

REVISED EDITION

DAGs: The Definitive Guide

Everything you need to know about Airflow DAGs



Powered by Astronomer

Editor's Note

Welcome to the ultimate guide to Apache Airflow DAGs, brought to you by the Astronomer team. This ebook covers everything you need to know to work with DAGs, from the building blocks that make them up to best practices for writing them, dynamically generating them, testing and debugging them, and more. It's a guide written by practitioners for practitioners.

Follow us on Twitter and LinkedIn!



Table of Contents

04	DAGs: Where to Begin?
04	What Exactly is a DAG?
11	From Operators to DagRuns: Implementing DAGs in Airflow
14	DAG Building Blocks
14	Scheduling and Timetables in Airflow
36	Operators 101
46	Hooks 101
53	Sensors 101
55	Deferrable Operators
62	DAG Design
62	DAG Writing Best Practices in Apache Airflow
74	Passing Data Between Airflow Tasks
87	Using Tasks Group in Airflow
96	Cross-DAG Dependencies
114	Dynamically Generating DAGs
132	Testing Airflow DAGs
144	Debugging DAGs
144	7 Common Errors to Check when Debugging DAGs
158	Error Notifications in Airflow

1. DAGs

Where to Begin?

D I R E C T E D

A C Y C L I C

G R A P H

What Exactly is a DAG?

A DAG is a Directed Acyclic Graph – a conceptual representation of a series of activities, or, in other words, a mathematical abstraction of a data pipeline. Although used in different circles, both terms, DAG and data pipeline, represent an almost identical mechanism. In a nutshell, a DAG (or a pipeline) defines a sequence of execution stages in any non-recurring algorithm.

The DAG acronym stands for:

DIRECTED – In general, if multiple tasks exist, each must have at least one defined upstream (previous) or downstream (subsequent) task, or one or more of both. (It's important to note however, that there are also DAGs that have multiple parallel tasks – meaning no dependencies.)

ACYCLIC – No task can create data that goes on to reference itself. That could cause an infinite loop, which could give rise to a problem or two. There are no cycles in DAGs.

GRAPH – In mathematics, a graph is a finite set of nodes, with vertices connecting the nodes. In the context of data engineering, each node in a graph represents a task. All tasks are laid out in a clear structure, with discrete processes occurring at set points and transparent relationships with other tasks.

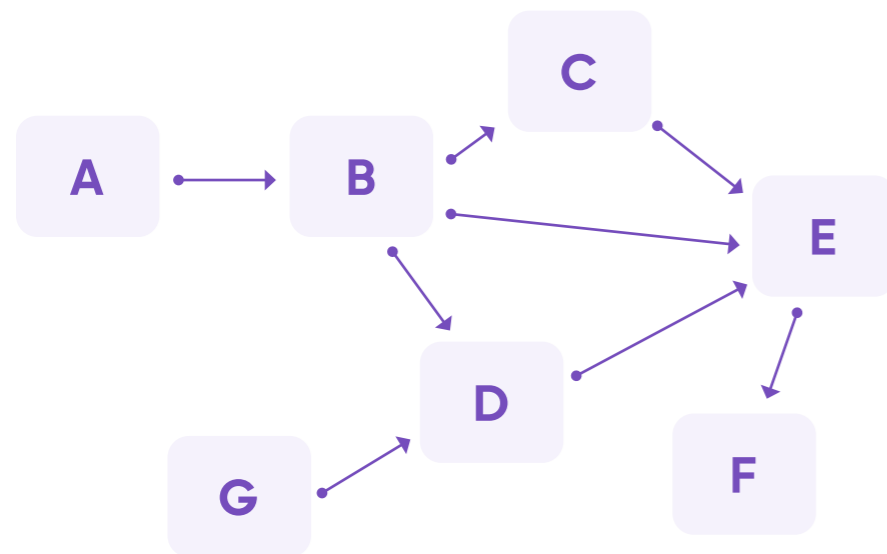
When Are DAGs Useful?

Manually building workflow code challenges the productivity of engineers, which is one reason there are a lot of helpful tools out there for automating the process, such as [Apache Airflow](#). A great first step to efficient automation is to realize that DAGs can be an optimal solution for moving data in nearly every computing-related area.

“At Astronomer, we believe using a code-based data pipeline tool like [Airflow](#) should be a standard,” says Kenten Danas, Lead Developer Advocate at Astronomer. There are many reasons for this, but these high-level concepts are crucial:

- **Code-based pipelines are extremely dynamic.** If you can write it in code, then you can do it in your [data pipeline](#).
- **Code-based pipelines are highly extensible.** You can integrate with basically every system out there, as long as it has an API.
- **Code-based pipelines are more manageable:** Since everything is in code, it can integrate seamlessly into your source controls CI/CD and general developer workflows. There’s no need to manage external things differently.

An Example of a DAG



Consider the directed acyclic graph above. In this DAG, each vertex (line) has a specific direction (denoted by the arrow) connecting different nodes.

This is the key quality of a directed graph: data can follow only in the direction of the vertex. In this example, data can go from A to B, but never B to A. In the same way that water flows through pipes in one direction, data must follow in the direction defined by the graph. Nodes from which a directed vertex extends are considered upstream, while nodes at the receiving end of a vertex are considered downstream.

In addition to data moving in one direction, nodes never become self-referential. That is, they can never inform themselves, as this could create an infinite loop. So data can go from A to B to C/D/E, but once there, no subsequent process can ever lead back to A/B/C/D/E as data moves down the graph. Data coming from a new source, such as node G, can still lead to nodes that are already connected, but no subsequent data can be passed back into G. This is the defining quality of an acyclic graph.

Why must this be true for data pipelines? If F had a downstream process in the form of D, we would see a graph where D informs E, which informs F, which informs D, and so on. It creates a scenario where the pipeline could run indefinitely without ever ending. Like water that never makes it to the faucet, such a loop would be a waste of data flow.

To put this example in real-world terms, imagine the DAG above represents a data engineering story:

- **Node A** could be the code for pulling data out of an API.
- **Node B** could be the code for anonymizing the data and dropping any IP address.
- **Node D** could be the code for checking that no duplicate record IDs exist.
- **Node E** could be putting that data into a database.
- **Node F** could be running a SQL query on the new tables to update a dashboard.

DAGs in Airflow

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:

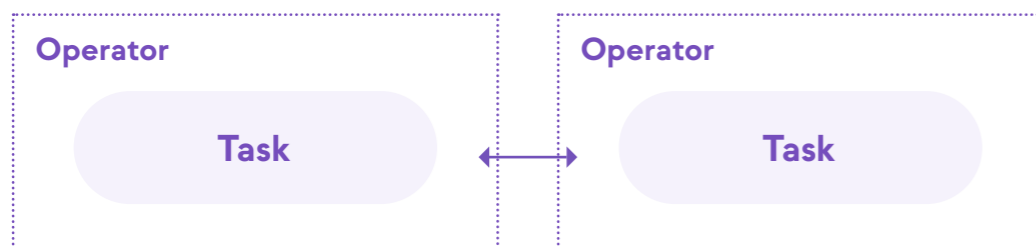
- **DAG dependencies** ensure that your data tasks are executed in the same order every time, making them reliable for your everyday data infrastructure.
- **The graphing component of DAGs** allows you to visualize dependencies in Airflow's user interface.
- **Because every path in a DAG is linear**, it's easy to develop and test your data pipelines against expected outcomes.

An Airflow DAG starts with a task written in Python. You can think of tasks as the nodes of your DAG: Each one represents a single action, and it can be dependent on both upstream and downstream tasks.

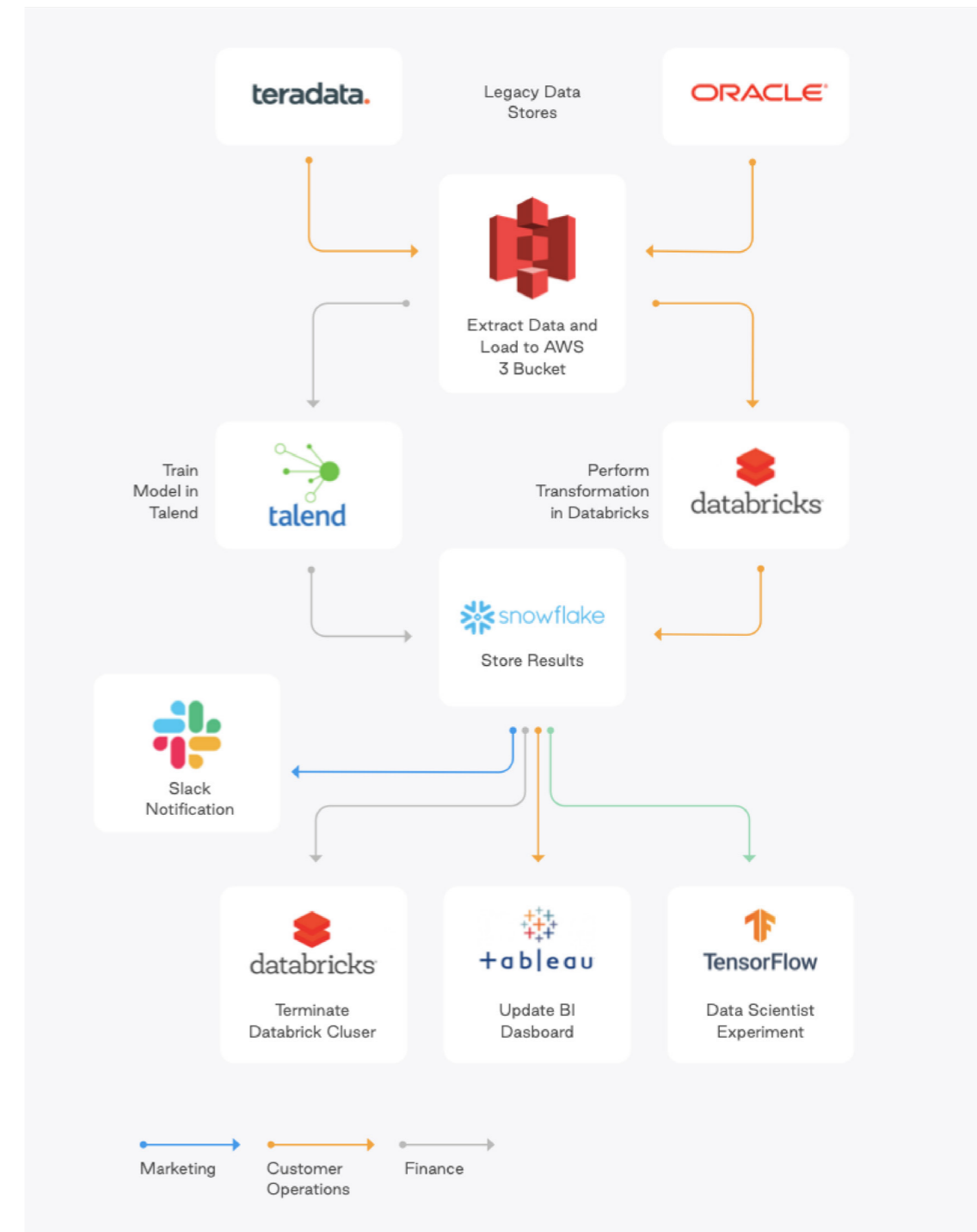
Tasks are wrapped by operators, which are the building blocks of Airflow, defining the behavior of their tasks. For example, a Python Operator task will execute a Python function, while a task wrapped in a Sensor Operator will wait for a signal before completing an action.

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

DAG



You can see the flexibility of DAGs in the following real-world example:



Using a single DAG (like the Customer Operations one shown in yellow), you are able to:

- **Extract data from a legacy data store and load it into an AWS S3 bucket.**
- **Either train a data model or complete a data transformation, depending on the data you're using.**
- **Store the results of the previous action in a database.**
- **Send information about the entire process to various metrics and reporting systems.**

Organizations use DAGs and pipelines that integrate with separate, interface-driven tools to extract, load, and transform data. But without an [orchestration platform like Astro from Astronomer](#), these tools aren't talking to each other. If there's an error during the loading, the other tools won't know about it. The transformation will be run on bad data, or yesterday's data, and deliver an inaccurate report. It's easy to avoid this, though — a data orchestration platform can sit on top of everything, tying the DAGs together, orchestrating the dataflow, and alerting in case of failures. Overseeing the end-to-end life cycle of data allows businesses to maintain interdependency across all systems, which is vital for effective management of data.

From Operators to DagRuns: Implementing DAGs in Airflow

While DAGs are simple structures, defining them in code requires some more complex infrastructure and concepts beyond nodes and vertices. This is especially true when you need to execute DAGs on a frequent, reliable basis.

Airflow includes a number of structures that enable us to define DAGs in code. While they have unique names, they roughly equate to various concepts that we've discussed in the book thus far.

How Work Gets Executed in Airflow

- **Operators are the building blocks of Airflow.**
Operators 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. We'll cover various types of operators in the Operators 101 chapter.
- **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.

- **Tasks are nodes in a DAG.**

In Airflow, a DAG is a group of tasks that have been configured to run in a directed, acyclic manner. Airflow's Scheduler parses DAGs to find tasks which are ready for execution based on their dependencies. If a task is ready for execution, the Scheduler sends it to an Executor.

A real-time run of a task is called a task instance (it's also common to call this a task run). Airflow logs information about task instances, including their running time and status, in a metadata database.

- **A DAG run is a single, specific execution of a DAG.**

If a task instance is a run of a task, then a DAG run is simply an instance of a complete DAG that has run or is currently running. At the code level, a DAG becomes a DAG run once it has an `execution_date`. Just like with task instances, information about each DAG run is logged in Airflow's metadata database.

ASTRONOMER

Want to know more?

Check out our comprehensive webinars, where Airflow experts dive deeper into DAGs.

Best Practices for Writing DAGs in Airflow 2

[SEE WEBINAR →](#)

Iterative Data Quality in Airflow DAGs

[SEE WEBINAR →](#)

Improve Your DAGs with Hidden Airflow Features

[SEE WEBINAR →](#)

2. DAG Building Blocks

Scheduling and Timetables in Airflow

One of the fundamental features of Apache Airflow is the ability to schedule jobs. Historically, Airflow users could schedule their DAGs by specifying a `schedule` with a cron expression, a `timedelta` object, or a preset Airflow schedule.

Timetables, released in Airflow 2.2, brought new flexibility to scheduling. Timetables allow users to create their own custom schedules using Python, effectively eliminating the limitations of cron. With timetables, you can now schedule DAGs to run at any time for any use case.

Additionally, Airflow 2.4 introduced datasets and the ability to schedule your DAGs on updates to a dataset rather than a time-based schedule. A more in-depth explanation on these features can be found in the [Datasets and Data Driven Scheduling](#) in Airflow guide.

In this guide, we'll walk through Airflow scheduling concepts and the different ways you can schedule a DAG with a focus on timetables. For additional instructions check out our [Scheduling in Airflow webinar](#).

Note: All code in this guide can be found in [this repo](#).

Assumed knowledge

To get the most out of this guide, you should have knowledge of:

- Basic Airflow concepts. See [Introduction to Apache Airflow](#).
- Configuring Airflow DAGs. See [Introduction to Airflow DAGs](#).
- Date and time modules in Python3. See the [Python documentation on the datetime package](#).

Scheduling concepts

There are a couple of terms and parameters in Airflow that are important to understand related to scheduling.

- **Data Interval:** The data interval is a property of each DAG run that represents the period of data that each task should operate on. For example, for a DAG scheduled hourly each data interval will begin at the top of the hour (minute 0) and end at the close of the hour (minute 59). The DAG run is typically executed at the end of the data interval, depending on whether your DAG's schedule has "gaps" in it.
- **Logical Date:** The logical date of a DAG run is the same as the start of the data interval. It does not represent when the DAG will actually be executed. Prior to Airflow 2.2, this was referred to as the execution date.
- **Timetable:** The timetable is a property of a DAG that dictates the data interval and logical date for each DAG run (i.e. it determines when a DAG will be scheduled).
- **Run After:** The earliest time the DAG can be scheduled. This date is shown in the Airflow UI, and may be the same as the end of the data interval depending on your DAG's timetable.
- **Backfilling and Catchup:** We won't cover these concepts in depth here, but they can be related to scheduling. We recommend reading [the Apache Airflow documentation](#) on them to understand how they work and whether they're relevant for your use case.

Note: In this guide we do not cover the `execution_date` concept, which has been deprecated as of Airflow 2.2. If you are using older versions of Airflow, review [this doc](#) for more on `execution_date`.

Parameters

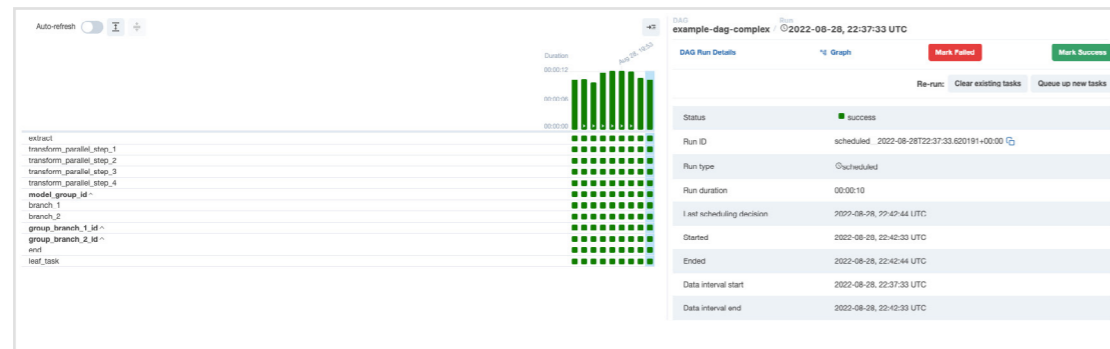
The following parameters are derived from the concepts described above and are important for ensuring your DAG runs at the correct time.

- **`data_interval_start`:** A datetime object defining the start date and time of the data interval. A DAG's timetable will return this parameter for each DAG run. This parameter is either created automatically by Airflow, or can be specified by the user when implementing a custom timetable
- **`data_interval_end`:** A datetime object defining the end date and time of the data interval. A DAG's timetable will return this parameter for each DAG run. This parameter is either created automatically by Airflow, or can be specified by the user when implementing a custom timetable
- **`schedule`:** A parameter that can be set at the DAG level to define when that DAG will be run. It accepts cron expressions, timedelta objects, timetables, and lists of datasets.
- **`start_date`:** The first date your DAG will be executed. This parameter is required for your DAG to be scheduled by Airflow.
- **`end_date`:** The last date your DAG will be executed. This parameter is optional.

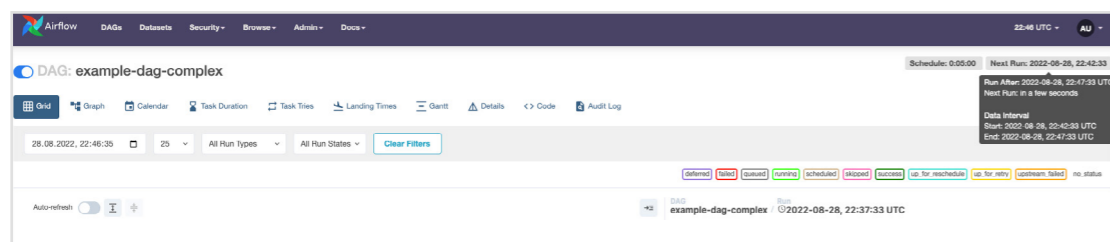
Note: Note In Airflow 2.3 or older, the `schedule` parameter is called `schedule_interval` and only accepts cron expressions or timedelta objects. Additionally, timetables have to be passed using the `timetable` parameter, which is deprecated in Airflow 2.4+. In versions of Airflow 2.2 and earlier, specifying `schedule_interval` is the only way to define a DAG's schedule.

Example

As a simple example of how these concepts work together, say we have a DAG that is scheduled to run every 5 minutes. Looking at the most recent DAG run, the logical date is `2022-08-28 22:37:33` (shown below **Run** next to the DAG name in the UI), which is the same as the **Data interval start** shown in the bottom right corner of in the screenshot below. The logical date is also the timestamp that will be incorporated into the **Run ID** of the DAG run which is how the DAG run is identified in the Airflow metadata database. The **Data interval end** is 5 minutes later.



If we look at the next DAG run in the UI, the logical date is `2022-08-28 22:42:33`, which is shown as the **Next Run** timestamp in the UI. This is 5 minutes after the previous logical date, and the same as the **Data interval end** of the last DAG run because there are no gaps in the schedule. If we hover over **Next Run**, we can see that **Run After**, which is the date and time that the next DAG run will actually start, is also the same as the next DAG run's **Data interval end**:



In summary we've described 2 DAG runs:

- DAG run 1 with the **Run ID** `scheduled_2022-08-28T22:37:33.620191+00:00` has a logical date of `2022-08-28 22:37:33`, a **Data interval start** of `2022-08-28 22:37:33` and a **Data interval end** of `2022-08-28 22:42:33`. This DAG run will actually start at `2022-08-28 22:42:33`.
- DAG run 2 with the **Run ID** `scheduled_2022-08-28T22:42:33.617231+00:00` has a logical date of `2022-08-28 22:42:33` (shown as **Next Run** in the UI in the second screenshot), a **Data interval start** of `2022-08-28 22:42:33` and a **Data interval end** of `2022-08-28 22:47:33`. This DAG run will actually start at `2022-08-28 22:47:33` (shown as **Run After** in the UI in the second screenshot).

In the sections below, we'll walk through how to use cron-based schedule, timetables, or datasets to schedule your DAG.

Cron-based schedules

For pipelines with simple scheduling needs, you can define a `schedule` in your DAG using:

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

Setting a cron-based schedule

Cron expressions

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

If you need help creating the correct cron expression, [crontab guru](#) is a great resource.

Cron presets

Airflow can utilize cron presets for common, basic schedules.

For example, `schedule='@hourly'` will schedule the DAG to run at the beginning of every hour. For the full list of presets, check out the [Airflow documentation](#). If your DAG does not need to run on a schedule and will only be triggered manually or externally triggered by another process, you can set `schedule=None`.

Timedelta objects

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

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

Cron-based schedules & the logical date

Airflow was originally developed for ETL under the expectation that data is constantly flowing in from some source and then will be summarized on a regular interval. If you want to summarize Monday's data, you can only do it after Monday is over (Tuesday at 12:01 AM). However, this assumption has turned out to be ill-suited to the many other things Airflow is being used for now. This discrepancy is what led to Timetables, which were introduced in Airflow 2.2.

Each DAG run therefore has a `logical_date` that is separate from the time that the DAG run is expected to begin (`logical_date` was called `execution_date` before Airflow 2.2). A DAG run is not actually allowed to run until the `logical_date` for the following DAG run has passed. So if you are running a daily DAG, Monday's DAG run will not actually execute until Tuesday. In this example, the `logical_date` would be Monday 12:01 AM, even though the DAG run will not actually begin until Tuesday 12:01 AM.

If you want to pass a timestamp to the DAG run that represents “the earliest time at which this DAG run could have started”, use `{{ next_ds }}` from the [jinja templating macros](#).

Note: It is best practice to make each DAG run idempotent (able to be re-run without changing the result) which precludes using `datetime.now()`.

Limitations of cron-based schedules

The relationship between a DAG's `schedule` and its `logical_date` leads to particularly unintuitive results when the spacing between DAG runs is irregular. The most common example of irregular spacing is when DAGs run only during business days (Mon-Fri). In this case, the DAG run with an `logical_date` of Friday will not run until Monday, even though all of Friday's data will be available on Saturday. This means that a DAG whose desired behavior is to summarize results at the end of each business day actually cannot be set using only the `schedule`. In versions of Airflow prior to 2.2, one must instead schedule the DAG to run every day (including the weekend) and include logic in the DAG itself to skip all tasks for days on which the DAG doesn't really need to run.

In addition, it is difficult or impossible to implement situations like the following using a traditional schedule:

- Schedule a DAG at different times on different days, like 2pm on Thursdays and 4pm on Saturdays.
- Schedule a DAG daily except for holidays.
- Schedule a DAG at multiple times daily with uneven intervals (e.g. 1pm and 4:30pm).

In the next section, we'll describe how these limitations were addressed in Airflow 2.2 with the introduction of timetables.

Timetables

[Timetables](#), introduced in Airflow 2.2, address the limitations of cron expressions and `timedelta` objects by allowing users to define their own schedules in Python code. All DAG schedules are ultimately determined by their internal timetable and if a cron expression or `timedelta` object is not sufficient for your use case, you can define your own.

Custom timetables can be registered as part of an Airflow plugin. They must be a subclass of `Timetable`, and they should contain the following methods, both of which return a `DataInterval` with a start and an end:

- `next_dagrun_info`: Returns the data interval for the DAG's regular schedule
- `infer_manual_data_interval`: Returns the data interval when the DAG is manually triggered

Below we'll show an example of implementing these methods in a custom timetable.

Example custom timetable

For this implementation, let's run our DAG at 6:00 and 16:30. Because this schedule has run times with differing hours and minutes, it can't be represented by a single cron expression. But we can easily implement this schedule with a custom timetable!

To start, we need to define the `next_dagrun_info` and `infer_manual_data_interval` methods. Before diving into the code, it's helpful to think through what the data intervals will be for the schedule we want. Remember that the time the DAG runs (`run_after`) should be the end of the data interval since our interval has no gaps. So in this case, for a DAG that we want to run at 6:00 and 16:30, we have two different alternating intervals:

- Run at 6:00: Data interval is from 16:30 on the previous day to 6:00 on the current day
- Run at 16:30: Data interval is from 6:00 to 16:30 on the current day

With that in mind, first we'll define `next_dagrun_info`. This method provides Airflow with the logic to calculate the data interval for scheduled runs. It also contains logic to handle the DAG's `start_date`, `end_date`, and `catch-up` parameters. To implement the logic in this method, we use the [Pendulum package](#), which makes dealing with dates and times simple. The method looks like this:

```

1 def next_dagrun_info(
2     self,
3     *,
4     last_automated_data_interval: Optional[DataInterval],
5     restriction: TimeRestriction,
6 ) -> Optional[DagRunInfo]:
7     if last_automated_data_interval is not None: # There
8 was a previous run on the regular schedule.
9         last_start = last_automated_data_interval.start
10        delta = timedelta(days=1)
11        if last_start.hour == 6: # If previous period
12 started at 6:00, next period will start at 16:30 and end
13 at 6:00 following day
14            next_start = last_start.set(hour=16, min-
15  ute=30).replace(tzinfo=UTC)
16            next_end = (last_start+delta).replace(tzin-
17 fo=UTC)
18        else: # If previous period started at 16:30, next
19 period will start at 6:00 next day and end at 16:30
20            next_start = (last_start+delta).set(hour=6,
21 minute=0).replace(tzinfo=UTC)
22            next_end = (last_start+delta).replace(tzin-
23 fo=UTC)
24        else: # This is the first ever run on the regular
25 schedule. First data interval will always start at 6:00
26 and end at 16:30
27            next_start = restriction.earliest
28            if next_start is None: # No start_date. Don't
29 schedule.
30                return None
31            if not restriction.catchup: # If the DAG has
32 catchup=False, today is the earliest to consider.
33                next_start = max(next_start, DateTime.com-
35 bine(Date.today(), Time.min).replace(tzinfo=UTC))

```

```

36         next_start = next_start.set(hour=6, minute=0).re-
37 place(tzinfo=UTC)
38         next_end = next_start.set(hour=16, minute=30).re-
39 place(tzinfo=UTC)
40         if restriction.latest is not None and next_start > re-
41 striction.latest:
42             return None # Over the DAG's scheduled end; don't
43 schedule.
44         return DagRunInfo.interval(start=next_start, end=next_
45 end)

```

Walking through the logic, this code is equivalent to:

- **If there was a previous run for the DAG:**
 - If the previous DAG run started at 6:00, then the next DAG run should start at 16:30 and end at 6:00 the next day.
 - If the previous DAG run started at 16:30, then the DAG run should start at 6:00 the next day and end at 16:30 the next day.
- **If it is the first run of the DAG:**
 - Check for a start date. If there isn't one, the DAG can't be scheduled.
 - Check if catchup=False. If so, the earliest date to consider should be the current date. Otherwise it is the DAG's start date.
 - We're mandating that the first DAG run should always start at 6:00, so update the time of the interval start to 6:00 and the end to 16:30.
- **If the DAG has an end date, do not schedule the DAG after that date has passed.**

Then we define the data interval for manually triggered DAG runs by defining the `infer_manual_data_interval` method. The code looks like this:

```

1 def infer_manual_data_interval(self, run_after: Date-
2 Time) -> DataInterval:
3     delta = timedelta(days=1)
4     # If time is between 6:00 and 16:30, period ends at
5     6am and starts at 16:30 previous day
6     if run_after >= run_after.set(hour=6, minute=0) and
7     run_after <= run_after.set(hour=16, minute=30):
8         start = (run_after-delta).set(hour=16, min-
9         ute=30, second=0).replace(tzinfo=UTC)
10        end = run_after.set(hour=6, minute=0, second=0).
11        replace(tzinfo=UTC)
10       # If time is after 16:30 but before midnight, period
11       is between 6:00 and 16:30 the same day
10       elif run_after >= run_after.set(hour=16, minute=30)
10       and run_after.hour <= 23:
11           start = run_after.set(hour=6, minute=0, sec-
12           ond=0).replace(tzinfo=UTC)
13           end = run_after.set(hour=16, minute=30, sec-
14           ond=0).replace(tzinfo=UTC)
15       # If time is after midnight but before 6:00, period
16       is between 6:00 and 16:30 the previous day
17       else:
18           start = (run_after-delta).set(hour=6, minute=0).
19           replace(tzinfo=UTC)
20           end = (run_after-delta).set(hour=16, minute=30).
21           replace(tzinfo=UTC)
22       return DataInterval(start=start, end=end)

```

This method figures out what the most recent complete data interval is based on the current time. There are three scenarios:

- The current time is between 6:00 and 16:30: In this case, the data interval is from 16:30 the previous day to 6:00 the current day.
- The current time is after 16:30 but before midnight: In this case, the data interval is from 6:00 to 16:30 the current day.
- The current time is after midnight but before 6:00: In this case, the data interval is from 6:00 to 16:30 the previous day.

We need to account for time periods in the same timeframe (6:00 to 16:30) on different days than the day that the DAG is triggered, which requires three sets of logic. When defining custom timetables, always keep in mind what the last complete data interval should be based on when the DAG should run.

Now we can take those two methods and combine them into a `Timetable` class which will make up our Airflow plugin. The full custom timetable plugin is below:

```

1  from datetime import timedelta
2  from typing import Optional
3  from pendulum import Date, DateTime, Time, timezone
4
5  from airflow.plugins_manager import AirflowPlugin
6  from airflow.timetables.base import DagRunInfo, DataInt-
7  erval, TimeRestriction, Timetable
8
9  UTC = timezone("UTC")
10
11 class UnevenIntervalsTimetable(Timetable):
12
13     def infer_manual_data_interval(self, run_after: Da-
14     teTime) -> DataInterval:
15         delta = timedelta(days=1)
16         # If time is between 6:00 and 16:30, period ends
17         at 6am and starts at 16:30 previous day
18         if run_after >= run_after.set(hour=6, minute=0)
19         and run_after <= run_after.set(hour=16, minute=30):
20             start = (run_after-delta).set(hour=16, min-
21             ute=30, second=0).replace(tzinfo=UTC)
22             end = run_after.set(hour=6, minute=0, sec-
23             ond=0).replace(tzinfo=UTC)
24             # If time is after 16:30 but before midnight,
25             period is between 6:00 and 16:30 the same day
26             elif run_after >= run_after.set(hour=16, min-
27             ute=30) and run_after.hour <= 23:
28                 start = run_after.set(hour=6, minute=0, sec-
29                 ond=0).replace(tzinfo=UTC)
30                 end = run_after.set(hour=16, minute=30, sec-
31                 ond=0).replace(tzinfo=UTC)
32                 # If time is after midnight but before 6:00, pe-
33                 riod is between 6:00 and 16:30 the previous day
34                 else:
35                     start = (run_after-delta).set(hour=6, min-

```

```

34     ute=0).replace(tzinfo=UTC)
35         end = (run_after-delta).set(hour=16, min-
36         ute=30).replace(tzinfo=UTC)
37         return DataInterval(start=start, end=end)
38
39     def next_dagrun_info(
40         self,
41         *,
42         last_automated_data_interval: Optional[DataInt-
43         erval],
44         restriction: TimeRestriction,
45     ) -> Optional[DagRunInfo]:
46         if last_automated_data_interval is not None: #
47         There was a previous run on the regular schedule.
48             last_start = last_automated_data_interval.
49             start
50             delta = timedelta(days=1)
51             if last_start.hour == 6: # If previous peri-
52             od started at 6:00, next period will start at 16:30 and
53             end at 6:00 following day
54                 next_start = last_start.set(hour=16,
55                 minute=30).replace(tzinfo=UTC)
56                 next_end = (last_start+delta).re-
57                 place(tzinfo=UTC)
58             else: # If previous period started at 14:30,
59             next period will start at 6:00 next day and end at 14:30
60                 next_start = (last_start+delta).
61                 set(hour=6, minute=0).replace(tzinfo=UTC)
62                 next_end = (last_start+delta).re-
63                 place(tzinfo=UTC)
64             else: # This is the first ever run on the reg-
65             ular schedule. First data interval will always start at
66             6:00 and end at 16:30
67                 next_start = restriction.earliest
68                 if next_start is None: # No start_date.

```

```

69 Don't schedule.
70     return None
71     if not restriction.catchup: # If the DAG has
72 catchup=False, today is the earliest to consider.
73         next_start = max(next_start, DateTime.
74 combine(Date.today(), Time.min).replace(tzinfo=UTC))
75         next_start = next_start.set(hour=6, min-
76 ute=0).replace(tzinfo=UTC)
77         next_end = next_start.set(hour=16, min-
78 ute=30).replace(tzinfo=UTC)
79         if restriction.latest is not None and next_start
80 > restriction.latest:
81             return None # Over the DAG's scheduled end;
82 don't schedule.
83             return DagRunInfo.interval(start=next_start,
84 end=next_end)
85
86 class UnevenIntervalsTimetablePlugin(AirflowPlugin):
87     name = "uneven_intervals_timetable_plugin"
88     timetables = [UnevenIntervalsTimetable]

```

Note that because timetables are plugins, you will need to restart the Airflow Scheduler and Webserver after adding or updating them.

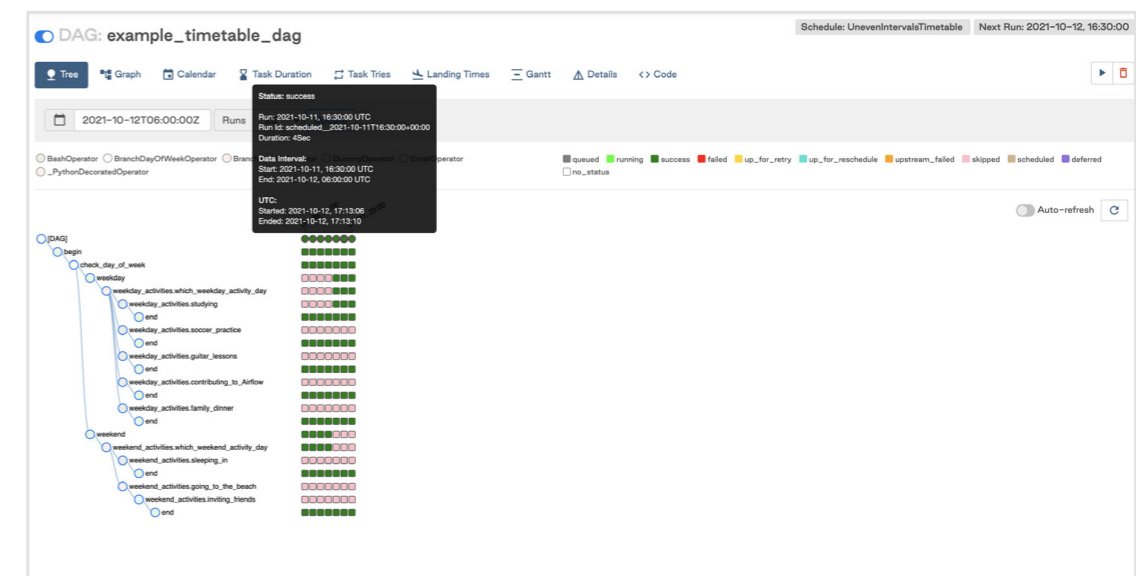
In the DAG, we can then import the custom timetable plugin and use it to schedule the DAG by setting the `timetable` parameter:

```

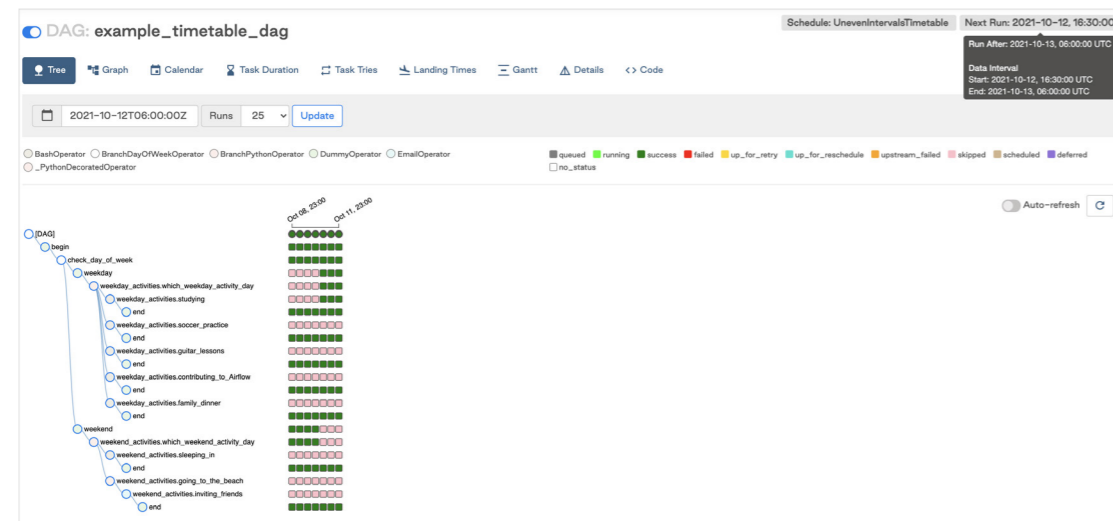
1  from uneven_intervals_timetable import UnevenIntervals-
2  Timetable
3
4  with DAG(
5      dag_id="example_timetable_dag",
6      start_date=datetime(2021, 10, 9),
7      max_active_runs=1,
8      timetable=UnevenIntervalsTimetable(),
9      default_args={
10         "retries": 1,
11         "retry_delay": timedelta(minutes=3),
10     },
11     catchup=True
12 ) as dag:

```

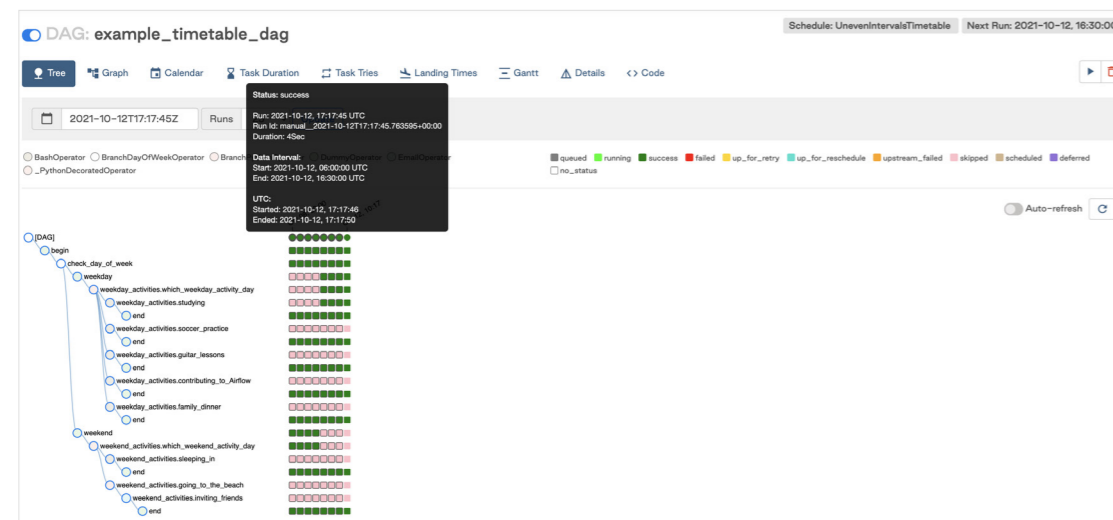
Looking at the Tree View in the UI, we can see that this DAG has run twice per day at 6:00 and 16:30 since the start date of 2021-10-09.



The next scheduled run is for the interval starting on 2021-10-12 at 16:30 and ending the following day at 6:00. This run will be triggered at the end of the data interval, so after 2021-10-13 6:00.



If we run the DAG manually after 16:30 but before midnight, we can see the data interval for the triggered run was between 6:00 and 16:30 that day as expected.



This is a simple timetable that could easily be adjusted to suit other use cases. In general, timetables are completely customizable as long as the methods above are implemented.

Note: Be careful when implementing your timetable logic that your `next_dagrun_info` method does not return a `data_interval_start` that is earlier than your DAG's `start_date`. This will result in tasks not being executed for that DAG run.

Current Limitations

There are some limitations to keep in mind when implementing custom timetables:

- Timetable methods should return the same result every time they are called (e.g. avoid things like HTTP requests). They are not designed to implement event-based triggering.
- Timetables are parsed by the scheduler when creating DAG runs, so avoid slow or lengthy code that could impact Airflow performance.

Dataset driven scheduling

Airflow 2.4 introduced the concept of datasets and data-driven DAG dependencies. You can now make Airflow detect when a task in a DAG updates a data object. Using that awareness, other DAGs can be scheduled depending on updates to these datasets. To create a dataset-based schedule, you simply pass the names of the datasets as a list to the `schedule` parameter.

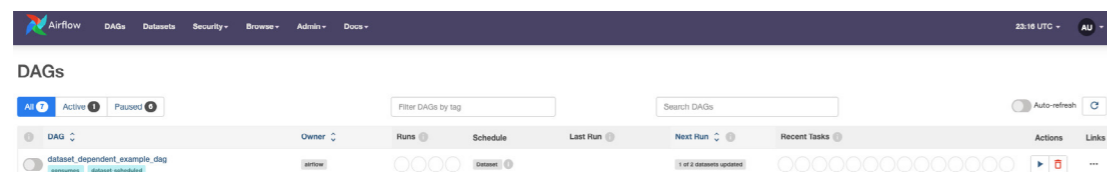
```

1 dataset1 = Dataset(f"{DATASETS_PATH}/dataset_1.txt")
2 dataset2 = Dataset(f"{DATASETS_PATH}/dataset_2.txt")
3
4 with DAG(
5     dag_id='dataset_dependent_example_dag',
6     catchup=False,
7     start_date=datetime(2022, 8, 1),
8     schedule=[dataset1, dataset2],
9     tags=['consumes', 'dataset-scheduled'],
10 ) as dag:

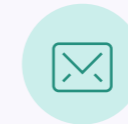
```

This DAG runs only when both `dataset1` and `dataset2` have been updated. These updates can occur by tasks in different DAGs as long as they are located in the same Airflow environment.

In the Airflow UI, the DAG now has a schedule of **Dataset** and the **Next Run** column shows how many datasets the DAG depends on and how many of them have been updated.



To learn more about datasets and data driven scheduling, check out the [Datasets and Data Driven Scheduling in Airflow](#) guide.



Never miss an update from us.

Sign up for the Astronomer newsletter.

Sign Up

Operators 101

Overview

Operators are the building blocks of Airflow DAGs. They contain the logic of how data is processed in a pipeline. Each task in a DAG is defined by instantiating an operator.

There are many different types of operators available in Airflow. Some operators execute general code provided by the user, like a Python function, while other operators perform very specific actions such as transferring data from one system to another.

In this guide, we'll cover the basics of using operators in Airflow and show an example of how to implement them in a DAG.

Note: To browse and search all of the available operators in Airflow, visit the [Astronomer Registry](#), the discovery and distribution hub for Airflow integrations.

Operator Basics

Under the hood, operators are Python classes that encapsulate logic to do a unit of work. They can be thought of as a wrapper around each unit of work that defines the actions that will be completed and abstracts away a lot of code you would otherwise have to write yourself. When you create an instance of an operator in a DAG and provide it with its required parameters, it becomes a task.

All operators inherit from the abstract [BaseOperator class](#), which contains the logic to execute the work of the operator within the context of a DAG.

The work each operator does varies widely. Some of the most frequently used operators in Airflow are:

- [PythonOperator](#): Executes a Python function.
- [BashOperator](#): Executes a bash script.
- [KubernetesPodOperator](#): Executes a task defined as a Docker image in a Kubernetes Pod.
- [SnowflakeOperator](#): Executes a query against a Snowflake database.

Operators are easy to use and typically only a few parameters are required. There are a few details that every Airflow user should know about operators:

- The [Astronomer Registry](#) is the best place to go to learn about what operators are out there and how to use them.
- The core Airflow package that contains basic operators such as the PythonOperator and BashOperator. These operators are automatically available in your Airflow environment. All other operators are part of provider packages, which must be installed separately. For example, the SnowflakeOperator is part of the [Snowflake provider](#).
- If an operator exists for your specific use case, you should always use it over your own Python functions or [hooks](#). This makes your DAGs easier to read and maintain.

- If an operator doesn't exist for your use case, you can extend operator to meet your needs. For more on how to customize operators, check out our previous [Anatomy of an Operator webinar](#).
- **Sensors** are a type of operator that wait for something to happen. They can be used to make your DAGs more event-driven.
- **Deferrable Operators** are a type of operator that release their worker slot while waiting for their work to be completed. This can result in cost savings and greater scalability. Astronomer recommends using deferrable operators whenever one exists for your use case and your task takes longer than about a minute. Note that you must be using Airflow 2.2+ and have a triggerer running to use deferrable operators.
- Any operator that interacts with a service external to Airflow will typically require a connection so that Airflow can authenticate to that external system. More information on how to set up connections can be found in our guide on [managing connections](#) or in the examples to follow.

Example Implementation

This example shows how to use several common operators in a DAG used to transfer data from S3 to Redshift and perform data quality checks.

Note: The full code and repository for this example can be found on the [Astronomer Registry](#).

The following operators are used:

- **EmptyOperator:** This operator is part of core Airflow and does nothing. It is used to organize the flow of tasks in the DAG.
- **PythonDecoratedOperator:** This operator is part of core Airflow and executes a Python function. It is functionally the same as the PythonOperator, but it is instantiated using the `@task decorator`.

- **LocalFilesystemToS3Operator:** This operator is part of the AWS provider and is used to upload a file from a local filesystem to S3.
- **S3ToRedshiftOperator:** This operator is part of the AWS provider and is used to transfer data from S3 to Redshift.
- **PostgresOperator:** This operator is part of the Postgres provider and is used to execute a query against a Postgres database.
- **SQLCheckOperator:** This operator is part of core Airflow and is used to perform checks against a database using a SQL query.

The following code shows how each of those operators can be instantiated in a DAG file to define the pipeline:

```

1  import hashlib
2  import json
3
4  from airflow import DAG, AirflowException
5  from airflow.decorators import task
6  from airflow.models import Variable
7  from airflow.models.baseoperator import chain
8  from airflow.operators.empty import EmptyOperator
9  from airflow.utils.dates import datetime
10 from airflow.providers.amazon.aws.hooks.s3 import S3Hook
11 from airflow.providers.amazon.aws.transfers.local_to_s3
12 import (
13     LocalFilesystemToS3Operator
14 )
15 from airflow.providers.amazon.aws.transfers.s3_to_redshift
16 import (
17     S3ToRedshiftOperator
18 )
19 from airflow.providers.postgres.operators.postgres import
20 PostgresOperator
21 from airflow.operators.sql import SQLCheckOperator

```

```

22 from airflow.operators.sql import SQLCheckOperator
23 from airflow.utils.task_group import TaskGroup
24
25
26 # The file(s) to upload shouldn't be hardcoded in a pro-
27 duction setting,
28 # this is just for demo purposes.
29 CSV_FILE_NAME = "forestfires.csv"
30 CSV_FILE_PATH = f"include/sample_data/forestfire_data/
31 {CSV_FILE_NAME}"
32
33 with DAG(
34     "simple_redshift_3",
35     start_date=datetime(2021, 7, 7),
36     description=""A sample Airflow DAG to load data from
37 csv files to S3
38         and then Redshift, with data integrity
39 and quality checks.""",
40     schedule_interval=None,
41     template_searchpath="/usr/local/airflow/include/sql/
42 redshift_examples/",
43     catchup=False,
44 ) as dag:
45
46     ""
47     Before running the DAG, set the following in an Air-
48 flow
49 or Environment Variable:
50 - key: aws_configs
51 - value: { "s3_bucket": [bucket_name], "s3_key_pre-
52 fix": [key_prefix],
53           "redshift_table": [table_name]}
54     Fully replacing [bucket_name], [key_prefix], and [ta-
55 ble_name].
56     ""

```

```

57     upload_file = LocalFileSystemToS3Operator(
58         task_id="upload_to_s3",
59         filename=CSV_FILE_PATH,
60         dest_key="{{ var.json.aws_configs.s3_key_prefix
61 }}/" + CSV_FILE_PATH,
62         dest_bucket="{{ var.json.aws_configs.s3_bucket
63 }}",
64         aws_conn_id="aws_default",
65         replace=True,
66     )
67
68     @task
69     def validate_etag():
70         ""
71         ##### Validation task
72         Check the destination ETag against the local MD5
73 hash to ensure
74 the file was uploaded without errors.
75         ""
76         s3 = S3Hook()
77         aws_configs = Variable.get("aws_configs", deseri-
78 alize_json=True)
79         obj = s3.get_key(
80             key=f"{aws_configs.get('s3_key_prefix')}/{CSV_
81 FILE_PATH}",
82             bucket_name=aws_configs.get("s3_bucket"),
83         )
84         obj_etag = obj.e_tag.strip('')
85         # Change `CSV_FILE_PATH` to `CSV_CORRUPT_FILE_
86 PATH` for the "sad path".
87         file_hash = hashlib.md5(
88             open(CSV_FILE_PATH).read().encode("utf-8")).
89 hexdigest()
90         if obj_etag != file_hash:

```

```

90 hexdigest()
91     if obj_etag != file_hash:
92         raise AirflowException(
93             f""""Upload Error: Object ETag in S3 did
94 not match
95             hash of local file.""")
96     )
97
98     # Tasks that were created using decorators have to be
99 called to be used
100 validate_file = validate_etag()
101
102 ##### Create Redshift Table
103 create_redshift_table = PostgresOperator(
104     task_id="create_table",
105     sql="create_redshift_forestfire_table.sql",
106     postgres_conn_id="redshift_default",
107 )
108
109 ##### Second load task
110 load_to_redshift = S3ToRedshiftOperator(
111     task_id="load_to_redshift",
112     s3_bucket="{{ var.json.aws_configs.s3_bucket }}",
113     s3_key="{{ var.json.aws_configs.s3_key_prefix }}"
114 + f"/{CSV_FILE_PATH}",
115     schema="PUBLIC",
116     table="{{ var.json.aws_configs.redshift_table }}",
117     copy_options=["csv"],
118 )
119
120 ##### Redshift row validation task
121 validate_redshift = SQLCheckOperator(
122     task_id="validate_redshift",
123     conn_id="redshift_default",

```

```

124     sql="validate_redshift_forestfire_load.sql",
125     params={"filename": CSV_FILE_NAME},
126 )
127
128 ##### Row-level data quality check
129 with open("include/validation/forestfire_validation.
130 json") as ffv:
131     with TaskGroup(group_id="row_quality_checks") as
132 quality_check_group:
133         ffv_json = json.load(ffv)
134         for id, values in ffv_json.items():
135             values["id"] = id
136             SQLCheckOperator(
137                 task_id=f"forestfire_row_quality_
138 check_{id}",
139                 conn_id="redshift_default",
140                 sql="row_quality_redshift_forestfire_
141 check.sql",
142                 params=values,
143             )
144
145 ##### Drop Redshift table
146 drop_redshift_table = PostgresOperator(
147     task_id="drop_table",
148     sql="drop_redshift_forestfire_table.sql",
149     postgres_conn_id="redshift_default",
150 )
151
152 begin = EmptyOperator(task_id="begin")
153 end = EmptyOperator(task_id="end")
154
155 ##### Define task dependencies
156 chain(
157     begin,

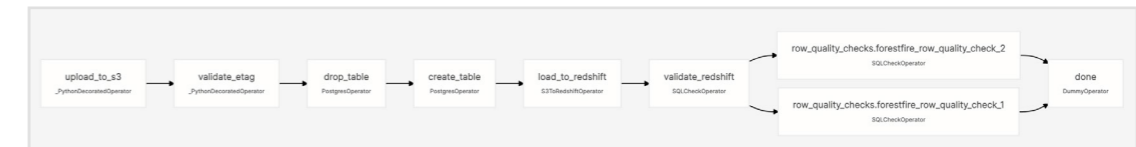
```

```

159     upload_file,
160     validate_file,
161     create_redshift_table,
162     load_to_redshift,
163     validate_redshift,
164     quality_check_group,
165     drop_redshift_table,
166     end
167 )

```

The resulting DAG looks like this:



There are a few things to note about the operators in this DAG:

- Every operator is given a `task_id`. This is a required parameter, and the value provided will be shown as the name of the task in the Airflow UI.
- Each operator requires different parameters based on the work it does. For example, the `PostgresOperator` has a `sql` parameter for the SQL script to be executed, and the `S3ToRedshiftOperator` has parameters to define the location and keys of the files being copied from S3 and the Redshift table receiving the data.
- Connections to external systems are passed in most of these operators. The parameters `conn_id`, `postgres_conn_id`, and `aws_conn_id` all point to the names of the relevant connections stored in Airflow.

Hooks 101

Overview

Hooks are one of the fundamental building blocks of Airflow. At a high level, a hook is an abstraction of a specific API that allows Airflow to interact with an external system. Hooks are built into many operators, but they can also be used directly in DAG code.

In this guide, we'll cover the basics of using hooks in Airflow and when to use them directly in DAG code. We'll also walk through an example of implementing two different hooks in a DAG.

Over 200 Hooks are currently listed in the Astronomer Registry. If there isn't one for your use case yet, you can write your own and share it with the community!

Hook Basics

Hooks wrap around APIs and provide methods to interact with different external systems. Because hooks standardize the way you can interact with external systems, using them makes your DAG code cleaner, easier to read, and less prone to errors.

To use a hook, you typically only need a connection ID to connect with an external system. More information on how to set up connections can be found in [Managing your Connections in Apache Airflow](#) or in the example section below.

All hooks inherit from the [BaseHook class](#), which contains the logic to set up an external connection given a connection ID. On top of making the connection to an external system, each hook might contain additional methods to perform various actions within that system. These methods might rely on different Python libraries for these interactions.

For example, the [S3Hook](#), which is one of the most widely used hooks, relies on the [boto3](#) library to manage its connection with S3.

The [S3Hook](#) contains [over 20 methods](#) to interact with S3 buckets, including methods like:

- [check_for_bucket](#): Checks if a bucket with a specific name exists.
- [list_prefixes](#): Lists prefixes in a bucket according to specified parameters.
- [list_keys](#): Lists keys in a bucket according to specified parameters.
- [load_file](#): Loads a local file to S3.
- [download_file](#): Downloads a file from the S3 location to the local file system.

When to Use Hooks

Since hooks are the building blocks of operators, their use in Airflow is often abstracted away from the DAG author. However, there are some cases when you should use hooks directly in a Python function in your DAG. The following are general guidelines when using hooks in Airflow:

- Hooks should always be used over manual API interaction to connect to external systems.
- If you write a custom operator to interact with an external system, it should use a hook to do so.
- If an operator with built-in hooks exists for your specific use case, then it is best practice to use the operator over setting up hooks manually.
- If you regularly need to connect to an API for which no hook exists yet, consider writing your own and sharing it with the community!

Example Implementation

The following example shows how you can use two hooks ([S3Hook](#) and [SlackHook](#)) to retrieve values from files in an S3 bucket, run a check on them, post the result of the check on Slack, and log the response of the Slack API.

For this use case, we use hooks directly in our Python functions because none of the existing S3 Operators can read data from several files within an S3 bucket. Similarly, none of the existing Slack Operators can return the response of a Slack API call, which you might want to log for monitoring purposes.

The full source code of the hooks used can be found here:

- [S3Hook source code](#)
- [SlackHook source code](#)

Before running the example DAG, make sure you have the necessary Airflow providers installed. If you are using the Astro CLI, you can do this by adding the following packages to your `requirements.txt`:

```
1 apache-airflow-providers-amazon
2 apache-airflow-providers-slack
```

Next you will need to set up connections to the S3 bucket and Slack in the Airflow UI.

1. Go to **Admin** -> **Connections** and click on the plus sign to add a new connection.
2. Select **Amazon S3** as connection type for the S3 bucket (if the connection type is not showing up, double check that you installed the provider correctly) and provide the connection with your AWS access key ID as `login` and your AWS secret access key as `password` ([See AWS documentation for how to retrieve your AWS access key ID and AWS secret access key](#)).
3. Create a new connection. Select **Slack Webhook** as the connection type and provide your [Bot User OAuth Token](#) as a password. This token can be obtained by going to **Features** > **OAuth & Permissions** on api.slack.com/apps.

The DAG below uses [Airflow Decorators](#) to define tasks and [XCom](#) to pass information between them. The name of the S3 bucket and the names of the files that the first task reads are stored as environment variables for security purposes.

```

1 # importing necessary packages
2 import os
3 from datetime import datetime
4 from airflow import DAG
5 from airflow.decorators import task
6 from airflow.providers.slack.hooks.slack import SlackHook
7 from airflow.providers.amazon.aws.hooks.s3 import S3Hook
8
9 # import environmental variables for privacy (set in Dockerfile)
10
11 S3BUCKET_NAME = os.environ.get('S3BUCKET_NAME')
12 S3_EXAMPLE_FILE_NAME_1 = os.environ.get('S3_EXAMPLE_FILE_NAME_1')
13 S3_EXAMPLE_FILE_NAME_2 = os.environ.get('S3_EXAMPLE_FILE_NAME_2')
14 S3_EXAMPLE_FILE_NAME_3 = os.environ.get('S3_EXAMPLE_FILE_NAME_3')
15
16 # task to read 3 keys from your S3 bucket
17 @task.python
18 def read_keys_form_s3():
19     s3_hook = S3Hook(aws_conn_id='hook_tutorial_s3_conn')
20     response_file_1 = s3_hook.read_key(key=S3_EXAMPLE_FILE_NAME_1,
21                                     bucket_name=S3BUCKET_NAME)
22     response_file_2 = s3_hook.read_key(key=S3_EXAMPLE_FILE_NAME_2,
23                                     bucket_name=S3BUCKET_NAME)
24     response_file_3 = s3_hook.read_key(key=S3_EXAMPLE_FILE_NAME_3,
25                                     bucket_name=S3BUCKET_NAME)
26
27     response = {'num1' : int(response_file_1),
28               'num2' : int(response_file_2),

```

```

35         'num3' : int(response_file_3)}
36
37     return response
38
39 # task running a check on the data retrieved from your S3 bucket
40 @task.python
41 def run_sum_check(response):
42     if response['num1'] + response['num2'] == response['num3']:
43         return (True, response['num3'])
44     else:
45         return (False, response['num3'])
46
47 # task posting to slack depending on the outcome of the above check
48 # and returning the server response
49 @task.python
50 def post_to_slack(sum_check_result):
51     slack_hook = SlackHook(slack_conn_id='hook_tutorial_slack_conn')
52
53     if sum_check_result[0] == True:
54         server_response = slack_hook.call(api_method='chat.postMessage',
55                                         json={"channel": "#test-airflow",
56                                               "text": f""""All is well in your bucket!
57
58                                     Correct sum: {sum_check_result[1]}!"""})
59     else:
60         server_response = slack_hook.call(api_method='chat.postMessage',
61                                         json={"channel": "#test-airflow",
62                                               "text": f""""A test on your bucket

```

```

69 contents failed!
70             Target sum not reached: {sum_
71 check_result[1]}""")
72
73     # return the response of the API call (for logging or
74 use downstream)
75     return server_response
76
77 # implementing the DAG
78 with DAG(dag_id='hook_tutorial',
79         start_date=datetime(2022,5,20),
80         schedule_interval='@daily',
81         catchup=False,
82         ) as dag:
83
84     # the dependencies are automatically set by XCom
85     response = read_keys_form_s3()
86     sum_check_result = run_sum_check(response)
87     post_to_slack(sum_check_result)

```

The DAG above completes the following steps:

1. Use a decorated Python Operator with a manually implemented `S3Hook` to read three specific keys from S3 with the `read_key` method. Returns a dictionary with the file contents converted to integers.
2. With the results of the first task, use a second decorated Python Operator to complete a simple sum check.
3. Post the result of the check to a Slack channel using the `call` method of the `SlackHook` and return the response from the Slack API.

Sensors 101

Sensors are a special kind of operator. When they run, they check to see if a certain criterion is met before they let downstream tasks execute. This is a great way to have portions of your DAG wait on some external check or process to complete.

To browse and search all of the available Sensors in Airflow, visit the [Astronomer Registry](#). Take the following sensor as an example:

```

1 s1 = S3KeySensor(
2     task_id='s3_key_sensor',
3     bucket_key='{{ ds_nodash }}/my_file.csv',
4     bucket_name='my_s3_bucket',
5     aws_conn_id='my_aws_connection',
6 )

```

S3 Key Sensor

The [S3KeySensor](#) checks for the existence of a specified key in S3 every few seconds until it finds it or times out. If it finds the key, it will be marked as a success and allow downstream tasks to run. If it times out, it will fail and prevent downstream tasks from running.

[S3KeySensor Code](#)

Sensor Params

There are sensors for many use cases, such as ones that check a database for a certain row, wait for a certain time of day, or sleep for a certain amount of time. All sensors inherit from the [BaseSensorOperator](#) and have 4 parameters you can set on any sensor.

- **soft_fail:** Set to true to mark the task as SKIPPED on failure.
 - **poke_interval:** Time in seconds that the job should wait in between each try. The poke interval should be more than one minute to prevent too much load on the scheduler.
 - **timeout:** Time, in seconds before the task times out and fails.
 - **mode:** How the sensor operates. Options are: { `poke` | `reschedule` }, default is `poke`. When set to `poke` the sensor will take up a worker slot for its whole execution time (even between pokes). Use this mode if the expected runtime of the sensor is short or if a short poke interval is required. When set to `reschedule` the sensor task frees the worker slot when the criteria is not met and it's rescheduled at a later time.
- soft_fail:** Set to true to mark the task as SKIPPED on failure.
- **poke_interval:** Time in seconds that the job should wait in between each try. The poke interval should be more than one minute to prevent too much load on the scheduler.
 - **timeout:** Time, in seconds before the task times out and fails.
 - **mode:** How the sensor operates. Options are: { `poke` | `reschedule` }, default is `poke`. When set to `poke` the sensor will take up a worker slot for its whole execution time (even between pokes). Use this mode if the expected runtime of the sensor is short or if a short poke interval is required. When set to `reschedule` the sensor task frees the worker slot when the criteria is not met and it's rescheduled at a later time.

Deferrable Operators

Prior to Airflow 2.2, all task execution occurred within your worker resources. For tasks whose work was occurring outside of Airflow (e.g. a Spark Job), your tasks would sit idle waiting for a success or failure signal. These idle tasks would occupy worker slots for their entire duration, potentially queuing other tasks and delaying their start times.

With the release of Airflow 2.2, Airflow has introduced a new way to run tasks in your environment: deferrable operators. These operators leverage Python's [asyncio](#) library to efficiently run tasks waiting for an external resource to finish. This frees up your workers, allowing you to utilize those resources more effectively. In this guide, we'll walk through the concepts of deferrable operators, as well as the new components introduced to Airflow related to this feature.

Deferrable Operator Concepts

There are some terms and concepts that are important to understand when discussing deferrable operators:

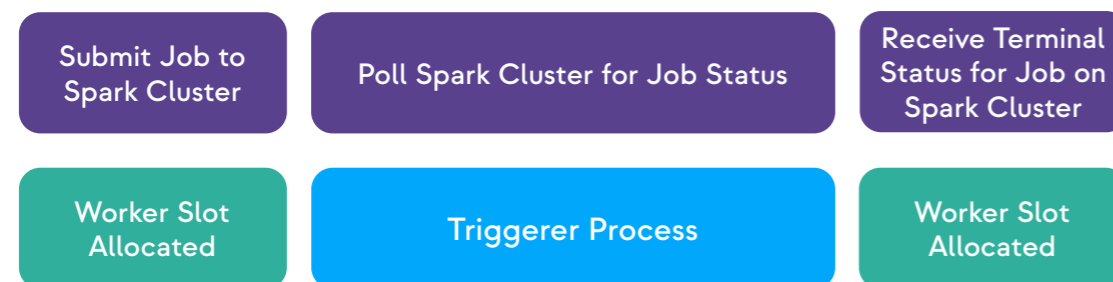
- **asyncio:** This Python library is used as a foundation for multiple asynchronous frameworks. This library is core to deferrable operator's functionality, and is used when writing triggers.
- **Triggers:** These are small, asynchronous pieces of Python code. Due to their asynchronous nature, they coexist efficiently in a single process known as the triggerer.
- **Triggerer:** This is a new airflow service (like a scheduler or a worker) that runs an [asyncio event loop](#) in your Airflow environment. Running a triggerer is essential for using deferrable operators.
- **Deferred:** This is a new Airflow task state (medium purple color) introduced to indicate that a task has paused its execution, released the worker slot, and submitted a trigger to be picked up by the triggerer process. All sensors inherit from the [BaseSensorOperator](#) and have 4 parameters you can set on any sensor.

Note: The terms “deferrable” and “async” or “asynchronous” are often used interchangeably. They mean the same thing in this context.

With traditional operators, a task might submit a job to an external system (e.g. a Spark Cluster) and then poll the status of that job until it completes. Even though the task might not be doing significant work, it would still occupy a worker slot during the polling process. As worker slots become occupied, tasks may be queued resulting in delayed start times. Visually, this is represented in the diagram below:



With deferrable operators, worker slots can be released while polling for job status. When the task is deferred (suspended), the polling process is offloaded as a trigger to the triggerer, freeing up the worker slot. The triggerer has the potential to run many asynchronous polling tasks concurrently, preventing this work from occupying your worker resources. When the terminal status for the job is received, the task resumes, taking a worker slot while it finishes. Visually, this is represented in the diagram below:



When and Why to Use Deferrable Operators

In general, deferrable operators should be used whenever you have tasks that occupy a worker slot while polling for a condition in an external system. For example, using deferrable operators for sensor tasks (e.g. poking for a file on an SFTP server) can result in efficiency gains and reduced operational costs. In particular, if you are currently working with [smart sensors](#), you should consider using deferrable operators instead. Compared to smart sensors, which were deprecated in Airflow 2.2.4, deferrable operators are more flexible and better supported by Airflow.

Currently, the following deferrable operators are available in Airflow:

- [TimeSensorAsync](#)
- [DateTimeSensorAsync](#)

However, this list will grow quickly as the Airflow community makes more investments into these operators. In the meantime, you can also create your own (more on this in the last section of this guide). Additionally, Astronomer maintains some deferrable operators [available only on Astro Runtime](#).

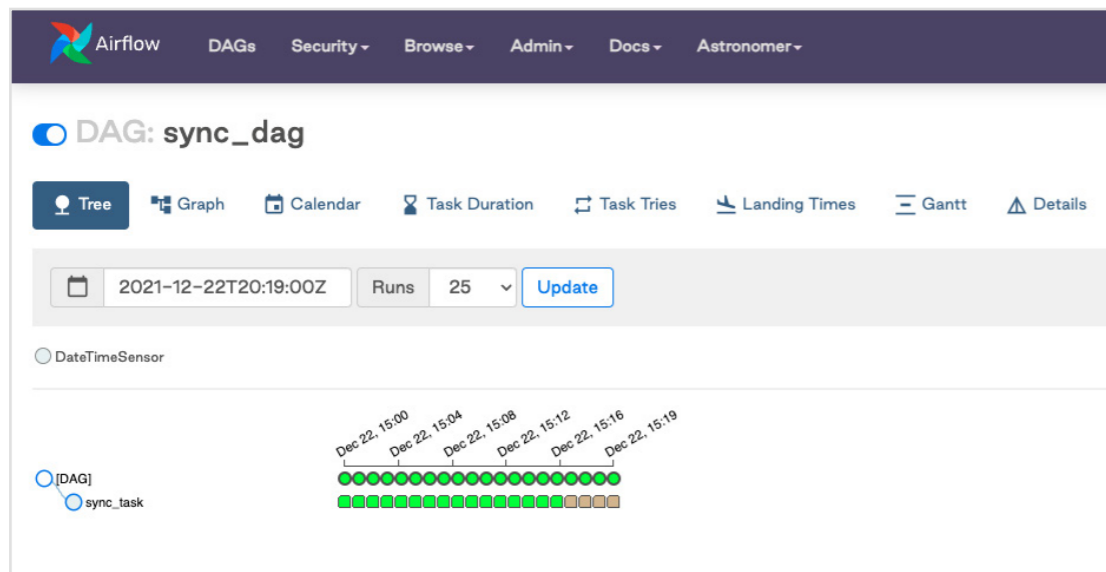
There are numerous benefits to using deferrable operators. Some of the most notable are:

- **Reduced resource consumption:** Depending on the available resources and the workload of your triggers, you can run hundreds to thousands of deferred tasks in a single triggerer process. This can lead to a reduction in the number of workers needed to run tasks during periods of high concurrency. With less workers needed, you are able to scale down the underlying infrastructure of your Airflow environment.
- **Resiliency against restarts:** Triggers are stateless by design. This means your deferred tasks will not be set to a failure state if a triggerer needs to be restarted due to a deployment or infrastructure issue. Once a triggerer is back up and running in your environment, your deferred tasks will resume.

- **Paves the way to event-based DAGs:** The presence of `asyncio` in core Airflow is a potential foundation for event-triggered DAGs.

Example Workflow Using Deferrable Operators

Let's say we have a DAG that is scheduled to run a sensor every minute, where each task can take up to 20 minutes. Using the default settings with 1 worker, we can see that after 20 minutes we have 16 tasks running, each holding a worker slot:



Because worker slots are held during task execution time, we would need at least 20 worker slots available for this DAG to ensure that future runs are not delayed. To increase concurrency, we would need to add additional resources to our Airflow infrastructure (e.g. another worker pod).

```

1  from datetime import datetime
2  from airflow import DAG
3  from airflow.sensors.date_time import DateTimeSensor
4
5  with DAG(
6      "sync_dag",
7      start_date=datetime(2021, 12, 22, 20, 0),
8      end_date=datetime(2021, 12, 22, 20, 19),
9      schedule_interval="* * * * *",
10     catchup=True,
11     max_active_runs=32,
12     max_active_tasks=32
13 ) as dag:
14
15     sync_sensor= DateTimeSensor(
16         task_id="sync_task",
17         target_time="{{ macros.datetime.utcnow() + mac-
18 macros.timedelta(minutes=20) }}"
19     )

```

By leveraging a deferrable operator for this sensor, we are able to achieve full concurrency while allowing our worker to complete additional work across our Airflow environment. With our updated DAG below, we see that all 20 tasks have entered a state of deferred, indicating that these sensing jobs (triggers) have been registered to run in the triggerer process.

```

1 from datetime import datetime
2 from airflow import DAG
3 from airflow.sensors.date_time import DateTimeSensorAsync
4
5 with DAG(
6     "async_dag",
7     start_date=datetime(2021, 12, 22, 20, 0),
8     end_date=datetime(2021, 12, 22, 20, 19),
9     schedule_interval="* * * * *",
10    catchup=True,
11    max_active_runs=32,
12    max_active_tasks=32
13 ) as dag:
14
15     async_sensor = DateTimeSensorAsync(
16         task_id="async_task",
17         target_time="""{{ macros.datetime.utcnow() + mac-
18         ros.timedelta(minutes=20) }}""",
19     )

```

Running Deferrable Tasks in Your Airflow Environment

To start a triggerer process, run `airflow triggerer` in your Airflow environment. You should see an output similar to the below image.

```

2021-12-22 22:18:32,367] {triggerer_job.py:101} INFO - Starting the triggerer
2021-12-22 22:19:32,665] {triggerer_job.py:251} INFO - 0 triggers currently running
2021-12-22 22:20:32,938] {triggerer_job.py:251} INFO - 0 triggers currently running
2021-12-22 22:21:26,140] {triggerer_job.py:356} INFO - Trigger <airflow.triggers.temporal.DateTimeTrigger moment=2021-12-22T22:41:24.717838+00:00>
2021-12-22 22:21:29,155] {triggerer_job.py:356} INFO - Trigger <airflow.triggers.temporal.DateTimeTrigger moment=2021-12-22T22:41:27.807735+00:00>
2021-12-22 22:21:31,165] {triggerer_job.py:356} INFO - Trigger <airflow.triggers.temporal.DateTimeTrigger moment=2021-12-22T22:41:30.722945+00:00>
2021-12-22 22:21:33,172] {triggerer_job.py:251} INFO - 3 triggers currently running

```

Note that if you are running Airflow on [Astro](#), the triggerer runs automatically if you are on Astro Runtime 4.0+. If you are using Astronomer Software 0.26+, you can add a triggerer to an Airflow 2.2+ deployment in the **Deployment Settings** tab. This [guide](#) details the steps for configuring this feature in the platform.

As tasks are raised into a deferred state, triggers are registered in the triggerer. You can set the number of concurrent triggers that can run in a single triggerer process with the `default_capacity` configuration setting in Airflow. This can also be set via the `AIRFLOW__TRIGGERER__DEFAULT_CAPACITY` environment variable. By default, this variable's value is `1,000`.

High Availability

Note that triggers are designed to be highly-available. You can implement this by starting multiple triggerer processes. Similar to the [HA scheduler](#) introduced in Airflow 2.0, Airflow ensures that they co-exist with correct locking and HA. You can reference the [Airflow docs](#) for further information on this topic.

Creating Your Own Deferrable Operator

If you have an operator that would benefit from being asynchronous but does not yet exist in OSS Airflow or Astro Runtime, you can create your own. [The Airflow docs](#) have great instructions to get you started.

3. DAG Design

Because Airflow is 100% code, knowing the basics of Python is all it takes to get started writing DAGs. However, writing DAGs that are efficient, secure, and scalable requires some Airflow-specific finesse. In this section, we will cover some best practices for developing DAGs that make the most of what Airflow has to offer.

In general, most of the best practices we cover here fall into one of two categories:

- **DAG design**
- **Using Airflow as an orchestrator**

Reviewing Idempotency

- Before we jump into best practices specific to Airflow, we need to review one concept which applies to all data pipelines.

Idempotency is the foundation for many computing practices, including the Airflow best practices in this section. Specifically, it is a quality: A computational operation is considered idempotent if it always produces the same output.

In the context of Airflow, a DAG is considered idempotent if every DAG Run generates the same results even when run multiple times. Designing idempotent DAGs decreases recovery time from failures and prevents data loss.

DAG Design

The following DAG design principles will help to make your DAGs idempotent, efficient, and readable.

Keep Tasks Atomic

When breaking up your pipeline into individual tasks, ideally each task should be atomic. This means each task should be responsible for one operation that can be rerun independently of the others. Said another way, in an automated task, a success in the part of the task means a success of the entire task.

For example, in an ETL pipeline you would ideally want your Extract, Transform, and Load operations covered by three separate tasks. Atomizing these tasks allows you to rerun each operation in the pipeline independently, which supports idempotence

Use Template Fields, Variables, and Macros

With template fields in Airflow, you can pull values into DAGs using environment variables and jinja templating. Compared to using Python functions, using template fields helps keep your DAGs idempotent and ensures you aren't executing functions on every Scheduler heartbeat (see "Avoid Top Level Code in Your DAG File" below for more about Scheduler optimization).

Contrary to our best practices, the following example defines variables based on `datetime` Python functions:

```
1 # Variables used by tasks
2 # Bad example - Define today's and yesterday's date using
3 datetime module
4 today = datetime.today()
5 yesterday = datetime.today() - timedelta(1)
```

If this code is in a DAG file, these functions will be executed on every Scheduler heartbeat, which may not be performant. Even more importantly, this doesn't produce an idempotent DAG: If you needed to rerun a previously failed DAG Run for a past date, you wouldn't be able to because `datetime.today()` is relative to the current date, not the DAG execution date. A better way of implementing this is by using an Airflow variable:

```
1 # Variables used by tasks
2 # Good example - Define yesterday's date with an Airflow
3 variable
4 yesterday = {{ yesterday_ds_nodash }}
```

You can use one of Airflow's many built-in [variables and macros](#), or you can create your own templated field to pass in information at runtime. For more on this topic check out our guide on [templating and macros in Airflow](#).

Incremental Record Filtering

It is ideal to break out your pipelines into incremental extracts and loads wherever possible. For example, if you have a DAG that runs hourly, each DAG Run should process only records from that hour, rather than the whole dataset. When the results in each DAG Run represent only a small subset of your total dataset, a failure in one subset of the data won't prevent the rest of your DAG Runs from completing successfully. And if your DAGs are idempotent, you can rerun a DAG for only the data that failed rather than reprocessing the entire dataset.

There are multiple ways you can achieve incremental pipelines. The two best and most common methods are described below.

- **Last Modified Date**
Using a "last modified" date is the gold standard for incremental loads. Ideally, each record in your source system has a column containing the last time the record was modified. With this design, a DAG Run looks for records that were updated within specific dates from this column. For example, with a DAG that runs hourly, each DAG Run will be responsible for loading any records that fall between the start and end of its hour. If any of those runs fail, it will not impact other Runs.
- **Sequence IDs**
When a last modified date is not available, a sequence or incrementing ID can be used for incremental loads. This logic works best when the source records are only being appended to and never updated. While we recommend implementing a "last modified" date system in your records if possible, basing your incremental logic off of a sequence ID can be a sound way to filter pipeline records without a last modified date.

Avoid Top-Level Code in Your DAG File

In the context of Airflow, we use “top-level code” to mean any code that isn’t part of your DAG or operator instantiations.

Airflow executes all code in the `dags_folder` on every `min_file_process_interval`, which defaults to 30 seconds (you can read more about this parameter in the Airflow docs). Because of this, top-level code that makes requests to external systems, like an API or a database, or makes function calls outside of your tasks can cause performance issues. Additionally, including code that isn’t part of your DAG or operator instantiations in your DAG file makes the DAG harder to read, maintain, and update.

Treat your DAG file like a config file and leave all of the heavy lifting to the hooks and operators that you instantiate within the file. If your DAGs need to access additional code such as a SQL script or a Python function, keep that code in a separate file that can be read into a DAG Run.

For one example of what not to do, in the DAG below a `PostgresOperator` executes a SQL query that was dropped directly into the DAG file:

```
1 from airflow import DAG
2 from airflow.providers.postgres.operators.postgres import
3 PostgresOperator
4 from datetime import datetime, timedelta
5
6 #Default settings applied to all tasks
7 default_args = {
8     'owner': 'airflow',
9     'depends_on_past': False,
10    'email_on_failure': False,
11    'email_on_retry': False,
12    'retries': 1,
13    'retry_delay': timedelta(minutes=1)
14 }
```

```
15 #Instantiate DAG
16 with DAG('bad_practices_dag_1',
17         start_date=datetime(2021, 1, 1),
18         max_active_runs=3,
19         schedule_interval='@daily',
20         default_args=default_args,
21         catchup=False
22         ) as dag:
23
24     t0 = DummyOperator(task_id='start')
25
26     #Bad example with top level SQL code in the DAG file
27     query_1 = PostgresOperator(
28         task_id='covid_query_wa',
29         postgres_conn_id='postgres_default',
30         sql=''with yesterday_covid_data as (
31             SELECT *
32             FROM covid_state_data
33             WHERE date = {{ params.today }}
34             AND state = 'WA'
35         ),
36         today_covid_data as (
37             SELECT *
38             FROM covid_state_data
39             WHERE date = {{ params.yesterday }}
40             AND state = 'WA'
41         ),
42         two_day_rolling_avg as (
43             SELECT AVG(a.state, b.state) as two_day_avg
44                 FROM yesterday_covid_data a
45                 JOIN yesterday_covid_data b
46                 ON a.state = b.state
47         )
48         SELECT a.state, b.state, c.two_day_avg
49         FROM yesterday_covid_data a
```

```

50         JOIN today_covid_data b
51         ON a.state=b.state
52         JOIN two_day_rolling_avg c
53         ON a.state=b.two_day_avg;''',
54         params={'today': today, 'yesterday':yesterday}
55     )

```

Keeping the query in the DAG file like this makes the DAG harder to read and maintain. Instead, in the DAG below we call in a file named `covid_state_query.sql` into our PostgresOperator instantiation, which embodies the best practice:

```

1  from airflow import DAG
2  from airflow.providers.postgres.operators.postgres import
3  PostgresOperator
4  from datetime import datetime, timedelta
5
6  #Default settings applied to all tasks
7  default_args = {
8      'owner': 'airflow',
9      'depends_on_past': False,
10     'email_on_failure': False,
11     'email_on_retry': False,
12     'retries': 1,
13     'retry_delay': timedelta(minutes=1)
14 }
15
16 #Instantiate DAG

```

```

17 with DAG('good_practices_dag_1',
18         start_date=datetime(2021, 1, 1),
19         max_active_runs=3,
20         schedule_interval='@daily',
21         default_args=default_args,
22         catchup=False,
23         template_searchpath='/usr/local/airflow/include'
24 #include path to look for external files
25     ) as dag:
26
27     query = PostgresOperator(
28         task_id='covid_query_{0}'.format(state),
29         postgres_conn_id='postgres_default',
30         sql='covid_state_query.sql', #reference query
31 #kept in separate file
32         params={'state': "" + state + ""}
33     )
34     'email_on_retry': False,
35     'retries': 1,
36     'retry_delay': timedelta(minutes=1)
37 }
38
39 #Instantiate DAG

```

Use a Consistent Method for Task Dependencies

In Airflow, task dependencies can be set multiple ways. You can use `set_upstream()` and `set_downstream()` functions, or you can use `<<` and `>>` operators. Which method you use is a matter of personal preference, but for readability it's best practice to choose one method and stick with it.

For example, instead of mixing methods like this:

```
1 task_1.set_downstream(task_2)
2 task_3.set_upstream(task_2)
3 task_3 >> task_4
```

Try to be consistent with something like this:

```
1 task_1 >> task_2 >> [task_3, task_4]
```

Leverage Airflow Features

The next category of best practices relates to using Airflow as what it was originally designed to be: an orchestrator. Using Airflow as an orchestrator makes it easier to scale and pull in the right tools based on your needs.

Make Use of Provider Packages

One of the best aspects of Airflow is its robust and active community, which has resulted in integrations between Airflow and other tools known as [provider packages](#).

Provider packages enable you to orchestrate third party data processing jobs directly from Airflow. Wherever possible, it's best practice to make use of these integrations rather than writing Python functions yourself (no need to reinvent the wheel). This makes it easier for teams using existing tools to adopt Airflow, and it means you get to write less code.

For easy discovery of all the great provider packages out there, check out the [Astronomer Registry](#).

Decide Where to Run Data Processing Jobs

Because DAGs are written in Python, you have many options available for implementing data processing. For small to medium scale workloads, it is typically safe to do your data processing within Airflow as long as you allocate enough resources to your Airflow infrastructure. Large data processing jobs are typically best offloaded to a framework specifically optimized for those use cases, such as [Apache Spark](#). You can then use Airflow to orchestrate those jobs.

We recommend that you consider the size of your data now and in the future when deciding whether to process data within Airflow or offload to an external tool. If your use case is well suited to processing data within Airflow, then we would recommend the following:

- Ensure your Airflow infrastructure has the necessary resources.
- Use the Kubernetes Executor to isolate task processing and have more control over resources at the task level.
- Use a [custom XCom backend](#) if you need to pass any data between the tasks so you don't overload your metadata database.

Use Intermediary Data Storage

Because it requires less code and fewer pieces, it can be tempting to write your DAGs to move data directly from your source to destination. However, this means you can't individually rerun the extract or load portions of the pipeline. By putting an intermediary storage layer such as S3 or SQL Staging tables in between your source and destination, you can separate the testing and rerunning of the extract and load.

Depending on your data retention policy, you could modify the load logic and rerun the entire historical pipeline without having to rerun the extracts. This is also useful in situations where you no longer have access to the source system (e.g. you hit an API limit).

Use an ELT Framework

Whenever possible, look to implement an ELT (extract, load, transform) data pipeline pattern with your DAGs. This means that you should look to offload as much of the transformation logic to the source systems or the destinations systems as possible, which leverages the strengths of all tools in your data ecosystem. Many modern data warehouse tools, such as [Snowflake](#), give you easy to access to compute to support the ELT framework, and are easily used in conjunction with Airflow.

Other Best Practices

Finally, here are a few other noteworthy best practices that don't fall under the two categories above.

Use a Consistent File Structure

Having a consistent file structure for Airflow projects keeps things organized and easy to adopt. At Astronomer, we use:

```
1 |— dags/ # Where your DAGs go
2 |   |— example-dag.py # An example dag that comes with
3 |   the initialized project
4 |— Dockerfile # For Astronomer's Docker image and runtime
5 |   overrides
6 |— include/ # For any other files you'd like to include
7 |— plugins/ # For any custom or community Airflow plugins
8 |— packages.txt # For OS-level packages
9 |— requirements.txt # For any Python packages
```

Use DAG Name and Start Date Properly

You should always use a static `start_date` with your DAGs. A dynamic `start_date` is misleading and can cause failures when clearing out failed task instances and missing DAG runs.

Additionally, if you change the `start_date` of your DAG you should also change the DAG name. Changing the `start_date` of a DAG creates a new entry in Airflow's database, which could confuse the scheduler because there will be two DAGs with the same name but different schedules.

Changing the name of a DAG also creates a new entry in the database, which

powers the dashboard, so follow a consistent naming convention since changing a DAG's name doesn't delete the entry in the database for the old name.

Set Retries at the DAG Level

Even if your code is perfect, failures happen. In a distributed environment where task containers are executed on shared hosts, it's possible for tasks to be killed off unexpectedly. When this happens, you might see Airflow's logs mention a [zombie process](#).

Issues like this can be resolved by using task retries. The best practice is to set retries as a `default_arg` so they are applied at the DAG level and get more granular for specific tasks only where necessary. A good range to try is ~2–4 retries.

Passing Data Between Airflow Tasks

Introduction

Sharing data between tasks is a very common use case in Airflow. If you've been writing DAGs, you probably know that breaking them up into appropriately small tasks is the best practice for debugging and recovering quickly from failures. But, maybe one of your downstream tasks requires metadata about an upstream task or processes the results of the task immediately before it.

There are a few methods you can use to implement data sharing between your Airflow tasks. In this section, we will walk through the two most commonly used methods, discuss when to use each, and show some example DAGs to demonstrate the implementation. Before we dive into the specifics, there are a couple of high-level concepts that are important when writing DAGs where data is shared between tasks.

Ensure Idempotency

An important concept for any data pipeline, including an Airflow DAG, is idempotency. This is the property whereby an operation can be applied multiple times without changing the result. We often hear about this concept as it applies to your entire DAG; if you execute the same DAGRun multiple times, you will get the same result. However, this concept also applies to tasks within your DAG; if every task in your DAG is idempotent, your full DAG will be idempotent as well.

When designing a DAG that passes data between tasks, it is important to ensure that each task is idempotent. This will help you recover and ensure no data is lost should you have any failures.

Consider the Size of Your Data

Knowing the size of the data you are passing between Airflow tasks is important when deciding which implementation method to use. As we will describe in detail below, XComs are one method of passing data between tasks, but they are only appropriate for small amounts of data. Large data sets will require a method making use of intermediate storage and possibly utilizing an external processing framework.

XCom

The first method for passing data between Airflow tasks is to use XCom, which is a key Airflow feature for sharing task data.

What is XCom

[XCom](#) (short for cross-communication) is a native feature within Airflow. XComs allow tasks to exchange task metadata or small amounts of data. They are defined by a key, value, and timestamp.

XComs can be “pushed”, meaning sent by a task, or “pulled”, meaning received by a task. When an XCom is pushed, it is stored in Airflow's metadata database and made available to all other tasks. Any time a task returns a val-

ue (e.g. if your Python callable for your [PythonOperator](#) has a return), that value will automatically be pushed to XCom. Tasks can also be configured to push XComs by calling the `xcom_push()` method. Similarly, `xcom_pull()` can be used in a task to receive an XCom.

You can view your XComs in the Airflow UI by navigating to Admin XComs.

You should see something like this:

Key	Value	Timestamp	Execution Date	Task Id	Dag Id
return_value	68999	2021-03-01, 17:14:14	2021-03-01, 16:44:01	testing_increase_task_group.get_testing_increase_data_tx	xcom_dag
return_value	25679	2021-03-01, 17:14:14	2021-03-01, 16:44:01	testing_increase_task_group.get_testing_increase_data_co	xcom_dag
return_value	0	2021-03-01, 17:14:14	2021-03-01, 16:44:01	testing_increase_task_group.get_testing_increase_data_or	xcom_dag
return_value	21008	2021-03-01, 17:14:14	2021-03-01, 16:44:01	testing_increase_task_group.get_testing_increase_data_wa	xcom_dag
return_value	249616	2021-03-01, 17:14:14	2021-03-01, 16:44:01	testing_increase_task_group.get_testing_increase_data_ca	xcom_dag

When to Use XComs

XComs should be used to pass **small** amounts of data between tasks. Things like task metadata, dates, model accuracy, or single value query results are all ideal data to use with XCom.

While there is nothing stopping you from passing small data sets with XCom, be very careful when doing so. This is not what XCom was designed for, and using it to pass data like pandas dataframes can degrade the performance of your DAGs and take up storage in the metadata database.

XCom cannot be used for passing large data sets between tasks. The limit for the size of the XCom is determined by which metadata database you are using:

- **Postgres: 1 Gb**
- **SQLite: 2 Gb**
- **MySQL: 64 Kb**

You can see that these limits aren't very big. And even if you think your data might squeak just under, don't use XComs. Instead, see the section below on using intermediary data storage, which is more appropriate for larger chunks of data.

Custom XCom Backends

[Custom XCom Backends](#) are a new feature available in Airflow 2.0 and greater. Using an XCom backend means you can push and pull XComs to and from an external system such as S3, GCS, or HDFS rather than the default of Airflow's metadata database. You can also implement your own serialization / deserialization methods to define how XComs are handled. This is a concept in its own right, so we won't go into too much detail here, but you can learn more by reading our [guide on implementing custom XCom backends](#).

Example DAGs

This section will show a couple of example DAGs that use XCom to pass data between tasks. For this example, we are interested in analyzing the increase in a total number of Covid tests for the current day for a particular state. To implement this use case, we will have one task that makes a request to the [Covid Tracking API](#) and pulls the `totalTestResultsIncrease` parameter from the results. We will then use another task to take that result and complete some sort of analysis. This is a valid use case for XCom because the data being passed between the tasks is a single integer.

```

1  from airflow import DAG
2  from airflow.operators.python_operator import PythonOp-
3  erator
4  from datetime import datetime, timedelta
5
6  import requests
7  import json
8
9  url = 'https://covidtracking.com/api/v1/states/'
10 state = 'wa'
11
12 def get_testing_increase(state, ti):
13     """
14     Gets totalTestResultsIncrease field from Covid API
15     for given state and returns value
16     """
17     res = requests.get(url+'{0}/current.json'.for-
18 mat(state))
19     testing_increase = json.loads(res.text)['totalT-
20 estResultsIncrease']
21
22     ti.xcom_push(key='testing_increase', value=testing_
23 increase)
24
25 def analyze_testing_increases(state, ti):
26     """
27     Evaluates testing increase results
28     """
29     testing_increases=ti.xcom_pull(key='testing_in-
30 crease', task_ids='get_testing_increase_data_{0}'.for-
31 mat(state))
32     print('Testing increases for {0}:'.format(state),
33 testing_increases)
34     #run some analysis here

```

```

30 # Default settings applied to all tasks
31 default_args = {
32     'owner': 'airflow',
33     'depends_on_past': False,
34     'email_on_failure': False,
35     'email_on_retry': False,
36     'retries': 1,
37     'retry_delay': timedelta(minutes=5)
38 }
39
40 with DAG('xcom_dag',
41         start_date=datetime(2021, 1, 1),
42         max_active_runs=2,
43         schedule_interval=timedelta(minutes=30),
44         default_args=default_args,
45         catchup=False
46         ) as dag:
47
48     opr_get_covid_data = PythonOperator(
49         task_id = 'get_testing_increase_data_{0}'.for-
50 mat(state),
51         python_callable=get_testing_increase,
52         op_kwargs={'state':state}
53     )
54
55     opr_analyze_testing_data = PythonOperator(
56         task_id = 'analyze_data',
57         python_callable=analyze_testing_increases,
58         op_kwargs={'state':state}
59     )
60
61     opr_get_covid_data >> opr_analyze_testing_data

```


In this DAG we have two `PythonOperator` tasks which share data using the `xcom_push` and `xcom_pull` functions. Note that in the `get_testing_increase` function, we used the `xcom_push` method so that we could specify the `key` name. Alternatively, we could have made the function return the `testing_increase` value, because any value returned by an operator in Airflow will automatically be pushed to XCom; if we had used this method, the XCom key would be “returned_value”.

For the `xcom_pull` call in the `analyze_testing_increases` function, we specify the key and `task_ids` associated with the XCom we want to retrieve. Note that this allows you to pull any XCom value (or multiple values) at any time into a task; it does not need to be from the task immediately prior, as shown in this example.

If we run this DAG and then go to the XComs page in the Airflow UI, we see that a new row has been added for our `get_testing_increase_data_wa` task with the key `testing_increase` and value returned from the API.

Key	Value	Timestamp	Execution Date	Task Id	Dag Id
testing_increase	21008	2021-03-01 21:51:20	2021-03-01 21:51:18	get_testing_increase_data_wa	xcom_dag

In the logs for the `analyze_data` task, we can see the value from the prior task was printed, meaning the value was successfully retrieved from XCom.

```

Task Instance: analyze_data at 2021-03-01 21:51:18+00
Log by attempts
1
*** Reading local file: /usr/local/airflow/logs/xcom_dag/analyze_data/2021-03-01T21:51:18.669693+00:00/1.log
[2021-03-01 21:51:21,230] {taskinstance.py:1851} INFO - Dependencies all met for <TaskInstance: xcom_dag.analyze_data 2021-03-01T21:51:18.669693+00:00 [queued]>
[2021-03-01 21:51:21,340] {taskinstance.py:1851} INFO - Dependencies all met for <TaskInstance: xcom_dag.analyze_data 2021-03-01T21:51:18.669693+00:00 [queued]>
[2021-03-01 21:51:21,340] {taskinstance.py:1842} INFO -
[2021-03-01 21:51:21,340] {taskinstance.py:1843} INFO - Starting attempt 1 of 2
[2021-03-01 21:51:21,340] {taskinstance.py:1844} INFO -
[2021-03-01 21:51:21,353] {taskinstance.py:1863} INFO - Executing <task(PythonOperator): analyze_data> on 2021-03-01T21:51:18.669693+00:00
[2021-03-01 21:51:21,356] {standard_task_runner.py:52} INFO - Started process 3522 to run task
[2021-03-01 21:51:21,360] {standard_task_runner.py:76} INFO - Running: ['airflow', 'tasks', 'run', 'xcom_dag', 'analyze_data', '2021-03-01T21:51:18.669693+00:00', '--job-id', '49', '--pool', 'default_pool', 'airflow']
[2021-03-01 21:51:21,361] {standard_task_runner.py:77} INFO - Job 49: Subtask analyze_data
[2021-03-01 21:51:21,399] {logging_mixin.py:183} INFO - Running <TaskInstance: xcom_dag.analyze_data 2021-03-01T21:51:18.669693+00:00 [running]> on host 76b978e0291e
[2021-03-01 21:51:21,439] {taskinstance.py:1256} INFO - Exporting the following env vars:
AIRFLOW_CTX_DAG_OWNER=airflow
AIRFLOW_CTX_DAG_ID=xcom_dag
AIRFLOW_CTX_TASK_ID=analyze_data
AIRFLOW_CTX_EXECUTION_DATE=2021-03-01T21:51:18.669693+00:00
AIRFLOW_CTX_DAG_RUN_ID=manual_2021-03-01T21:51:18.669693+00:00
[2021-03-01 21:51:21,445] {logging_mixin.py:183} INFO - Testing increases for wa: 21008
[2021-03-01 21:51:21,445] {python.py:118} INFO - Done. Returned value was: None
[2021-03-01 21:51:21,458] {taskinstance.py:1166} INFO - Marking task as SUCCESS. dag_id=xcom_dag, task_id=analyze_data, execution_date=20210301T215118, start_date=20210301T215121, end_date=20210301T215121
[2021-03-01 21:51:21,480] {taskinstance.py:1219} INFO - 0 downstream tasks scheduled from follow-on schedule check
[2021-03-01 21:51:21,493] {local_task_job.py:142} INFO - Task exited with return code 0
  
```

TaskFlow API

Another way to implement this use case is to use the [TaskFlow API](#) that was released with Airflow 2.0. With the TaskFlow API, returned values are pushed to XCom as usual, but XCom values can be pulled simply by adding the key as an input to the function as shown in the following DAG:

```

1 from airflow.decorators import dag, task
2 from datetime import datetime
3
4 import requests
5 import json
6
7 url = 'https://covidtracking.com/api/v1/states/'
8 state = 'wa'
9
10 default_args = {
11     'start_date': datetime(2021, 1, 1)
12 }
  
```

```

13 @dag('xcom_taskflow_dag', schedule_interval='@daily', de-
14 fault_args=default_args, catchup=False)
15 def taskflow():
16
17     @task
18     def get_testing_increase(state):
19         """
20         Gets totalTestResultsIncrease field from Covid API
21         for given state and returns value
22         """
23         res = requests.get(url+'{0}/current.json'.for-
24 mat(state))
25         return{'testing_increase': json.loads(res.text)
26 ['totalTestResultsIncrease']}
27
28     @task
29     def analyze_testing_increases(testing_increase: int):
30         """
31         Evaluates testing increase results
32         """
33         print('Testing increases for {0}:'.format(state),
34 testing_increase)
35         #run some analysis here
36
37         analyze_testing_increases(get_testing_increase(state))
38
39 dag = taskflow()

```

This DAG is functionally the same as the first one, but thanks to the TaskFlow API there is less code required overall and no additional code required for passing the data between the tasks using XCom.

Intermediary Data Storage

As mentioned above, XCom can be a great option for sharing data between tasks because it doesn't rely on any tools external to Airflow itself. However, it is only designed to be used for very small amounts of data. What if the data you need to pass is a little bit larger, for example, a small dataframe?

The best way to manage this use case is to use intermediary data storage. This means saving your data to some system external to Airflow at the end of one task, then reading it in from that system in the next task. This is commonly done using cloud file storage such as S3, GCS, Azure Blob Storage, etc., but it could also be done by loading the data in either a temporary or persistent table in a database.

We will note here that while this is a great way to pass data that is too large to be managed with XCom, you should still exercise caution. Airflow is meant to be an orchestrator, not an execution framework. If your data is very large, it is probably a good idea to complete any processing using a framework like Spark or compute-optimized data warehouses like Snowflake or dbt.

Example DAG

Building on our Covid example above, let's say instead of a specific value of testing increases, we are interested in getting all of the daily Covid data for a state and processing it. This case would not be ideal for XCom, but since the data returned is a small dataframe, it is likely okay to process using Airflow.

```

1  from airflow import DAG
2  from airflow.operators.python_operator import PythonOperator
3
4  from airflow.providers.amazon.aws.hooks.s3 import S3Hook
5  from datetime import datetime, timedelta
6
7  from io import StringIO
8  import pandas as pd
9  import requests
10
11 s3_conn_id = 's3-conn'
12 bucket = 'astro-workshop-bucket'
13 state = 'wa'
14 date = '{{ yesterday_ds_nodash }}'
15
16 def upload_to_s3(state, date):
17     '''Grabs data from Covid endpoint and saves to flat
18     file on S3
19     '''
20     # Connect to S3
21     s3_hook = S3Hook(aws_conn_id=s3_conn_id)
22
23     # Get data from API
24     url = 'https://covidtracking.com/api/v1/states/'
25     res = requests.get(url+'{0}/{1}.csv'.format(state,
26     date))
27
28     # Save data to CSV on S3
29     s3_hook.load_string(res.text, '{0}_{1}.csv'.format(
30     state, date), bucket_name=bucket, replace=True)
31
32 def process_data(state, date):
33     '''Reads data from S3, processes, and saves to new S3
34     file

```

```

35     # Connect to S3
36     s3_hook = S3Hook(aws_conn_id=s3_conn_id)
37
38     # Read data
39     data = StringIO(s3_hook.read_key(key='{0}_{1}.csv'.
40     format(state, date), bucket_name=bucket))
41     df = pd.read_csv(data, sep=',')
42
43     # Process data
44     processed_data = df[['date', 'state', 'positive', 'neg-
45     ative']]
46
47     # Save processed data to CSV on S3
48     s3_hook.load_string(processed_data.to_string(), '{0}_
49     {1}_processed.csv'.format(state, date), bucket_name=bucket,
50     replace=True)
51
52 # Default settings applied to all tasks
53 default_args = {
54     'owner': 'airflow',
55     'depends_on_past': False,
56     'email_on_failure': False,
57     'email_on_retry': False,
58     'retries': 1,
59     'retry_delay': timedelta(minutes=1)
60 }
61
62 with DAG('intermediary_data_storage_dag',
63         start_date=datetime(2021, 1, 1),
64         max_active_runs=1,
65         schedule_interval='@daily',
66         default_args=default_args,
67         catchup=False
68

```

```

69         ) as dag:
70
71         generate_file = PythonOperator(
72             task_id='generate_file_{0}'.format(state),
73             python_callable=upload_to_s3,
74             op_kwargs={'state': state, 'date': date}
75         )
76
77         process_data = PythonOperator(
78             task_id='process_data_{0}'.format(state),
79             python_callable=process_data,
80             op_kwargs={'state': state, 'date': date}
81         )
82
83         generate_file >> process_data
'''

```

In this DAG we make use of the [S3Hook](#) to save data retrieved from the API to a CSV on S3 in the `generate_file` task. The `process_data` task then grabs that data from S3, converts it to a dataframe for processing, and then saves the processed data back to a new CSV on S3.

Using Task Groups in Airflow

Overview

Prior to the release of [Airflow 2.0](#) in December 2020, the only way to group tasks and create modular workflows within Airflow was to use SubDAGs. SubDAGs were a way of presenting a cleaner-looking DAG by capitalizing on code patterns. For example, ETL DAGs usually share a pattern of tasks that extract data from a source, transform the data, and load it somewhere. The SubDAG would visually group the repetitive tasks into one UI task, making the pattern between tasks clearer.

However, SubDAGs were really just DAGs embedded in other DAGs. This caused both performance and functional issues:

- When a SubDAG is triggered, the SubDAG and child tasks take up worker slots until the entire SubDAG is complete. This can delay other task processing and, depending on your number of worker slots, can lead to deadlocking.
- SubDAGs have their own parameters, schedule, and enabled settings. When these are not consistent with their parent DAG, unexpected behavior can occur.

Unlike SubDAGs, Task Groups are just a UI grouping concept. Starting in Airflow 2.0, you can use Task Groups to organize tasks within your DAG's graph view in the Airflow UI. This avoids the added complexity and performance issues of SubDAGs, all while using less code!

In this section, we will walk through how to create [Task Groups](#) and show some example DAGs to demonstrate their scalability.

Creating Task Groups

To use Task Groups you'll need to use the following import statement.

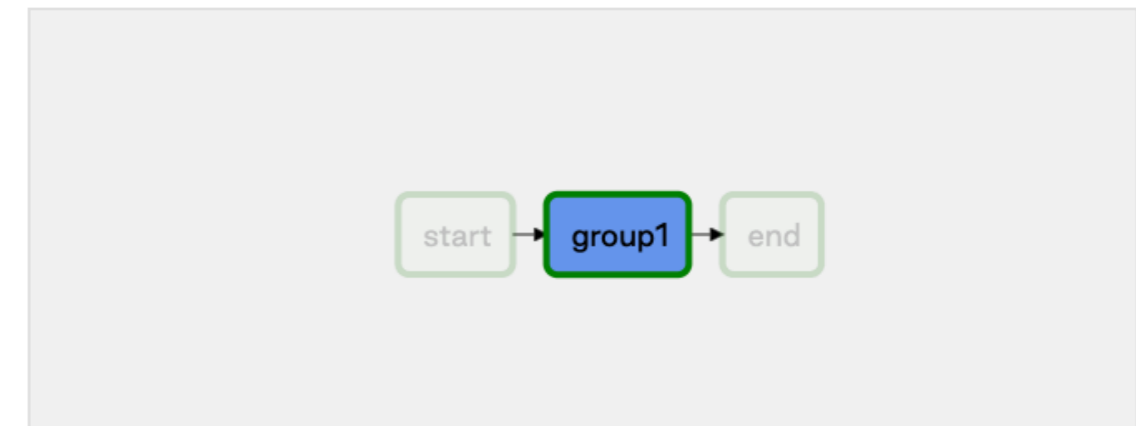
```
1 from airflow.utils.task_group import TaskGroup
```

For our first example, we will instantiate a Task Group using a `with` statement and provide a `group_id`. Inside our Task Group, we will define our two tasks, `t1` and `t2`, and their respective dependencies.

You can use dependency operators (`<<` and `>>`) on Task Groups in the same way that you can with individual tasks. Dependencies applied to a Task Group are applied across its tasks. In the following code, we will add additional dependencies to `t0` and `t3` to the Task Group, which automatically applies the same dependencies across `t1` and `t2`:

```
1 t0 = DummyOperator(task_id='start')
2
3 # Start Task Group definition
4 with TaskGroup(group_id='group1') as tg1:
5     t1 = DummyOperator(task_id='task1')
6     t2 = DummyOperator(task_id='task2')
7
8     t1 >> t2
9 # End Task Group definition
10
11 t3 = DummyOperator(task_id='end')
12
13 # Set Task Group's (tg1) dependencies
14 t0 >> tg1 >> t3
```

In the Airflow UI, Task Groups look like tasks with blue shading. When we expand `group1` by clicking on it, we see blue circles where the Task Group's dependencies have been applied to the grouped tasks. The task(s) immediately to the right of the first blue circle (`t1`) get the group's upstream dependencies and the task(s) immediately to the left (`t2`) of the last blue circle get the group's downstream dependencies.



Note: When your task is within a Task Group, your callable `task_id` will be the `task_id` prefixed with the `group_id` (i.e. `group_id.task_id`). This ensures the uniqueness of the `task_id` across the DAG. This is important to remember when calling specific tasks with XCOM passing or branching operator decisions.

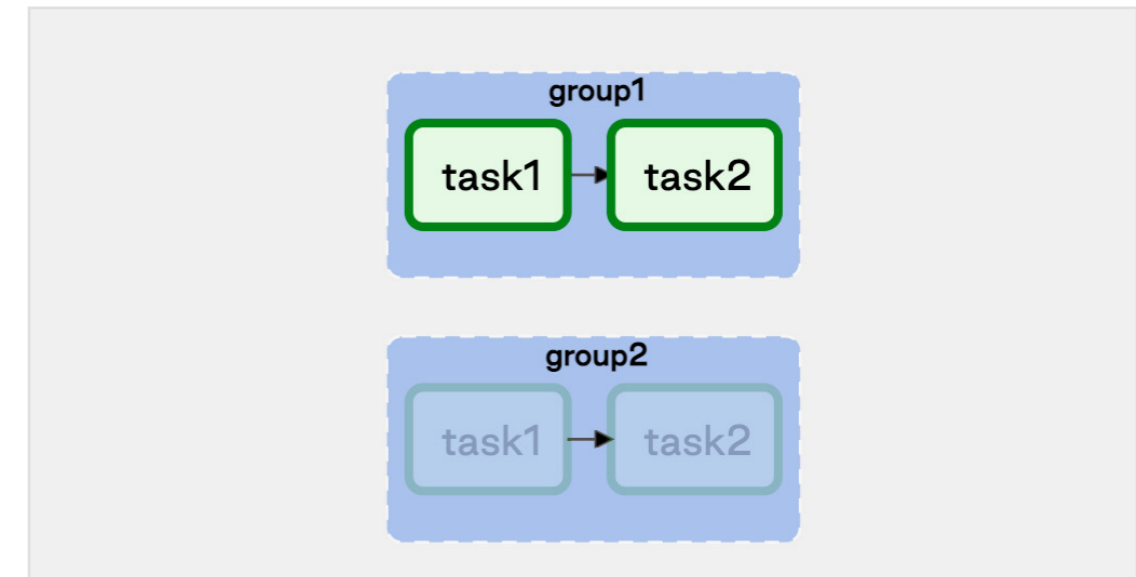
Dynamically Generating Task Groups

Just like with DAGs, Task Groups can be dynamically generated to make use of patterns within your code. In an ETL DAG, you might have similar downstream tasks that can be processed independently, such as when you call different API endpoints for data that needs to be processed and stored in the same way. For this use case, we can dynamically generate Task Groups by API endpoint. Just like with manually written Task Groups, generated Task Groups can be drilled into from the Airflow UI to see specific tasks.

In the code below, we use iteration to create multiple Task Groups. While the tasks and dependencies remain the same across Task Groups, we can change which parameters are passed in to each Task Group based on the `group_id`:

```
1 for g_id in range(1,3):
2     with TaskGroup(group_id=f'group{g_id}') as tg1:
3         t1 = DummyOperator(task_id='task1')
4         t2 = DummyOperator(task_id='task2')
5
6         t1 >> t2
```

This screenshot shows the expanded view of the Task Groups we generated above in the Airflow UI:



What if your Task Groups can't be processed independently? Next, we will show how to call Task Groups and define dependencies between them.

Ordering Task Groups

By default, using a loop to generate your Task Groups will put them in parallel. If your Task Groups are dependent on elements of another Task Group, you'll want to run them sequentially. For example, when loading tables with foreign keys, your primary table records need to exist before you can load your foreign table.

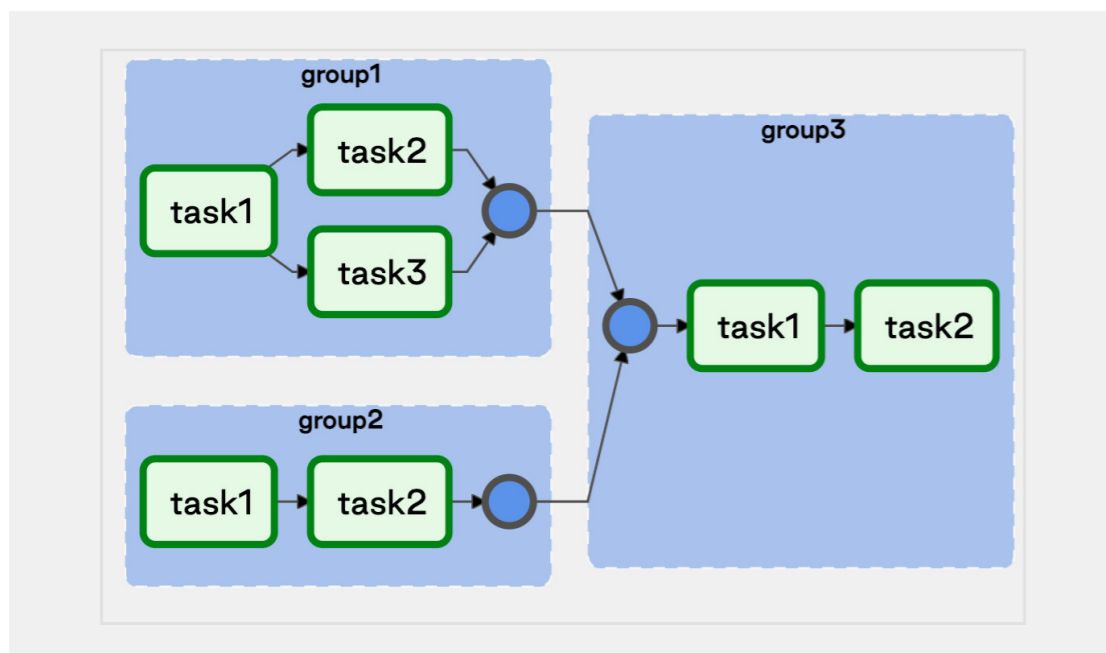
In the example below, our third dynamically generated Task Group has a foreign key constraint on both our first and second dynamically generated Task Groups, so we will want to process it last. To do this, we will create an empty list and append our Task Group objects as they are generated. Using this list, we can reference the Task Groups and define their dependencies to each other:

```

1 groups = []
2 for g_id in range(1,4):
3     tg_id = f'group{g_id}'
4     with TaskGroup(group_id=tg_id) as tg1:
5         t1 = DummyOperator(task_id='task1')
6         t2 = DummyOperator(task_id='task2')
7
8         t1 >> t2
9
10        if tg_id == 'group1':
11            t3 = DummyOperator(task_id='task3')
12            t1 >> t3
13
14        groups.append(tg1)
15
16 [groups[0] , groups[1]] >> groups[2]

```

The following screenshot shows how these Task Groups appear in the Airflow UI:



Conditioning on Task Groups

In the above example, we added an additional task to `group1` based on our `group_id`. This was to demonstrate that even though we are dynamically creating Task Groups to take advantage of patterns, we can still introduce variations to the pattern while avoiding code redundancies from building each Task Group definition manually.

Nesting Task Groups

For additional complexity, you can nest Task Groups. Building on our previous ETL example, when calling API endpoints, we may need to process new records for each endpoint before we can process updates to them.

In the following code, our top-level Task Groups represent our new and updated record processing, while the nested Task Groups represent our API endpoint processing:

```

1 groups = []
2 for g_id in range(1,3):
3     with TaskGroup(group_id=f'group{g_id}') as tg1:
4         t1 = DummyOperator(task_id='task1')
5         t2 = DummyOperator(task_id='task2')
6
7         sub_groups = []
8         for s_id in range(1,3):
9             with TaskGroup(group_id=f'sub_group{s_id}') as
10            tg2:
11
12                st1 = DummyOperator(task_id='task1')
13                st2 = DummyOperator(task_id='task2')
14
15                st1 >> st2
16                sub_groups.append(tg2)

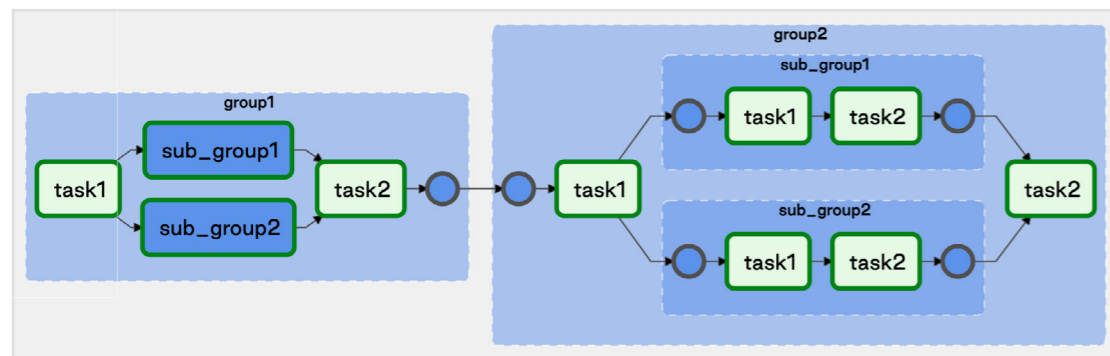
```

```

16     t1 >> sub_groups >> t2
17     groups.append(tg1)
18
19 groups[0] >> groups[1]

```

The following screenshot shows the expanded view of the nested Task Groups in the Airflow UI:



Takeaways

Task Groups are a dynamic and scalable UI grouping concept that eliminates the functional and performance issues of SubDAGs.

Ultimately, Task Groups give you the flexibility to group and organize your tasks in a number of ways. To help guide your implementation of Task Groups, think about:

- **What patterns exist in your DAGs?**
- **How can simplifying your DAG's graph better communicate its purpose?**

Note: Astronomer highly recommends avoiding SubDAGs if the intended use of the SubDAG is to simply group tasks within a DAG's Graph View. Airflow 2.0 introduces Task Groups which is a UI grouping concept that satisfies this purpose without the performance and functional issues of SubDAGs. While the `SubDagOperator` will continue to be supported, Task Groups are intended to replace it long-term.

Cross-DAG Dependencies

When designing Airflow DAGs, it is often best practice to put all related tasks in the same DAG. However, it's sometimes necessary to create dependencies between your DAGs. In this scenario, one node of a DAG is its own complete DAG, rather than just a single task. Throughout this guide, we'll use the following terms to describe DAG dependencies:

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

According to the Airflow documentation [on cross-DAG dependencies](#), designing DAGs in this way can be useful when:

- A DAG should only run after one or more datasets have been updated by tasks in other DAGs.
- Two DAGs are dependent, but they have different schedules.
- Two DAGs are dependent, but they are owned by different teams.
- A task depends on another task but for a different execution date.

For any scenario where you have dependent DAGs, we've got you covered! In this guide, we'll discuss multiple methods for implementing cross-DAG dependencies, including how to implement dependencies if your dependent DAGs are located in different Airflow deployments.

Note: All code in this section can be found in [this Github repo](#).

Assumed knowledge

To get the most out of this guide, you should have knowledge of:

- Dependencies in Airflow. See [Managing Dependencies in Apache Airflow](#).
- Airflow DAGs. See [Introduction to Airflow DAGs](#).
- Airflow operators. See [Operators 101](#).
- Airflow sensors. See [Sensors 101](#).

Implementing Cross-DAG Dependencies

There are multiple ways to implement cross-DAG dependencies in Airflow, including:

- [Dataset driven scheduling](#).
- The [TriggerDagRunOperator](#).
- The [ExternalTaskSensor](#).
- The [Airflow API](#).

In this section, we detail how to use each method and ideal scenarios for each, as well as how to view dependencies in the Airflow UI.

Note: It can be tempting to use SubDAGs to handle DAG dependencies, but we highly recommend against doing so as SubDAGs can create performance issues. Instead, use one of the other methods described below.

In Airflow 2.4+, you can use datasets to create data-driven dependencies between DAGs. This means that DAGs which access the same data can have explicit, visible relationships, and DAGs can be scheduled based on updates to this data.

You should use this method if you have a downstream DAG that should only run after a dataset has been updated by an upstream DAG, especially if those updates can be very irregular. This type of dependency also provides you with increased observability into the dependencies between your DAGs and datasets in the Airflow UI.

Using datasets requires knowledge of the following scheduling concepts:

- **Producing task:** A task that updates a specific dataset, defined by its `outlets` parameter.
- **Consuming DAG:** A DAG that will run as soon as a specific dataset(s) are updated.

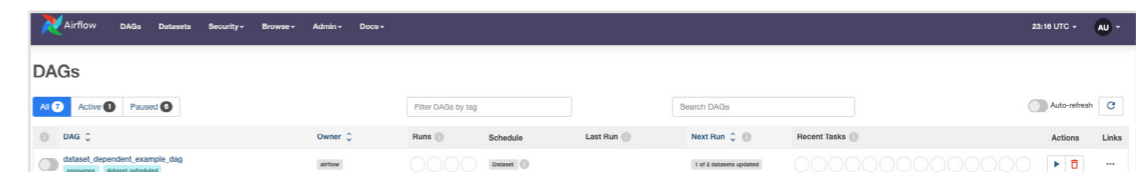
Any task can be made into a producing task by providing one or more datasets to the `outlets` parameter as shown below.

```
1 dataset1 = Dataset('s3://folder1/dataset_1.txt')
2
3 # producing task in the upstream DAG
4 EmptyOperator(
5     task_id="producing_task",
6     outlets=[dataset1] # flagging to Airflow that data-
7     set1 was updated
8 )
```

The downstream DAG is scheduled to run after `dataset1` has been updated by providing it to the `schedule` parameter.

```
1 dataset1 = Dataset('s3://folder1/dataset_1.txt')
2
3 # consuming DAG
4 with DAG(
5     dag_id='consuming_dag_1',
6     catchup=False,
7     start_date=datetime.datetime(2022, 1, 1),
8     schedule=[dataset1]
9 ) as dag:
```

In the Airflow UI, the **Next Run** column for the downstream DAG shows how many datasets the DAG depends on and how many of those have been updated since the last DAG run. The screenshot below shows that the DAG `dataset_dependent_example_dag` runs only after two different datasets have been updated. One of those datasets has already been updated by an upstream DAG.



Check out the [Datasets and Data Driven Scheduling in Airflow](#) guide to learn more and see an example implementation of this feature.

TriggerDagRunOperator

The TriggerDagRunOperator is an easy way to implement cross-DAG dependencies from the upstream DAG. This operator allows you to have a task in one DAG that triggers another DAG in the same Airflow environment.

Read more in-depth documentation about this operator on the [Astronomer Registry](#).

You can trigger a downstream DAG with the TriggerDagRunOperator from any point in the upstream DAG. If you set the operator's `wait_for_completion` parameter to `True`, the upstream DAG will pause and resume only once the downstream DAG has finished running.

A common use case for this implementation is when an upstream DAG fetches new testing data for a machine learning pipeline, runs and tests a model, and publishes the model's prediction. In case of the model underperforming, the TriggerDagRunOperator is used to kick off a separate DAG that retrains the model while the upstream DAG waits. Once the model is retrained and tested by the downstream DAG, the upstream DAG resumes and publishes the new model's results.

Below is an example DAG that implements the TriggerDagRunOperator to trigger the dependent-dag between two other tasks. The `trigger-dagrun-dag` will wait until `dependent-dag` has finished its run until it moves onto running `end_task`, since `wait_for_completion` in the `TriggerDagRunOperator` has been set to `True`.

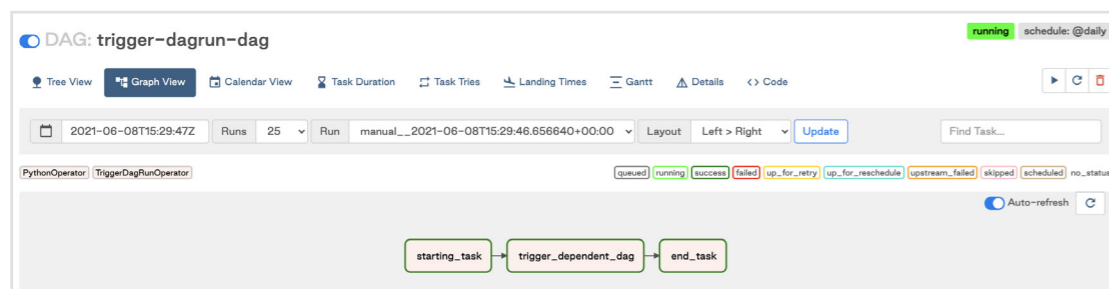
```
1 from airflow import DAG
2 from airflow.operators.python import PythonOperator
3 from airflow.operators.trigger_dagrun import TriggerDagRunOperator
4
5 from datetime import datetime, timedelta
6
7 def print_task_type(**kwargs):
8     """
9     Dummy function to call before and after dependent DAG.
10    """
11    print(f"The {kwargs['task_type']} task has completed.")
12
13
14 # Default settings applied to all tasks
15 default_args = {
16     'owner': 'airflow',
17     'depends_on_past': False,
18     'email_on_failure': False,
19     'email_on_retry': False,
20     'retries': 1,
21     'retry_delay': timedelta(minutes=5)
22 }
23
24 with DAG(
25     'trigger-dagrun-dag',
26     start_date=datetime(2021, 1, 1),
27     max_active_runs=1,
28     schedule_interval='@daily',
29     default_args=default_args,
30     catchup=False
31 ) as dag:
32
33     start_task = PythonOperator(
```

```

34     task_id='starting_task',
35     python_callable=print_task_type,
36     op_kwargs={'task_type': 'starting'}
37 )
38
39     trigger_dependent_dag = TriggerDagRunOperator(
40         task_id="trigger_dependent_dag",
41         trigger_dag_id="dependent-dag",
42         wait_for_completion=True
43     )
44
45     end_task = PythonOperator(
46         task_id='end_task',
47         python_callable=print_task_type,
48         op_kwargs={'task_type': 'ending'}
49     )
50
51     start_task >> trigger_dependent_dag >> end_task

```

In the following graph view, you can see that the `trigger_dependent_dag` task in the middle is the `TriggerDagRunOperator`, which runs the `dependent-dag`.



Note that if your dependent DAG requires a config input or a specific execution date, these can be specified in the operator using the `conf` and `execution_date` params respectively.

ExternalTaskSensor

To create cross-DAG dependencies from a downstream DAG, consider using one or more [ExternalTaskSensors](#). The downstream DAG will pause until a task is completed in the upstream DAG before resuming.

This method of creating cross-DAG dependencies is especially useful when you have a downstream DAG with different branches that depend on different tasks in one or more upstream DAGs. Instead of defining an entire DAG as being downstream of another DAG like with datasets, you can set a specific task in a downstream DAG to wait for a task to finish in an upstream DAG.

For example, you could have upstream tasks modifying different tables in a data warehouse and one downstream DAG running one branch of data quality checks for each of those tables. You can use one `ExternalTaskSensor` at the start of each branch to make sure that the checks running on each table only start, once the update to that specific table has finished.

Note: In Airflow 2.2+, a deferrable version of the `ExternalTaskSensor` is available, the [ExternalTaskSensorAsync](#). For more info on deferrable operators and their benefits, see [this guide](#).

An example DAG using three `ExternalTaskSensors` at the start of three parallel branches in the same DAG is shown below.

```

1 from airflow import DAG
2 from airflow.operators.python import PythonOperator
3 from airflow.sensors.external_task import ExternalTaskSensor
4
5 from airflow.operators.empty import EmptyOperator
6 from datetime import datetime, timedelta
7
8 def downstream_function_branch_1():
9     print('Upstream DAG 1 has completed. Starting tasks of
10 branch 1.')

```

```

34     catchup=False
35 ) as dag:
36
37     start = EmptyOperator(task_id="start")
38     end = EmptyOperator(task_id="end")
39
40     ets_branch_1 = ExternalTaskSensor(
41         task_id="ets_branch_1",
42         external_dag_id='upstream_dag_1',
43         external_task_id='my_task',
44         allowed_states=['success'],
45         failed_states=['failed', 'skipped']
46     )
47
48     task_branch_1 = PythonOperator(
49         task_id='task_branch_1',
50         python_callable=downstream_function_branch_1,
51     )
52
53     ets_branch_2 = ExternalTaskSensor(
54         task_id="ets_branch_2",
55         external_dag_id='upstream_dag_2',
56         external_task_id='my_task',
57         allowed_states=['success'],
58         failed_states=['failed', 'skipped']
59     )
60
61     task_branch_2 = PythonOperator(
62         task_id='task_branch_2',
63         python_callable=downstream_function_branch_2,
64     )
65
65     ets_branch_3 = ExternalTaskSensor(

```

```

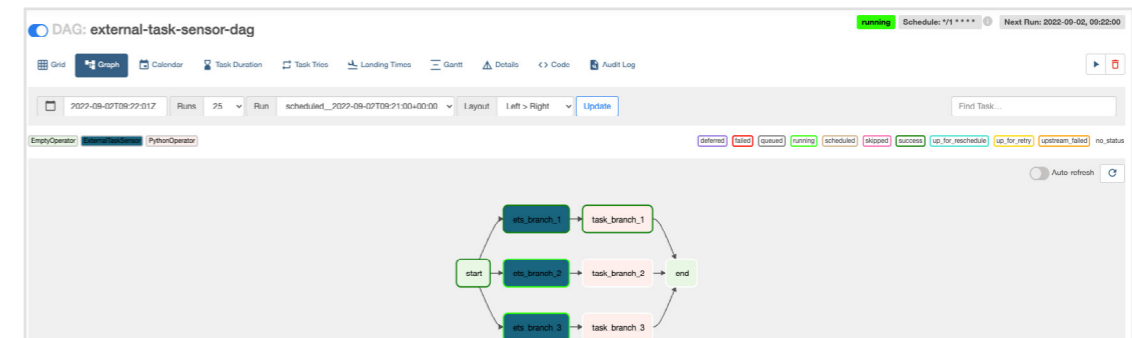
67     task_id="ets_branch_3",
68     external_dag_id='upstream_dag_3',
69     external_task_id='my_task',
70     allowed_states=['success'],
71     failed_states=['failed', 'skipped']
72 )
73
74 task_branch_3 = PythonOperator(
75     task_id='task_branch_3',
76     python_callable=downstream_function_branch_3,
77 )
78
79
80
81 start >> [ets_branch_1, ets_branch_2, ets_branch_3]
82
83 ets_branch_1 >> task_branch_1
84 ets_branch_2 >> task_branch_2
85 ets_branch_3 >> task_branch_3
86
87 [task_branch_1, task_branch_2, task_branch_3] >> end

```

In this DAG

- `ets_branch_1` waits for the `my_task` task of `upstream_dag_1` to complete before moving on to execute `task_branch_1`.
- `ets_branch_2` waits for the `my_task` task of `upstream_dag_2` to complete before moving on to execute `task_branch_2`.
- `ets_branch_3` waits for the `my_task` task of `upstream_dag_3` to complete before moving on to execute `task_branch_3`.

These processes happen in parallel and independent of each other. The graph view shows the state of the DAG after `my_task` in `upstream_dag_1` has finished which caused `ets_branch_1` and `task_branch_1` to run. `ets_branch_2` and `ets_branch_3` are still waiting for their upstream tasks to finish.



If you want the downstream DAG to wait for the entire upstream DAG to finish instead of a specific task, you can set the `external_task_id` to `None`. In the example above, we specify that the external task must have a state of `success` for the downstream task to succeed, as defined by the `allowed_states` and `failed_states`.

Also note that in the example above, the upstream DAG (`example_dag`) and downstream DAG (`external-task-sensor-dag`) must have the same start date and schedule interval. This is because the ExternalTaskSensor will look for completion of the specified task or DAG at the same `logical_date` (previously called `execution_date`). To look for completion of the external task at a different date, you can make use of either of the `execution_delta` or `execution_date_fn` parameters (these are described in more detail in the documentation linked above).

Airflow API

The [Airflow API](#) is another way of creating cross-DAG dependencies. This is especially useful in [Airflow 2.0](#), which has a fully stable REST API. To use the API to trigger a DAG run, you can make a POST request to the `DAGRuns` endpoint as described in the [Airflow API documentation](#).

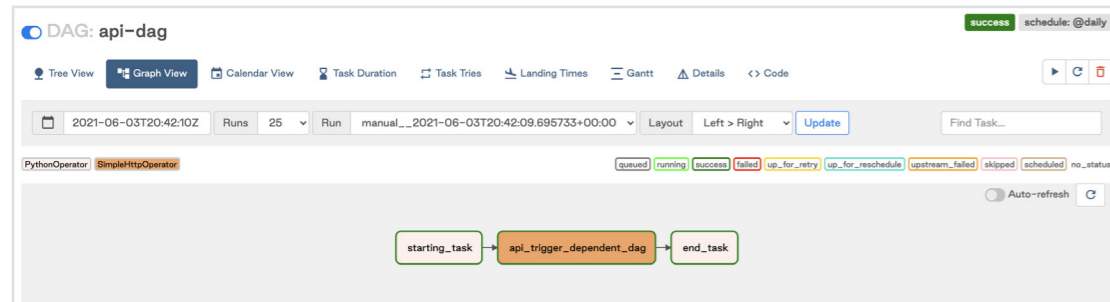
This method is useful if your dependent DAGs live in different Airflow environments (more on this in the Cross-Deployment Dependencies section below). The task triggering the downstream DAG will complete once the API call is complete.

Using the API to trigger a downstream DAG can be implemented within a DAG by using the [SimpleHttpOperator](#) as shown in the example DAG below:

```
1 from airflow import DAG
2 from airflow.operators.python import PythonOperator
3 from airflow.providers.http.operators.http import Simple-
4 HttpOperator
5 from datetime import datetime, timedelta
6 import json
7
8 # Define body of POST request for the API call to trigger
9 another DAG
10 date = '{{ execution_date }}'
11 request_body = {
12     "execution_date": date
13 }
14 json_body = json.dumps(request_body)
15
16 def print_task_type(**kwargs):
17     """
18     Dummy function to call before and after downstream DAG.
19     """
20     print(f"The {kwargs['task_type']} task has completed.")
21     print(request_body)
22
23 default_args = {
24     'owner': 'airflow',
25     'depends_on_past': False,
26     'email_on_failure': False,
27     'email_on_retry': False,
28     'retries': 1,
29     'retry_delay': timedelta(minutes=5)
30 }
31
32
```

```
1 with DAG(
2     'api-dag',
3     start_date=datetime(2021, 1, 1),
4     max_active_runs=1,
5     schedule_interval='@daily',
6     catchup=False
7 ) as dag:
8
9     start_task = PythonOperator(
10         task_id='starting_task',
11         python_callable=print_task_type,
12         op_kwargs={'task_type': 'starting'}
13     )
14
15     api_trigger_dependent_dag = SimpleHttpOperator(
16         task_id="api_trigger_dependent_dag",
17         http_conn_id='airflow-api',
18         endpoint='/api/v1/dags/dependent-dag/dagRuns',
19         method='POST',
20         headers={'Content-Type': 'application/json'},
21         data=json_body
22     )
23
24     end_task = PythonOperator(
25         task_id='end_task',
26         python_callable=print_task_type,
27         op_kwargs={'task_type': 'ending'}
28     )
29
30 start_task >> api_trigger_dependent_dag >> end_task
```

This DAG has a similar structure to the TriggerDagRunOperator DAG above, but instead uses the SimpleHttpOperator to trigger the `dependent-dag` using the Airflow API. The graph view looks like this:



In order to use the SimpleHttpOperator to trigger another DAG, you need to define the following:

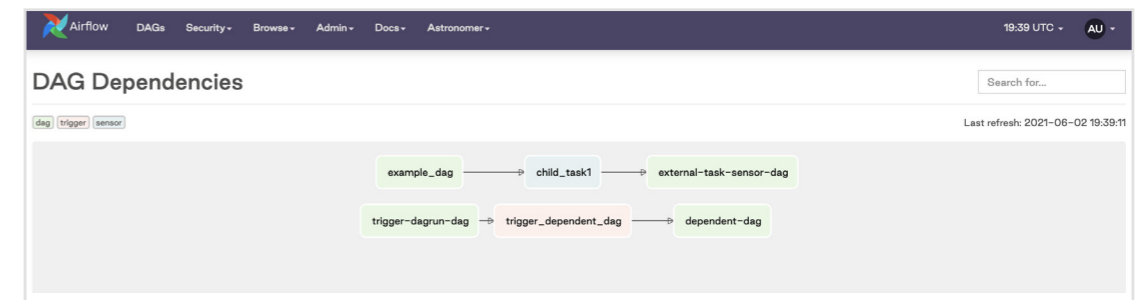
- **endpoint:** This should be of the form `"/api/v1/dags/<dag-id>/dagRuns"` where `<dag-id>` is the ID of the DAG you want to trigger.
- **data:** To trigger a DAG run using this endpoint, you must provide an execution date. In the example above, we use the `execution_date` of the upstream DAG, but this can be any date of your choosing. You can also specify other information about the DAG run as described in the API documentation linked above.
- **http_conn_id:** This should be an [Airflow connection](#) of **type HTTP**, with your Airflow domain as the Host. Any authentication should be provided either as a Login/Password (if using Basic auth) or as a JSON-formatted Extra. In the example below, we use an authorization token.

DAG Dependencies View

In [Airflow 2.1](#), a new cross-DAG dependencies view was added to the Airflow UI. This view shows all dependencies between DAGs in your Airflow environment as long as they are implemented using one of the following methods:

- **Using dataset driven scheduling**
- **Using a TriggerDagRunOperator**
- **Using an ExternalTaskSensor**

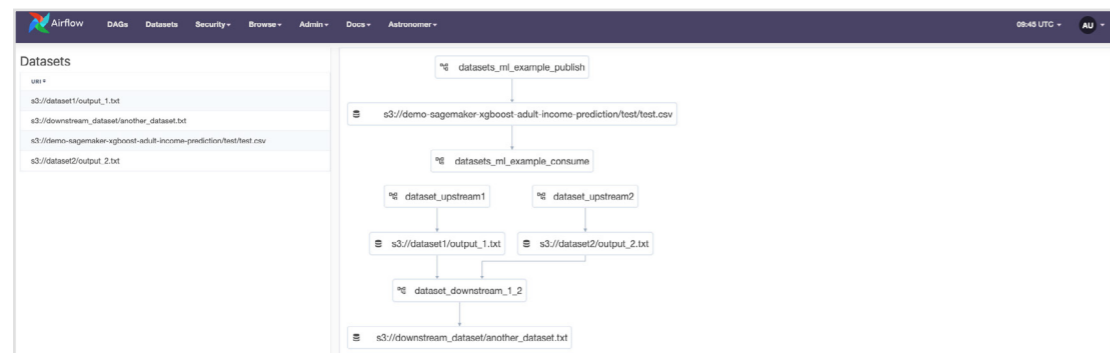
Dependencies can be viewed in the UI by going to **Browse » DAG Dependencies** or by clicking on the **Graph** button from within the Datasets tab. The screenshot below shows the dependencies created by the TriggerDagRunOperator and ExternalTaskSensor example DAGs in the sections above.



When DAGs are scheduled depending on datasets, both the DAG containing the producing task, as well as the dataset itself will be shown upstream of the consuming DAG.



In Airflow 2.4 an additional **Datasets** tab was added, which shows all dependencies between datasets and DAGs.



Cross-Deployment Dependencies

It is sometimes necessary to implement cross-DAG dependencies where the DAGs do not exist in the same Airflow deployment. The TriggerDagRunOperator, ExternalTaskSensor, and dataset methods described above are designed to work with DAGs in the same Airflow environment, so they are not ideal for cross-Airflow deployments. The Airflow API, on the other hand, is perfect for this use case. In this section, we'll focus on how to implement this method on Astro, but the general concepts will likely be similar wherever your Airflow environments are deployed.

Cross-Deployment Dependencies with Astronomer

To implement cross-DAG dependencies on two different Airflow environments on Astro, we can follow the same general steps for triggering a DAG using the Airflow API described above. It may be helpful to first read our documentation on [making requests to the Airflow API](#) from Astronomer. When you're ready to implement a cross-deployment dependency, follow these steps:

1. In the upstream DAG, create a SimpleHttpOperator task that will trigger the downstream DAG. Refer to the section above for details on configuring the operator.
2. In the downstream DAG Airflow environment, [create a Service Account](#) and copy the API key.
3. In the upstream DAG Airflow environment, create an Airflow connection as shown in the Airflow API section above. The Host should be `https://<your-base-domain>/<deployment-release-name>/airflow` where the base domain and deployment release name are from your downstream DAG's Airflow deployment. In the Extras, use `{"Authorization": "api-token"}` where `api-token` is the service account API key you copied in step 2.
4. Ensure the downstream DAG is turned on, then run the upstream DAG.

4. Dynamically Generating DAGs in Airflow

Overview

In Airflow, **DAGs** are defined as Python code. Airflow executes all Python code in the **DAG_FOLDER** and loads any **DAG** objects that appear in `globals()`. The simplest way of creating a DAG is to write it as a static Python file.

However, sometimes manually writing DAGs isn't practical. Maybe you have hundreds or thousands of DAGs that do similar things with just a parameter changing between them. Or perhaps you need a set of DAGs to load tables but don't want to manually update DAGs every time those tables change. In these cases and others, it can make more sense to generate DAGs dynamically.

Because everything in Airflow is code, you can dynamically generate DAGs using Python alone. As long as a **DAG** object in `globals()` is created by Python code that lives in the **DAG_FOLDER**, Airflow will load it. In this section, we will cover a few of the many ways of generating DAGs. We will also discuss when DAG generation is a good option and some pitfalls to watch out for when doing this at scale.

Single-File Methods

One method for dynamically generating DAGs is to have a single Python file that generates DAGs based on some input parameter(s) (e.g., a list of APIs or tables). An everyday use case for this is an ETL or ELT-type pipeline with many data sources or destinations. It would require creating many DAGs that all follow a similar pattern.

Some benefits of the single-file method:

- + It's simple and easy to implement.
- + It can accommodate input parameters from many different sources (see a few examples below).
- + Adding DAGs is nearly instantaneous since it requires only changing the input parameters.

However, there are also drawbacks:

- ✗ Since a DAG file isn't actually being created, your visibility into the code behind any specific DAG is limited.
- ✗ Since this method requires a Python file in the **DAG_FOLDER**, the generation code will be executed on every Scheduler heartbeat. It can cause performance issues if the total number of DAGs is large or if the code is connecting to an external system such as a database. For more on this, see the Scalability section below.

In the following examples, the single-file method is implemented differently based on which input parameters are used for generating DAGs.

EXAMPLE:

Use a Create_DAG Method

To dynamically create DAGs from a file, we need to define a Python function that will generate the DAGs based on an input parameter. In this case, we are going to define a DAG template within a `create_dag` function. The code here is very similar to what you would use when creating a single DAG, but it is wrapped in a method that allows for custom parameters to be passed in.

```
1 from airflow import DAG
2 from airflow.operators.python_operator import PythonOperator
3
4 from datetime import datetime
5
6
7 def create_dag(dag_id,
8               schedule,
9               dag_number,
10              default_args):
11
12     def hello_world_py(*args):
13         print('Hello World')
14         print('This is DAG: {}'.format(str(dag_number)))
15
16     dag = DAG(dag_id,
17              schedule_interval=schedule,
18              default_args=default_args)
19     with dag:
20         t1 = PythonOperator(
```

```
21         task_id='hello_world',
22         python_callable=hello_world_py,
23         dag_number=dag_number)
24
25     return dag              schedule,
26                             dag_number,
27                             default_args):
```

In this example, the input parameters can come from any source that the Python script can access. We can then set a simple loop (`range(1, 4)`) to generate these unique parameters and pass them to the global scope, thereby registering them as valid DAGs within the Airflow scheduler:

```
1 from airflow import DAG
2 from airflow.operators.python_operator import PythonOperator
3
4 from datetime import datetime
5
6
7 def create_dag(dag_id,
8               schedule,
9               dag_number,
10              default_args):
11
12     def hello_world_py(*args):
13         print('Hello World')
14         print('This is DAG: {}'.format(str(dag_number)))
15
16     dag = DAG(dag_id,
17              schedule_interval=schedule,
18              default_args=default_args)
```

```

1     dag = DAG(dag_id,
2     schedule_interval=schedule,
3     default_args=default_args)
4
5     with dag:
6         t1 = PythonOperator(
7             task_id='hello_world',
8             python_callable=hello_world_py)
9
10    return dag
11
12
13    # build a dag for each number in range(10)
14    for n in range(1, 4):
15        dag_id = 'loop_hello_world_{}'.format(str(n))
16
17        default_args = {'owner': 'airflow',
18                        'start_date': datetime(2021, 1, 1)}
19    }

```

And if we look at the Airflow UI, we can see the DAGs have been created.

Success!

The screenshot shows the Airflow web interface. At the top, there is a navigation bar with the Airflow logo and menu items: DAGs, Security, Browse, Admin, Docs, and Astronomer. Below the navigation bar, the main heading is "DAGs". There are three tabs: "All 4", "Active 1", and "Paused 3". Below the tabs is a table with the following structure:

DAG	Owner
<input type="checkbox"/> loop_hello_world_1	airflow
<input type="checkbox"/> loop_hello_world_2	airflow
<input type="checkbox"/> loop_hello_world_3	airflow

EXAMPLE:

Generate DAGs from Variables

As previously mentioned, the input parameters don't have to exist in the DAG file itself. Another common form of generating DAGs is by setting values in a Variable object.

The screenshot shows the "List Variable" page in the Airflow UI. At the top, there is a navigation bar with the Airflow logo and menu items: DAGs, Security, Browse, Admin, Docs, and Astronomer. Below the navigation bar, there is a "Choose file" button and a "No file chosen" message. To the right, there is a blue button labeled "Import Variables". Below this is a "List Variable" section with a search bar and a table of variables.

Key	Val
<input type="checkbox"/> dag_number	10

We can retrieve this value by importing the Variable class and passing it into our `range`. We want the interpreter to register this file as valid – regardless of whether the variable exists, the `default_var` is set to 3.

```
1 from airflow import DAG
2 from airflow.models import Variable
3 from airflow.operators.python_operator import PythonOperator
4
5 from datetime import datetime
6
7
8 def create_dag(dag_id,
9               schedule,
10              dag_number,
11              default_args):
12
13     def hello_world_py(*args):
14         print('Hello World')
15         print('This is DAG: {}'.format(str(dag_number)))
16
17     dag = DAG(dag_id,
18              schedule_interval=schedule,
19              default_args=default_args)
20
21     with dag:
22         t1 = PythonOperator(
23             task_id='hello_world',
24             python_callable=hello_world_py)
25
26     return dag
27
28 number_of_dags = Variable.get('dag_number', default_var=3)
29 number_of_dags = int(number_of_dags)
```

```
29 for n in range(1, number_of_dags):
30     dag_id = 'hello_world_{}'.format(str(n))
31
32     default_args = {'owner': 'airflow',
33                   'start_date': datetime(2021, 1, 1)}
34
35
36     schedule = '@daily'
37     dag_number = n
38     globals()[dag_id] = create_dag(dag_id,
39                                   schedule,
40                                   dag_number,
41                                   default_args)
```

If we look at the scheduler logs, we can see this variable was pulled into the DAG and, and 15 DAGs were added to the DagBag based on its value.

```
04 04:06:06,279] {jobs.py:1754} INFO - DAG(s) dict_keys(['hello_world_1', 'hello_world_2', 'hello_world_3', 'hello_world_4', 'hello_world_5', 'hello_world_6', 'hello_world_7', 'hello_world_8', 'hello_world_9', 'hello_world_10', 'hello_world_11', 'hello_world_12', 'hello_world_13', 'hello_world_14']) retrieved from /usr/local/airflow/dags/example
```

We can then go to the Airflow UI and see all of the new DAGs that have been created.

DAG	Owner	Runs	Schedule
hello_world_1	airflow	○○○	@daily
hello_world_2	airflow	○○○	@daily
hello_world_3	airflow	○○○	@daily
hello_world_4	airflow	○○○	@daily
hello_world_5	airflow	○○○	@daily
hello_world_6	airflow	○○○	@daily
hello_world_7	airflow	○○○	@daily
hello_world_8	airflow	○○○	@daily
hello_world_9	airflow	○○○	@daily

Conn Id	Conn Type
MY_DATABASE_CONN_1	mysql
MY_DATABASE_CONN_2	mysql
MY_DATABASE_CONN_3	mysql
MY_DATABASE_CONN_4	mysql

EXAMPLE:

Generate DAGs from Connections

Another way to define input parameters for dynamically generating DAGs is by defining Airflow connections. It can be a good option if each of your DAGs connects to a database or an API. Because you will be setting up those connections anyway, creating the DAGs from that source avoids redundant work.

To implement this method, we can pull the connections we have in our Airflow metadata database by instantiating the “Session” and querying the “Connection” table. We can also filter this query so that it only pulls connections that match specific criteria.

```

1  from airflow import DAG, settings
2  from airflow.models import Connection
3  from airflow.operators.python_operator import PythonOperator
4  from datetime import datetime
5
6
7  def create_dag(dag_id,
8                schedule,
9                dag_number,
10               default_args):
11
12     def hello_world_py(*args):
13         print('Hello World')
14         print('This is DAG: {}'.format(str(dag_number)))
15
16     dag = DAG(dag_id,
17               schedule_interval=schedule

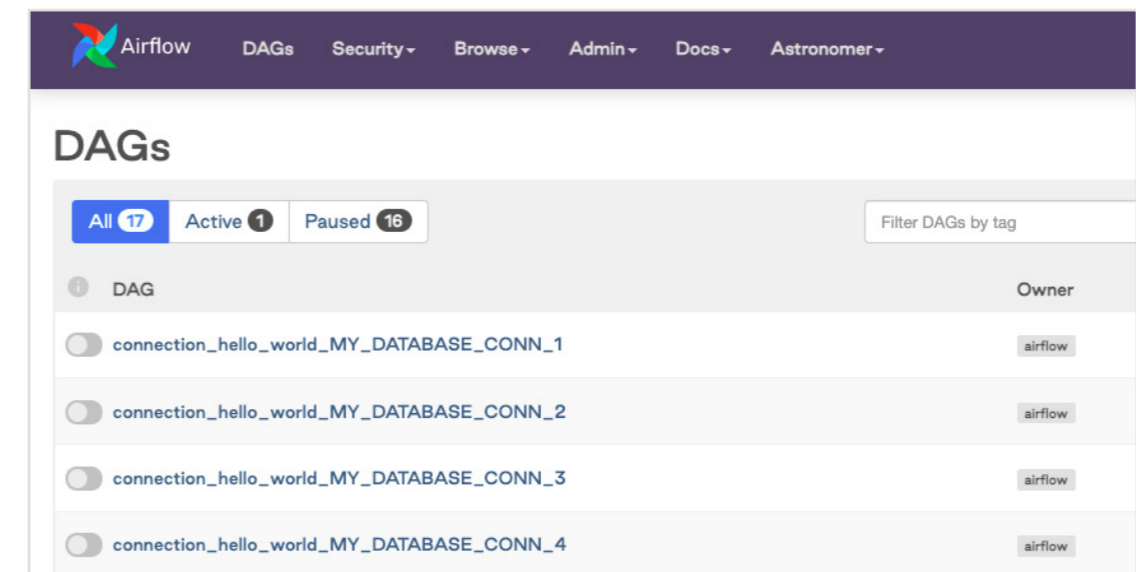
```

```

18         default_args=default_args)
19
20     with dag:
21         t1 = PythonOperator(
22             task_id='hello_world',
23             python_callable=hello_world_py)
24
25     return dag
26
27
28 session = settings.Session()
29 conns = (session.query(Connection.conn_id)
30         .filter(Connection.conn_id.ilike('%MY_DATABASE_
31 CONN%')))
32         .all())
33
34 for conn in conns:
35     dag_id = 'connection_hello_world_{}'.format(conn[0])
36
37     default_args = {'owner': 'airflow',
38                   'start_date': datetime(2018, 1, 1)
39                   }
40
41     schedule = '@daily'
42     dag_number = conn
43
44     globals()[dag_id] = create_dag(dag_id,
45                                   schedule,
46                                   dag_number,
47                                   default_args)

```

Notice that, as before, we access the Models library to bring in the `Connection` class (as we did previously with the `Variable` class). We are also accessing the `Session()` class from `settings`, which will allow us to query the current database session.



We can see that all of the connections that match our filter have now been created as a unique DAG. The one connection we had which did not match (`SOME_OTHER_DATABASE`) has been ignored.

Multiple-File Methods

Another method for dynamically generating DAGs is to use code to create full Python files for each DAG. The end result of this method is having one Python file per generated DAG in your `DAG_FOLDER`.

One way of implementing this method in production is to have a Python script that, when executed, generates DAG files as part of a CI/CD workflow. The DAGs are generated during the CI/CD build and then deployed to Airflow. You could also have another DAG that runs the generation script periodically.

Some benefits of this method:

- + It's more scalable than single-file methods. Because the DAG files aren't generated by parsing code in the `DAG_FOLDER`, the DAG generation code isn't executed on every scheduler heartbeat.
- + Since DAG files are being explicitly created before deploying to Airflow, you have a full visibility into the DAG code.

On the other hand, this method includes drawbacks:

- ✗ It can be complex to set up.
- ✗ Changes to DAGs or additional DAGs won't be generated until the script is run, which in some cases requires deployment.

Let's see a simple example of how this method could be implemented.

EXAMPLE:

Generate DAGs from JSON Config Files

One way of implementing a multiple-file method is using a Python script to generate DAG files based on a set of JSON configuration files. For this simple example, we will assume that all DAGs have the same structure: each has a single task that uses the `PostgresOperator` to execute a query. This use case might be relevant for a team of analysts who need to schedule SQL queries, where the DAG is mostly the same, but the query and the schedule are changing.

To start, we create a DAG 'template' file that defines the DAG's structure. It looks like a regular DAG file, but we have added specific variables where we know information will be dynamically generated, namely the `dag_id`, `schedulereplace`, and `queryreplace`.

```
1 from airflow import DAG
2 from airflow.operators.postgres_operator import PostgresOperator
3
4 from datetime import datetime
5
6 default_args = {'owner': 'airflow',
7                 'start_date': datetime(2021, 1, 1)}
8
9
10 dag = DAG(dag_id,
11           schedule_interval=schedulereplace,
12           default_args=default_args,
13           catchup=False)
```



```

1 with dag:
2     t1 = PostgresOperator(
3         task_id='postgres_query',
4         postgres_conn_id=connection_id
5         sql=querytoreplace)

```

Next, we create a `dag-config` folder that will contain a JSON config file for each DAG. The config file should define the parameters that we noted above, the DAG ID, schedule interval, and query to be executed.

```

1 {
2     "DagId": "dag_file_1",
3     "Schedule": "@daily",
4     "Query": "SELECT * FROM table1;"
5 }

```

Finally, we write a Python script to create the DAG files based on the template and the config files. The script loops through every config file in the `dag-config/` folder, makes a copy of the template in the `dags/` folder and overwrites the parameters in that file (including the parameters from the config file).

```

1 import json
2 import os
3 import shutil
4 import fileinput
5
6 config_filepath = 'include/dag-config/'
7 dag_template_filename = 'include/dag-template.py'

```

```

8 for filename in os.listdir(config_filepath):
9     f = open(filepath + filename)
10    config = json.load(f)
11
12    new_filename = 'dags/'+config['DagId']+'.py'
13    shutil.copyfile(dag_template_filename, new_filename)
14
15
16    for line in fileinput.input(new_filename, in-
17    place=True):
18        line.replace("dag_id", ""+config['DagId']+""")
19        line.replace("scheduletoreplace", config['Schedule'])
20    line.replace("querytoreplace", config['Query'])
21    print(line, end="")
22

```

To generate our DAG files, we either run this script ad-hoc as part of our CI/CD workflow, or we create another DAG that would run it periodically. After running the script, our final directory would look like the example below, where the `include/` directory contains the files shown above, and the `dags/` directory contain the two dynamically generated DAGs:

```

1 dags/
2 |— dag_file_1.py
3 |— dag_file_2.py
4 include/
5 |— dag-template.py
6 |— generate-dag-files.py
7 |— dag-config
8   |— dag1-config.json
9   |— dag2-config.json

```

This is obviously a simple starting example that works only if all DAGs follow the same pattern. However, it could be expanded upon to have dynamic inputs for tasks, dependencies, different operators, etc.

DAG Factory

A notable tool for dynamically creating DAGs from the community is [dag-factory](#). `dag-factory` is an open-source Python library for dynamically generating Airflow DAGs from YAML files.

To use `dag-factory`, you can install the package in your Airflow environment and create YAML configuration files for generating your DAGs. You can then build the DAGs by calling the `dag-factory.generate_dags()` method in a Python script, like this example from the `dag-factory` README:

```
1 from airflow import DAG
2 import dagfactory
3
4 dag_factory = dagfactory.DagFactory("/path/to/dags/config_
5 file.yml")
6
7 dag_factory.clean_dags(globals())
8 dag_factory.generate_dags(globals())
```

Scalability

Dynamically generating DAGs can cause performance issues when used at scale. Whether or not any particular method will cause problems is dependent on your total number of DAGs, your Airflow configuration, and your infrastructure. Here are a few general things to look out for:

- Any code in the `DAG_FOLDER` will run on every Scheduler heartbeat. Methods where that code dynamically generates DAGs, such as the single-file method, are more likely to cause performance issues at scale.
- If the DAG parsing time (i.e., the time to parse all code in the `DAG_FOLDER`) is greater than the Scheduler heartbeat interval, the scheduler can get locked up, and tasks won't get executed. If you are dynamically generating DAGs and tasks aren't running, this is a good metric to review in the beginning of troubleshooting.

Upgrading to Airflow 2.0 to make use of the [_HA Scheduler](#) should help with these performance issues. But it can still take some additional optimization work depending on the scale you're working at. There is no single right way to implement or scale dynamically generated DAGs. Still, the flexibility of Airflow means there are many ways to arrive at a solution that works for a particular use case.

5. Testing Airflow DAGs

Overview

One of the core principles of Airflow is that your DAGs are defined as Python code. Because you can treat data pipelines like you would any other piece of code, you can integrate them into a standard software development lifecycle using source control, CI/CD, and automated testing.

Although DAGs are 100% Python code, effectively testing DAGs requires accounting for their unique structure and relationship to other code and data in your environment. This guide will discuss a couple of types of tests that we would recommend to anybody running Airflow in production, including DAG validation testing, unit testing, and data and pipeline integrity testing.

Before you begin

If you are newer to test-driven development, or CI/CD in general, we'd recommend the following resources to get started:



TUTORIAL

Getting Started With Testing in Python

[SEE TUTORIAL →](#)



TUTORIAL

Continuous Integration With Python: An Introduction

[SEE TUTORIAL →](#)



ARTICLE

The Challenge of Testing Data Pipelines

[SEE ARTICLE →](#)



DOCUMENTATION

Deploying to Astronomer via CI/CD

[SEE DOCUMENTATION →](#)

We also recommend checking out [Airflow's documentation on testing DAGs](#) and [testing guidelines for contributors](#); we will walk through some of the concepts covered in those docs in more detail below.

Note on test runners: Before we dive into different types of tests for Airflow, we have a quick note on test runners. There are multiple test runners available for Python, including `unittest`, `pytest`, and `nose2`. The OSS Airflow project uses `pytest`, so we will do the same in this section. However, Airflow doesn't require using a specific test runner. In general, choosing a test runner is a matter of personal preference and experience level, and some test runners might work better than others for a given use case.

DAG Validation Testing

DAG validation tests are designed to ensure that your DAG objects are defined correctly, acyclic, and free from import errors.

These are things that you would likely catch if you were starting with the local development of your DAGs. But in cases where you may not have access to a local Airflow environment or want an extra layer of security, these tests can ensure that simple coding errors don't get deployed and slow down your development.

DAG validation tests apply to all DAGs in your Airflow environment, so you only need to create one test suite.

To test whether your DAG can be loaded, meaning there aren't any syntax errors, you can run the Python file:

```
1 python your-dag-file.py
```

Or to test for import errors specifically (which might be syntax related but could also be due to incorrect package import paths, etc.), you can use something like the following:

```
1 import pytest
2 from airflow.models import DagBag
3
4 def test_no_import_errors():
5     dag_bag = DagBag()
6     assert len(dag_bag.import_errors) == 0, "No Import Fail-
7     ures"
```

You may also use DAG validation tests to test for properties that you want to be consistent across all DAGs. For example, if your team has a rule that all DAGs must have two retries for each task, you might write a test like this to enforce that rule:

```
1 def test_retries_present():
2     dag_bag = DagBag()
3     for dag in dag_bag.dags:
4         retries = dag_bag.dags[dag].default_args.get('re-
5         tries', [])
6         error_msg = 'Retries not set to 2 for DAG {id}'.for-
7         mat(id=dag)
8         assert retries == 2, error_msg
```

To see an example of running these tests as part of a CI/CD workflow, check out [this repo](#), which uses GitHub Actions to run the test suite before deploying the project to an Astronomer Airflow deployment.

Unit Testing

Unit testing is a software testing method where small chunks of source code are tested individually to ensure they function as intended. The goal is to isolate testable logic inside of small, well-named functions, for example:

```
1 def test_function_returns_5():
2     assert my_function(input) == 5
```

In the context of Airflow, you can write unit tests for any part of your DAG, but they are most frequently applied to hooks and operators. All official Airflow hooks, operators, and provider packages have unit tests that must pass before merging the code into the project. For an example, check out the [AWS S3Hook](#), which has many accompanying [unit tests](#).

If you have your custom hooks or operators, we highly recommend using unit tests to check logic and functionality. For example, say we have a custom operator that checks if a number is even:

```
1 from airflow.models import BaseOperator
2 from airflow.utils.decorators import apply_defaults
3
4 class EvenNumberCheckOperator(BaseOperator):
5     @apply_defaults
6     def __init__(self, my_operator_param, *args,
7 **kwargs):
8         self.operator_param = my_operator_param
9         super(EvenNumberCheckOperator, self).__init__(*args,
10 **kwargs)
11     def execute(self, context):
12         if self.operator_param % 2:
13             return True
14         else:
15             return False
```

We would then write a `test_evencheckoperator.py` file with unit tests like the following:

```
1 import unittest
2 import pytest
3 from datetime import datetime
4 from airflow import DAG
5 from airflow.models import TaskInstance
6 from airflow.operators import EvenNumberCheckOperator
7
8 DEFAULT_DATE = datetime(2021, 1, 1)
9
10 class EvenNumberCheckOperator(unittest.TestCase):
11
12     def setUp(self):
13         super().setUp()
14         self.dag = DAG('test_dag', default_args={'owner':
15 'airflow', 'start_date': DEFAULT_DATE})
16         self.even = 10
17         self.odd = 11
18
19     def test_even(self):
20         """Tests that the EvenNumberCheckOperator returns True for
21 10."""
22         task = EvenNumberCheckOperator(my_operator_param=-
23 self.even, task_id='even', dag=self.dag)
24         ti = TaskInstance(task=task, execution_date=date-
25 time.now())
26         result = task.execute(ti.get_template_context())
27         assert result is True
28
29     def test_odd(self):
30         """Tests that the EvenNumberCheckOperator returns False
31  for 11."""
```

```

32     task = EvenNumberCheckOperator(my_operator_param==
33 self.odd, task_id='odd', dag=self.dag)
34     ti = TaskInstance(task=task, execution_date=date-
35 time.now())
36     result = task.execute(ti.get_template_context())
37     assert result is False
38

```

Note that if your DAGs contain PythonOperators that execute your Python functions, it is a good idea to write unit tests for those functions as well.

The most common way of implementing unit tests in production is to automate them as part of your CI/CD process. Your CI tool executes the tests and stops the deployment process if any errors occur.

Mocking

Sometimes unit tests require mocking: the imitation of an external system, dataset, or another object. For example, you might use mocking with an Airflow unit test if you are testing a connection but don't have access to the metadata database. Another example could be testing an operator that executes an external service through an API endpoint, but you don't want to wait for that service to run a simple test.

Many [Airflow tests](#) have examples of mocking. [This blog post](#) also has a helpful section on mocking Airflow that may help get started.

Data Integrity Testing

Data integrity tests are designed to prevent data quality issues from breaking your pipelines or negatively impacting downstream systems. These tests could also be used to ensure your DAG tasks produce the expected output when processing a given piece of data. They are somewhat different in scope than the code-related tests described in previous sections since your data is not static like a DAG.

One straightforward way of implementing data integrity tests is to build them directly into your DAGs. This allows you to use Airflow dependencies to manage any errant data in whatever way makes sense for your use case.

There are many ways you could integrate data checks into your DAG. One method worth calling out is using [Great Expectations](#) (GE), an open-source Python framework for data validations. You can make use of the [Great Expectations provider package](#) to easily integrate GE tasks into your DAGs. In practice, you might have something like the following DAG, which runs an Azure Data Factory pipeline that generates data then runs a GE check on the data before sending an email.

```

1  from airflow import DAG
2  from datetime import datetime, timedelta
3  from airflow.operators.email_operator import EmailOperator
4  from airflow.operators.python_operator import PythonOperator
5  tor
6  from airflow.providers.microsoft.azure.hooks.azure_data_
7  factory import AzureDataFactoryHook
8  from airflow.providers.microsoft.azure.hooks.wasb import
9  WasbHook
10 from great_expectations_provider.operators.great_expecta-
11 tions import GreatExpectationsOperator
12 #Get yesterday's date, in the correct format
13 yesterday_date = '{{ yesterday_ds_nodash }}'
14
15 #Define Great Expectations file paths
16 data_dir = '/usr/local/airflow/include/data/'
17 data_file_path = '/usr/local/airflow/include/data/'
18 ge_root_dir = '/usr/local/airflow/include/great_expectations'

```

```

19     #Make connection to ADF, and run pipeline with parame-
20     ter
21     hook = AzureDataFactoryHook('azure_data_factory_conn')
22     hook.run_pipeline(pipeline_name, parameters=params)
23
24     def get_azure_blob_files(blobname, output_filename):
25         '''Downloads file from Azure blob storage
26         '''
27         azure = WasbHook(wasb_conn_id='azure_blob')
28         azure.get_file(output_filename, container_
29         name='covid-data', blob_name=blobname)
30
31
32     default_args = {
33         'owner': 'airflow',
34         'depends_on_past': False,
35         'email_on_failure': False,
36         'email_on_retry': False,
37         'retries': 0,
38         'retry_delay': timedelta(minutes=5)
39     }
40
41     with DAG('adf_great_expectations',
42             start_date=datetime(2021, 1, 1),
43             max_active_runs=1,
44             schedule_interval='@daily',
45             default_args=default_args,
46             catchup=False
47
48             ) as dag:
49
50         run_pipeline = PythonOperator(
51             task_id='run_pipeline',
52             python_callable=run_adf_pipeline,
53             op_kwargs={'pipeline_name': 'pipeline1', 'date':

```

```

54     yesterday_date}
55         )
56
57         download_data = PythonOperator(
58             task_id='download_data',
59             python_callable=get_azure_blob_files,
60             op_kwargs={'blobname': 'or/'+ yesterday_date
61             +'.csv', 'output_filename': data_file_path+'or_'+yesterday_
62             date+'.csv'}
63         )
64
65         ge_check = GreatExpectationsOperator(
66             task_id='ge_checkpoint',
67             expectation_suite_name='azure.demo',
68             batch_kwargs={
69                 'path': data_file_path+'or_'+yesterday_
70             date+'.csv',
71                 'datasource': 'data__dir'
72             },
73             data_context_root_dir=ge_root_dir
74         )
75
76         send_email = EmailOperator(
77             task_id='send_email',
78             to='noreply@astronomer.io',
79             subject='Covid to S3 DAG',
80             send_email = EmailOperator(
81                 task_id='send_email',
82                 to='noreply@astronomer.io',
83                 subject='Covid to S3 DAG',
84
85                 html_content='<p>The great expectations checks passed success-
86                 fully. <p>'
87         )

```

If the GE check fails, any downstream tasks will be skipped. Implementing checkpoints like this allows you to either conditionally branch your pipeline to deal with data that doesn't meet your criteria or potentially skip all downstream tasks so problematic data won't be loaded into your data warehouse or fed to a model. For more information on conditional DAG design, check out the documentation on [Airflow Trigger Rules](#) and our guide on [branching in Airflow](#).

It's also worth noting that data integrity testing will work better at scale if you design your DAGs to load or process data incrementally. We talk more about incremental loading in our [Airflow Best Practices guide](#). Still, in short, processing smaller, incremental chunks of your data in each DAG Run ensures that any data quality issues have a limited blast radius and are easier to recover from.



WEBINAR

Testing Airflow to Bulletproof Your Code with Bas Harens

[SEE WEBINAR →](#)



DAG Authoring for Apache Airflow

The Astronomer Certification: DAG Authoring for Apache Airflow gives you the opportunity to challenge yourself and show the world your ability to create incredible data pipelines. And don't worry, we've also prepared a preparation course to give you the best chance of success!

Concepts Covered:

- Variables
- Pools
- Trigger Rules
- DAG Dependencies
- Idempotency
- Dynamic DAGs
- DAG Best Practices
- DAG Versioning and much more

[Get Certified](#)

6. Debugging DAGs

7 Common Errors to Check when Debugging Airflow DAGs

Apache Airflow is the industry standard for workflow orchestration. It's an incredibly flexible tool that powers mission-critical projects, from machine learning model training to traditional ETL at scale, for startups and Fortune 50 teams alike.

Airflow's breadth and extensibility, however, can make it challenging to adopt – especially for those looking for guidance beyond day-one operations. In an effort to provide best practices and expand on existing resources, our team at Astronomer has collected some of the most common issues we see Airflow users face.

Whether you're new to Airflow or an experienced user, check out this list of common errors and some corresponding fixes to consider.

Note: Following the [Airflow 2.0](#) release in December of 2020, the open-source project has addressed a significant number of pain points commonly reported by users running previous versions. We strongly encourage your team to upgrade to Airflow 2.x.

If your team is running Airflow 1 and would like help establishing a migration path, [reach out to us](#).

1. Your DAG Isn't Running at the Expected Time

You [wrote a new DAG](#) that needs to run every hour and you're ready to turn it on. You set an hourly interval beginning today at 2pm, setting a reminder to check back in a couple of hours. You hop on at 3:30pm to find that your DAG did in fact run, but your logs indicate that there was only one recorded execution at 2pm. Huh – what happened to the 3pm run?

Before you jump into debugging mode (you wouldn't be the first), rest assured that this is expected behavior. The functionality of the Airflow Scheduler can be counterintuitive, but you'll get the hang of it.

The two most important things to keep in mind about scheduling are:

- By design, an Airflow DAG will run at the end of its `schedule_interval`. Airflow operates in UTC by default.

Airflow's Schedule Interval

As stated above, an Airflow DAG will execute at the completion of its `schedule_interval`, which means one `schedule_interval` **AFTER the start date**. An hourly DAG, for example, will execute its 2:00 PM run when the clock strikes 3:00 PM. This happens because Airflow can't ensure that all of the data from 2:00 PM - 3:00 PM is present until the end of that hourly interval.

This quirk is specific to Apache Airflow, and it's important to remember — especially if you're using [default variables and macros](#). Thankfully, Airflow 2.2+ simplifies DAG scheduling with the introduction of the timetables!

Use Timetables for Simpler Scheduling

There are some data engineering use cases that are difficult or even impossible to address with Airflow's original scheduling method. Scheduling DAGs to skip holidays, run only at certain times, or otherwise run on varying intervals can cause major headaches if you're relying solely on cron jobs or timedeltas.

This is why Airflow 2.2 introduced [timetables](#) as the new default scheduling method. Essentially, `timetable` is a DAG-level parameter that you can set to a Python function that contains your execution schedule.

A timetable is significantly more customizable than a cron job or timedelta. You can program varying schedules, conditional logic, and more, directly within your DAG schedule. And because timetables are imported as Airflow plugins, you can use community-developed timetables to quickly — and literally — get your DAG up to speed.

We recommend using timetables as your de facto scheduling mechanism in Airflow 2.2+. You might be creating timetables without even knowing it: if you define a `schedule_interval`, Airflow 2.2+ will convert it to a timetable behind the scenes.

Airflow Time Zones

Airflow stores datetime information in UTC internally and in the database. This behavior is shared by many databases and APIs, but it's worth clarifying. You should not expect your DAG executions to correspond to your local time-zone. If you're based in US Pacific Time, a DAG run of 19:00 will correspond to 12:00 local time.

In recent releases, the community has added more time zone-aware features to the Airflow UI. For more information, refer to [Airflow documentation](#).

2. One of Your DAGs Isn't Running

If workflows on your Deployment are generally running smoothly but you find that one specific DAG isn't scheduling tasks or running at all, it might have something to do with how you set it to schedule.

Make Sure You Don't Have `datetime.now()` as Your `start_date`

It's intuitive to think that if you tell your DAG to start "now" that it'll execute immediately. But that's not how Airflow reads `datetime.now()`.

For a DAG to be executed, the `start_date` must be a time in the past, otherwise Airflow will assume that it's not yet ready to execute. When Airflow evaluates your DAG file, it interprets `datetime.now()` as the current timestamp (i.e. NOT a time in the past) and decides that it's not ready to run.

To properly trigger your DAG to run, make sure to insert a fixed time in the past and set `catchup=False` if you don't want to perform a backfill.

Note: You can manually trigger a DAG run via Airflow's UI directly on your dashboard (it looks like a "Play" button). A manual trigger executes immediately and will not interrupt regular scheduling, though it will be limited by any concurrency configurations you have at the deployment level, DAG level, or task level. When you look at corresponding logs, the run_id will show manual_ instead of scheduled_.

If your team is running Airflow 1 and would like help establishing a migration path, reach out to us.

3. You're Seeing a 503 Error on Your Deployment

If your Airflow UI is entirely inaccessible via web browser, you likely have a Webserver issue.

If you've already refreshed the page once or twice and continue to see a 503 error, read below for some Webserver-related guidelines.

Your Webserver Might Be Crashing

A 503 error might indicate an issue with your Deployment's Webserver, which is the Airflow component responsible for rendering task state and task execution logs in the [Airflow UI](#). If it's underpowered or otherwise experiencing an issue, you can expect it to affect UI loading time or web browser accessibility.

In our experience, a 503 often indicates that your Webserver is crashing. If you push up a deploy and your Webserver takes longer than a few seconds to start, it might hit a timeout period (10 secs by default) that "crashes" the Webserver before it has time to spin up. That triggers a retry, which crashes again, and so on and so forth.

If your Deployment is in this state, your Webserver might be hitting a memory limit when loading your DAGs even as your Scheduler and Worker(s) continue to schedule and execute tasks.

Increase Webserver Resources

If your Webserver is hitting the timeout limit, a bump in Webserver resources usually does the trick.

If you're using Astronomer, we generally recommend running the Webserver with a minimum of 5 AUs (Astronomer Units), which is equivalent to 0.5 CPUs and 1.88 GiB of memory. Even if you're not running anything particularly heavy, underprovisioning your Webserver will likely return some funky behavior.

Increase the Webserver Timeout Period

If bumping Webserver resources doesn't seem to have an effect, you might want to try increasing `web_server_master_timeout` or `web_server_worker_timeout`.

Raising those values will tell your Airflow Webserver to wait a bit longer to load before it hits you with a 503 (a timeout). You might still experience slow loading times if your Webserver is underpowered, but you'll likely avoid hitting a 503.

Avoid Making Requests Outside of an Operator

If you're making API calls, JSON requests, or database requests outside of an Airflow operator at a high frequency, your Webserver is much more likely to timeout.

When Airflow interprets a file to look for any valid DAGs, it first runs all code at the top level (i.e. outside of operators). Even if the operator itself only gets executed at execution time, everything outside of an operator is called every heartbeat, which can be very taxing on performance.

We'd recommend taking the logic you have currently running outside of an operator and moving it inside of a Python Operator if possible.

4. Sensor Tasks are Failing Intermittently

If your sensor tasks are failing, it might not be a problem with your task. It might be a problem with the sensor itself.

Be Careful When Using Sensors

By default, Airflow sensors run continuously and occupy a task slot in perpetuity until they find what they're looking for, often causing concurrency issues. Unless you never have more than a few tasks running concurrently, we recommend avoiding them unless you know it won't take too long for them to exit.

For example, if a worker can only run X number of tasks simultaneously and you have three sensors running, then you'll only be able to run X-3 tasks at any given point. Keep in mind that if you're running a sensor at all times, that limits how and when a scheduler restart can occur (or else it will fail the sensor).

Depending on your use case, we'd suggest considering the following:

- **Create a DAG that runs at a more frequent interval.**
- **Trigger a [Lambda function](#).**
- Set `mode='reschedule'`. If you have more sensors than worker slots, the sensor will now get thrown into an `up_for_reschedule` state, which frees up its worker slot.

Replace Sensors with Deferrable Operators

If you're running Airflow 2.2+, we recommend almost always using Deferrable Operators instead of sensors. These operators never use a worker slot when waiting for a condition to be met. Instead of using workers, deferrable operators poll for a status using a new Airflow component called the **triggerer**. Compared to using sensors, tasks with deferrable operators use a fraction of the resources to poll for a status.

As the Airflow community continues to adopt deferrable operators, the number of available deferrable operators is quickly growing. For more information on how to use deferrable operators, see our Deferrable Operators Guide.

5. Tasks are Executing Slowly

If your tasks are stuck in a bottleneck, we'd recommend taking a closer look at:

- **Environment variables and concurrency configurations**
Worker and Scheduler resources

Update Concurrency Settings

The potential root cause for a bottleneck is specific to your setup. For example, are you running many DAGs at once, or one DAG with hundreds of concurrent tasks?

Regardless of your use case, configuring a few settings as parameters or [environment variables](#) can help improve performance. Use this section to learn what those variables are and how to set them.

Most users can set parameters in Airflow's `airflow.cfg` file. If you're using Astro, you can also set environment variables [via the Astro UI or your project's Dockerfile](#). We've formatted these settings as parameters for readability – the environment variables for these settings are formatted as `AIRFLOW__CORE__PARAMETER_NAME`. For all default values, [refer here](#).

Parallelism

`parallelism` determines how many task instances can run in parallel across all DAGs given your environment resources. Think of this as “maximum active tasks anywhere.” To increase the limit of tasks set to run in parallel, set this value higher than its default of 32.

DAG Concurrency

`max_active_tasks_per_dag` (formerly `dag_concurrency`) determines how many task instances your Scheduler is able to schedule at once per DAG. Think of this as “maximum tasks that can be scheduled at once, per DAG.” The default is 16, but you should increase this if you're not noticing an improvement in performance after provisioning more resources to Airflow.

Max Active Runs per DAG

`max_active_runs_per_dag` determines the maximum number of active DAG runs per DAG. This setting is most relevant when backfilling, as all of your DAGs are immediately vying for a limited number of resources. The default value is 16.

Pro-tip: If you consider setting DAG or deployment-level concurrency configurations to a low number to protect against API rate limits, we'd recommend instead using “pools” – they'll allow you to limit parallelism at the task level and won't limit scheduling or execution outside of the tasks that need it.

Worker Concurrency

Defined as `AIRFLOW__CELERY__WORKER_CONCURRENCY=9`, `worker_concurrency` determines how many tasks each Celery Worker can run at any given time. The Celery Executor will run a max of 16 tasks concurrently by default. Think of this as “how many tasks each of my workers can take on at any given time.”

It's important to note that this number will naturally be limited by `dag_concurrency`. If you have 1 Worker and want it to match your Deployment's capacity, `worker_concurrency` should be equal to `parallelism`. The default value is 16.

Pro-tip: If you consider setting DAG or deployment-level concurrency configurations to a low number to protect against API rate limits, we'd recommend instead using “pools” – they'll allow you to limit parallelism at the task level and won't limit scheduling or execution outside of the tasks that need it.

Try Scaling Up Your Scheduler or Adding a Worker

If tasks are getting bottlenecked and your concurrency configurations are already optimized, the issue might be that your Scheduler is underpowered or that your Deployment could use another worker. If you're running on Astro, we generally recommend 5 AU (0.5 CPUs and 1.88 GiB of memory) as the default minimum for the Scheduler and 10 AU (1 CPUs and 3.76 GiB of memory) for workers.

Whether or not you scale your current resources or add an extra Celery Worker depends on your use case, but we generally recommend the following:

- If you're running a relatively high number of light tasks across DAGs and at a relatively high frequency, you're likely better off having 2 or 3 "light" workers to spread out the work.
- If you're running fewer but heavier tasks at a lower frequency, you're likely better off with a single but "heavier" worker that can more efficiently execute those tasks.

For more information on the differences between Executors, we recommend reading [Airflow Executors: Explained](#).

6. You're Missing Tasks Logs

Generally speaking, logs fail to show up because of a process that died on your Scheduler or one or more of your Celery Workers.

If you're missing logs, you might see something like this under "Log by attempts" in the Airflow UI:

```
Failed to fetch log file from worker. Invalid URL 'http://:8793/log/staging_to_presentation_pipeline_v5/redshift_to_s3_Order_Payment_17461/2019-01-11T00:00:00+00:00/1.log': No host supplied
```

A few things to try:

- Clear the task instance via the Airflow UI to see if logs show up. This will prompt your task to run again.
- Change the `log_fetch_timeout_sec` to something greater than 5 seconds. Defined in seconds, this setting determines the amount of time that the Webserver will wait for an initial handshake while fetching logs from other workers.
- Give your workers a little more power. If you're using Astro, you can do this in the **Configure** tab of the Astro UI.
- Are you looking for a log from over 15 days ago? If you're using Astro, the log retention period is an Environment Variable we have hard-coded on our platform. For now, you won't have access to logs over 15 days old.
- Exec into one of your Celery workers to look for the log files. If you're running Airflow on Kubernetes or Docker, you can use `kubectl` or Docker commands to `run $ kubectl exec -it {worker_name} bash`. Log files should be in `~/logs`. From there, they'll be split up by DAG/TASK/RUN.
- Try checking your Scheduler and Webserver logs to see if there are any errors that might tell you why your task logs are missing. If your tasks are slower than usual to get scheduled, you might need to update Scheduler settings to increase performance and optimize your environment.

7. Tasks are Slow to Schedule and/or Have Stopped Being Scheduled Altogether

If your tasks are slower than usual to get scheduled, you might need to update Scheduler settings to increase performance and optimize your environment.

Just like with concurrency settings, users can set parameters in Airflow's `airflow.cfg` file. If you're using Astro, you can also set environment variables via the Astro UI or your project's Dockerfile. We've formatted these settings as parameters for readability – the environment variables for these settings are formatted as `AIRFLOW__CORE__PARAMETER_NAME`. For all default values, refer [here](#).

- **`min_file_process_interval`**: The Scheduler parses your DAG files every `min_file_process_interval` number of seconds. Airflow starts using your update DAG code only after this interval ends. Because the Scheduler will parse your DAGs more often, setting this value to a low number will increase Scheduler CPU usage. If you have dynamic DAGs or otherwise complex code, you might want to increase this value to avoid poor Scheduler performance. By default, it's set to 30 seconds.
- **`dag_dir_list_interval`**: This setting determines how often Airflow should scan the DAGs directory in seconds. A lower value here means that new DAGs will be processed faster, but this comes at the cost of CPU usage. By default, this is set to 300 seconds (5 minutes). You might want to check how long it takes to parse your DAGs (`dag_processing.total_parse_time`) to know what value to choose for `dag_dir_list_interval`. If your `dag_dir_list_interval` is less than this value, then you might see performance issues.
- **`parsing_processes`**: (formerly `max_threads`) The Scheduler can run multiple processes in parallel to parse DAGs, and this setting determines how many of those processes can run in parallel. We recommend setting this to 2x your available vCPUs. Increasing this value can help to serialize DAGs if you have a large number of them. By default, this is set to 2.

Pro-tip: Scheduler performance was a critical part of the Airflow 2 release and has seen significant improvements since December of 2020. If you are experiencing Scheduler issues, we strongly recommend upgrading to Airflow 2.x. For more information, read our blog post: [The Airflow 2.0 Scheduler](#).



Ready for more?

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

[Discover Guides](#)

Error Notifications in Airflow

Overview

A key question when using any data orchestration tool is “How do I know if something has gone wrong?” Airflow users always have the option to check the UI to see the status of their DAGs, but this is an inefficient way of managing errors systematically, especially if certain failures need to be addressed promptly or by multiple team members. Fortunately, Airflow has built-in notification mechanisms that can be leveraged to configure error notifications in a way that works for your team.

In this section, we will cover the basics of Airflow notifications and how to set up common notification mechanisms including email, Slack, and SLAs. We will also discuss how to make the most of Airflow alerting when using the Astronomer platform.

Airflow Notification Basics

Airflow has an incredibly flexible notification system. Having your DAGs defined as Python code gives you full autonomy to define your tasks and notifications in whatever way makes sense for your use case.

In this section, we will cover some of the options available when working with notifications in Airflow.

Notification Levels

- Sometimes it makes sense to standardize notifications across your entire DAG. Notifications set at the DAG level will filter down to each task in the DAG. These notifications are usually defined in `default_args`.
- For example, in the following DAG, `email_on_failure` is set to `True`, meaning any task in this DAG’s context will send a failure `email` to all addresses in the email array.

```
1 from datetime import datetime
2 from airflow import DAG
3
4 default_args = {
5     'owner': 'airflow',
6     'start_date': datetime(2018, 1, 30),
7     'email': ['noreply@astronomer.io'],
8     'email_on_failure': True
9 }
10
11 with DAG('sample_dag',
12         default_args=default_args,
13         schedule_interval='@daily',
14         catchup=False) as dag:
15
16     ...
```

In contrast, it’s sometimes useful to have notifications only for certain tasks. The `BaseOperator` that all Airflow Operators inherit from has support for built-in notification arguments, so you can configure each task individually as needed. In the DAG below, email notifications are turned off by default at the DAG level but are specifically enabled for the `will_email` task.


```

1 from datetime import datetime
2 from airflow import DAG
3 from airflow.operators.dummy_operator import DummyOperator
4
5 default_args = {
6     'owner': 'airflow',
7     'start_date': datetime(2018, 1, 30),
8     'email_on_failure': False,
9     'email': ['noreply@astronomer.io'],
10    'retries': 1
11 }
12
13 with DAG('sample_dag',
14         default_args=default_args,
15         schedule_interval='@daily',
16         catchup=False) as dag:
17
18     wont_email = DummyOperator(
19         task_id='wont_email'
20     )
21
22     will_email = DummyOperator(
23         task_id='will_email',
24         email_on_failure=True
25     )

```

Notification Triggers

The most common trigger for notifications in Airflow is a task failure. However, notifications can be set based on other events, including retries and successes.

Emails on retries can be useful for debugging indirect failures; if a task needed to retry but eventually succeeded, this might indicate that the problem was caused by extraneous factors like a load on an external system. To turn on email notifications for retries, simply set the `email_on_retry` parameter to `True` as shown in the DAG below.

```

1 from datetime import datetime, timedelta
2 from airflow import DAG
3
4 default_args = {
5     'owner': 'airflow',
6     'start_date': datetime(2018, 1, 30),
7     'email': ['noreply@astronomer.io'],
8     'email_on_failure': True,
9     'email_on_retry': True,
10    'retry_exponential_backoff': True,
11    'retry_delay' = timedelta(seconds=300)
12    'retries': 3
13 }
14
15 with DAG('sample_dag',
16         default_args=default_args,
17         schedule_interval='@daily',
18         catchup=False) as dag:
19
20     ...

```

When working with retries, you should configure a `retry_delay`. This is the amount of time between a task failure and when the next try will begin. You can also turn on `retry_exponential_backoff`, which progressively increases the wait time between retries. This can be useful if you expect that extraneous factors might cause failures periodically.

Finally, you can also set any task to email on success by setting the `email_on_success` parameter to `True`. This is useful when your pipelines have conditional branching, and you want to be notified if a certain path is taken (i.e. certain tasks get run).

Custom Notifications

The email notification parameters shown in the sections above are an example of built-in Airflow alerting mechanisms. These simply have to be turned on and don't require any configuration from the user.

You can also define your own notifications to customize how Airflow alerts you about failures or successes. The most straightforward way of doing this is by defining `on_failure_callback` and `on_success_callback` Python functions. These functions can be set at the DAG or task level, and the functions will be called when a failure or success occurs respectively. For example, the following DAG has a custom `on_failure_callback` function set at the DAG level and an `on_success_callback` function for just the `success_task`.

```
1 from datetime import datetime
2 from airflow import DAG
3 from airflow.operators.dummy_operator import DummyOperator
4
5 def custom_failure_function(context):
6     "Define custom failure notification behavior"
```

```
1     dag_run = context.get('dag_run')
2     task_instances = dag_run.get_task_instances()
3     print("These task instances failed:", task_instances)
4
5 def custom_success_function(context):
6     "Define custom success notification behavior"
7     dag_run = context.get('dag_run')
8     task_instances = dag_run.get_task_instances()
9     print("These task instances succeeded:", task_instances)
10
11
12 default_args = {
13     'owner': 'airflow',
14     'start_date': datetime(2018, 1, 30),
15     'on_failure_callback': custom_failure_function
16     'retries': 1
17 }
18
19 with DAG('sample_dag',
20         default_args=default_args,
21         schedule_interval='@daily',
22         catchup=False) as dag:
23
24     failure_task = DummyOperator(
25         task_id='failure_task'
26     )
27
28     success_task = DummyOperator(
29         task_id='success_task',
30         on_success_callback=custom_success_function
31     )
```

Note that custom notification functions can be used in addition to email notifications.

Email Notifications

Email notifications are a native feature in Airflow and are easy to set up. As shown above, the `email_on_failure` and `email_on_retry` parameters can be set to `True` either at the DAG level or task level to send emails when tasks fail or retry. The `email` parameter can be used to specify which email(s) you want to receive the notification. If you want to enable email alerts on all failures and retries in your DAG, you can define that in your default arguments like this:

```
1 from datetime import datetime, timedelta
2 from airflow import DAG
3
4 default_args = {
5     'owner': 'airflow',
6     'start_date': datetime(2018, 1, 30),
7     'email': ['noreply@astronomer.io'],
8     'email_on_failure': True,
9     'email_on_retry': True,
10    'retry_delay' = timedelta(seconds=300)
11    'retries': 1
12 }
13
14 with DAG('sample_dag',
15         default_args=default_args,
16         schedule_interval='@daily',
17         catchup=False) as dag:
18
19     ...
```

In order for Airflow to send emails, you need to configure an SMTP server in your Airflow environment. You can do this by filling out the SMTP section of your `airflow.cfg` like this:

```
1 [smtp]
2 # If you want airflow to send emails on retries, failure,
3 # and you want to use
4 # the airflow.utils.email.send_email_smtp function, you
5 # have to configure an
6 # smtp server here
7 smtp_host = your-smtp-host.com
8 smtp_starttls = True
9 smtp_ssl = False
10 # Uncomment and set the user/pass settings if you want to
11 # use SMTP AUTH
12 # smtp_user =
13 # smtp_password =
14 smtp_port = 587
15 smtp_mail_from = noreply@astronomer.io
```

You can also set these values using environment variables. In this case, all parameters are preceded by `AIRFLOW__SMTP__`, consistent with Airflow environment variable naming convention. For example, `smtp_host` can be specified by setting the `AIRFLOW__SMTP__SMTP_HOST` variable. For more on Airflow email configuration, check out the [Airflow documentation](#).

Note: If you are running on the Astronomer platform, you can set up SMTP using environment variables since the `airflow.cfg` cannot be directly edited. For more on email alerting on the Astronomer platform, see the 'Notifications on Astronomer' section below.

Customizing Email Notifications

By default, email notifications will be sent in a standard format as defined in the `email_alert()` and `get_email_subject_content()` methods of the `TaskInstance` class. The default email content is defined like this:

```
1 default_subject = 'Airflow alert: {{ti}}'
2 # For reporting purposes, we report based on 1-indexed,
3 # not 0-indexed lists (i.e. Try 1 instead of
4 # Try 0 for the first attempt).
5 default_html_content = (
6     'Try {{try_number}} out of {{max_tries + 1}}<br>'
7     'Exception:<br>{{exception_html}}<br>'
8     'Log: <a href="{{ti.log_url}}">Link</a><br>'
9     'Host: {{ti.hostname}}<br>'
10    'Mark success: <a href="{{ti.mark_success_url}}">Link</
11    a><br>'
12 )
13
14 ...
```

To see the full method, check out the source code [here](#).

You can overwrite this default with your custom content by setting the `subject_template` and/or `html_content_template` variables in your `airflow.cfg` with the path to your jinja template files for subject and content respectively.

Slack Notifications

Sending notifications to Slack is another common way of alerting with Airflow.

There are multiple ways you can send messages to Slack from Airflow. In this section, we will cover how to use the [Slack Provider's `SlackWebhookOperator`](#) with a Slack Webhook to send messages, since this is Slack's recommended way of posting messages from apps. To get started, follow these steps:

- 1. From your Slack workspace, create a Slack app and an incoming Webhook.** The Slack documentation [here](#) walks through the necessary steps. Make a note of the Slack Webhook URL to use in your Python function.
- 2. Create an Airflow connection to provide your Slack Webhook to Airflow.** Choose an HTTP connection type (if you are using Airflow 2.0 or greater, you will need to install the `apache-airflow-providers-http` provider for the HTTP connection type to appear in the Airflow UI). Enter `https://hooks.slack.com/services/` as the Host, and enter the remainder of your Webhook URL from the last step as the Password (formatted as `T00000000/B00000000/XXXXXXXXXXXXXXXXXXXXXXXXXX`).

The screenshot shows the 'Add Connection' form in the Airflow web interface. The form is titled 'Add Connection' and contains the following fields:

- Conn Id:** A text input field containing 'slack_webhook'.
- Conn Type:** A dropdown menu set to 'HTTP'. Below it, a message reads: 'Conn Type missing? Make sure you've installed the corresponding Airflow Provider Package.'
- Description:** A large text area for providing a description.
- Host:** A text input field containing 'https://hooks.slack.com/services/'.
- Schema:** An empty text input field.
- Login:** An empty text input field.
- Password:** A password input field with masked characters (dots).
- Port:** An empty text input field.
- Extra:** A large text area for additional configuration.

At the bottom left of the form, there is a 'Save' button with a left-pointing arrow.

3. Create a Python function to use as your `on_failure_callback` method.

Within the function, define the information you want to send and invoke the `SlackWebhookOperator` to send the message. Here's an example:

```
1 from airflow.providers.slack.operators.slack_webhook import
2 SlackWebhookOperator
3
4 def slack_notification(context):
5     slack_msg = """
6         :red_circle: Task Failed.
7         *Task*: {task}
8         *Dag*: {dag}
9         *Execution Time*: {exec_date}
10        *Log Url*: {log_url}
11        """.format(
12            task=context.get('task_instance').task_id,
13            dag=context.get('task_instance').dag_id,
14            ti=context.get('task_instance'),
15            exec_date=context.get('execution_date'),
16            log_url=context.get('task_instance').log_url,
17        )
18    failed_alert = SlackWebhookOperator(
19        task_id='slack_notification',
20        http_conn_id='slack_webhook',
21        message=slack_msg)
22    return failed_alert.execute(context=context)
```

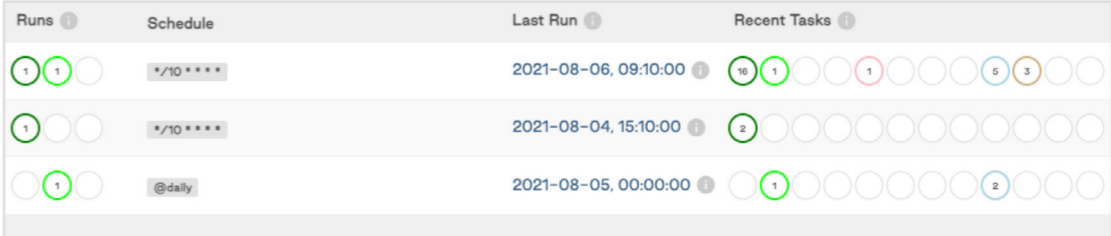
Note: In Airflow 2.0 or greater, to use the `SlackWebhookOperator` you will need to install the `apache-airflow-providers-slack` provider package.

4. Define your `on_failure_callback` parameter in your DAG either as a `default_arg` for the whole DAG, or for specific tasks. Set it equal to the function you created in the previous step. You should now see any failure notifications show up in Slack.

States

One of the key pieces of data stored in Airflow's metadata database is State. States are used to keep track of what condition task instances and DAG Runs are in. In the screenshot below, we can see how states are represented in the Airflow UI:

DAG Runs and tasks can have the following states:



Runs	Schedule	Last Run	Recent Tasks
1	* / 10 * * * * *	2021-08-06, 09:10:00	16 (1) (1) (1) (1) (1) (1) (1) (1) (1) (1) (1) (1) (1) (1) (1) (1)
1	* / 10 * * * * *	2021-08-04, 15:10:00	2 (1) (1) (1) (1) (1) (1) (1) (1) (1) (1) (1) (1) (1) (1) (1) (1)
1	@daily	2021-08-05, 00:00:00	1 (1) (1) (1) (1) (1) (1) (1) (1) (1) (1) (1) (1) (1) (1) (1) (1)

- **Running (Lime):** DAG is currently being executed.
- **Success (Green):** DAG was executed successfully.
- **Failed (Red):** The task or DAG failed.

Task States

- **None (Light Blue):** No associated state. Syntactically - set as Python None.

- **Queued (Gray)**: The task is waiting to be executed, set as queued.
- **Scheduled (Tan)**: The task has been scheduled to run.
- **Running (Lime)**: The task is currently being executed.
- **Failed (Red)**: The task failed.
- **Success (Green)**: The task was executed successfully.
- **Skipped (Pink)**: The task has been skipped due to an upstream condition.
- **Shutdown (Blue)**: The task is up for retry.
- **Removed (Light Grey)**: The task has been removed.
- **Retry (Gold)**: The task is up for retry.
- **Upstream Failed (Orange)**: The task will not run because of a failed upstream dependency.

Airflow SLAs

[Airflow SLAs](#) are a type of notification that you can use if your tasks are taking longer than expected to complete. If a task takes longer than a maximum amount of time to complete as defined in the SLA, the SLA will be missed and notifications will be triggered. This can be useful in cases where you have potentially long-running tasks that might require user intervention after a certain period of time or if you have tasks that need to complete by a certain deadline.

Note that exceeding an SLA will not stop a task from running. If you want tasks to stop running after a certain time, try using [timeouts](#) instead.

You can set an SLA for all tasks in your DAG by defining `'sla'` as a default argument, as shown in the DAG below:

```

1  from airflow import DAG
2  from airflow.operators.dummy_operator import DummyOperator
3  from airflow.operators.python_operator import PythonOperator
4  tor
5  from datetime import datetime, timedelta
6  import time
7
8  def my_custom_function(ts,**kwargs):
9      print("task is sleeping")
10     time.sleep(40)
11
12     # Default settings applied to all tasks
13     default_args = {
14         'owner': 'airflow',
15         'depends_on_past': False,
16         'email_on_failure': True,
17         'email': 'noreply@astronomer.io',
18         'email_on_retry': False,
19         'sla': timedelta(seconds=30)
20     }
21
22     # Using a DAG context manager, you don't have to specify
23     the dag property of each task
24     with DAG('sla-dag',
25             start_date=datetime(2021, 1, 1),
26             max_active_runs=1,
27             schedule_interval=timedelta(minutes=2),
28             default_args=default_args,
29             catchup=False
30             ) as dag:

```

```

31     t0 = DummyOperator(
32         task_id='start'
33     )
34
35     t1 = DummyOperator(
36         task_id='end'
37     )
38
39     sla_task = PythonOperator(
40         task_id='sla_task',
41         python_callable=my_custom_function
42     )
43     t0 >> sla_task >> t1

```

SLAs have some unique behaviors that you should consider before implementing them:

- **SLAs are relative to the DAG execution date, not the task start time.** For example, in the DAG above the `sla_task` will miss the 30 second SLA because it takes at least 40 seconds to complete. The `t1` task will also miss the SLA, because it is executed more than 30 seconds after the DAG execution date. In that case, the `sla_task` will be considered “blocking” to the `t1` task.
- **SLAs will only be evaluated on scheduled DAG Runs.** They will not be evaluated on manually triggered DAG Runs.
- **SLAs can be set at the task level if a different SLA is required for each task.** In this case, all task SLAs are still relative to the DAG execution date. For example, in the DAG below, `t1` has an SLA of 500 seconds. If the upstream tasks (`t0` and `sla_task`) combined take 450 seconds to complete, and `t1` takes 60 seconds to complete, then `t1` will miss its SLA even though the task did not take more than 500 seconds

```

1     from airflow import DAG
2     from airflow.operators.dummy_operator import DummyOp-
3         erator
4     from airflow.operators.python_operator import Pytho-
5         nOperator
6     from datetime import datetime, timedelta
7     import time
8
9     def my_custom_function(ts,**kwargs):
10         print("task is sleeping")
11         time.sleep(40)
12
13     # Default settings applied to all tasks
14     default_args = {
15         'owner': 'airflow',
16         'depends_on_past': False,
17         'email_on_failure': True,
18         'email': 'noreply@astronomer.io',
19         'email_on_retry': False
20     }
21
22     # Using a DAG context manager, you don't have to spec-
23         ify the dag property of each task
24     with DAG('sla-dag',
25             start_date=datetime(2021, 1, 1),
26             max_active_runs=1,
27             schedule_interval=timedelta(minutes=2),
28             default_args=default_args,
29             catchup=False
30         ) as dag:

```

```

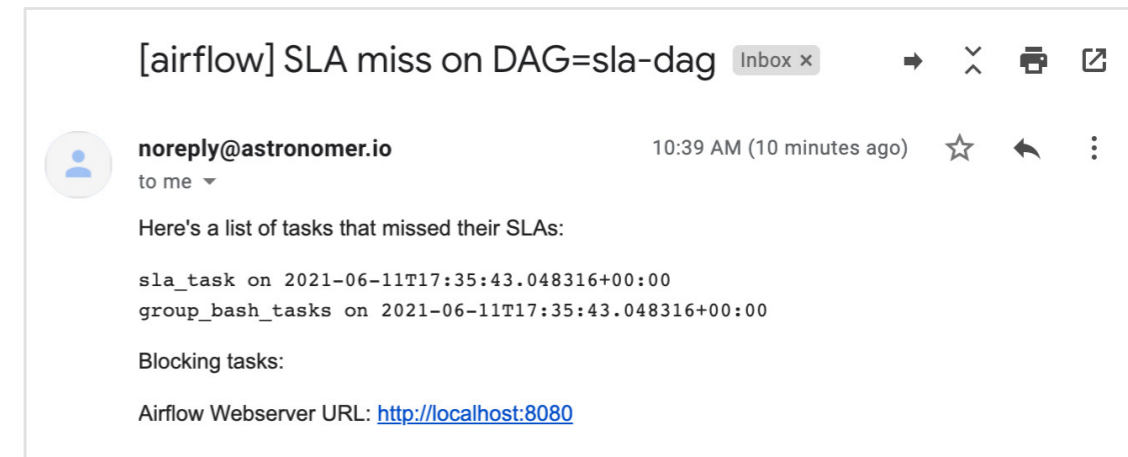
31     t0 = DummyOperator(
32         task_id='start',
33         sla=timedelta(seconds=50)
34     )
35
36     t1 = DummyOperator(
37         task_id='end',
38         sla=timedelta(seconds=500)
39     )
40
41     sla_task = PythonOperator(
42         task_id='sla_task',
43         python_callable=my_custom_function,
44         sla=timedelta(seconds=5)
45     )
46
46     t0 >> sla_task >> t1

```

Any SLA misses will be shown in the Airflow UI. You can view them by going to Browse SLA Misses, which looks something like this:

Dag Id	Task Id	Execution Date	Email Sent	Timestamp
sla-dag	end	2021-06-11, 17:43:50	True	2021-06-11, 17:46:30
sla-dag	sla_task	2021-06-11, 17:43:50	True	2021-06-11, 17:46:30
sla-dag	sla_task	2021-06-11, 17:41:19	True	2021-06-11, 17:43:49
sla-dag	group_bash_tasks	2021-06-11, 17:39:16	True	2021-06-11, 17:42:00
sla-dag	sla_task	2021-06-11, 17:39:16	True	2021-06-11, 17:42:00
sla-dag	group_bash_tasks	2021-06-11, 17:37:43	True	2021-06-11, 17:41:18
sla-dag	sla_task	2021-06-11, 17:37:43	True	2021-06-11, 17:41:18
sla-dag	group_bash_tasks	2021-06-11, 17:35:43	True	2021-06-11, 17:41:18
sla-dag	sla_task	2021-06-11, 17:35:43	True	2021-06-11, 17:41:18

If you configured an SMTP server in your Airflow environment, you will also receive an email with notifications of any missed SLAs.



Note that there is no functionality to disable email alerting for SLAs. If you have an `'email'` array defined and an SMTP server configured in your Airflow environment, an email will be sent to those addresses for each DAG Run that has missed SLAs.

Notifications on Astronomer

If you are running Airflow on the Astronomer platform, you have multiple options for managing your Airflow notifications. All of the methods above for sending task notifications from Airflow are easily implemented on Astronomer. Our [documentation discusses](#) how to leverage these notifications on the platform, including how to set up SMTP to enable email alerts.

Astronomer also provides deployment and platform-level alerting to notify you if any aspect of your Airflow or Astronomer infrastructure is unhealthy.

ASTRONOMER

Thank you

We hope you've enjoyed our guide to DAGs. 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 the guide:

Marc Lamberti | Head of Customer Training at Astronomer

Kenten Danas | Lead Developer Advocate at Astronomer

Jake Witz | Technical Writer at Astronomer

Created by © Astronomer 2022, Revised edition

ASTRONOMER