Apache Airflow 101

Essential concepts and tips for beginners

Powered by Astronomer

Editor’s Note

Welcome to **The Apache Airflow 101 ebook**, brought to you by Astronomer. We believe that Airflow is the best-in-class open source technology for adapting to the ever-changing data landscape. In this ebook, we’ve gathered and explained key Airflow concepts and components to help you get started using this data orchestration platform. We’ve also compiled some guides and tutorials for more advanced Airflow features. This ebook is everything you need to successfully kick off your Airflow journey.

Follow us on Twitter and LinkedIn!

Table of Contents

1. What is Apache Airflow?

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

Today, with a thriving community of more than **2k contributors, 25k+ stars on Github**, and thousands of users—including organizations ranging from early-stage startups to Fortune 500s and tech giants—Airflow is widely recognized as the industry's leading data orchestration solution.

Apache Airflow's popularity as a tool for data pipeline automation has grown for a few reasons:

- **Proven core functionality for data pipelining.**
  Airflow's core capabilities are used by thousands of organizations in production, delivering value across scheduling, scalable task execution, and UI-based task management and monitoring.

- **An extensible framework.**
  Airflow was designed to make integrating existing data sources as simple as possible. Today, it supports over 80 providers, including AWS, GCP, Microsoft Azure, Salesforce, Slack, and Snowflake. Its ability to meet the needs of simple and complex use cases alike makes it both easy to adopt and scale.

- **Scalability.**
  From running a few pipelines to thousands every day, Airflow manages workflows using a reliable scheduler. Additionally, you can use parameters to fine-tune the performance of Airflow to fit any use case.

- **A large, vibrant community.**
  Airflow boasts thousands of users, and over 2k+ contributors who regularly submit features, plugins, content and bug fixes to ensure continuous momentum, and improvement. **So far Airflow has reached 15k+ commits and 25k+ GitHub stars.**

Apache Airflow continues to grow thanks to its active and expanding community. And because it belongs to the Apache Software Foundation and is governed by a group of PMC members, it can live forever.

As a result, hundreds of companies — including Fortune 500s, tech giants, and early-stage startups — are adopting Airflow as their preferred tool to programmatically author, schedule, and monitor workflows. Among the biggest names, you'll find Adobe, Bloomberg, Asana, Dropbox, Glassdoor, HBO, PayPal, Tesla, ThoughtWorks, WeTransfer, and more!

Apache Airflow — Built on a Strong and Growing Community

27k+ Github Stars	25k+ Slack community
12m+ Monthly downloads	2k+ Contributors

Apache Airflow Core Principles

Airflow is built on a set of core ideals that allow you to leverage the most popular open source workflow orchestrator on the market while maintaining enterprise-ready flexibility and reliability.

Flexible

Fully programmatic workflow authoring allows you to maintain full control of the logic you wish to execute.

Extensible

Leverage a robust ecosystem of open source mintegrations to connect natively to any third party datastore or API.

Open Source

Get the optionality of an open-source codebase while tapping into a buzzing and action-packed community.

Scalable

Scale your Airflow environment to infinity with a modular and highly-available architecture across a variety of execution frameworks.

Secure

Integrate with your internal authentication systems and secrets managers for an platform ops experience that

Modular

Plug into your internal logging and monitoring systems to keep all of the metrics you care about in one place.

2. Why Apache Airflow?

Viraj Parekh
Field CTO at Astronomer

Kenten Danas
Lead Developer Advocate at Astronomer

Our Field CTO Viraj Parekh and Lead Developer Advocate Kenten Danas gathered some of the most common questions data practitioners ask us on a daily basis. If you'd like to find out why Airflow is the best tool for data orchestration, here are their answers.

Q: Why would you run your pipelines as code?

At Astronomer, we believe using a code-based data pipeline tool like Airflow should be a standard. There are many benefits to that solution, but a few come to mind as high-level concepts:

Firstly, code-based pipelines are **dynamic**. If you can write it in code, then you can do it in your data pipeline. And that's really powerful.

Firstly, code-based pipelines are **dynamic**. If you can write it in code, then you can do it in your data pipeline. And that's really powerful.

Secondly, code-based pipelines are **highly extensible**. If your external system has an API, there's a way to integrate it with Airflow.

Finally, they are **more manageable**. Because everything is in code, these pipelines can integrate seamlessly into your source controls CICT and general developer workflows.

Q: What about a company that is looking at a drag-and-drop tool or other low/no-code approach? Why should they consider Airflow?

We never think of it as "Airflow or"– instead, it's always "Airflow and". Because Airflow is so extensible, you can still use those drag-and-drop tools and then orchestrate and integrate them with other pipelines using Airflow. **This approach allows you to have a one-stop-shop for orchestration without necessarily having to make a giant migration effort upfront or convince everybody at one time.**

As those internal teams get more comfortable with Airflow and see all the benefits, they'll naturally start to transfer to Airflow.

Q: How can functional data pipelines improve my business?

Data pipelines help users manage data workflows that need to be run on a consistent, reliable schedule. You can use data pipelines to help you with simple use cases, such as running a report about yesterday's sales. Data pipelines empower those users who are looking for baseline insights into the business.

You can also use data pipelines to help with more complex use cases, such as running machine learning models, training AI, and generating complex business intelligence reports that help you make critical decisions.

So, if you want something that's flexible, standardized, and scalable that has the ongoing support of the open-source community— Airflow is perfect for you.

Q: Is Apache Airflow hard to learn?

Many resources exist for learning how to write code for Airflow, including Astronomer's Airflow Guides. Airflow becomes more difficult to learn once you try to run it at business scale. Luckily, there are tools like Astro that allow for abstracting and managing the infrastructure easily and efficiently.

Q: What companies use Apache Airflow?

Walmart, Tinder, JP Morgan, Tesla, Revolut....just to name a few. Currently, hundreds of companies around the world, from various industries like healthcare, AI, fintech, or e-commerce, use Apache Airflow as part of their modern data stack to help them make sense of their data and drive better business decisions.

Get Apache Airflow Support

Schedule time with Astronomer experts to learn how we can help you address your Airflow issues.

Contact Us

3. Why Open Source Software?

The massive increase in open source software (OSS) projects is changing both the tech scene and the business scene on a global scale. So why do we love open-source? **Because it is:**

Innovative

According to Wired in the 2016 article, Facebook decided to stop *"treating data center design like **Fight Club**"* by making its architecture open and accessible to everyone. Companies like Microsoft, HP, and even Google followed suit by open-sourcing many of their technologies that had been kept secret for years. By making code for their projects accessible to the public, companies could freely adapt golden standards in the open-source world and create better integrations between products, all while freeing up time to innovate elsewhere.

Reliable

We've all had "technical difficulties" when our technology operated unreliably. Professionally, that can cause a variety of outcomes, from mild annoyance to genuine detriment. When everybody's using the same components, everybody's optimizing the same components. As Linux Creator Linus Torvalds said, "Given enough eyeballs, all bugs are shallow" (**Linus's Law**). In other words, the more people who can see and test a set of code, the more likely any flaws will be caught and fixed quickly.

Diverse

It's a huge benefit to have a large, diverse community of developers look at and contribute to the same system. Instead of a few people with the same view of the world-building components, an entire community with varying perspectives and strengths contributes to the system (and has a stake in its success and adoption), which makes the system more resilient.

Transparent

When software is transparent people aren't tied down to a proprietary system protected in a black box. Users and builders of open source software can see everything; they can open GitHub, look at the source code, and trust that the code has been reviewed publicly. This gives decision-makers insight into what's going on — where components can be swapped around — and the control they need to do it.

Agile

When components are living in a community-curated open source world, they are built to “play nicely” with other components. This flexibility allows us to stay on the cutting edge. With open-source, any organization can take a piece of code and customize it to best fit their needs. When it comes to tech, we want to live in a world where we’re free to explore, invent best-in-class technology and use it in revolutionary ways. We can do it through open source.

Apache Airflow Fundamentals

The Astronomer Certification for Apache Airflow Fundamentals exam assesses an understanding of the basics of the Airflow architecture and the ability to create basic data pipelines for scheduling and monitoring tasks.

Concepts Covered:

- User Interface
- Scheduling
- Backfill & Catchup
- DAG Structure
- Architectural Components
- Use Cases

Get Certified

4. Apache Airflow 101 Core Concepts & Components

Airflow Core Concepts

DAGs

In Airflow, a DAG is your data pipeline and represents a set of instructions that must be completed in a specific order. This is beneficial to data orchestration for a few reasons:

• **DAG dependencies** ensure that your data tasks are executed in the same order every time, making them reliable for your everyday data infrastructure.

• **The graphing component of DAGs** allows you to visualize dependencies in Airflow's user interface.
• **Because every path in a DAG is linear**, it's easy to develop and test your data pipelines against expected outcomes.

Learn more:

ARITCLE
What exactly is a DAG?
SEE ARTICLE →

Operators

Operators are the building blocks of Airflow. They contain the logic of how data is processed in a pipeline. There are different operators for different types of work: some operators execute general types of code, while others are designed to complete very specific types of work.

```
operator
     file = open("myfile", "r")
     print (f.read() )
```

- **Action Operators** execute pieces of code. For example, a Python action operator will run a Python function, a bash operator will run a bash script, etc.
- **Transfer Operators** are more specialized, designed to move data from one place to another.
- **Sensor Operators,** frequently called “sensors,” are designed to wait for something to happen — for example, for a file to land on S3, or for another DAG to finish running.

When you create an instance of an operator in a DAG and provide it with its required parameters, it becomes a task.

Learn more:

TUTORIAL
Operators 101
SEE TUTORIAL ⟶

TUTORIAL
Deferrable Operators
SEE TUTORIAL ⟶

TUTORIAL
Sensors 101
SEE TUTORIAL ⟶

Tasks

A task is an instance of an operator. In order for an operator to complete work within the context of a DAG, it must be instantiated through a task. Generally speaking, you can use tasks to configure important context for your work, including when it runs in your DAG.

Learn more:

GUIDE
Using Task Groups in Airflow
SEE GUIDE ⟶

GUIDE
Passing Data Between Airflow Tasks
SEE GUIDE ⟶

A data pipeline

A “data pipeline” describes the general process by which data moves from one system into another. From an engineering perspective, it can also be represented by a DAG. Each task in a DAG is defined by an operator, and there are specific downstream or upstream dependencies set between tasks. A DAG run either extracts, transforms, or loads data - becoming a data pipeline, essentially.

Learn more:

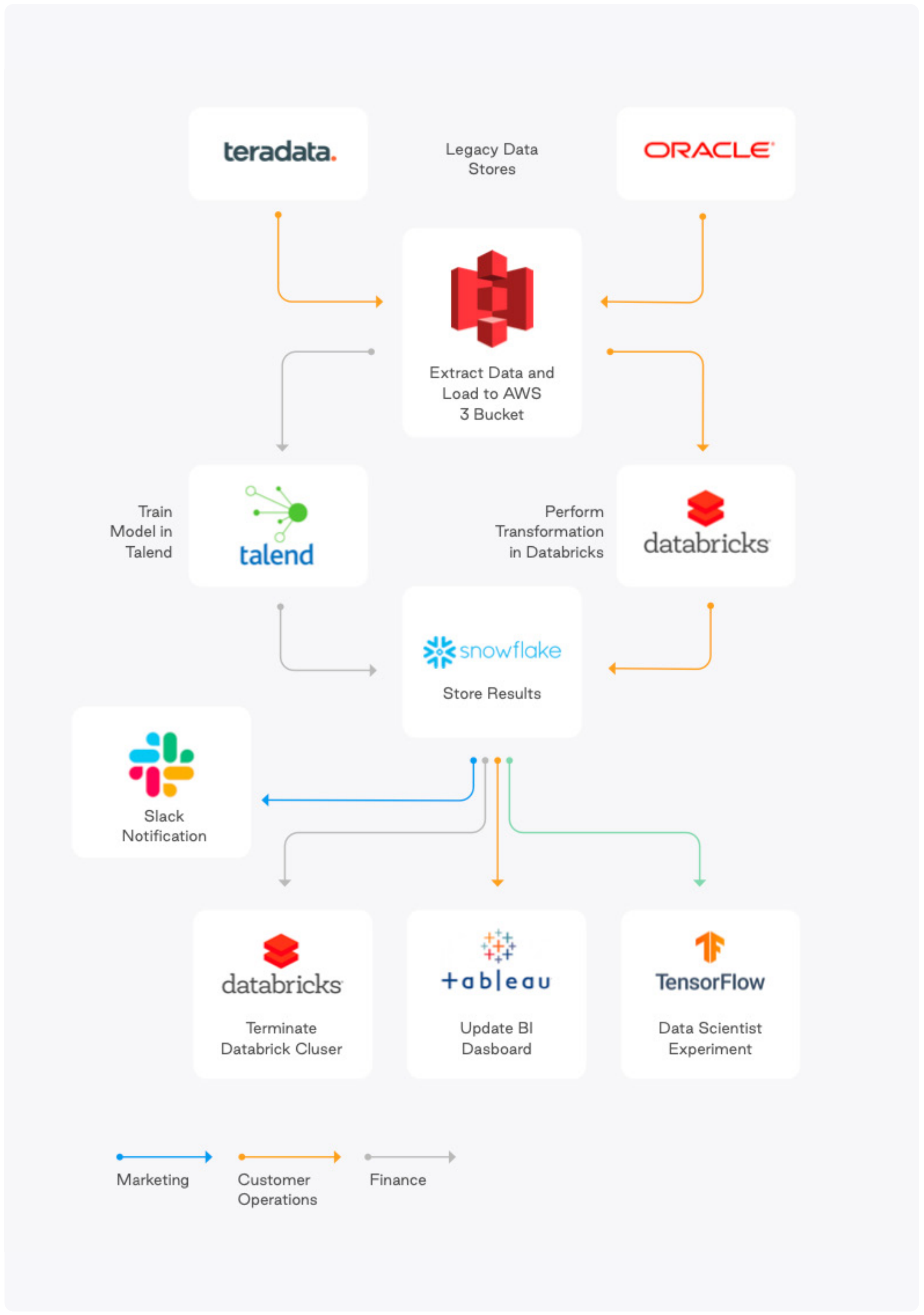
ARITCLE
Data Pipeline: Components, Types, and Best Practices
SEE ARTICLE ⟶

Providers

Airflow providers are Python packages that contain all of the relevant Airflow modules for interacting with external services. Airflow is designed to be an agnostic workflow orchestrator: You can do your work within Airflow, but you can also use other tools with it, like AWS, Snowflake, or Databricks.

Most of these tools already have community-built Airflow modules, giving Airflow spectacular flexibility. Check out the Astronomer Registry to find all the providers.

The following diagram shows how these concepts work in practice. As you can see, by writing a single DAG file in Python using existing provider packages, you can begin to define complex relationships between data and actions.



Airflow Core Components

When working with Airflow, it is important to understand the underlying components of its infrastructure. Even if you mostly interact with Airflow as a DAG author, knowing which components are “under the hood” and why they are needed can be helpful for developing your DAGs, debugging, and running Airflow successfully. The first four components below run at all times, and the last two are situational components that are used only to run tasks or make use of certain features.

The infrastructure:

webserver	Flask server running with Gunicorn serving the UI
scheduler	Daemon responsible for scheduling jobs
metastore	A database where all metadata are stored
executor	Defines **how** tasks are executed
worker	Process **executing** the tasks, defined by the executor
triggerer	Process running **asyncio** to support deferrable operators

5. Apache Airflow 2

As **Apache Airflow** grows in adoption, there's no question that a major release to expand on the project's core strengths was long overdue. Astronomer was delighted to release Airflow 2.0 together with the community in December 2020.

Throughout the year various organizations and leaders within the Airflow community have been in close collaboration refining the scope of Airflow 2.0 and actively working towards enhancing existing functionality and introducing changes to make Airflow faster, more reliable, and more performant at scale.

Major Features in Airflow 2.0

Airflow 2.0 includes hundreds of features and bug fixes both large and small. Many of the significant improvements were influenced and inspired by feedback from **Airflow's 2019 Community Survey,** which garnered over 300 responses.

A New Scheduler: Low-Latency + High-Availability

The Airflow scheduler as a core component has been key to the growth and success of the project following its creation in 2014. In fact, "Scheduler Performance" was the most asked for improvement in the Community Survey. Airflow users have found that while the Celery and **Kubernetes Executors** allow for task execution at scale, the scheduler often limits the speed at which tasks are scheduled and queued for execution. While effects vary across use cases, it's not unusual for users to grapple with induced downtime and a long recovery in the case of a failure and experience high latency between short-running tasks.

It is for that reason we introduced a new, refactored scheduler with the Airflow 2.0 release. The most impactful Airflow 2.0 change in this area is support for running multiple schedulers concurrently in an active/active model. Coupled with **DAG** Serialization, Airflow's refactored scheduler is now highly available, significantly faster, and infinitely scalable. Here's a quick overview of the new functionality:

1. **Horizontal Scalability.** If task load one scheduler increases, a user can now launch additional "replicas" of the scheduler to increase the throughput of their Airflow Deployment.

2. **Lowered Task Latency.** In Airflow 2.0, even a single scheduler has proven to schedule tasks at much faster speeds with the same level of CPU and Memory.

3. **Zero Recovery Time.** Users running 2+ Schedulers see zero downtime and no recovery time in the case of a failure.

4. **Easier Maintenance.** The Airflow 2.0 model allows users to make changes to individual schedulers without impacting the rest and inducing downtime.

The scheduler's now-zero recovery time and readiness for scale eliminate it as a single point of failure within Apache Airflow. Given the significance of this change, our team published "**The Airflow 2.0 Scheduler**", a blog post that dives deeper into the story behind scheduler improvements alongside an architecture overview and benchmark metrics.

For more information on how to run more than 1 scheduler concurrently, refer to the official documentation on the Airflow Scheduler.

Full REST API

Data engineers have been using Airflow's "Experimental API" for years, most often for **triggering DAG runs programmatically**. With that said, the API has historically remained narrow in scope and lacked critical elements of functionality, including a robust authorization and permissions framework. Airflow 2.0 introduces a new, comprehensive REST API that sets a strong foundation for a new Airflow UI and CLI in the future. Additionally, the new API:

- **Makes for easy access by third-parties.**
- **Is based on the Swagger/OpenAPI Spec.**
- **Implements CRUD (Create, Update, Delete) operations on all Airflow resources.**
- **Includes authorization capabilities (parallel to those of the Airflow UI).**

These capabilities enable a variety of use cases and create new opportunities for automation. For example, users now have the ability to programmatically set Connections and Variables, show import errors, create Pools, and monitor the status of the Metadata Database and scheduler.

For more information, reference Airflow's REST API documentation.

TaskFlow API

While Airflow has historically shined in scheduling and running idempotent tasks, it has historically lacked a simple way to pass information between tasks. Let's say you are writing a DAG to train some set of Machine Learning models. The first set of tasks in that DAG generates an identifier for each model and the second set of tasks outputs the results generated by each of those models. In this scenario, what's the best way to pass output from that first set of tasks to the latter?

Historically, **XComs** have been the standard way to pass information between tasks and would be the most appropriate method to tackle the use case above. As most users know, however, XComs are often cumbersome to use and require redundant boilerplate code to set return variables at the end of a task and retrieve them in downstream tasks.

With Airflow 2.0, we were excited to introduce the TaskFlow API and Task Decorator to address this challenge. The TaskFlow API implemented in 2.0 makes DAGs significantly easier to write by abstracting the task and dependency management layer from users. Here's a breakdown of the functionality:

- **A framework that automatically creates PythonOperator tasks from Python functions and handles variable passing.** Now, variables such as Python Dictionaries can simply be passed between tasks as return and input variables for cleaner and more efficient code.
- **Task dependencies are abstracted and inferred as a result of the Python function invocation.** This again makes for much cleaner and more simple DAG writing for all users.
- **Support for Custom XCom Backends. Airflow 2.0 includes support for a new `xcom_backend parameter` that allows users to pass even more objects between tasks.** Out-of-the-box support for S3, HDFS, and other tools is coming soon.

It's worth noting that the underlying mechanism here is still XCom and data is still stored in Airflow's Metadata Database, but the XCom operation itself is hidden inside the PythonOperator and is completely abstracted from the DAG developer. Now, Airflow users can pass information and manage dependencies between tasks in a standardized Pythonic manner for cleaner and more efficient code.

To learn more, refer to Airflow documentation on the TaskFlow API and the accompanying tutorial.

Task Groups

Airflow SubDAGs have long been limited in their ability to provide users with an easy way to manage a large number of tasks. The lack of parallelism coupled with confusion around the fact that SubDAG tasks can only be executed by the Sequential Executor, regardless of which executor is employed for all other tasks, made for a challenging and unreliable user experience.

Airflow 2.0 introduced Task Groups as a UI construct that doesn't affect task execution behavior but fulfills the primary purpose of SubDAGs. Task Groups give a DAG author the management benefits of "grouping" a logical set of tasks with one another without having to look at or process those tasks any differently.

While Airflow 2.0 continues to support the SubDAG Operator, Task Groups are intended to replace it in the long term.

Independent Providers

One of Airflow's signature strengths is its sizable collection of community-built operators, hooks, and sensors — all of which enable users to integrate with external systems like AWS, GCP, Microsoft Azure, Snowflake, Slack and many more.

Providers have historically been bundled into the core Airflow distribution and versioned alongside every Apache Airflow release. As of Airflow 2.0, they are now split into their own airflow/providers directory such that they can be released and versioned independently from the core Apache Airflow distribution. Cloud service release schedules often don't align with the Airflow release schedule and either result in incompatibility errors or prohibit users from being able to run the latest versions of certain providers. The separation in Airflow 2.0 allows the most up-to-date versions of Provider packages to be made generally available and removes their dependency on core Airflow releases.

It's worth noting that some operators, including the Bash and Python Operators, remain in the core distribution given their widespread usage.

To learn more, refer to Airflow documentation on Provider Packages.

Simplified Kubernetes Executor

Airflow 2.0 includes a re-architecture of the **Kubernetes Executor** and **KubernetesPodOperator**, both of which allow users to dynamically launch tasks as individual Kubernetes Pods to optimize overall resource consumption.

Given the known complexity users previously had to overcome to successfully leverage the executor and operator, we drove a concerted effort towards simplification that ultimately involved removing over 3,000 lines of code. The changes incorporated in Airflow 2.0 made the executor and operator easier to understand, faster to execute and offers far more flexibility in configuration.

Data Engineers have now access to the full Kubernetes API to create a yaml 'pod_template_file' instead of being restricted to a partial set of configurations through parameters defined in the airflow.cfg file. We've also replaced the executor_config dictionary with the `pod_override` parameter, which takes a Kubernetes V1Pod object for a clear 1:1 override setting.

For more information, we encourage you to follow the documentation on the new **pod_template file** and **pod_override** functionality.

UI/UX Improvements

Perhaps one of the most welcomed sets of changes brought by Airflow 2.0 has been the visual refreshment of the **Airflow UI**.

In an effort to give users a more sophisticated and intuitive front-end experience, we've made over 30 UX improvements.

Major Features in Airflow 2.2

Airflow 2.2 Big Features

Deferrable tasks

Turn any task into a super-efficient asynchronous loop

Custom timetables

Run DAGs exactly when you want

@task.docker

Easily run Python functions in seperate docker containers

Custom Timetables

Cron expressions got us as far as regular time intervals, which is only convenient for predictable data actions. Before Airflow 2.2.0, it was impossible to schedule data activities from Monday to Friday and stop them from running over the weekend. Now that the custom timetables are finally here, the scheduling sky is the limit. With this feature, it's also now possible to look at a given data for a specific period of time.

New concept: data_interval — A period of data that a task should operate on.

Since the concept of execution_date was confusing to every new user, a better version is now available. No more *"why didn't my* ***DAG*** *run?"*, as this feature has been replaced with data_interval, which is the period of data that a task should operate on.

It includes:

- `logical_date` (aka `execution_date`)
- `data_interval_start` (same value as `execution_date for cron`)
- `data_interval_end` (aka `next_execution_date`)

Key Benefits of Custom Timetables

- Flexibility
- Run DAGs
- Introducing explicit "data interval"
- Full back-compability maintained — `schedule_interval` not going away

Bonus for Astronomer Customers only: NYSE Trading Timetable

The trading hours timetable allows users to run DAGs based on the start and end of trading hours for NYSE and Nasdaq. It includes historic trading hours as well as holidays and half-days where the markets have irregular hours.

Deferrable operators

Do you know the feeling of tasks or sensors clogging up worker resources when waiting for external systems and events? Airbnb introduced smart sensors, the first tackle to this issue. Deferrable operators go further than sensors — they are perfect for anything that submits a job to an external system and then polls for status.

With this feature, operators or sensors can postpone themselves until a lightweight async check succeeds, at which time they can resume executing. This causes the worker slot, as well as any resources it uses, to be returned to Airflow. A deferred task does not consume a worker slot while in deferral mode — instead, one triggerer process (a new component, which is the daemon process that executes the asyncio event loop) can run 100s of async deferred tasks concurrently. As a result, tasks like monitoring a job on an external system or watching for an event become far less expensive.

Custom @task decorators and @task.docker
The '@task.docker' decorator allows for running a function inside a docker container. Airflow handles putting the code into the container and returning xcom. This is especially beneficial when there are competing dependencies between Airflow and tasks that must run.

Key Benefits of Deferrable Tasks

- Turn any task into a super-efficient asynchronous loop
- Resilient against restarts
- Resilient against restarts
- Doesn't use worker resources when deferred
- **Paves the way to event-based DAGs!**

Major Features in Airflow 2.3

Note: Airflow 2.3 was released in April 2022

Dynamic Task Mapping

Dynamic task mapping permits Airflow's scheduler to trigger tasks based on context: given this input — a set of files in an S3 bucket — run these tasks. The number of tasks can change based on the number of files; you no longer have to configure a static number of tasks.

As of Airflow 2.3, you can use dynamic task mapping to hook into S3, list any new files, and run separate instances of the same task for each file. You don't have to worry about keeping your Airflow workers busy, because Airflow's scheduler automatically optimizes for available parallelism.

Dynamic task mapping not only gives you a means to easily parallelize these operations across your available Airflow workers, but also makes it easier to rerun individual tasks (e.g., in the case of failure) by giving you vastly improved visibility into the success or failure of these tasks. Imagine that you create a monolithic task to process all of the files in the S3 bucket, and that one or more steps in this task fail. In past versions of Airflow, you'd have to parse the Airflow log file generated by your monolithic task to determine which step failed and why. In 2.3, when a dynamically mapped task fails, it generates a discrete alert for that step in Airflow. This enables you to zero in on the specific task and troubleshoot from there. If appropriate, you can easily requeue the task and run it all over again.

Learn more:

GUIDE
Dynamic Tasks in Airflow
SEE GUIDE →

New Grid View

Airflow's new grid view replaces the tree view which was not ideal for representing DAGs and their topologies, since a tree cannot natively represent a DAG that has more than one path, such as a task with branching dependencies. The tree view could only represent these paths by displaying multiple, separate instances of the same task. So if a task had three paths, Airflow's tree view would show three instances of the same task — confusing even for expert users. The grid view, by contrast, is ideal for displaying complex DAGs, such as tasks that have multiple dependencies. The grid view also offers first-class support for Airflow task groups. The tree view chained task group IDs together, resulting in repetitive text and, occasionally, broken views. In the new grid view, task groups display summary information and can be expanded or collapsed as needed. The new grid view also dynamically generates lines and hover effects based on the task you're inspecting, and displays the durations of your DAG runs; this lets you quickly track performance — and makes it easier to spot potential problems. These are just a few of the improvements the new grid view brings.

A New LocalKubenetesExecutor

Airflow 2.3's new local K8s executor allows you to be selective about which tasks you send out to a new pod in your K8s cluster — i.e., you can either use a local Airflow executor to run your tasks within the scheduler service or send them out to a discrete K8s pod. Kubernetes is powerful, to be sure, but it's overkill for many use cases. (It also adds layers of latency that the local executor does not.) The upshot is that for many types of tasks — especially lightweight ones — it's faster (and, arguably, just as reliable) to run them in Airflow's local executor, as against spinning up a new K8s pod.

Storing Airflow Connections in JSON Instead of URI Format

And the new release's **ability to store Airflow connections in JSON** (rather than in Airflow URI) format is a relatively simple feature that — for some users — could nevertheless be a hugely welcomed change. Earlier versions stored connection information in Airflow's URI format, and there are cases in which that format can be tricky to work with. JSON provides a simple, human-readable alternative.

–

Altogether, Airflow 2.3 introduces more than a dozen new features, including, in addition to the above, a new command in the Airflow CLI **for reserializing DAGs, a new listener plugin API** that tracks TaskInstance state changes, a new **REST API endpoint for bulk-pausing/resuming DAGs**, and other ease-of-use (or fit-and-finish) features. Learn more about **Airflow 2.3 on our blog**.

Major Features in Airflow 2.4

Note: Airflow 2.4 was released in September 2022

When **Airflow 2.3** dropped just over four months ago, we called it one of the most important-ever releases of Apache Airflow, thanks mainly to its introduction of dynamic task mapping.

The same can be said of today's Airflow 2.4 release, which introduces a **"datasets"** feature that augments Airflow with powerful new data-driven scheduling capabilities.

In Airflow 2.4, you can use data-driven scheduling to break up large, monolithic DAGs into multiple upstream and downstream DAGs, and explicitly define dependencies between them. Doing so makes it easier for you to ensure the timely delivery of data to consumers in different roles, as well as optimize the runtime performance of your business-critical DAGs. Basically, if you have any downstream use case that depends on one or more upstream data sources, data-driven scheduling is going to transform how you use Airflow.

Other benefits include:

- Ensuring that data scientists, data analysts, and other self-service users always have access to the up-to-date data they need to do their work.
- Guaranteeing the timely delivery of the cleansed, conditioned data used to feed the business-critical KPIs, metrics, and measures that power operational dashboards.
- Enabling ML engineers and ops personnel to automate the process of retraining, testing, and redeploying production ML models, radically simplifying maintenance.

Data-driven scheduling is Airflow 2.4's top-line feature, but it isn't the only major change. The new release also introduces a `schedule` parameter that consolidates all of Airflow's extant scheduling parameters. Now, authors can use that one parameter for all their tasks. And Airflow 2.4 expands the input types that DAG authors can use with its dynamic task mapping capabilities, fleshing out that capability first introduced in 2.3.

The rapidly improving **Airflow UI** benefits from a number of ease-of-use updates, too, while — behind the scenes — Airflow 2.4 packs several improvements that promise to reduce the amount of code DAG authors and ops personnel need to write and maintain. Once again, Airflow has gotten easier to code for, operate, maintain, and govern.

Data-Driven Scheduling Explained

Until now, DAG authors had to write, debug, and maintain their own logic to manage data dependencies. In earlier versions of Airflow, they sometimes used **cross-DAG dependencies** for this purpose, but these introduced external dependencies of their own, and — because Airflow lacked a native `Dataset` class — authors still had to use sensors to trigger dependent tasks.

A more common solution was to consolidate all dependencies into a single, monolithic DAG. This guaranteed that a successful upstream task run would automatically trigger downstream tasks in the same DAG; moreover, if an upstream dependency failed, the downstream tasks would not run, so DAG authors didn't have to use (or write their own) task sensors to control for this. It was a workable approach, but monolithic DAGs can be difficult to debug and maintain; are prone to outages (because if a single task fails, the entire DAG fails); are not generally reusable; and do not give organizations a way to visualize and track datasets in Airflow.

The new release introduces a `Dataset` class that augments Airflow with the built-in logic it needs to run and manage different kinds of data-driven dependencies, exposing a single unified interface for inter-task signaling.

Data-driven scheduling works at the task level: you create a `Dataset` class and associate that with a specific task, which becomes a "producer" for one or more downstream "consumer" DAGs. A successful run by the producer task automatically triggers runs by any consumer DAGs. This makes a lot of sense: an upstream DAG might consist of hundreds of tasks, only one of which produces the dataset that a consumer DAG depends on. Or an upstream DAG might contain dozens of producer tasks, each corresponding to a separate consumer DAG.

By triggering data-driven dependencies at the task level, consumer DAGs can start sooner, enabling organizations to refresh the data consumed by their reports, dashboards, alerts, and other analytics in a timely manner.

Data-Driven Scheduling in Action

To take advantage of Airflow's new datasets feature, DAG authors:

1. Define a `Dataset` class and give a name to the dataset in their upstream DAG(s).
2. Reference that `Dataset` in all producer tasks in their upstream DAG(s).
3. Define the `Dataset` in their consumer DAG(s), if any, and use that as the schedule parameter.

A good example involves one or more producer tasks that update the dimension and fact tables in a data warehouse. Once these producer tasks run and exit successfully, Airflow "knows" to run any consumer DAGs that depend on this data.

A dataset is any output of a producer task. It can be a JSON or CSV file; a column in an Apache Iceberg table; a table, or a specific column in a table, in a database; etc. The dataset is usually used as a source of data for one or more downstream consumer DAGs.

A consumer DAG might, for example, condense multiple upstream datasets into the CSV files that business analysts load into their data visualization tools. Or it might pull data from multiple upstream CSV files, Parquet files, and database tables to create training datasets for machine learning models.

Datasets are powerful because you can chain them together, such that one consumer DAG can function as a producer for the next DAG downstream from it. It's easy to imagine a chain of datasets in which the first consumer DAG — triggered by a producer task in an upstream ETL DAG — extracts a fresh dataset from the data warehouse, triggering a run by the next consumer DAG in the sequence, which cleanses, joins, and models this dataset, outputting the results to a second, entirely new dataset. The next and last consumer DAG in this imagined sequence then uses this second dataset to compute different types of measurements over time, outputting the results to a third and final dataset.

The `Dataset` class is fungible enough to be adapted to many common event-dependent use cases — including cutting-edge use cases, like model maintenance in ML engineering. New data added to upstream sources can trigger a consumer DAG that processes and loads it into the feature store used to train your ML models. This DAG can be a producer for the next steps in your pipeline, triggering consumer DAGs which validate (and retrain if necessary) your existing models on the updated data. In theory, you can even use datasets to more or less automate the maintenance of your ML models — for example, by creating consumer DAGs that automatically test and deploy your retrained models in production.

For DAG authors charged with creating and maintaining datasets, the new `Dataset` class promises to be a huge time-saver; for organizations concerned about governing datasets — which have a tendency to proliferate and become ungoverned **data silos** — it makes tracking and visualizing them much easier. (Admittedly, this ability is fairly basic in Airflow 2.4 today — users can see a "Datasets" tab in the Airflow UI; clicking on it displays the extant dataset objects — but is poised to improve over time.)

Beyond Datasets, One Scheduling Parameter to Consolidate Them All

Another big change in Airflow 2.4 is its consolidated `schedule` parameter, which was prompted in part by the new `Dataset` class.

In prior versions of Airflow, DAG authors used a pair of parameters — `schedule_interval` and `timetable` — to tell Airflow when to run their DAGs. To accommodate `Dataset` scheduling in Airflow 2.4, maintainers initially planned to introduce a third parameter, `schedule_on`. Ultimately, though, they decided to consolidate the functions of all three parameters into a new one: `schedule`, which can accept cron expressions, as well as timedelta, timetable, and dataset objects.

Fortunately, updating your DAGs to use the new `schedule` parameter is extremely simple. In existing DAGs, authors can paste it in as a drop-in replacement for the deprecated `schedule_interval` and `timetable` parameters. For new, dataset-aware DAGs, authors can use the `schedule` parameter to schedule DAG runs for both upstream DAGs that have producer tasks and any consumer DAGs that depend on upstream producer datasets.

Although `schedule_interval` and `timetable` are officially deprecated as of Airflow 2.4, Airflow will continue to support all three parameters for now. Support for the deprecated ones won't be sunsetted until a future major release, giving organizations plenty of time to update their DAGs. And because `schedule` is literally a drop-in replacement for the older parameters, implementing this change should be as easy as using an editor to find-and-replace them. In practice, many organizations will likely write scripts to bulk-automate this process.

Other Airflow Improvements Simplify DAG Writing and Maintenance, Reduce Custom Coding

Another scheduling-related change in Airflow 2.4 is `CronTriggerTimetable`, a **cron timetable** that tells Airflow's scheduler to use cron-like conventions. The background to this is that, by default, Airflow runs daily task instances 24 hours after their specified start time. So, if an author scheduled an end-of-week task to run on Sunday at 12 AM, it would actually run on Monday at 12 AM. (There's a similar default interval for hourly tasks: if you schedule a task to run from 9-10 AM, it won't actually start running until 10 AM.) A delay interval makes sense for certain types of batch jobs (e.g., end-of-day reports), but is a constant source of confusion for new Airflow users. Now, tasks can run on the day and at the time you specify, in accordance with cron conventions.

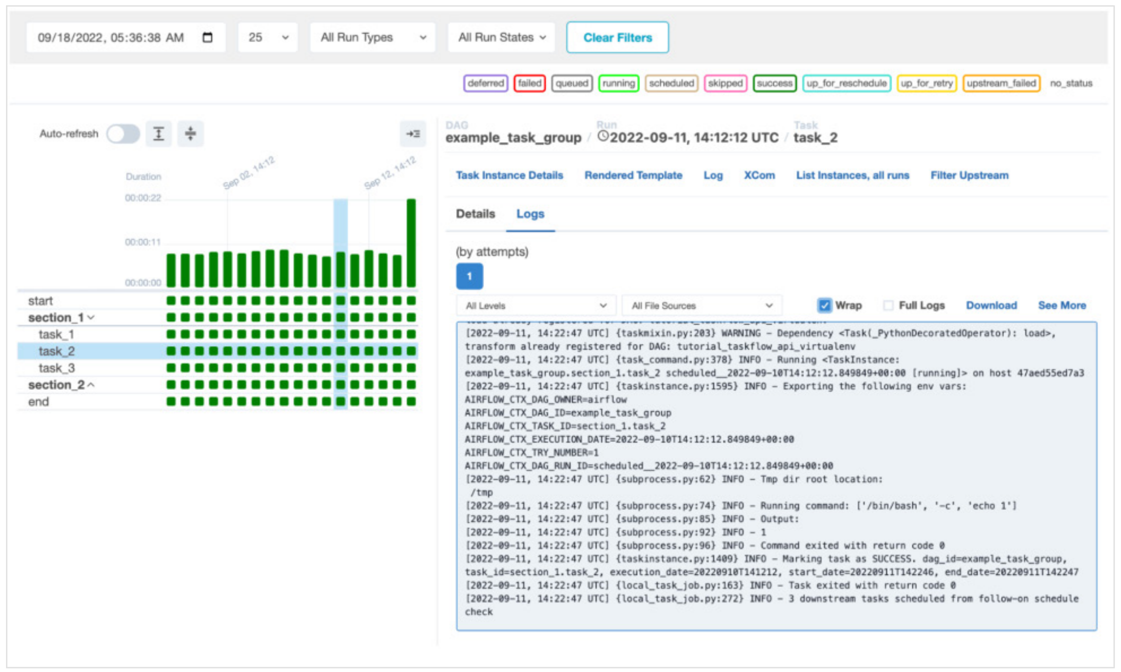
The new 2.4 release also augments Airflow's dynamic task mapping capabilities, with support for expanded input types — namely `expand_kwargs` and `expand_zip`. `Expand_kwargs` is an **escape hatch** of sorts that enables DAG authors to address a large number of use cases; `expand_zip` , by contrast, is quite specific, albeit also quite useful: it tells Airflow to "zip" two lists together, such that a,b,c and d,e,f become (a,d), (b,e), and (c,f).

These additions, along with several others, help to reduce the amount of code that DAG authors need to write and maintain, because — as with the data-driven scheduling logic undergirding Airflow's new `Dataset` class — the necessary logic has now moved into Airflow itself.

Another example is a new UI option that administrators can use to **configure a global task retry delay**. Airflow's default task retry delay is 300 seconds; previously, authors could change this by **hardcoding it into their tasks**. You can still do this — e.g., if you want to use a time limit other than the global value — but the new feature is a labor-saving alternative.

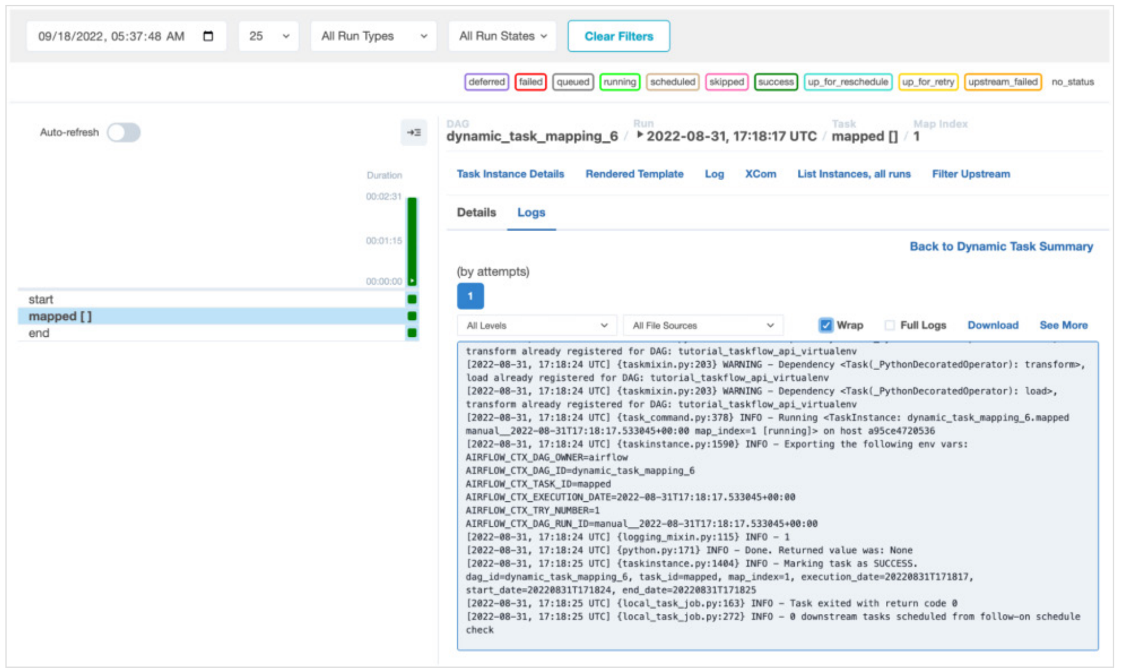
Similarly, Airflow 2.4's option to **reuse decorated task groups** makes it much easier for authors to reuse collections of tasks to accommodate repeatable patterns in DAGs.

The Airflow UI **is a locus of ongoing innovation**. Also new in Airflow 2.4 is an ability to **drill-down into logs from the default grid view**, which enables users — either DAG authors or DevOps/SRE personnel — to click on a task (represented by a block in Airflow's grid view) to see the log file associated with it, rather than being redirected to another page. The idea is to expose as much useful information as possible in an integrated UI workflow, so users don't have to switch contexts.



```
[2022-09-11, 14:22:47 UTC] {taskmixin.py:203} WARNING - Dependency <Task(_PythonDecoratedOperator): load>,
transform already registered for DAG: tutorial_taskflow_api_virtualenv
[2022-09-11, 14:22:47 UTC] {task_command.py:378} INFO - Running <TaskInstance:
example_task_group.section_1.task_2 scheduled__2022-09-10T14:12:12.849849+00:00 [running]> on host 47aed55ed7a3
[2022-09-11, 14:22:47 UTC] {taskinstance.py:1595} INFO - Exporting the following env vars:
AIRFLOW_CTX_DAG_OWNER=airflow
AIRFLOW_CTX_DAG_ID=example_task_group
AIRFLOW_CTX_TASK_ID=section_1.task_2
AIRFLOW_CTX_EXECUTION_DATE=2022-09-10T14:12:12.849849+00:00
AIRFLOW_CTX_TRY_NUMBER=1
AIRFLOW_CTX_DAG_RUN_ID=scheduled__2022-09-10T14:12:12.849849+00:00
[2022-09-11, 14:22:47 UTC] {subprocess.py:62} INFO - Tmp dir root location:
 /tmp
[2022-09-11, 14:22:47 UTC] {subprocess.py:74} INFO - Running command: ['/bin/bash', '-c', 'echo 1']
[2022-09-11, 14:22:47 UTC] {subprocess.py:85} INFO - Output:
[2022-09-11, 14:22:47 UTC] {subprocess.py:92} INFO - 1
[2022-09-11, 14:22:47 UTC] {subprocess.py:96} INFO - Command exited with return code 0
[2022-09-11, 14:22:47 UTC] {taskinstance.py:1409} INFO - Marking task as SUCCESS. dag_id=example_task_group,
task_id=section_1.task_2, execution_date=20220910T141212, start_date=20220911T142246, end_date=20220911T142247
[2022-09-11, 14:22:47 UTC] {local_task_job.py:163} INFO - Task exited with return code 0
[2022-09-11, 14:22:47 UTC] {local_task_job.py:272} INFO - 3 downstream tasks scheduled from follow-on schedule
check
```

Airflow 2.4 UI showing a task group



```
transform already registered for DAG: tutorial_taskflow_api_virtualenv
[2022-08-31, 17:18:24 UTC] {taskmixin.py:203} WARNING - Dependency <Task(_PythonDecoratedOperator): transform>,
load already registered for DAG: tutorial_taskflow_api_virtualenv
[2022-08-31, 17:18:24 UTC] {taskmixin.py:203} WARNING - Dependency <Task(_PythonDecoratedOperator): load>,
transform already registered for DAG: tutorial_taskflow_api_virtualenv
[2022-08-31, 17:18:24 UTC] {task_command.py:378} INFO - Running <TaskInstance: dynamic_task_mapping_6.mapped
manual__2022-08-31T17:18:17.533045+00:00 map_index=1 [running]> on host a95ce4720536
[2022-08-31, 17:18:24 UTC] {taskinstance.py:1590} INFO - Exporting the following env vars:
AIRFLOW_CTX_DAG_OWNER=airflow
AIRFLOW_CTX_DAG_ID=dynamic_task_mapping_6
AIRFLOW_CTX_TASK_ID=mapped
AIRFLOW_CTX_EXECUTION_DATE=2022-08-31T17:18:17.533045+00:00
AIRFLOW_CTX_TRY_NUMBER=1
AIRFLOW_CTX_DAG_RUN_ID=manual__2022-08-31T17:18:17.533045+00:00
[2022-08-31, 17:18:24 UTC] {logging_mixin.py:115} INFO - 1
[2022-08-31, 17:18:24 UTC] {python.py:171} INFO - Done. Returned value was: None
[2022-08-31, 17:18:25 UTC] {taskinstance.py:1404} INFO - Marking task as SUCCESS.
dag_id=dynamic_task_mapping_6, task_id=mapped, map_index=1, execution_date=20220831T171817,
start_date=20220831T171824, end_date=20220831T171825
[2022-08-31, 17:18:25 UTC] {local_task_job.py:163} INFO - Task exited with return code 0
[2022-08-31, 17:18:25 UTC] {local_task_job.py:272} INFO - 0 downstream tasks scheduled from follow-on schedule
check
```

Airflow 2.4 UI showing a dynamically mapped task

Flexible Deferrable Operators to Replace Smart Sensors

For all its useful improvements, Airflow 2.4 is taking one capability away: **smart sensors**, first introduced in Airflow 2.0 (December 2020) as an experimental feature — a short-term fix, explicitly excepted from Airflow's **versioning policy** — which DAG authors could use to trigger tasks to run in response to specific events. Smart sensors were useful, but they also had a few drawbacks: they worked for only a limited range of sensor-style workloads, had no redundancy, and had to run in the context of a custom DAG. **Airflow 2.2** introduced a permanent solution to this problem — deferrable operators, a term used to describe a category of hooks, sensors, and operators capable of asynchronous operation — and included two novel async sensors, **TimeSensorAsync** and **DateTimeSensorAsync**. **Deferrable operators** are a more robust and flexible solution for supporting sensor-style event triggering. They support a broader range of use cases and workloads. Functionally, they far outstrip what you can do with smart sensors.

When Airflow 2.2.4 was released in February, support for smart sensors **was officially deprecated**, with a warning that support would be terminated as of Airflow 2.4; now, that Airflow 2.4 is available, smart sensors are no longer supported. The Airflow community produced only **one known pre-built smart sensor**, which has been superseded by an **equivalent deferrable operator**.

If you wrote smart sensors of your own, you can now **reimplement them as deferrable operators**. If you have questions about doing this, consider reaching out to us at Astronomer. Not only did we **build the first deferrable operators for Airflow**, we made them available under the Apache 2.0 license, and we've helped our customers reimplement their own smart sensors as deferrable operators. We can definitely be of assistance to you as you manage this breaking change.

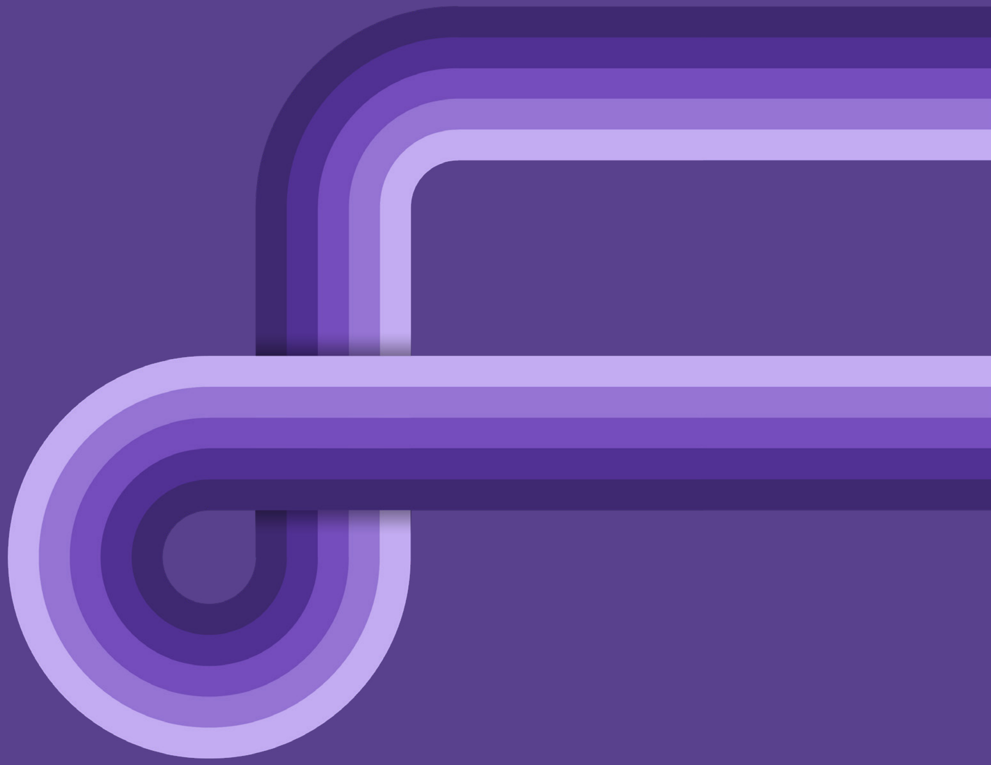
An Upgrade That Cuts Complexity and Promotes Timely Delivery of Data

There's a lot of amazing stuff in Airflow 2.4, but by any criteria, its support for datasets is the breakout star. Data-driven scheduling is going to change what organizations can do with Airflow. By augmenting Airflow with the built-in logic it needs to automatically manage different kinds of data-dependent events, the feature makes it much easier for data teams to break up large, monolithic DAGs into smaller, function- or process-specific DAGs — i.e., "datasets." Data-driven scheduling allows organizations to more effectively optimize how, when, or under what conditions their pipelines run, enabling them to prioritize the timely delivery of data for business-critical processes.

For DAG authors, Airflow's new `Dataset` class does away with a big source of tedium and frustration: the need to design elaborate logic to manage dependencies and control for different kinds of task failures between their DAGs. And from an operational standpoint, breaking up monolithic DAGs into discrete datasets ensures that they run reliably and are much easier to troubleshoot and maintain.

For the moment, data-driven scheduling in Airflow does have one significant limitation: dataset dependencies don't (yet) span separate Airflow deployments. You cannot have a run by a producer task in one deployment triggering a run by a consumer DAG in another. But for a novel implementation of a major new feature, data-driven scheduling is already hugely useful.

As this feature improves, it will become easier to track and visualize datasets within Airflow, as well as to manage and improve the quality of the data used to produce them. Organizations will be able to better understand how and for whom they're producing datasets, along with what those people and teams are doing with them. They will be able to more easily identify redundant, infrequently used, or incomplete datasets, along with datasets that contain sensitive information, enabling them to more effectively govern the data produced and consumed by self-service users. The datasets capabilities we're getting in Airflow 2.4 are just the beginning.



Ready for more?

Discover our guides where we cover everything from introductory content to advanced tutorials around Airflow.

Discover Guides

6. Useful Resources to Get Started with Airflow

Airflow Essentials

WEBINAR
Intro to Airflow
SEE WEBINAR →

WEBINAR
Intro to Data Orchestration with Apache Airflow
SEE WEBINAR →

VIDEO
Coding Your First DAG for Beginners
SEE VIDEO →

WEBINAR
Best Practices For Writing DAGs in Airflow 2
SEE WEBINAR →

GUIDE
Understanding the Airflow UI
SEE GUIDE →

GUIDE
Airflow Executors Explained
SEE GUIDE →

GUIDE
Managing Your Connections in Apache Airflow
SEE GUIDE →

GUIDE
DAG Writing Best Practices
SEE GUIDE →

GUIDE
Using Airflow to Execute SQL
SEE GUIDE →

GUIDE
Using TaskGroups in Airflow
SEE GUIDE ⟶

GUIDE
Debugging DAGs
SEE GUIDE ⟶

GUIDE
Rerunning DAGs
SEE GUIDE ⟶

GUIDE
Airflow Decorators
SEE GUIDE ⟶

GUIDE
Deferrable Operators
SEE GUIDE ⟶

WEBINAR
Scaling out Airflow
SEE WEBINAR ⟶

WEBINAR
Scheduling in Airflow
SEE WEBINAR ⟶

Integrations

Make the most of Airflow by connecting it to other tools.

Airflow + dbt

ARITCLE
Building a Scalable Analytics Architecture With Airflow and dbt
SEE ARTICLE ⟶

ARITCLE
Airflow and dbt, Hand in Hand
SEE ARTICLE ⟶

GUIDE
Integrating Airflow and dbt
SEE GUIDE ⟶

Airflow + Great Expectations

GUIDE
Integrating Airflow and Great Expectations
SEE GUIDE ⟶

Airflow + Azure

GUIDE
Executing Azure Data Factory Pipelines with Airflow
SEE GUIDE ⟶

GUIDE
Executing Azure Data Explorer Queries with Airflow
SEE GUIDE ⟶

GUIDE
Orchestrating Azure Container Instances with Airflow
SEE GUIDE ⟶

Airflow + Redshift

GUIDE
Orchestrating Redshift Operations from Airflow
SEE GUIDE ⟶

Airflow + Sagemaker

GUIDE
Using Airflow with SageMaker
SEE GUIDE ⟶

Airflow + Notebooks

GUIDE
Executing Notebooks with Airflow
SEE GUIDE ⟶

Airflow + Kedro

GUIDE
Deploying Kedro Pipelines to Apache Airflow
SEE GUIDE ⟶

Airflow + Databricks

GUIDE
Orchestrating Databricks Jobs with Airflow
SEE GUIDE ⟶

Airflow + Talend

GUIDE
Executing Talend Jobs with Airflow
SEE GUIDE →

Airflow + AWS Lambda

GUIDE
Best Practices Calling AWS Lambda from Airflow
SEE GUIDE →

Setting Up Error Notifications in Airflow Using Email, Slack, and SLAs

GUIDE
Error Notifications in Airflow
SEE GUIDE →

Can’t find the integration you’re looking for?

Check out the Astronomer Registry — a go-to discovery and distribution hub for Apache Airflow integrations.

Browse Registry

7. About Astronomer

With its rapidly growing number of downloads and contributors, Airflow is at the center of the data management conversation—a conversation Astronomer is helping to drive.

Our commitment is evident in our people. We've got a robust Airflow Engineering team and sixteen active committers on board, including seven PMC members: **Ash Berlin-Taylor, Kaxil Naik, Daniel Imberman, Jed Cunningham, Bolke de Bruin, Sumit Maheshwari, and Ephraim Anierobi.**

A significant portion of the work all these people do revolves around Airflow releases—creating new features, testing, troubleshooting, and ensuring that the project continues to improve and grow. Their hard work, combined with that of the community at large, allowed us to deliver Airflow 2.0 in late 2020, Airflow 2.2 in 2021, and Airflow 2.3 and 2.4 in 2022.

With every new release, Astronomer is unlocking more of Airflow's potential and moving closer to the goal of democratizing the use of Airflow, so that all members of the data team can work with or otherwise benefit from it. We've built Astro—the essential data orchestration platform. With the strength and vibrancy of Airflow at its core, Astro offers a scalable, adaptable, and efficient blueprint for successful data orchestration.

If you'd like to learn more about how Astronomer drives the Apache Airflow project together with the community check out our recent article.

ASTRONOMER

Thank you

We hope you've enjoyed our guide to Airflow. Please follow us on Twitter and LinkedIn, and share your feedback, if any.

Start building your next-generation data platform with Astro

Get Started

Experts behind this ebook:
Viraj Parekh | Field CTO at Astronomer
Kenten Danas | Lead Developer Advocate at Astronomer
Jake Witz | Technical Writer at Astronomer

ASTRONOMER