourly partitions had to be part of a dag that ran every hour or had to process all 24 hours' worth of data in a single daily dag run. The following listing shows how to partition a dag written using the asset-based approach on a time-based interval, independent from the materialization schedule of the asset.

Listing 9.1 Time-based partition syntax example

```
@asset(..., schedule="@daily", partition=PartitionByInterval("@hourly"))
def hourly_data_generated_daily():
    # Code that generates the data for one hour
```

> **NOTE** Listings 9.1 and 9.2 show example syntax that may change during feature development. Please refer to the Airflow documentation (https://mng.bz/rZBx) upon release of the feature for the final syntax.

In this listing, the asset is materialized on a daily schedule, whereas the partition is based on an hourly interval. In practice, this means that every day at midnight, 24 dag runs are triggered, each materializing an asset for one hour of the previous day.

The big advantage of this decoupled approach is that these 24 dag runs can run in parallel and independent of one another. If the data for one hour is flawed, for example, only the run for that hour fails and needs to be rerun.

Segment-based partitions allow you to process user-defined segments of the data in separate dag runs. The data could be segmented by team to calculate their cloud spend, for example, as shown in the following listing.

Listing 9.2 Segment-based partition syntax example

```
@asset(
    ...,
    schedule="@daily",
    partition=PartitionBySequence(["marketing-dwh", "engineering-dwh"]),
)
def dwh_daily_cloud_spend():
    # Code that analyzes cloud spend for one partition
```

Also, it will be possible to combine time- and segment-based partitions. For more information, see AIP-76 (https://mng.bz/NwAv).

9.2 Community-led development

Airflow is an open source project led by the Airflow community, which consists of more than 58,000 Slack members and more than 3,200 contributors on GitHub as of April 2025 (figure 9.2), with new engineers joining every day.

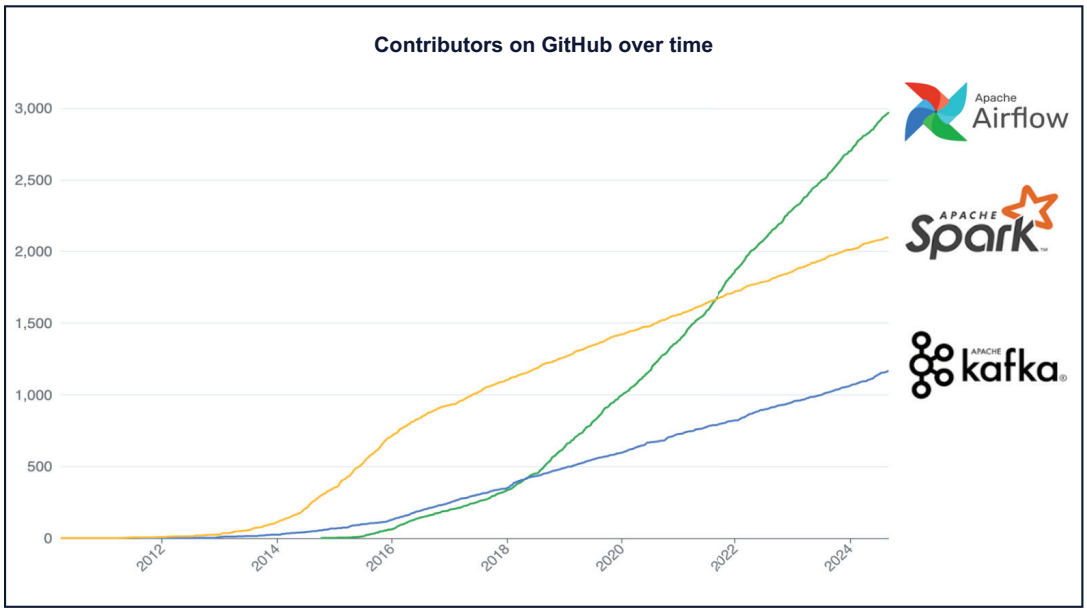


Figure 9.2 Growth of Apache Airflow contributors over time

Because Airflow is an Apache Software Foundation (ASF; https://www.apache.org) project, all development and release decisions are governed by Apache rules, with formal discussions and announcements taking place on the Airflow Dev mailing list. Anyone can subscribe to the Dev list (https://airflow.apache.org/community) to stay up to date on the latest discussions about the project, such as new AIPs being announced and new versions being released.

The core of the developer community is the Project Management Committee (PMC) and the Airflow committers. PMC members can cast binding votes on the Dev list to accept or reject new AIPs and Airflow release candidates. Which features are prioritized depends heavily on community input, so you have direct influence on the future of Airflow. The most important way that feedback is gathered from the community is the annual Airflow survey (https://airflow.apache.org/blog/airflow-survey-2024), so make

sure to respond every year. The Airflow developers also listen to input in the Airflow Slack and respond to bug reports submitted as Airflow problems (https://github.com/apache/airflow/issues).

Chris, while expanding on his newsletter pipeline (which by then had become one of the most-subscribed-to newsletters at *AllThingsLookingUp*), discovered a bug in Airflow. Now a seasoned Airflow user, he went to the Airflow source code (https://github.com/apache/airflow) to understand what was going on and thought he might know how to fix the bug. But he wasn't sure how to proceed.

Become an Airflow contributor

We welcome new contributors at all levels. Whether this is your first time contributing to open source software (OSS) or you are a seasoned engineer who wants to work on cutting-edge cross-technology problems, we are happy to help you set up your development environment and get you started. Read our contributors' guide (https://mng.bz/xZJ7), and join the Airflow Slack (https://apache-airflow-slack.herokuapp.com) #new-contributors channel for more information. If you are looking for inspiration about what you might work on, check out the Airflow issues page (https://github.com/apache/airflow/issues).

Most contributors begin their journey by making documentation improvements and code changes for Airflow provider packages. If you are not a Python developer and are eager to add Airflow task-execution support for a different programming language, please let me know. Can't wait to see your first pull request!

— Vikram Koka, Airflow committer and PMC member

Chris joined the Airflow Slack and asked for help in the #new-contributors channel. An Airflow committer helped him set up the development environment, and Chris submitted his first pull request (PR). After two rounds of reviews and adding more tests, the PR got merged into the next Airflow release. Chris had become an Airflow contributor.

9.3 The future of Astronomer

Astronomer's future is closely tied to the evolution of open source Airflow. Astro (https://www.astronomer.io/trial-3), the unified DataOps platform built on Airflow, benefits from its comprehensive control of and visibility into both data and metadata, especially as data-awareness features continue to expand.

Building a unified DataOps platform on Airflow enables rapid adaptation to changes in the data ecosystem, whether you are switching to a lower-cost data transformation tool or integrating the latest large language models (LLMs). With new tools and models emerging daily, Airflow's tool-agnostic design ensures that you're never locked into a single vendor, giving you the flexibility to adapt and grow.

To power this enterprise platform, Astronomer is making significant investments in expanding features across the DataOps stack beyond orchestration, integrating

observability, ML and AI Ops, data integration, and data cataloging. Figure 9.3 illustrates Astronomer's vision of Astro as the unified DataOps platform. At the same time, we continue to deliver the best managed Airflow experience, capable of supporting mission-critical workloads.

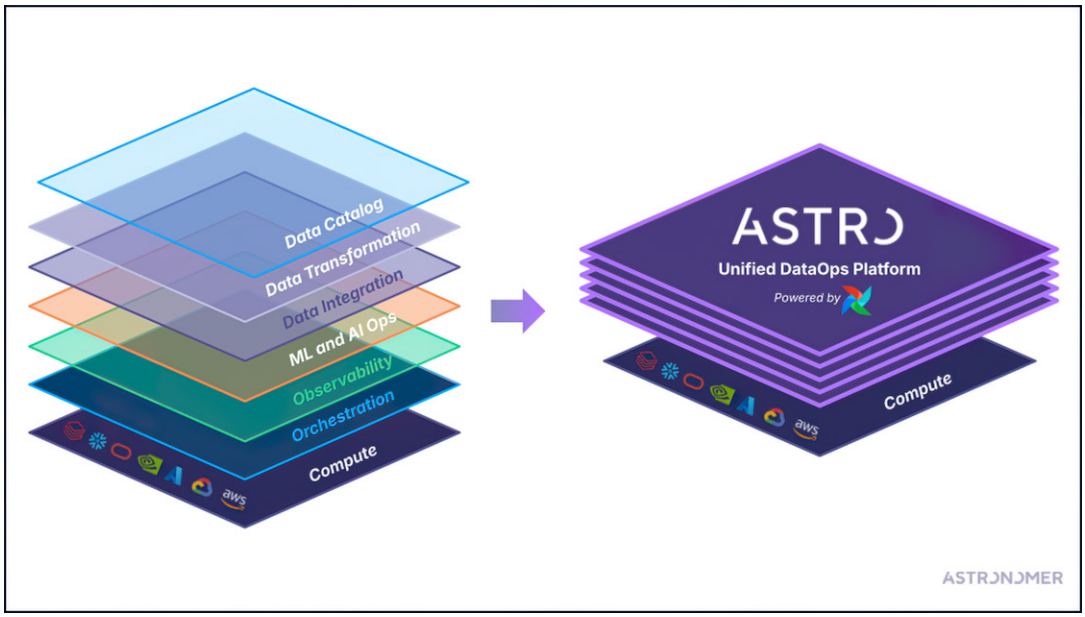


Figure 9.3 Astro's vision of a unified DataOps platform powered by Airflow

Summary

- Airflow 3 contains two important paradigm shifts that lay the groundwork for additional features: architectural changes decoupling task execution from Airflow system components and a new asset-oriented approach to defining Airflow dags.
- As of April 2025, five accepted AIPs describe features to be released in Airflow 3.1+. One of the proposals covers asset partitioning, which will allow you to process data incrementally based on partitions that are independent of the asset's schedule.
- The Airflow project is led by the community, and development of new features is directly informed by community feedback, gathered primarily from the Airflow survey and Airflow Slack.
- Everyone is welcome to contribute to Airflow. All contributions, from fixing typos in the documentation to writing substantive features, are welcome. The

best way to start is to read the contributors' guide and join the Airflow Slack's #new-contributors channel.

- Astronomer continues to build a unified DataOps platform on top of Airflow and is heavily invested in the open source project and community.

10 Resources

This chapter covers

- Key resources to expand your knowledge of Airflow
- How to get involved in the Airflow community
- Where to find answers to your Airflow-related questions

We hope that this book has sparked your interest in exploring Airflow further and joining its vibrant, global community! This chapter provides essential resources for learning more about Airflow, along with guidance on getting involved, whether as an Airflow user, a contributor to the project, or both.

All the code shown in this book, as well as Chris's full newsletter pipeline, is available in this GitHub repository: https://github.com/astronomer/practical-guide-to-apache-airflow-3.

10.1 Learn more about Apache Airflow

Thanks to Airflow's huge community, you can find an abundance of resources explaining Airflow concepts, showing examples, and providing best-practices guidance. Table 10.1 gives an overview of the best places to learn more.

Table 10.1 Resources to learn more about Airflow

Resource	Description
Data Pipelines with Apache Airflow, 2nd ed. (https://mng.bz/Wwjw)	This comprehensive book dives deep into Airflow concepts and best practices while expanding on topics such as dag testing and using Airflow to train machine learning (ML) and generative AI (GenAI) models.
Apache Airflow documentation (https://airflow.apache.org/docs)	The official Airflow documentation contains detailed instructions for using the latest Airflow features, including example code and reference pages such as these: ■ *Configuration Reference*—A list of all Airflow config variables (https://mng.bz/8X7B) ■ *Airflow API*—Docs for the Airflow API (https://mng.bz/EwRj) ■ *Command Line Interface and Environment Variables Reference*—A list of all Airflow CLI commands (https://mng.bz/Nwy7) ■ *Templates Reference*—A list of variables and macros that can be used in Jinja templates in Airflow (https://mng.bz/Dwoa)
Astronomer Academy (https://academy.astronomer.io)	In the Astronomer Academy, you can take full-length video-based courses and even get Airflow-certified.
Astronomer webinars (https://www.astronomer.io/events/webinars)	Astronomer webinars are hour-long deep dives into topics surrounding Airflow and Astro, such as best practices for dag writing and running Airflow at scale.
The Data Flowcast (https://www.astronomer.io/podcast)	The Airflow podcast, offering weekly insights into how some of the biggest companies use Airflow for all sorts of use cases.
Astronomer Learn documentation (https://www.astronomer.io/docs/learn)	The Astronomer Learn docs contain a large number of guides and tutorials expanding on the official documentation with example code and best-practices guidance.
Astronomer Registry (https://registry.astronomer.io)	The Astronomer Registry is the best place to discover Airflow providers and their modules.
Ask Astro (https://ask.astronomer.io)	Ask Astro is a conversational chatbot with advanced knowledge of Airflow and Astronomer. The code powering the chatbot serves as an open source reference architecture on building retrieval augmented generation (RAG) apps with Airflow (https://mng.bz/lZvB).

You can extend Airflow's functionality by using it in conjunction with other open source and proprietary tools. Table 10.2 lists some of the tools we recommend that you consider using with Airflow.

Table 10.2 Tools to use with Airflow

Resource	Description
Astro (https://www.astronomer.io/product)	Astro is the easiest way to run Airflow in production in a cloud-based environment as part of a unified data operations platform. A free trial of Astro is available (https://www.astronomer.io/trial-3). See chapter 6 for more information.

Table 10.2 Tools to use with Airflow (*continued*)

Resource	Description
Astro CLI (https://mng.bz/Bz0r)	The Astro CLI is a free tool developed by Astronomer to run Airflow in containers on your local machine. See chapter 2 for instructions on using the Astro CLI.
`dag-factory` (https://astronomer.github.io/dag-factory/latest)	The `dag-factory` package allows you to create Airflow dags programmatically based on configurations written in YAML.
Astronomer Cosmos (https://mng.bz/dW7v)	If you are using Airflow with dbt Core, you will love Cosmos, an advanced open source Airflow provider package for dbt Core that automatically turns dbt workflows into Airflow dags.
Airflow AI SDK (https://github.com/astronomer/airflow-ai-sdk)	An SDK for using Airflow to work with LLMs and AI Agents, based on Pydantic AI.

10.2 Get involved in the community

The Airflow community is a vibrant, welcoming place where Airflow users and developers interact to give feedback and drive the development of Airflow forward.

The first step in getting involved in the community is joining the Apache Airflow Community Slack (https://apache-airflow-slack.herokuapp.com), the hub where more than 58,000 Airflow users and developers interact and the best place to stay informed about what is happening in the community.

If you are looking to network with other Airflow users, check out Astronomer & Airflow Events (https://www.astronomer.io/events) and Airflow's local Meetup groups (https://www.meetup.com/pro/astronomer-inc).

Official Airflow discussions and votes on new features and releases are held on the Airflow Dev mailing list (https://airflow.apache.org/community). Larger feature proposals are publicly posted as Airflow Improvement Proposals (https://mng.bz/rZGe).

10.3 Getting answers to Airflow questions

Thousands of people use Airflow every day. Chances are that if you have a question about best practices, how to connect Airflow to a specific tool, or an error message, someone can answer it. Table 10.3 lists places to find help.

Table 10.3 Where to get help with Airflow questions

Resource	Description
Airflow Slack (https://apache-airflow-slack.herokuapp.com)	Join the Apache Airflow Slack, and open a thread in the #user-troubleshooting channel. The Airflow Slack is the best place to get answers to complex Airflow-specific questions.
Stack Overflow (https://stackoverflow.com)	You can 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, because experts on different tools will see your question.

Table 10.3 Where to get help with Airflow questions (*continued*)

Resource	Description
Airflow issues (https://github.com/apache/airflow/issues)	If you found a bug in Airflow or one of its providers, please open an issue in the Airflow GitHub repository. For bugs in Astronomer open source tools, please open an issue in the relevant Astronomer repository (https://github.com/astronomer).
Ask Astro (https://ask.astronomer.io)	Ask Astro is a conversational chatbot with advanced knowledge of Airflow and Astronomer. Always test and double-check any AI-generated code!

No matter where you ask your Airflow question, it is helpful for the person who answers to have as much detail as possible about your Airflow setup and use case. In your question, try to include the following:

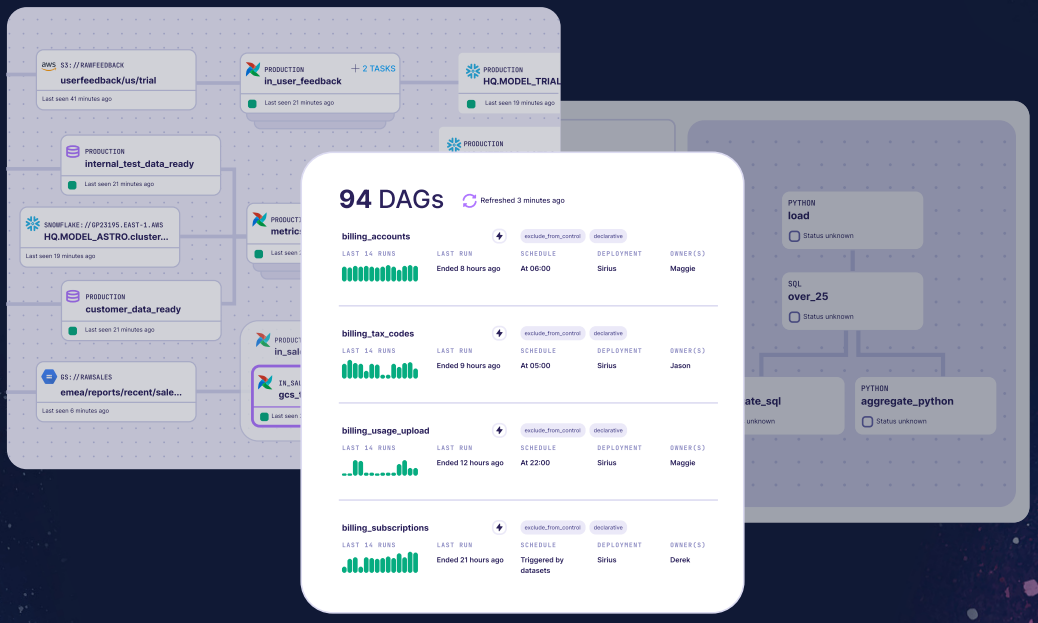
- Your method for running Airflow (Astro CLI, standalone, Docker, managed services, and so on)
- Your Airflow version and the versions of relevant providers
- The full error with the error trace, if applicable
- The full code of the dag causing the error, if applicable
- What you are trying to accomplish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

Summary

- You can find many resources to learn more about Airflow, including the Apache Airflow documentation (https://airflow.apache.org/docs), a dedicated podcast (https://www.astronomer.io/podcast), and full-length courses at Astronomer Academy (https://academy.astronomer.io).
- The first step in getting involved in the Airflow community as an Airflow user and contributor is joining the Airflow Slack (https://apache-airflow-slack.herokuapp.com).
- If you have questions about Airflow, ask them in the #user-troubleshooting channel of the Airflow Slack, and make sure to include detailed information about your Airflow setup and use case.

Experience fully-managed Airflow with Astro.

Build, run, & observe your data workflows all in one unified platform.



Try for free today: astronomer.io/trial-3

ASTRONOMER

astronomer.io